Multicore-Aware Reuse Distance Analysis

Derek L. Schuff, Benjamin S. Parsons, and Vijay S. Pai
Purdue University
West Lafayette, IN 47907
dschuff@purdue.edu, bsparson@purdue.edu, vpai@purdue.edu

***Abstract*—This paper presents and validates methods to extend reuse distance analysis of application locality characteristics to shared-memory multicore platforms by accounting for invalidation-based cache-coherence and inter-core cache sharing. Existing reuse distance analysis methods track the number of distinct addresses referenced between reuses of the same address by a given thread, but do not model the effects of data references by other threads. This paper shows several methods to keep reuse stacks consistent so that they account for invalidations and cache sharing, either as references arise in a simulated execution or at synchronization points. These methods are evaluated against a Simics-based coherent cache simulator running several OpenMP and transaction-based benchmarks. The results show that adding multicore-awareness substantially improves the ability of reuse distance analysis to model cache behavior, reducing the error in miss ratio prediction (relative to cache simulation for a specific cache size) by an average of 70% for per-core caches and an average of 90% for shared caches.**

I. Introduction

Reuse distance analysis (also called stack distance analysis) is a method for characterizing application memory access behavior without being tied to a specific memory hierarchy [1]. The reuse distance of a reference to element x is the number of distinct data elements that have been referenced since the last access of x (or ∞ if the element has not been referenced before). The data elements can be pages, cache lines, individual words, or even instructions, yielding a generalized machine-independent measure of program locality. Conceptually, this is implemented by tracking the depth of the access in a stack, though more efficient implementations also exist [2], [3], [4], [5], [6]. Forming a cumulative histogram of reuse distances gives a prediction of cache hit ratio; in particular, the hit ratio of a cache with N one-word blocks organized in a single LRU set would correspond to the fraction of references with reuse distance less than N. A single run of reuse distance analysis can thus predict the behavior of an application over a variety of cache sizes, but does not account for real cache constraints like associativity, block size, or replacement policy details. Despite those limitations, reuse distance analysis has been used and validated for predicting cache performance (even for low-associativity caches), modeling program locality, providing cache hints during code generation, and for feeding models that restructure code and data to improve locality [7], [8], [4], [9], [10], [11], [12], [13].

When considering multicore systems, however, existing reuse distance analysis methods are insufficient, as their abstract notion of access ordering by counting only makes sense within a single reference stream. Considering events within only a single reference stream ignores inter-core data communication that may impact or even dominate parallel program performance. These inter-core interactions depend largely on the cache configuration, which may consist of private caches or shared caches. Private caches are typically kept coherent using invalidations; shared caches can allow fast inter-thread communication when more than one core sharing the cache accesses the same block (inter-core prefetching), and can also use cache capacity more efficiently by reducing the number of replicated copies of widely-read data. Reuse distances are supposed to represent a measure of locality, with shorter distances being more likely to hit and longer ones less likely. However, if one thread has written to the same address between two reuses of that address by another thread with a different cache, an invalidation may take place and the access will miss regardless of how short the reuse distance is. Similarly, if a thread fetches an address into a shared cache, another thread will see that access as a hit even if it has never referenced that data before. Thus, reuse distance alone does not consistently correspond to locality in multicore systems.

There is thus a fundamental tradeoff between completely machine-independent metrics and those that include details of the machine such as the organization of the memory hierarchy. Machine independent metrics are advantageous in that they require less information about machine parameters, and data from a single run may be more broadly applicable, but at the cost of some accuracy. Although reuse distance is intended to be largely machine-independent, applying it to multithreaded programs requires striking a balance: adding enough machine awareness to accurately capture locality characteristics but not so much as to diminish its simplicity and broad applicability.

This paper enables reuse distance analysis for parallel systems by modeling the cache sharing configuration of the multicore processors used. Although the analysis methods are independent of cache sizes, they do require an understanding of which (if any) cores share a cache. In practice, real systems include private L1 caches even if they have a shared L2 or L3; this paper only considers the configuration

This work is supported in part by the National Science Foundation under Grant Nos. CCF-0532448, CNS-0532452, and CCF-0621457.

of the outermost level of on-chip cache since that determines off-chip bandwidth consumption and typically has the most pronounced impact on performance [14]. Systems with per-core caches use private reuse distance stacks, but must guarantee that addresses are removed from the reuse distance stack when they would be invalidated in a real system; a reuse after an invalidation is just as "non-local" as a cold miss, and is thus treated as such.

