Studying Multicore Processor Scaling via Reuse Distance Analysis

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

ABSTRACT

The trend for multicore processors is towards increasing numbers of cores, with 100s of cores–*i.e.* large-scale chip multiprocessors (LCMPs)–possible in the future. The key to realizing the potential of LCMPs is the cache hierarchy, so studying how memory performance will scale is crucial. Reuse distance (RD) analysis can help architects do this. In particular, recent work has developed *concurrent reuse distance (CRD)* and *private reuse distance (PRD)* profiles to enable analysis of shared and private caches. Also, techniques have been developed to predict profiles across problem size and core count, enabling the analysis of configurations that are too large to simulate.

This paper applies RD analysis to study the scalability of multicore cache hierarchies. We present a framework based on CRD and PRD profiles for reasoning about the locality impact of core count and problem scaling. We find *interference-based* locality degradation is more significant than *sharing-based* locality degradation. For 256 cores running small problems, the former occurs at small cache sizes, allowing moderate capacity scaling of multicore caches to achieve the same cache performance (MPKI) as a single-core cache. At very large problems, interference-based locality degradation increases significantly in many of our benchmarks. For shared caches, this prevents most of our benchmarks from achieving constant-MPKI scaling within a 256 MB capacity budget; for private caches, all benchmarks cannot achieve constant-MPKI scaling within 256 MB.

1. INTRODUCTION

The trend for high-performance microprocessors is towards an increasing number of on-chip cores. Today, CPUs with 8–10 state-of-the-art cores or 10s of smaller cores [1, 6] are commonplace. In the near future, CPUs with 100s of cores–*i.e.* large-scale chip multiprocessors (LCMPs) [7, 21]–will be possible. Such processors will offer enormous performance potential for programs that exhibit sufficient thread-level parallelism to utilize all of the available cores.

Fully realizing the performance potential of LCMPs, however, will be challenging. The problem is *scalability*. Many bottlenecks will threaten to limit performance scaling, one of which will be the memory hierarchy. As the number of on-chip cores increases–and potentially, as problem size increases too–greater pressure will be placed on the cache hierarchy. Although caches will also scale, memory performance may still degrade if the cache hierarchy is unable to keep on- and off-chip traffic at reasonable levels. In this case, the performance gains from core count scaling will be limited.

Given the importance of memory performance to multicore scalability, studying how parallel programs utilize the on-chip cache hierarchy as processors and problems scale to the LCMP level will be crucial. This will not only shed light on the nature and severity of the bottlenecks mentioned above, it will also inform architects on how best to scale the cache hierarchy to keep bottlenecks at bay.

Currently, the main approach for studying memory performance is architectural simulation. Unfortunately, simulating CPUs with 100s of cores is extremely slow, at best. Simulating LCMPs running moderate–let alone large–input problems is beyond the capabilities of even the fastest simulators. Moreover, simulations only represent individual architecture configurations and input problems. So, studying scaling trends requires running simulation sweeps. Unfortunately, the cross-product of core counts, cache sizes, cache organizations, and problem sizes can yield 1000s to millions of configurations. With simulation's high cost, it is infeasible to explore such combinatorially large configuration spaces.

A powerful tool that can help architects study multicore scaling 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. Once acquired, RD profiles can be used to predict cache performance across numerous configurations which can dramatically accelerate scaling analysis. For example, in sequential programs, an RD profile can predict performance at any cache size, permitting exhaustive analysis along the cache capacity dimension after acquiring just one profile.

Compared to the sequential case, RD analysis for multicore processors is much more complex because data locality

This research was supported in part by NSF under grant #CCF-1117042, and in part by DARPA under grant #HR0011-13-2-0005. 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.

ISCA'13 Tel-Aviv, Israel

in parallel programs depends not only on per-thread reuse, but also on how simultaneous threads' memory references interact. Many thread interactions can occur across different types of caches. For instance, inter-thread memory reference interleaving leads to *interference* in shared caches. In addition, *data sharing* leads to replication and communication across private caches.

Recent research has tried to account for these thread interactions to enable RD analysis for multicore processors. A key innovation is the introduction of new locality profiles that quantify thread interactions' impact on locality in shared and private caches. *Concurrent reuse distance (CRD) profiles* [4, 9, 16, 15, 19, 20] quantify reuse across thread-interleaved memory reference streams, and account for interactions in shared caches. *Private-stack reuse distance (PRD) profiles* [16, 15, 20] quantify reuse within per-thread memory reference streams under invalidation-based coherence, and account for interactions in private caches.

Much like RD profiles for uniprocessors, CRD and PRD profiles can accelerate cache capacity scaling analysis for multicore CPUs. But the leverage does not end there. Recent research has also investigated *profile prediction techniques*. In particular, CRD and PRD profiles can be predicted across different core counts by analyzing and accounting for the effects of increased memory interleaving [4, 9, 19, 20]. Locality profiles can also be predicted for different problem sizes. Early techniques worked for RD profiles and uniprocessors only [22], but recent research has applied similar techniques to CRD and PRD profiles [19, 20].

One limitation of these techniques is their sensitivity to memory interleaving: both CRD and PRD are defined only for a given interleaving of threads' memory reference streams. This has two repercussions. First, because memory interleaving depends on timing, CRD and PRD profiles are *architecture dependent*. This can complicate cache scaling analysis from a single pair of CRD/PRD profiles since the profiles may not accurately reflect locality for the predicted caches if per-thread timing changes substantially. And second, interleaving sensitivity also complicates profile prediction. As core count or problem size scale, it is unclear how scaled memory reference streams interleave. Composition techniques for predicting core count scaling [4, 9] analyze traces and consider all possible interleavings of the scaled reference streams. Unfortunately, this approach is intractable for even modest machine and problem sizes, let alone LCMPs.

A silver lining for multicore RD analysis is that interleaving sensitivity is minor for certain types of parallel programs. Parallel programs express either *task-based* or *loop-based* parallelism. In task-based parallel programs, threads execute dissimilar code, giving rise to irregular memory interleavings which tend to exacerbate interleaving-related problems. In loop-based parallel programs, however, simultaneous threads execute similar code–*i.e.*, from the same parallel loop–so they exhibit similar locality characteristics. Such *symmetric threads* produce regular memory interleavings which tend to ameliorate interleaving-related problems.

In particular, research has shown symmetric threads speed-up or slow down by similar amounts as cache size changes. So, CRD and PRD profiles are fairly stable across different cache capacities, permitting them to make very accurate cache performance predictions [9, 20]. Moreover, while CRD and PRD profiles change across core count and problem size, they do so in a systematic fashion when threads are symmetric. This permits scaled profiles to be predicted with high accuracy using extremely simple prediction algorithms [19, 20]. Hence, for parallel programs with symmetric threads, state-of-the-art RD analysis is capable of performing sophisticated and accurate scaling analyses.

While significant research has investigated RD analysis for multicores, existing work has focused on developing the analysis techniques. Far less research has *applied* the techniques to actually study memory behavior. In this paper, we use RD analysis to investigate the scalability of multicore cache hierarchies, focusing on LCMP-sized processors and problems. An important thrust of our work is to explore scaling beyond what is possible to simulate, thus providing insights that would be impossible to obtain via traditional simulation-based methodologies. To our knowledge, this is the first-ever scalability study using RD analysis.

Our work proceeds in several parts, making many contributions along the way. First, we present a framework for reasoning about multicore scaling based on previous RD analysis work. This framework uses CRD and PRD profiles to quantify locality degradation due to core count scaling, and to break down the overall locality impact into *interference-based* and *sharing-based* components. Two metrics, C_{core} and C_{share} [20], are presented that identify the cache capacities over which these components occur. We also propose a new profile, PRD^{C_l}, which breaks down the amount of sharing-based locality degradation that *cluster caches* can alleviate. Then, we present *reference groups* [22] for predicting the locality profiles of core count and problem scaled configurations.