For systems with shared caches, the most important issue is to make sure that accesses by one thread become promptly available in cache for other threads to access. This is done by using a shared reuse distance stack that sees all references from all threads and measures reuse distances in terms of total number of distinct addresses referenced by all threads since the last reuse by any thread. This paper also considers caches shared by a subset of the cores, such as pairwise-shared caches in a quad-core processor (e.g., the Intel Core 2 Quad Processor [15]).

The experimental results show that adding multicore-awareness to reuse distance analysis substantially improves locality modeling when compared to various cache configurations for a simulated quad-core system. When comparing the predictions from reuse distance analysis against actual cache simulation results for 1 MB, 2 MB, and 4 MB total cache sizes split across 4 per-core caches, adding multicore-awareness reduces the error in miss ratio prediction by an average of 70% across 13 applications. This result arises because the unaware method does not account for the fact that a reuse will not take place if another thread has issued an intervening write. Comparing against simulation results for shared caches of the same total cache size, multicore-awareness reduces the errors in predicted miss ratios by an average of 83%. This result arises because the unaware method does not account for the positive effects of inter-thread reuse when considering limited cache sizes.

II. Multicore-Aware Reuse Distance Models

Reuse distance is a model of program locality, which determines cache performance. Its fundamental limitation with respect to multicore systems is that cache performance for a given thread is affected by the actions of other threads. Because these interactions can have a significant impact on performance, they should not be ignored by models that predict performance. Since intra-thread locality is still the primary factor affecting cache performance, the basic model of reuse distance is the same; however, it must be extended to account for inter-thread interactions because behaviors such as coherence and sharing are fundamental elements in multicore systems. The modifications to the reuse distance model depend on the cache configuration used in a multicore system. Caches may be private to each processor core or shared among two or more cores; each policy has a different effect on reuse distance since it has a different impact on coherence and sharing. On the other hand, issues such as cache size or associativity do not have any additional impact on multicore systems, so the existing reuse distance models may remain independent of these constraints.

Modeling private caches. When caches are private to each processor core in a shared-memory system, a coherence protocol is employed to prevent caches from containing stale data. Most coherence protocols are write-invalidate, in which a write to a cache block in one cache invalidates copies of that block in other caches. If a subsequent reference is made to the overwritten block, it will miss in the cache no matter how recently the core accessed it. (This is called a coherence miss.)

Per-core caches with invalidation-based coherence can be modeled by using per-thread reuse distance stacks for which a write to any address removes that address from all other stacks which contain it, leaving it only in the stack belonging to the thread which wrote it. Then a subsequent reference to that address by some other thread will have infinite reuse distance, as though it had never been seen before — just like a cold miss. When a block is invalidated from a cache, it leaves behind a free slot that can be re-used for the next block of data that maps to that set. Blocks that have been evicted will not go back into the cache, but the slot can be filled without evicting any other blocks. In a fully-associative cache, any block can be mapped to this slot. To model this behavior, blocks that are overwritten are left in the stack but marked as invalid, so an access deeper in the stack has the same reuse distance as if the overwritten block were still in the stack. When the deeper block is brought to the top, the empty slot moves to the former depth of the deeper block; this ensures that still deeper blocks do not reduce their depth. If a new block is accessed, the topmost empty slot is filled in. This overwriting process is referred to in this paper as "invalidation". It is not precisely equivalent to invalidation in a real cache, because in a real cache, only blocks still in the cache get invalidated. In the reuse stack, invalidation affects all blocks which the thread has ever referenced, resulting in more interaction between the stacks than real caches would have. Invalidation is performed on every write, as with real caches. However, Section IV-C discusses the impact of variations in this timing.

False sharing is an additional concern in write-invalidate protocols. It can be detected by extending the analysis such that each stack entry includes information on which words in the block have been accessed by which thread, allowing differentiation between true and false sharing.

Modeling shared caches. When threads run on cores that share a cache, the locality of each thread is affected by the behavior of the others in several ways. First, one thread accessing data will effectively prefetch that data for all others with the same shared cache. Second, all threads sharing a cache can use just one copy of a given widely-read data element, thus reducing unnecessary replication and using the cache capacity more efficiently. Third, different threads may have different working set sizes and thus require different partitions of total cache capacity, possibly freeing up space for others, or taking space away from them.

Reuse distance can be measured across all references from all threads; in this case the distance seen by one thread's

references are affected by the data use patterns of the other threads. This type of analysis can be enabled by directly using a shared reuse stack.

Additionally, caches may have hierarchical sharing. For example, a 4-core system could have 2 caches, each shared by 2 cores, which use a coherence protocol between them (e.g., the Intel Core 2 Quad Processor [15]). In this case, any of the private techniques could be used across caches.

Definition of reuse distance. These modifications can be summarized in a new definition of reuse distance. For a multithreaded program, the reuse distance of a reference by thread T to address x is

(a) ∞ if x has never been referenced before;
(b) ∞ if x has been overwritten by a thread which does not share a cache with T since the last reference by a thread which shares a cache with T; or
(c) the number of distinct data accessed by threads which share a cache with T since the last reference to x by a thread which shares a cache with T.

III. Experimental Methodology

Simulation Platform. The reuse distance stack implementation is based on Sugumar and Abraham's splay tree code as distributed with SimpleScalar 4.0 [16], [6]. The reuse distance stacks were implemented as a memory hierarchy module in the Virtutech Simics full-system simulator, which executes all code from the Linux kernel and system daemons [17]. However, only userspace references from the benchmark were fed to the stacks. In addition the stacks were tied to the thread rather than the processor to eliminate any effects of thread migration. Timing of the instructions and memory references was not modeled; Simics' default timing model of one instruction per cycle was used for most tests, with round-robin switching among processors on each cycle. Some additional tests were also performed with different switching frequencies to determine if the results were sensitive to that particular assumption.

Workloads. This system is evaluated with thirteen benchmarks: six from SPEC OMP2001, three from the OpenMP implementation of the NAS Parallel Benchmarks version 3.3, and four from the STAMP transactional memory benchmark suite [18], [19], [20]. They include `312.swim`, `316.applu`, `318.galgel`, `320.equake`, `324.apsi`, and `326.gafort` from SPEC OMP; `CG.W`, `FT.W`, and `MG.W` from NAS; and `genome`, `intruder`, `kmeans`, and `vacation` from STAMP. All benchmarks were run with 4 threads. The "test" data input was used for SPEC benchmarks and the "W" input size for NAS benchmarks. High-contention input parameters were used for `kmeans` and `vacation`. For the OpenMP benchmarks under the lazy and oracular invalidation models, the library itself was modified to notify the simulator at the end of each parallel region and thus allow the simulator to ignore the implementation of barriers between parallel regions. For the STAMP benchmarks, all transactions are implemented in software using the TL2-x86 STM system distributed with STAMP [21].

Validation against simulated caches. This study also validates the accuracy of reuse distance as a predictor of cache performance by comparing against Simics' coherent cache simulation module called *g-cache*. The caches operated on the same reference stream as the reuse distance stacks. The stacks were then used to predict hits and misses at each reference for 3 specific cache sizes by comparing the reuse distance for the reference with the size of the simulated cache. The base tests use caches with single-word (8 byte) lines and 8-way set associativity; as discussed in Section II, some sensitivity tests also consider 64 byte cache block sizes.

The cache simulations model configurations with private per-thread caches with MSI coherence, with a single cache shared among all 4 threads, and with a pair of caches, each of which is shared by 2 threads, and which use MSI coherence between them. The private cache experiments use 256KB, 512KB and 1MB sizes. The shared cache experiments use 1MB, 2MB, and 4MB sizes, so as to correspond to the same total cache size used in the private cache tests. Similarly, the pairwise shared cache tests used 512KB, 1MB, and 2MB sizes.

IV. Experimental Results

A. Private Cache Configurations

Reuse distance results. Figure 1 shows the cumulative reuse distance function for each application using unaware per-thread reuse distance analysis without considering multicore effects and using the multicore-aware reuse distance analysis method discussed in Section II. The X axis represents a given reuse distance value x, while the Y axis represents the percentage of application data accesses that exhibit a reuse distance of x or less under each model. Because these applications are dominated by double-precision floating-point accesses, addresses are considered at 64-bit granularity. A particular (x, y) point on these curves thus predicts a cache hit ratio y for a fully-associative LRU cache of size x 64-bit words. The graphs in this section do not include the `312.swim` benchmark because both multicore-unaware and multicore-aware analyses give nearly the same results and see practically no error compared to simulated caches. However, the benchmark is included in all tables and averages.