Second, we apply our framework to study the locality degradation due to core count scaling. This initial study considers configurations small enough to be profiled: CPUs with 256 cores running problems that complete in 10s of billions of instructions. Across 16 SPLASH2 and PARSEC benchmarks, we find interference-based locality degradation is much more significant than sharing-based locality degradation. Due to the small problem sizes, the former is confined to small capacities, so small shared caches can contain this locality impact. Sharing-based locality degradation is suite dependent, with some SPLASH2 programs exhibiting coarse-grain sharing and the remaining programs exhibiting fine-grain sharing. On average, sharing-induced locality degradation is noticeable starting at 13.7 MB on average.

Third, we also use our framework to study the scaling of cache capacity and cluster size to compensate for the effects of core count scaling–in particular, to match the MPKI observed on a 1-core CPU. We find the rate of capacity scaling needed for shared caches is much smaller than for private caches due to the limited impact of interference-based locality degradation. For the baseline problem sizes, constant-MPKI scaling at 256 cores is always achievable within 256 MB of on-chip cache.

Finally, we use profile prediction to study input problems that complete in 100s of trillions of instructions, and CPUs with up to 1024 cores. We find problem scaling increases the capacity across which interference-based locality degradation occurs for many benchmarks. This increases the shared cache scaling needed to maintain constant MPKI. For our largest problems, only 3 benchmarks achieve constant-MPKI scaling within 256 MB of shared cache; for private caches, no benchmarks can achieve constant-MPKI scaling within 256 MB. Cluster caches can help bridge the gap between private

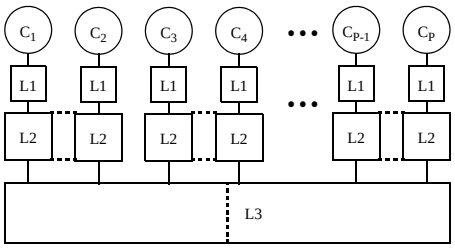


Figure 1: Example 3-level multicore cache hierarchies.

Time:	1	2	3	4	5	6	7	8	9	10	11
Core C_1:	A	B	C		D		E			A	C
Core C_2:				F	C (A)		G	H			

Figure 2: Two interleaved memory reference streams, illustrating different thread interactions.

and shared caches, but they do not change the poor scaling behavior that larger problems exhibit fundamentally.

The rest of this paper is organized as follows. Section 2 reviews and extends previous multicore RD analyses, establishing a framework for analyzing multicore memory performance scaling. Next, Section 3 applies the framework to investigate core count scaling on baseline configurations. Then, Section 4 investigates how problem scaling and continued core count scaling affects the results. Finally, Sections 5 and 6 discuss related work and conclusions.

2. MULTICORE SCALING FRAMEWORK

In the past, RD profiles have been used to analyze cache capacity scaling for uniprocessors. An RD profile is a histogram of RD values for all memory references in a sequential program, where each RD value measures the number of unique memory blocks referenced since the last reference to the same data block. Because a cache of capacity C can satisfy references with RD $< C$ (assuming LRU management), the cache-miss count is the sum of all reference counts in an RD profile above the RD value for capacity C.

This section discusses CRD, PRD, and other profiles which extend this basic notion of reuse distance to handle multicore CPUs (Section 2.1). Then, we show how these profiles breakdown different sources of locality degradation, indicating how effective different cache scaling techniques will be (Section 2.2). Lastly, we discuss the impact of core count and problem scaling on the locality profiles (Section 2.3).

2.1 Locality Profiles

2.1.1 CRD and PRD.

Multicore CPUs often contain shared and private caches. For example, the solid lines in Figure 1 illustrate a CPU consisting of two levels of private cache backed by a shared L3 cache. Threads interact differently in each type of cache, requiring separate locality profiles.

CRD profiles report locality for thread-interleaved memory reference streams, thus capturing interference in shared caches. CRD profiles can be measured by applying the interleaved stream on a single (global) LRU stack [9, 16, 15, 19, 20]. For example, Figure 2 illustrates the interleaving of 2 cores' memory references accessing 8 memory blocks, A–H. Figure 3(a) shows the state of the global LRU stack

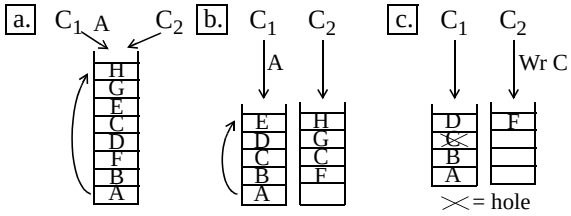


Figure 3: LRU stacks showing (a) dilation and overlap for CRD, (b) scaling, replication, and (c) holes for PRD.

when core C_1 re-references A at $t = 10$. C_1's reuse of A exhibits an intra-thread RD = 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 parallel programs, threads share data which offsets dilation in two ways. First, it introduces *overlapping references*. For example, in Figure 2, while C_2's reference to C interleaves with C_1's reuse of A, this doesn't increase A's CRD because C_1 already references C in the reuse interval. Second, sharing also introduces *intercepts*. For example, if C_2 references A instead of C at $t = 6$ in Figure 2, then C_1's reuse of A has CRD = 3, so CRD is actually *less* than RD.

PRD profiles report locality across per-thread memory reference streams that access coherent private caches. PRD profiles can be measured by applying threads' references on private LRU stacks that are kept coherent. Without writes, the private stacks do not interact. For example, Figure 3(b) shows the PRD stacks corresponding to Figure 3(a) 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. (Here, the capacity needed to satisfy PRD = 4 is 10, assuming 2 caches with 5 cache blocks each). To capture this effect, we compute the *scaled PRD* (sPRD) which equals $T \times PRD$, where T is the number of threads. Hence, for C_1's reuse of A, sPRD = 8.

In PRD profiles, read sharing causes *replication*, increasing overall capacity pressure. Figure 3(b) shows duplication of C in the private stacks. Because PRD scaling aggregates private LRU stacks, replication effects are automatically captured in sPRD profiles. In contrast, write sharing causes *invalidation*. For example, if C_2's reference to C is a write instead of a read, then invalidation would occur in C_1's stack, as shown in Figure 3(c). To prevent invalidations from promoting blocks further down the LRU stack, invalidated blocks become *holes* rather than being removed from the stack [16]. Holes are unaffected by references to blocks above the hole, 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 3(c), D would be pushed down and the hole would move to depth 3, preserving A's stack depth.

2.1.2 $\mathbf{PRD^{C_1}}$.

CRD and PRD profiles address the basic shared and private caches in muticore processors. Multicores may also employ *cluster caches*, especially when scaling to LCMPs. These are hybrid caches in which the cores from a cluster access a common shared cache, but all of the per-cluster caches are treated as private caches requiring coherence. For example, Figure 1's dotted lines suggest pair-wise clustered

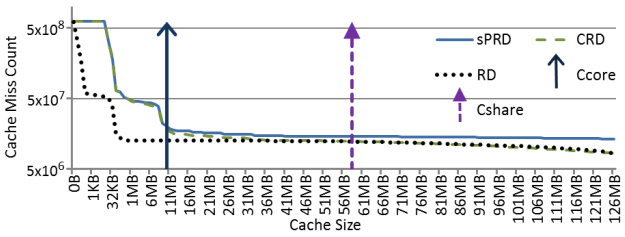


Figure 4: CMC as a function of cache capacity computed from the RD, CRD, and PRD profiles for the FFT benchmark.

L2s, and two clustered L3s each shared by half the CPU. We assume all clusters in a CPU contain the same number of cores, C_l (the cluster size), which always evenly divides P, the total number of cores.

The locality profile for a group of clustered caches is the PRD profile acquired using $\frac{P}{C_l}$ private stacks in which each private stack's memory reference stream is formed by interleaving the streams from the C_l cores belonging to the same cluster. We call the locality profile acquired in this fashion PRD^{C_l} (sPRD^{C_l}). Notice, the C_l parameter defines a family of caches in which private and shared caches are the degenerate cases, $C_l = 1$ and P, respectively. In other words, PRD = PRD^1 and CRD = PRD^P.

2.2 Locality Degradation

RD analysis can quantify the magnitude and source of locality degradation due to core count scaling. Figure 4 illustrates the insights. We plot the cache-miss count (CMC) profiles corresponding to the CRD and sPRD profiles for FFT from SPLASH2 [18] running on a 256-core CPU; the CMC profile for FFT's RD profile (1-core case) is also plotted. Each CMC profile reports the cache misses predicted by its corresponding locality profile as a function of reuse distance (*e.g.*, $CMC[i] = \sum_{j=i}^{\infty} CRD[j]$ for the CRD profile). Along the X-axis, reuse distance is plotted in terms of capacity; along the Y-axis, CMC is plotted on a log scale.

As Figure 4 shows, scaling core count shifts the RD profile to larger capacities, creating gaps that represent locality degradation. In particular, the gap between the CRD and RD profiles quantifies the cache-miss increase for shared caches. This quantifies the impact of destructive interference from interleaving memory references to non-overlapping data.

The gap between the sPRD and RD profiles in Figure 4 quantifies the cache-miss increase for private caches. This quantifies the impact of PRD scaling, which includes the accumulation of both per-thread non-overlapping data as well as shared data, *i.e.* replication, and the impact of holes created by write invalidations. Previous work has shown that for symmetric threads, PRD scaling of non-overlapping data has an identical locality impact as memory interleaving interference in shared caches [19, 20]. So, in the absence of sharing effects–replication and invalidation–CRD and sPRD profiles are coincident. This also implies sPRD profiles are always above or equal to CRD profiles, and any gap between the two quantifies the extra misses due to replicas and invalidations.

Finally, since clustered caches are hybrids, with private and shared caches as degenerate cases, sPRD^{C_l} profiles lie in between sPRD and CRD profiles, starting from sPRD and moving towards CRD as C_l increases from 1 to P. (These profiles have been omitted from Figure 4 for clarity). This group of sPRD^{C_l} profiles shows the benefit of increasing cluster size to reduce inter-cluster replicas and invalidations.

2.2.1 Scaling Capacity vs. Clustering.

Figure 4 defines two types of locality degradation: interference-based (CRD-RD gap) and sharing-based (sPRD-CRD gap). *Breaking down different types of locality degradation informs architects on how effective different cache scaling techniques will be.* In both cases, scaling cache capacity can improve cache performance. This is because both CRD and sPRD profiles decrease monotonically along the X-axis. However, for sharing-based locality degradation, scaling cluster size can also improve cache performance. This reduces replicas and invalidations along the sPRD-CRD gap at a given cache capacity.

In addition to breaking down interference and sharing, Figure 4 also shows across which cache capacities each type of locality degradation occurs. *This informs architects on the range of cache sizes, and hence the caching levels, where capacity and cluster size scaling will be effective.*

Interference-based degradation occurs below some capacity, called C_{core} [19, 20]. For example, Figure 4 shows the CRD-RD gap narrows at C_{core} = 10 MB, and eventually closes shortly thereafter. One reason for this is parallelization scope in programs is limited, *e.g.* to a single parallel loop. Because parallel regions typically access less data than a program's total memory footprint, the amount of destructive interference is also limited. Essentially, C_{core} quantifies a program's "parallel working set size." Although not the case in Figure 4, the CRD-RD gap may not completely close in some programs. In these cases, the CRD-RD gap still becomes narrow, so C_{core} can be defined where the gap is near its minimum [20].

Sharing-based degradation occurs beyond some capacity, called C_{share} [19, 20]. For example, Figure 4 shows the sPRD-CRD gap opens at C_{share} = 58 MB. The reason for this is certain parallelization schemes, such as blocking, limit the minimum sharing distance. Essentially, C_{share} quantifies the resulting "granularity of sharing" that a parallel program exhibits.

2.3 Core Count and Problem Scaling

Figure 4 shows the analysis at fixed core count and problem size. Our multicore scaling framework can also analyze core count and problem scaling effects. This is done by comparing CRD/PRD profiles at different configurations to identify trends.

Core count scaling usually increases both interference and sharing. The former shifts the leading edge of CRD and sPRD profiles towards higher cache capacities. For CRD profiles, this increases CMC in the interference region, and also increases C_{core}. But beyond C_{core}, the CRD-RD gap remains small or completely closed due to the limited scope of interference. Increased sharing raises the height of the "horizontal" portion of sPRD profiles. In some applications, this can cause C_{share} to shift towards smaller cache sizes.

Problem scaling more universally degrades locality. It increases the height of CRD and sPRD profiles since the number of memory references scales with run time. In data-parallel programs, problem scaling also increases memory footprint size. This shifts most of CRD/sPRD profiles to larger cache sizes, often increasing C_{core} and/or C_{share}.

Previous research has shown profile shift can be highly

Benchmarks	Profiled Problems S1/S2	Instruction Count (in millions) S1	S2		Interference Region ΔM_m	ΔM_a	MPKI	C_{core}
fft	$2^{20}/2^{22}$ elements	598.2	2571.8		11.8	3.6	10.8	10.0MB
lu	$1024^2/2048^2$ elements	2979.9	23815.9		128.7	28.0	49.7	640KB
radix	$2^{22}/2^{24}$ keys	784.4	3137.4		10.4	3.0	13.0	14.0MB
barnes	$2^{17}/2^{19}$ particles	4470.1	19305.7		189.3	12.7	27.6	6.5MB
fmm	$2^{17}/2^{19}$ particles	4090.1	16497.5		42.6	4.4	7.6	8.8MB
ocean	$514^2/1026^2$ grid	417.3	1623.6		12.6	5.9	110.6	256KB
water	$25^3/40^3$ molecules	497.5	1862.3		105.0	15.1	45.8	2.4MB
kmeans	$2^{20}/2^{22}$ objects, 18 features	2666.8	10674.7		623.9	149.9	49.2	384KB
blackscholes	$2^{20}/2^{22}$ options	989.4	3872.3		197.4	56.7	87.3	128KB
bodytrack	B_4,8k/B_261,16k particles	5262.0	10462.9		59.4	10.1	14.5	4.6MB
canneal	400000/2500000.net	9.1	10.0		3.4	2.3	127.6	8KB
fluidanimate	in_300k/500k.fluid	2311.0	3729.0		312.8	42.4	39.8	2.0MB
raytrace	1920x1080/3840x2160 pixels	2834.6	7402.2		689.7	71.9	29.0	4.0MB
swaptions	$2^{16}/2^{18}$ swaptions	1263.9	5055.8		107.6	19.2	56.2	1.4MB
streamcluster	$2^{16}/2^{18}$ data points	6351.0	26656.3		13.7	6.7	74.5	128KB
vips	2336x2336/2662x5500 pixels	11953.8	33500.4		708.0	362.0	1.3	85.6MB
Average					201.0	49.6	46.5	8.8MB
Benchmark	Predicted Problems S3/S4/S5	Instruction Count (in billions) S3	S4	S5				
ALL	-	60	3600	604800				

Table 1: Parallel benchmarks, problems S1 and S2 and their instruction counts (in millions), and characterization of the interference-based locality degradation region. The last row reports instruction counts for predicted problems S3 – S5 (in billions).

systematic, especially for programs with symmetric threads. In this case, core count scaling increases interleaving from threads with almost identical locality characteristics. This causes both dilation of intra-thread RD in the interference region and replication in the sharing region to occur systematically [19, 20]. Also, problem scaling expands computation structures (*e.g.*, loops) and data structures systematically [22]. The net effect is *profile shift due to core count and problem scaling preserves the shape of CRD and sPRD profiles, making the shift highly predictable.*

Zhong proposed reference groups [22] to predict shape-preserving profile shift. The idea is to "diff" two profiles, aligning groups of references that "correspond" across the shift and measuring their shift rates. Then, scaled shift rates are applied to each reference group to form scaled-up profiles. Individual reference groups are allowed to use different shift rates, so the technique can track non-uniform shift. Zhong originally predicted RD profiles across problem scaling. Our own previous work [19, 20] extended reference groups to predict CRD and sPRD profiles across core count scaling, and applied the technique for predicting CRD/sPRD across problem scaling. Results show the technique is very accurate.