In most cases, there is little discernible difference in predicted hit ratio between unaware and multicore-aware reuse distance measurement until the right end where reuse distance is the longest, and the plots are zoomed to show these differences. The fact that the differences primarily appear for large reuse distances stems from traditional working set analysis. Reuse distance plots often have one or more knees and plateaus, which correspond to different working sets. These working sets can be seen clearly, for example, in the `equake` benchmark. In data-parallel applications the largest datasets would be the ones most commonly divided among the cores in the system, so any differences between traditional and multicore-aware reuse distance would be seen

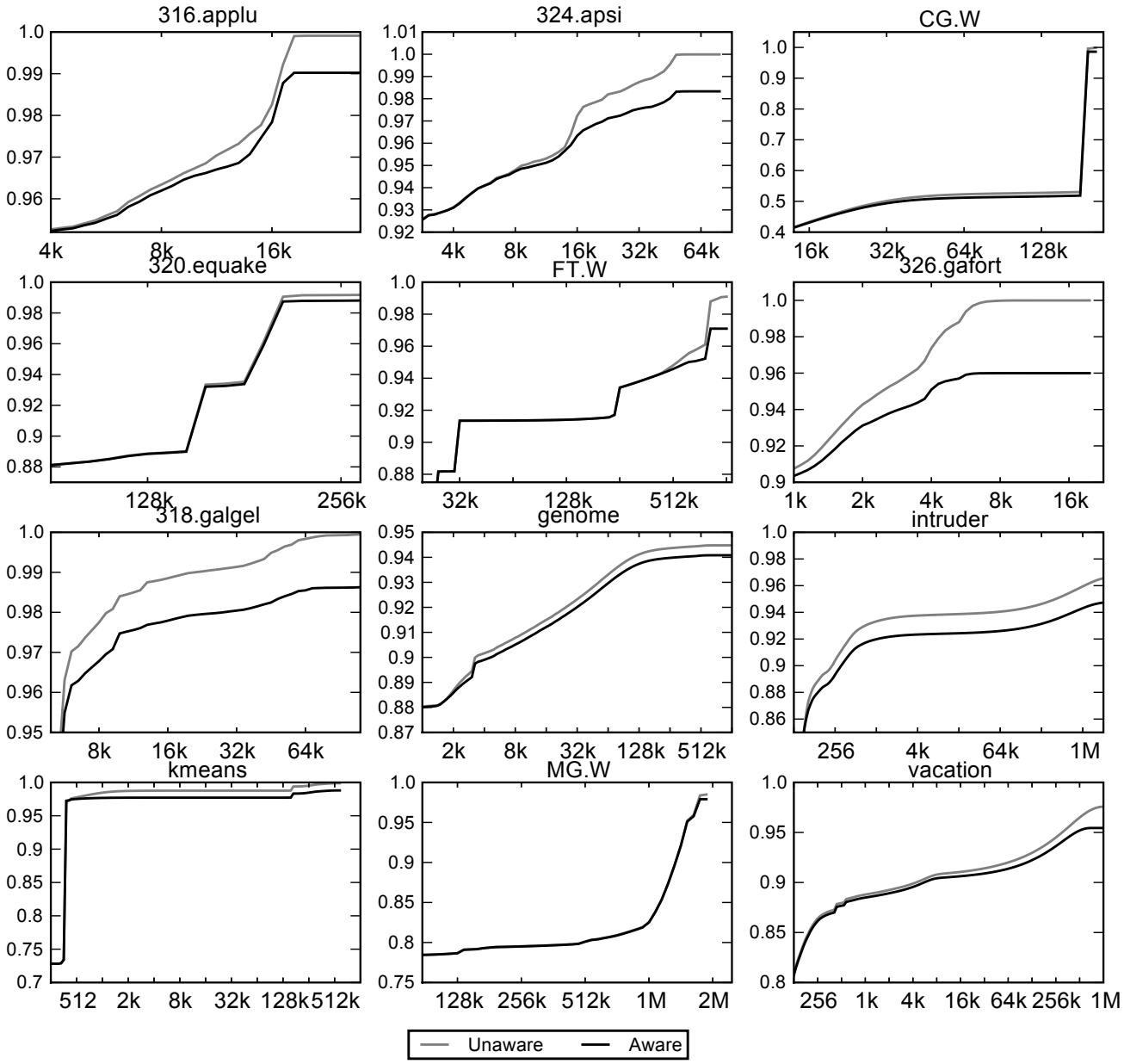

Figure 1. CDFs of reuse distance using private per-thread stacks. X axis is reuse distance in 64-bit words, Y axis is fraction of references with reuse distance less than x.

at this end of the plot. In addition, with smaller caches and lower overall hit ratios, capacity misses increase, causing coherence misses to become a smaller fraction of overall misses, mitigating their effects. The STAMP benchmarks do not have the clean data distribution or the well-defined (and often long) periods between synchronizations that the OpenMP benchmarks do, and write more frequently to actively shared data. Thus, a higher fraction of the misses are coherence misses even at small cache sizes, causing unaware and invalidating schemes to diverge in the plots even at small reuse distances.

Impact on performance prediction. Comparing the multicore-aware model to the unaware model shows that for most applications, the multicore-aware cumulative reuse level reaches a peak at least a few percent below that of the unaware case. The multicore-aware version never reaches 100% reuse because coherence-related misses behave just like cold misses, with an infinite reuse distance. When seen on a plot as hit ratio variations of just a few or even fractions of a percent, these differences may seem insignificant. However the memory hierarchy is such an important factor in system performance because the difference in stall time between a hit and a miss is so great. This is especially significant when overall miss ratios are very low. Consequently, the difference between a 98% hit ratio (2% miss ratio) and a 99% hit ratio (1% miss ratio) leads to a

Benchmark	Percent error	
	Private	
	Unaware	Aware
316.applu	91.46	5.53
324.apsi	82.63	5.84
CG.W	2.6	0.63
320.equake	1.29	1.36
FT.W	5.2	5.2
326.gafort	99.89	0.37
318.galgel	79.68	4.45
genome	9.42	4.79
intruder	19.71	0.11
kmeans	45.58	0.69
312.swim	4.3E-04	1.7E-04
vacation	8.39	1.25
Average	37.15	2.52

Table I
PERCENT ERROR OF EACH REUSE DISTANCE METHOD COMPARED TO SIMULATED CACHES.

Benchmark	Percent error			
	Shared cache		Pairwise shared	
	Unaware	Aware	Unaware	Aware
316.applu	69.84	5.51	84.78	5.5
324.apsi	12477.1	0.22	68.25	6.57
CG.W	2.27	0.02	1.69	0.14
320.equake	4.03	0.25	2.67	1.24
FT.W	4.56	4.02	4.2	1.83
326.gafort	125.68	1.0E-05	99.8	0.51
318.galgel	511.7	41.08	60.55	4.12
genome	23.05	1.03	6.77	2.56
intruder	28.74	0.34	7.71	0.38
kmeans	0.97	1.4E-03	35.51	1.1
312.swim	0.19	1.1E-03	9.7E-02	5.1E-04
vacation	73.16	2.77	8.03	2.47
Average	1110.11	4.6	31.67	2.2

Table II
PERCENT ERROR OF EACH REUSE DISTANCE METHOD COMPARED TO A SIMULATED SHARED OR PAIRWISE SHARED CACHE CONFIGURATION

performance impact far greater than 1%, and indeed often closer to doubling.

Validating against simulated caches. Table I shows a comparison of miss ratios predicted by the reuse distance analysis schemes with miss ratios determined by simulated caches using MSI coherence. Reuse distances for each thread are used to predict the miss ratio for each of the cache sizes on each of the threads, and then averaged together. The predicted and simulated cache miss ratios for cache sizes of 256 KB, 512 KB, and 1 MB are compared, and the reported prediction error for each application represents the average error when modeling these three cache sizes.

The prediction error for the unaware non-invalidating method varies from 1.29% for `equake` to more than 99% for `gafort`, with an average of 37%. The accuracy of Unaware correlates strongly with overall miss ratio, especially for the OpenMP benchmarks. For example, `applu`, `galgel`, `apsi`, and `gafort` have miss ratios ranging from 1.2-4.0% for the smallest cache size, and their error for Unaware is greater than 79%. The remaining OpenMP benchmarks have miss ratios ranging from 10-51%, with error of less than 5.5%. The STAMP benchmarks fall in between, with an average of 6.9% miss ratio and 21% error for Unaware. In these cases where the overall number of misses is small, the proportion of coherence misses, unaccounted for by Unaware, is much higher and its accuracy is reduced. In contrast, the multicore-aware method improves accuracy dramatically, seeing less than 6% inaccuracy for all benchmarks; the average reduction in error is 70%.