3. BASELINE SCALING RESULTS

This section uses the framework defined in Section 2 to measure the locality impact of scaling core count on several parallel benchmarks assuming a baseline configuration. Then, we study how the cache hierarchy must scale in order to address the locality impact.

3.1 Experimental Methodology

We implemented locality profiling within the Intel PIN tool [11]. We modified PIN to maintain a global LRU stack to acquire CRD profiles and coherent private LRU stacks to acquire PRD profiles following the techniques discussed in Section 2.1. Our coherent private LRU stacks are also used to acquire PRD^{C_l} profiles. In this case, we group C_l memory reference streams at a time, interleaving the streams from each group and applying the per-cluster interleaved streams on the private LRU stacks. From the measured PRD and PRD^{C_l} profiles, we also derive sPRD and sPRD^{C_l} profiles. Finally, we implemented single-core RD profiling in PIN as well. We assume 64-byte memory blocks in all LRU stacks.

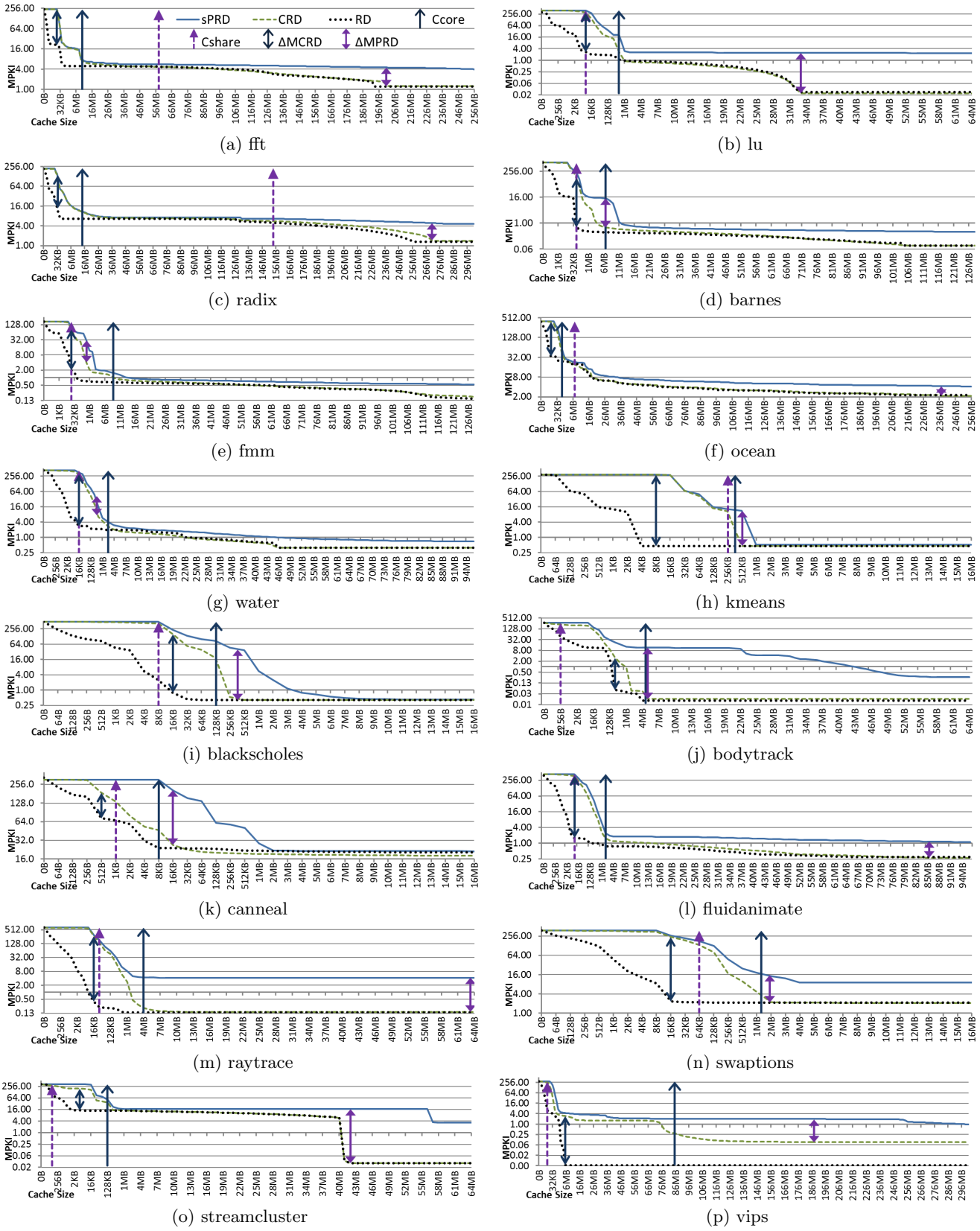
Our PIN tool follows McCurdy's method [12] which performs functional execution only, context switching threads after every memory reference. This interleaves threads' memory references uniformly in time. Studies have shown that for symmetric threads, this approach yields locality profiles that accurately reflect locality on real CPUs [9, 20].

Using PIN, we profiled 16 parallel benchmarks across 2 different problem sizes. Table 1 lists the benchmarks: the first 7 are from SPLASH2 [18], kmeans is from MineBench [13], and the last 8 are from PARSEC [2]. The 2^{nd} column in Table 1 specifies the 2 profiled problem sizes, S1 and S2, and the next 2 columns in Table 1 specify the number of instructions executed by each input problem (assuming sequential runs). For each benchmark and problem size, we acquired the RD (1-core) profile, and the CRD, sPRD, and sPRD^{C_l} profiles for 128 and 256 cores. For sPRD^{C_l} profiles, we profiled cluster sizes of 4, 8, and 16 cores. Because results across cluster sizes are similar, we only present detailed results for the sPRD^{8} profiles.

In addition to profiling runs, we also used reference groups, as described in Section 2.3, to predict additional configurations. Section 4 will discuss our prediction experiments.

3.2 Locality Impact

Figure 5 presents our baseline results for core count scaling across all 16 benchmarks. In Figure 5, we plot the misses per 1000 instructions (MPKI) associated with the RD, CRD, and sPRD profiles for each benchmark. (Each graph is in exactly the same format as Figure 4 except CMC values are normalized by $\frac{InstructionCount}{1000}$). We plot MPKI rather than CMC because MPKI better reflects performance. All of the multicore locality profiles assume S2 running on 256 cores, the largest problem and machine configuration we profiled on PIN. The RD profiles assume S2 running on 1 core.



(a) fft (b) lu

(c) radix (d) barnes

(e) fmm (f) ocean

(g) water (h) kmeans

(i) blackscholes (j) bodytrack

(k) canneal (l) fluidanimate

(m) raytrace (n) swaptions

(o) streamcluster (p) vips

Figure 5: 256-core CRD and sPRD profiles and RD profiles for the S2 problem showing interference and sharing regions in our benchmarks.

Benchmarks	Sharing Region			
	ΔM_m	ΔM_a	MPKI	C_{share}
fft	3.6	2.3	2.4	57.8MB
lu	129.7	90.6	0.1	8KB
radix	3.5	1.9	2.8	154.3MB
barnes	22.0	3.8	0.1	64KB
fmm	12.5	2.2	0.3	16KB
ocean	2.0	1.7	3.6	7MB
water	3.2	1.8	0.5	16KB
kmeans	23.5	1.1	0.5	256KB
blackscholes	101.2	1.3	0.4	8KB
bodytrack	692.5	69.5	0.1	256B
canneal	7.6	1.2	20.7	1KB
fluidanimate	4.1	3.5	0.3	8KB
raytrace	30.3	29.9	0.1	32KB
swaptions	6.9	4.3	2.2	64KB
streamcluster	657.4	136.2	1.6	128B
vips	21.4	13.4	0.02	2KB
Average	107.6	22.8	2.2	13.7MB

Table 2: Characterization of the sharing-based locality degradation region.

Notice, all graphs in Figure 5 exhibit the same locality behaviors described in Section 2.2. There is always a CRD-RD gap at small RD values that eventually (or nearly) closes by some capacity, C_{core}. Also, CRD and sPRD profiles are coincident initially, and then at some capacity, C_{share}, a gap opens. While the magnitude of these locality degradations and the capacities across which they occur vary, *all benchmarks exhibit interference-based and sharing-based degradation regions*. Given the diverse nature of our benchmarks, this suggests that interference- and sharing-based locality impact along with their demarcation at C_{core} and C_{share} are fairly general notions, and can be used to reason about multicore scaling across many different parallel programs.

To better understand the scaling impact, we measure different features on the profiles, as indicated by the arrows in Figure 5, and report results in the columns labeled "Interference Region" and "Sharing Region" in Tables 1 and 2, respectively. In particular, we measure the ratio of the CRD and RD profiles and the sPRD and CRD profiles, quantifying the locality degradation in the interference and sharing regions, respectively. In Tables 1 and 2, we report both the maximum ratio, ΔM_m (indicated by the double-headed arrows in Figure 5), as well as the average across each region, ΔM_a. We also report the average 1-core MPKI over which the interference and sharing regions span. Lastly, we report C_{core} and C_{share} (solid and dashed single-headed arrows, respectively, in Figure 5).

As Table 1 shows, the cache-miss increase in the interference region due to scaling from 1 to 256 cores is substantial. The average increase within the region, ΔM_a is between 2.3× and 362.0×. Across all 16 benchmarks, it is 49.6×. The maximum increase, ΔM_m, is $> 100\times$ in 9 benchmarks, and is $> 10\times$ in all but one benchmark. Moreover, the interference region spans large MPKI values. As Table 1 shows, the MPKI in the interference region is > 10.0 in all but two benchmarks. So, the large locality degradation will likely impact overall CPU performance significantly.

As Table 2 shows, the cache-miss increase in the sharing region due to scaling from 1 to 256 cores can be large, but not as consistently as in the interference region. The ΔM_a within the region is between 1.1× and 136.2×. In 5 of the benchmarks, there are double-digit ($> 10\times$) increases, but in the remaining 11 benchmarks, increases are much more modest. Averaged across all 16 benchmarks, ΔM_a is 22.8×. Similar behavior is observed for ΔM_m. Also, compared to the interference region, the sharing region occurs across much smaller MPKI values. In 10 benchmarks, the MPKI is ≤ 0.5, and on average it's only 2.2. So, even in those benchmarks where cache-miss increase is large, there still may not be a significant impact on overall performance. In Figure 5, we see locality degradation due to sharing rises to non-trivial MPKI for fft, lu, radix, raytrace, swaptions, and streamcluster. It also has an impact in blackscholes, kmeans, bodytrack, and canneal, but only for a very limited range of cache capacities. In the remaining cases, sharing will not have a big impact on overall CPU performance.

Next, Tables 1 and 2 also report where different locality degradation regions occur. The last column in Table 1 shows that except for vips, C_{core} is modest or small. In most cases, $C_{core} \leq 14$MB. So, for the profiles in Figure 5, interference-based locality degradation is confined to small caches–*i.e.*, our benchmarks' parallel working sets fit on-chip (at least for S2). In vips, scaling to 256 cores results in $C_{core} = 86$MB.

The last column in Table 2 shows C_{share} is somewhat suite dependent. For a few SPLASH2 benchmarks (fft, radix, and ocean), C_{share} is large, between 7–155MB. But for kmeans and the entire PARSEC suite, C_{share} is always small, $<$ 1MB. In other words, SPLASH2 benchmarks can exhibit coarse-grain sharing, whereas PARSEC benchmarks tend to exhibit finer-grained sharing. Granted, several SPLASH2 benchmarks in Table 2 have small C_{share} too, but Section 4 will show the suite-dependent nature of sharing granularity becomes more pronounced with problem scaling.

3.3 Cache Hierarchy Scaling

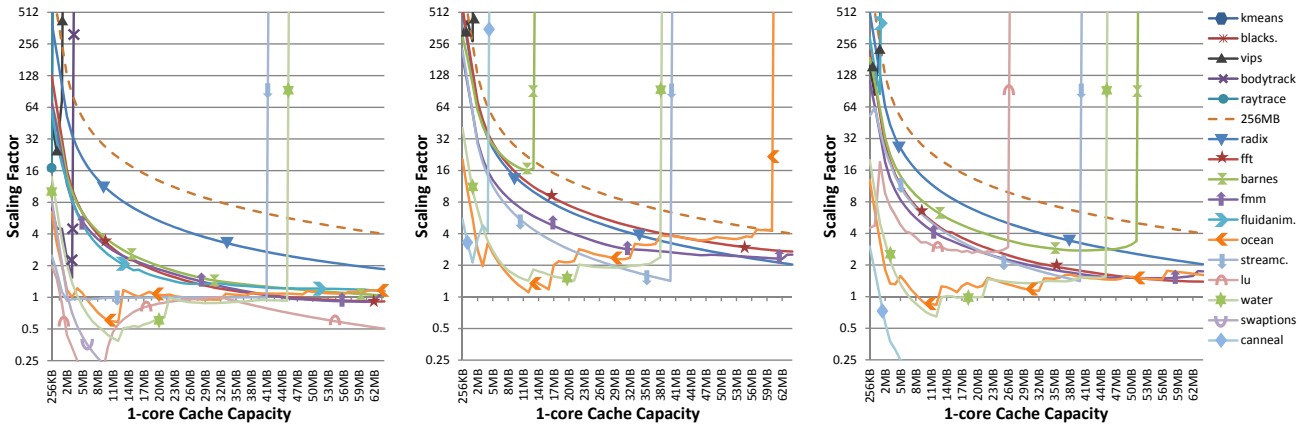
Computer architects can scale the cache hierarchy to address the locality impact from core count scaling, which we measured in Section 3.2. An important question is can cache capacity and/or cluster size scaling compensate for the observed locality degradation? And if so, how much scaling is needed, and why? As discussed in Section 2.2, our scaling framework can help answer these questions.

3.3.1 Cache Capacity.

We first quantify how much cache size should be scaled. In particular, we measure the increase in capacity needed for both shared and private caches to achieve an MPKI on our baseline 256-core CPU that is equal to a particular MPKI value on a 1-core CPU. This *constant-MPKI scaling factor* can be measured in Figure 5 by drawing a horizontal line at a particular MPKI, and computing the ratio of the line's X-intercept at the CRD or sPRD profile to the X-intercept at the RD profile. Since this scaling factor varies across different 1-core MPKI values, we measure it for all 1-core MPKI–or equivalently, across all 1-core cache sizes.

Ideally, under constant-MPKI scaling, per-thread performance is constant with respect to core count, yielding linear speedup. In actuality, constant-MPKI scaling will achieve sub-linear speedup since on-chip communication latency increases with core count. In addition, memory bandwidth consumption will also increase with core count, further reducing speedup if contention occurs. While our framework can analyze larger scaling factors, for simplicity, we focus on constant MPKI.

Figure 6(a) presents our analysis for shared caches. We plot the cache capacity scaling factor needed to maintain



(a) Shared Cache (b) Private Cache (c) 8-Core Cluster Cache

Figure 6: Constant-MPKI cache capacity scaling factor as a function of 1-core cache size at 256 cores and the S2 problem for (a) shared, (b) private, and (c) 8-core cluster caches.

constant MPKI (when scaling from 1–256 cores) as a function of single-core cache size for all 16 benchmarks running the S2 problem. Figure 6(a) shows a large scaling factor is needed at small cache sizes: between 32× and 128× for most benchmarks. For small caches, dilation-based inter-thread interference is significant, causing CRD profiles to shift by large amounts relative to RD profiles (see Figure 5). So, a large capacity increase is needed to compensate for this large locality degradation.