B. Shared Cache Configurations

Figure 2 shows the cumulative reuse distance function for each application using a single shared reuse distance stack (solid black line) and pairwise shared stacks that invalidate each other using the eager method (dotted line). The unaware reuse distance with no invalidation or sharing is shown for comparison. The lines are shifted for the unaware and pairwise shared stacks such that the x value represents the total cache size in all cases (4x the size of the unaware caches and 2x the size of the pairwise caches).

When reuse distances are shortened by effects such as inter-core prefetching or reduced replication, the shared plots reflect higher fractions of accesses less than a given distance (and thus a higher predicted hit ratio for a cache of that size). For most benchmarks (especially the OpenMP benchmarks), this corresponding distinction between unaware and multicore-aware models is seen over a wider range of sizes than with the private caches. Though not easily seen on all plots, the line for the pairwise shared stacks dips below those of the shared stacks in the upper right corner and flattens out at a hit ratio less than 100% due to invalidations between the shared caches.

Table II shows a comparison of miss ratios predicted by the reuse distance analysis schemes with miss ratios determined by simulated shared caches and pairwise-shared caches. For the unaware method, the predictions are made for individual caches of size $1/4$ the size of the shared cache or $1/2$ the size of the pairwise-shared caches, so that the total cache size is the same. Reuse distances for each thread are used to predict the miss ratio for each of the cache sizes on each of the threads (for the shared stack there is only one stack for all threads), and then averaged together.

Focusing first on the single shared cache, there are two main sources of error when attempting to model this with independent unaware reuse stacks. The first is that they will indicate capacity misses that will not be seen by a shared stack; this is due to duplication of data reducing the effective aggregate capacity, and to accesses by other cores that keep shared blocks in the cache when they would have been