However, Figure 6(a) also shows the capacity scaling factors decrease rapidly for most benchmarks, eventually reaching 1.0. This is related to C_{core}. At a program's C_{core}, the amount of interference-based locality degradation reduces–*i.e.* the CRD-RD gap narrows–which can reduce the shared cache size needed to maintain constant MPKI. In programs where the CRD and RD profiles merge completely beyond C_{core}, multicore and single-core CPUs exhibit the same locality past the merge point, so no cache capacity scaling is needed (scaling factor = 1.0). From Table 1, we see C_{core} is usually small at S2, resulting in a small 1.0-intercept–by 40MB for most benchmarks in Figure 6(a).

Recall from Section 2.2.1 that the CRD-RD gap may never close completely. In these benchmarks, the CRD profile "flattens" beyond C_{core} (*e.g.*, vips in Figure 5(p)). As the 1-core MPKI continues to drop, shared cache capacity scaling ceases to achieve constant MPKI. This is the reason why some scaling factors in Figure 6(a) shoot up to ∞.

In Figure 6(a), we also plot constant ((scaling factor) × (1-core cache size))–*i.e.* constant scaled CPU cache size–as a dotted line. In particular, this dotted line shows cache scaling to 256 MB which is a reasonable on-chip cache capacity for 256-core chips. Except for the ∞ cases, every scaling factor at every 1-core cache size falls below this 256 MB line, showing that sufficient on-chip cache will be available for constant-MPKI scaling at S2.

Figure 6(b) presents our analysis for private caches. Similar to Figure 6(a), a large scaling factor is needed to maintain constant MPKI for small private caches. Because this small-cache region is below or near most benchmarks' C_{share}, sPRD profiles are coincident with or very similar to CRD profiles. Hence, shared and private caches exhibit similar scaling requirements.

In Figure 6(b), we also see the constant-MPKI scaling factor for private caches decreases with cache size. However, unlike Figure 6(a), the rate of decrease for many benchmarks is not as steep. This is due to C_{share}. At cache sizes beyond a program's C_{share}, sPRD and CRD profiles diverge due to sharing-induced replication and invalidation effects, requiring greater capacity scaling for private caches compared to shared caches to match the 1-core MPKI.

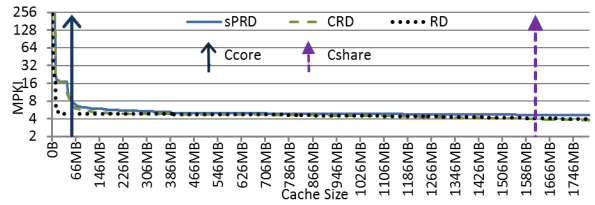
In addition, the private-cache scaling factor never reaches 1.0. Whereas CRD profiles eventually (or nearly) merge with RD profiles, sPRD profiles never merge, again due to sharing effects. Hence, some capacity scaling is always needed–between 2×–4× for most benchmarks in Figure 6(b). Also, sPRD profiles always "flatten" at some capacity beyond which further cache size scaling provides no additional MPKI reduction. Similar to Figure 6(a), this causes some scaling factors in Figure 6(b) to shoot up to ∞.

Figure 6(b) plots the same constant cache scaling dotted line from Figure 6(a). Across every benchmark's range of constant-MPKI scaling, every scaling factor falls below this 256 MB line. So, there is sufficient on-chip cache to facilitate the private-cache constant-MPKI scaling. Nevertheless, these results demonstrate that private caches require significantly larger scaling factors compared to shared caches.

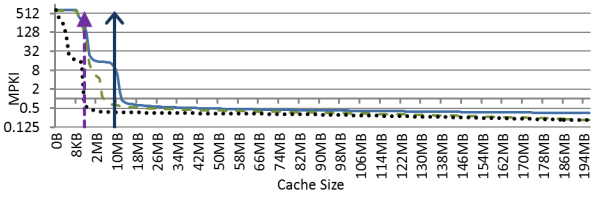
3.3.2 Cluster Size.

We now perform our constant-MPKI scaling factor analysis for cluster caches. Figure 6(c) shows our analysis assuming 8 cores per cluster. (This is the same as Figure 6(b) except the analysis is performed on sPRD[8] profiles instead of sPRD profiles). As in Figure 6(b), constant-MPKI scaling factors are large for small caches, decrease with cache size, and never reach 1.0.

However, compared to Figure 6(b), cluster caches can achieve lower scaling factors, between 1×–2×. As pointed out earlier, at cache sizes beyond a program's C_{share}, sPRD and CRD profiles diverge. But because cluster caches reduce sharing-induced replication and invalidation, the profile for the cluster cache lies in between sPRD and CRD. Therefore, cluster caches require less capacity scaling than private caches to match the 1-core MPKI. Moreover, due to the same C_{share}-related effect, the cache capacity where the scaling factor shoots to ∞ increases in Figure 6(c) because clustering reduces the MPKI at which sPRD^{C_l} profiles "flatten." Overall, Figure 6(c) shows that by reducing sharing effects, cluster caches require less capacity than private



(a) Both C_{core} and C_{share} shift significantly: fft at S3 (radix, ocean, and water are similar)



(b) C_{core} and/or C_{share} shift moderately: barnes at S3 (fmm, canneal, raytrace, and vips are similar)

Figure 7: 256-core CRD, sPRD and RD profiles showing problem scaling effects.

caches to maintain constant MPKI, and are scalable across a larger range of cache sizes.

4. EXTENDED SCALING RESULTS

Section 3 demonstrates our scaling framework for S2 and 256 cores. But on a 256-core CPU, the S2 problem for all our benchmarks would each run in under 1 second. While speeding up small problems is important, S2 does not represent the problem sizes that LCMPs are likely to run. Instead, studying larger problems–as well as larger core counts–is crucial for understanding *actual locality behavior on future LCMPs*. Unfortunately, our PIN tool (let alone a detailed architectural simulator) cannot run such extended configurations. But our scaling framework can study them via profile prediction.

Using the reference groups technique described in Section 2.3, we acquire the same profiles in Figure 5 for 3 larger problems, S3–S5. The last row in Table 1 reports the number of instructions for each predicted problem (assuming sequential runs). On a single 1 GHz core achieving an ideal CPI = 1, the S3–S5 problems represent wall-clock times of 1 minute, 1 hour, and 1 week, respectively.[1] In addition, we predicted locality profiles for 1024 cores assuming the S2 problem. (We also predicted the 512-core case, but omit detailed results because they are similar to the 1024-core case). When predicting, we use two profiled configurations to drive profile diffing: S1/S2 for problem size prediction, and 128/256 cores for core count prediction.

Similar to Section 3, we first quantify locality impact, and then study cache hierarchy scaling.

4.1 Locality Impact

As discussed in Section 2.3, problem scaling shifts locality profiles to larger capacities just like core count scaling, which can degrade locality. For example, Figure 7(a) shows fft's predicted locality profile for S3. By comparing against the corresponding profile for S2 in Figure 5(a), we can see the impact of increased problem size. For both S2 and S3, the first "hump" in the locality profiles ends at about the same capacity, 64KB. Memory references in this initial region access data unrelated to input size (*e.g.*, stack variables), so they do not shift with problem scaling. However, the portions of the profiles beyond the first hump do shift from S2 to S3, stretching regions of higher MPKI across larger cache capacities. This degrades cache performance across the affected capacities.

To quantify, Figure 8 reports the single-core MPKI at cache sizes of 16, 64, and 256 MB across our benchmarks. At each cache size, we show the MPKI for S2–S5 as a stacked bar, illustrating the cache performance degradation incurred as problem size is scaled. (Notice, MPKI is plotted on a log scale). As Figure 8 shows, 12 benchmarks–fft, lu, radix, barnes, fmm, ocean, water, bodytrack, canneal, fluidanimate, streamcluster, and vips–exhibit performance degradation as problem size increases. In most cases, the degradation is significant. For 4 benchmarks–kmeans, blackscholes, raytrace, and swaptions–there is no locality degradation. But overall, locality degradation is the dominant behavior.