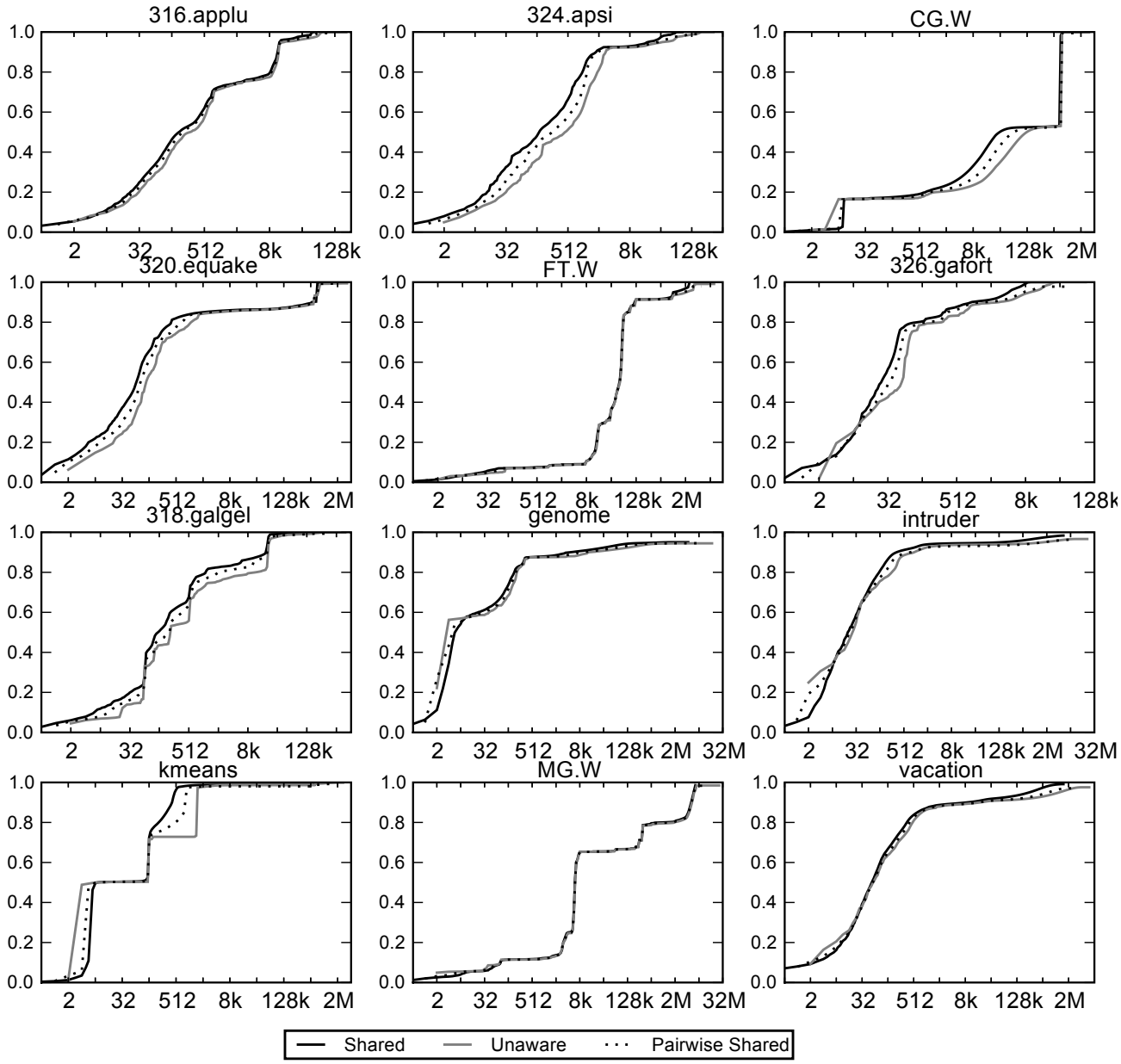

Figure 2. CDFs of reuse distance using a single shared or pairwise shared stacks. X axis is reuse distance in 64-bit words, Y axis is fraction of references with reuse distance less than x.

evicted from a private cache. The second is that accurately modeling inter-core prefetching will also cause the shared stacks to report fewer cold misses. These effects represent benefits to using a shared-cache configuration.

Both of these effects are strongly intensified when the working set is small compared to the total size of the shared cache. This can be seen in the inaccuracy of the unaware method on the `apsi`, `galgel`, `applu`, and `gafort` benchmarks. All 4 have very low simulated miss ratios (less than 0.13% in the smallest simulated cache size). The particularly extreme numbers seen in `apsi` and `galgel` are due to reductions in capacity misses; the working set fits entirely into a 1MB cache, so the shared 1MB cache correctly predicts the extremely low miss ratio, but individual caches of 256KB do not. Conversely, for `applu` and `gafort` the errors are due mostly to a discrepancy in cold misses; they too have very low miss ratios, so the difference in cold misses is significant. For the rest of the benchmarks, which do not have such small working sets, the breakdown is more mixed, but slightly favors the impact of reduced capacity misses.

Overall, the prediction accuracy for shared caches is improved by 90% by using the multicore-aware method. Even if the 4 benchmarks with extremely low miss ratios are ignored, the improvement is substantial, averaging 87%.

The `galgel` benchmark has significant error even with

the shared stack; this benchmark had significant numbers of conflict misses in the shared configuration that did not appear in the private configuration.

Looking next at the pairwise-shared cache results, the accuracy on most of the benchmarks is quite similar to that of the private invalidating stacks. Those where errors are higher tend to be those that also had higher error in the shared-cache configuration. The overall effect of adding multicore-awareness is a 83% improvement in modeling accuracy (78% if not counting the benchmarks with low miss ratios).

C. Sensitivity Analysis

Besides the basic tests described above, additional experiments considered the sensitivity of our methods to number of cycles between simulator context switches, cache block size, and timing of invalidations.

The base tests switch between simulating all of the processors in round-robin order on each instruction. Since this is unlikely to accurately represent a real execution, additional experiments were run with context switching every 100, 1000, 5000, and 10000 instructions. In all cases, all of these switching times caused less than 3% difference in accuracy for any of the reuse distance models tested: namely, the multicore-unaware models continued to show large and highly variable errors while the aware models had much smaller and more bounded errors. Most tests saw less than 1% difference betweenthe various switching times.