Not only can problem scaling cause cache performance degradation, it can also shift C_{core} and/or C_{share}, affecting the extent of locality degradation regions and how much cache scaling is needed to address performance losses. To illustrate, Figure 7 shows different behaviors across our benchmarks, and Table 3 reports C_{core} and C_{share} for S4 and S5.

In some benchmarks, both C_{core} and C_{share} increase significantly with problem size. We find this behavior in fft, radix, ocean, and water. For example, comparing Figures 7(a) and 5(a) shows both C_{core} and C_{share} shift in fft by large amounts when scaling from S2 to S3. For the 4 benchmarks with this behavior, C_{core} and C_{share} become very large. Table 3 shows that by S5, all C_{core} and C_{share} values for these benchmarks are $>$ 600MB, and most are $>$ 2GB. Notice also, these benchmarks are all from the SPLASH2 suite. Under problem scaling, SPLASH2 benchmarks become more coarse-grained, supporting the observation from Section 3.2 that sharing granularity tends to be coarser in SPLASH2 and finer in PARSEC.

For other benchmarks, C_{core} and C_{share} increase, but one or both of them increases only moderately. We find this behavior in barnes, fmm, canneal, raytrace, and vips. For example, comparing Figures 7(b) and 5(d) shows C_{core} shifts moderately in barnes when scaling from S2 to S3 (in this example, C_{share} does not shift at all). For the 5 benchmarks with this behavior, C_{core} usually shifts more than C_{share}. Table 3 shows that except for raytrace, the C_{share} for these benchmarks never exceeds 64KB. C_{core} can become more significant, but is usually $<$ 350MB.

Finally, C_{core} and C_{share} may not increase at all, or the increase is small. This occurs in the remaining 7 benchmarks where there is a much more moderate change in C_{core} and C_{share}. For these benchmarks, Table 3 shows that for both S4 and S5, C_{core} is always $<$ 40MB. C_{share} reaches 10MB in one case, but is $\leq$ 512KB in the other 6 cases.

Compared to problem scaling, continued core count scaling has much less impact on locality. In particular, the predicted profiles for 512 and 1024 cores are similar to those in Figure 5. As Table 3 shows, C_{core} at 1024 cores increases compared to 256 cores. This is because increased interleaving causes more inter-thread interference. In contrast, C_{share} may increase or decrease (Table 3 compared to

[1]Notice, a 1-week sequential run would take under an hour on 256 cores with linear speedup. So, even S5 is not that large.

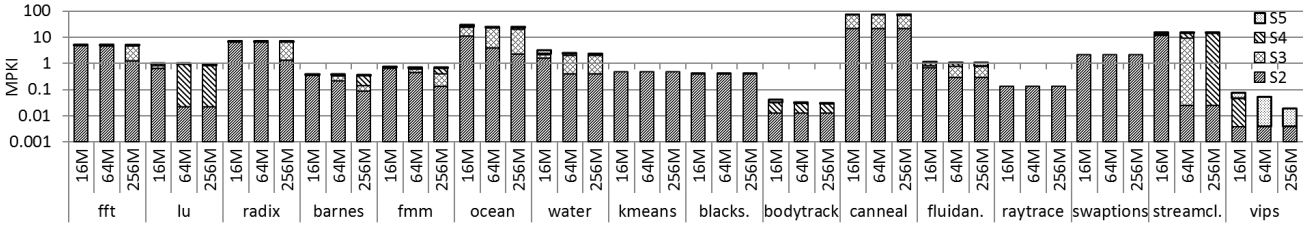


Figure 8: Single-core MPKI break down for S2–S5 at different cache capacities.

Benchmarks	C_{core}			C_{share}		
	S4	S5	1024	S4	S5	1024
fft	970M	>2G	37M	>2G	>2G	43M
radix	28M	>2G	49M	>2G	>2G	142M
ocean	18M	1.5G	1M	>2G	>2G	64K
water	53M	601M	6M	460M	695M	64K
barnes	35M	143M	18M	64K	64K	128K
fmm	65M	739M	33M	32K	32K	64K
canneal	1.1G	>2G	16K	1K	1K	1K
raytrace	2M	173M	15M	6M	661M	128K
vips	334M	1.6G	87M	4K	4K	2K
lu	3M	9M	1M	3M	10M	32K
kmeans	1M	12M	2M	128K	256K	256K
blackscholes	512K	12M	1M	8K	8K	8K
bodytrack	3M	38M	7M	32K	512K	256
fluidanimate	4M	29M	4M	16K	32K	8K
swaptions	1M	1M	6M	64K	64K	128K
streamcluster	128K	256K	1M	256	256	256

Table 3: C_{core} and C_{share} for the S4 and S5 problems on 256 cores, and the S2 problem on 1024 cores.

Table 2), but the change is relatively small. Overall, there is little impact on locality. For this reason, our main focus in the rest of the paper is to analyze problem scaling.

4.2 Cache Hierarchy Scaling

In this section, we re-perform the cache scaling analyses from Section 3.3 to study the impact of problem scaling and continued core count scaling on cache hierarchy design.

4.2.1 Cache Capacity.

Figure 9(a) reports the shared cache capacity scaling factor results for an extended problem size, S5. (In other words, this graph provides the same analysis for S5 that Figure 6(a) provides for S2). Qualitatively, Figure 9(a) resembles Figure 6(a). At small cache capacities, large scaling factors are needed to maintain constant MPKI. But as cache size increases, the constant-MPKI scaling factor drops and eventually reaches 1.0.

However, Figure 9(a) also shows problem scaling dramatically changes shared cache scaling. In particular, benchmarks exhibit much larger scaling factors at S5 compared to S2. Because problem scaling shifts locality profiles, the large scaling factors visible in Figure 6(a) are "stretched" to the right, resulting in a much wider range of cache sizes across which large scaling factors occur in Figure 9(a). Also, 1.0-intercepts increase significantly in most benchmarks as well. Recall from Section 3.3.1 that the 1.0-intercept is related to C_{core}. Since many benchmarks' C_{core} increase from S2 to S5 (as discussed in Section 4.1), so do their 1.0-intercepts. In fact, most benchmarks' 1.0-intercepts occur well beyond the end of the graph in Figure 9(a).

Both the larger scaling factors and extended 1.0-intercepts under S5 negatively impact shared cache scalability. In Figure 9(a), only 3 benchmarks (lu, bodytrack, and streamcluster) are contained entirely within the 256 MB dotted line. Three other benchmarks (blackscholes, kmeans, and swaptions) also appear below the dotted line, but quickly shoot up to ∞ due to the CRD flattening issue described in Section 3.3.1. The remaining 10 benchmarks–*i.e.* those with large C_{core} values in Table 3–have scaling factors well above the dotted line, and cannot achieve constant-MPKI scaling within a 256 MB on-chip cache capacity.

Not only does problem scaling negatively impact shared cache scalability, it also negatively impacts the scalability of private caches as well. Figure 9(b) reports the capacity scaling factor results for private caches at S5. Once again, Figures 9(b) and 6(b) are qualitatively similar. In particular, Figure 9(b) shows constant-MPKI scaling factors are larger for private caches compared to shared caches, and they never reach 1.0. (They also eventually shoot to ∞, though for most benchmarks this happens well beyond the end of the graph in Figure 9(b)).

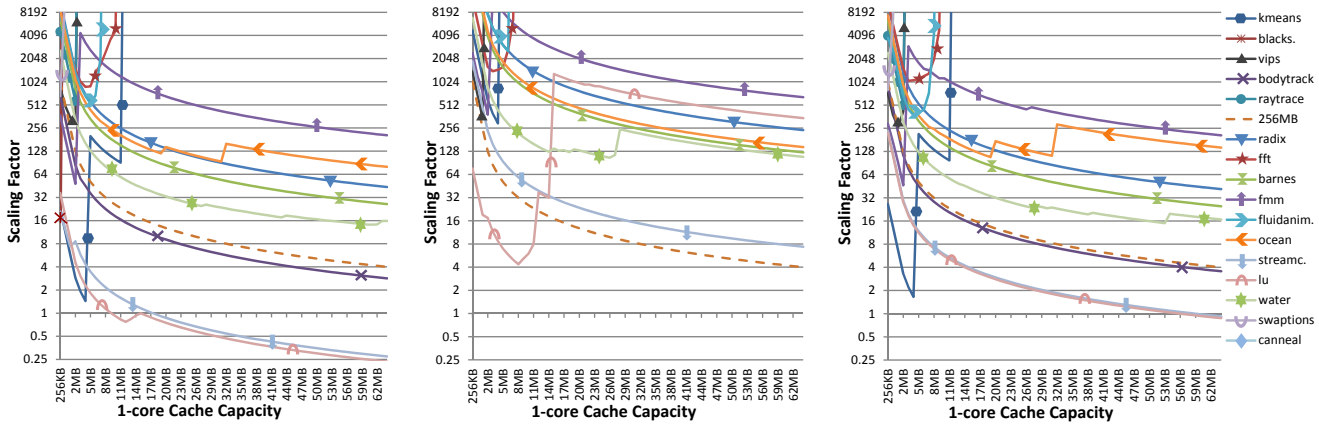
Because private caches exhibit the same or worse scaling factors as shared caches, the benchmarks in Figure 9(a) that are already above the 256 MB dotted line for a shared cache are also above the dotted line in Figure 9(b) for a private cache. Even worse, the benchmarks that are below the 256 MB dotted line in Figure 9(a) move above the dotted line across most cache capacities in Figure 9(b). These benchmarks have small C_{share}, as shown in Table 3. So, their sPRD and CRD profiles diverge at small cache sizes, leading to larger constant-MPKI scaling factors across most/all private cache capacities. Consequently, except for lu at small cache sizes, Figure 9(b) shows none of our benchmarks achieve constant-MPKI scaling for private caches within the 256 MB limit under the S5 problem.

As for continued core count scaling to 1024 cores, due to the small change in C_{core} and C_{share} shown in Table 3, the result for shared caches is qualitatively the same as Figure 6(a). But there are more sharing-induced invalidations at 1024 cores, thus many benchmarks' private-cache scaling factors shoot to ∞ earlier than in Figure 6(b). (Detailed results have been omitted to save space).

4.2.2 Cluster Size.

We now consider the predicted PRD^{C_l} profiles. Figure 9(c) presents our analysis for 8-core cluster caches at S5. As discussed in Section 3.3.2, the effects of clustering–lowering the scaling factor and increasing the scalable range–are related to C_{share}. For most benchmarks in Table 3, C_{share} remains small at S5. Therefore, there is potential for clustering to fill the gap between the sPRD and CRD profiles.

Comparing Figures 9(c) and 9(b), we can see there are scaling factor reductions for many benchmarks. In particular, clustering enables the same 3 benchmarks from Figure 9(a) to achieve constant-MPKI scaling within 256 MB of on-chip cache (lu, bodytrack, and streamcluster). Also,



(a) Shared Cache (b) Private Cache (c) 8-Core Cluster Cache

Figure 9: Constant-MPKI cache capacity scaling factor as a function of 1-core cache size at 256 cores and the S5 problem for (a) shared, (b) private, and (c) 8-core cluster caches.

the benchmarks that shoot up to ∞ generally do so at lager capacities in Figure 9(c) compared to Figure 9(b). However, the fact remains that scaling problem size from S2 to S5 makes many more benchmarks unscalable within 256 MB of on-chip cache, and clustering does not change this fundamental conclusion.

5. RELATED WORK

Our work is related to all of the recent research on RD analysis for multicore processors. In particular, we employ CRD profiles explored by Ding [4] as well as PRD profiles explored by Schuff [16, 15]. We also exploit the properties of symmetric threads first observed by Jiang [9]. However, while these prior research efforts focus on developing profiling techniques, we apply them to study scaling. Our work considers 256-1024 core CPUs and LCMP-sized problems, whereas prior papers investigate 32 cores at most (usually 4 cores or less), and small problems. We also extend prior analysis with PRD^{C_l} profiles for cluster caches.

This paper is closely related to our own previous work [19, 20], which defined many components of the scaling framework in Section 2. In particular, our previous work observed that core count scaling for symmetric threads causes systematic shift of CRD and PRD profiles, and proposed applying Zhong's reference groups [22] to predict the shift. To explain why profiles shift, we identified interference- and sharing-based locality degradation, and defined C_{core} and C_{share} to mark the capacities across which these regions occur.

Compared to our prior work, this paper studies how multicore cache hierarchies should scale in order to compensate for the observed locality degradation. This paper studies both cache capacity and cluster size scaling to address locality degradation; the prior work did not apply the techniques for studying cache hierarchies. In addition, this paper uses profile prediction to study very large problems and 1024-core CPUs; the prior work focused only on assessing the accuracy of the profile prediction techniques. This paper also performs a more detailed characterization of the interference- and sharing-based locality degradation regions, and considers many more benchmarks.

Chandra [3] and Suh [17] also developed locality models for multicore processors. However, these papers focus on multiprogrammed workloads consisting of sequential programs whereas we focus on multithreaded parallel programs. RD analysis has also been used to analyze uniprocessor cache hierarchies [22, 5, 23].

Finally, many studies have tried to characterize multicore memory behavior using simulation [7, 8, 10, 14]. Although our study is not as accurate, it explores a much larger portion of the design space, providing more insight into scaling trends. Moreover, because we can predict scaled configurations, our study can also consider much larger core counts and problem sizes.

6. CONCLUSION

This paper conducts the first-ever study of multicore cache hierarchy scalability using RD analysis. We present a powerful framework for analyzing parallel program locality based on existing RD analysis techniques, and apply it to study multicore CPU scaling. Across 16 SPLASH2 and PARSEC benchmarks, we find interference-based locality degradation is more significant than sharing-based locality degradation. For small input problems typically used to drive architectural simulation, interference-based locality degradation can be fully contained within capacities feasible for on-chip caches. Sharing-based locality degradation is suite dependent, with some SPLASH2 programs exhibiting coarse-grain sharing and the remaining benchmarks exhibiting fine-grain sharing. Given a 256 MB on-chip cache budget, our results show all of our benchmarks can achieve constant-MPKI scaling for both shared and private caches when going from 1 to 256 cores. This is due to the small input problem sizes and their relatively small C_{core} values.

For large input problems that are more appropriate for LCMPs but are impractical to simulate, we find interference-based locality degradation becomes much more significant, occurring across a wider range of cache sizes for many benchmarks. As a result, only 3 benchmarks achieve constant-MPKI scaling within 256 MB of shared cache, and no benchmarks can achieve constant-MPKI scaling within 256 MB for private caches. Cluster caches can help bridge the gap between private and shared cache scalability, but do not change the poor scaling behavior that plagues the larger problems. These results demonstrate the importance of RD analysis techniques: they can reveal qualitatively different behavior compared to what traditional simulation of smaller problem sizes would show.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments, and Abdel-Hameed Badawy for insightful discussions.

7. REFERENCES

[1] 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 *Hot Chips*, 2007.

[2] 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.

[3] 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.

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

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

[6] Y. Hoskote, S. Vangal, S. Dighe, N. Borkar, and S. Borkar. Teraflop Prototype Processor with 80 Cores. In *Hot Chips*, 2007.

[7] 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*, 33, 2005.

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

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

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

[11] 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 ACM Conference on Programming Language Design and Implementation*, 2005.

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

[13] 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.

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

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

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

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

[18] 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 International Symposium on Computer Architecture*, 1995.

[19] M.-J. Wu and D. Yeung. Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[20] M.-J. Wu and D. Yeung. Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. *ACM Transactions on Computer Systems*, 31(1), 2013.

[21] 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.

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

[23] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.