In addition to interleaving of references, the timing of the invalidations is also investigated. The multicore-aware scheme presented so far mimics the behavior of a cache, performing invalidations for each write reference as soon as it occurs during simulation. Two alternative methods were used: In *lazy* invalidation, each thread accesses its own reuse stack as normal, but write references are also buffered. All invalidations caused by writes in the buffer take place at the next synchronization point. *Oracular* invalidation uses future knowledge of the write references for the upcoming parallel region and invalidates the addresses that will be written before the region begins. These variants are valid for the OpenMP benchmarks because OpenMP requires the programs to be data-race-free (DRF); in particular this means that any data invalidated from a processor's will must not be accessed again by that processor until the next synchronization. For OpenMP this usually means the barrier at the end of parallel regions. The lazy and oracular schemes test the effect of invalidations freeing space in the cache to be used by other data. Since the DRF property guarantees that a thread will not reference an address in the same parallel region in which it is invalidated, these addresses just take up space until they are invalidated and removed. The lazy invalidation scheme leaves them in the stack as long as possible, which should increase the overall reuse distance and thus represent a worst case. The oracular invalidation scheme removes these addresses as soon as possible, thus freeing up their space early and reducing overall reuse distance; it represents a best case. Even with these relatively extreme timing differences, the lazy and oracular cache predictions fell within 4.5% of the simulated timing scheme. These results, along with the simulator context switching results, indicate that while it is important to add multicore-awareness to reuse distance analysis, the precise timing of references and invalidations is not as important for these applications.

Although the base tests use single-word cache lines, simulations were also run for all benchmarks under all sharing configurations with 64-byte line sizes (to more accurately reflect the designs of modern caches and to detect false sharing), and with 4-byte line sizes for `genome` and `kmeans` (the only benchmarks dominated by 4-byte instead of 8-byte accesses). The results were very similar, with average accuracy and improvement within 5% of the numbers reported above.

V. Related Work

Rather than extend reuse distance analysis directly, Ding and Chilimbi present a model which computes a set of statistics on data sharing and thread interleaving. Combined with per-thread reuse distance analysis, it can predict multithreaded reuse signatures and cache performance [22].

Several works have used reuse distance analysis to model locality in broader ways. For example, Marin and Mellor-Crummey extend reuse distance analysis to identify the source, destination, and scope of a data reuse pattern, providing a list of suggested transformations such as data-splitting and loop fusion that can address misses generated in various ways [11]. Ding and Zhong present a scheme to detect access patterns based on approximate reuse analysis results [4]. Zhong et al. use those access patterns to predict cache behavior for all possible data sizes using a limited number of actual observations [13]. Shen et al. show that reuse-distance in programs exhibits phase-dependent behavior and that information-processing methods (such as wavelet transformations) can be used to understand the time-frequency characteristics of reuse [12]. Other efforts have focused on per-instruction characterization of reuse distance [9], [10]. All of these analysis and transformation methods considered the behavior of a single reference stream; the strategies therein could be combined with the methods from this paper to provide a better understanding of locality in multicore programs.

Berg et al. present a statistical model of multiprocessor caches with full associativity and random replacement [23]. That work models coherence misses probabilistically using a definition of reuse distance that counts all intervening references, whether distinct or not. In contrast, our work models coherence misses directly, considers how synchronization points may be used to propagate coherence information, and performs reuse distance analysis using the more commonly-studied metric of stack distance. These distinctions makes their work well-suited for cache performance evaluation but less so for application locality analysis.

Shi et al. present an analytical model of data replication and a method to simulate multiple caches in a CMP in a

single pass [24]. Their method uses a list-based reuse stack to model private and shared caches and to account for data replication in the shared cache, but instead of calculating the actual reuse distance for each access, it simply accumulates the appropriate hit and miss counters for the various caches. Therefore it is unsuitable for uses other than simulation.

Agarwal and Gupta study shared memory reference patterns independent of a specific memory hierarchy by marking a reference as a *ping* if it accesses a block that was last accessed by a different core and as a *cling* otherwise [25]. The number of clings between pings on a block indicates how often that block remains associated with the same core (*processor locality*), the number of references (distinct or not) between clings at a given address represents temporal locality, and successive references by the same core to nearby addresses represent spatial locality. Their per-address approach aims primarily to explore cache coherence alternatives, whereas this work aims to model how application reuse characteristics interact with the memory hierarchy's sharing configuration to shape overall performance.

VI. Conclusions

Reuse distance analysis has previously proven to be a valuable tool for cache modeling, application locality prediction, and locality-aware code transformations [8], [4], [9], [10], [11], [1], [12], [13]. This paper extends reuse distance analysis to the multicore domain by exploring methods to make the reuse stack account for inter-thread interactions, such as invalidations for per-core caches or inter-core prefetching for shared caches. The results show that modeling these interactions can lead to reuse distance analysis that is much more accurate in predicting application locality characteristics. The net effect is to substantially improve the robustness and applicability of reuse distance analysis in the face of evolving multicore hardware.

References

[1] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[2] G. Almási, C. Caşcaval, and D. A. Padua, "Calculating stack distances efficiently," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 37–43, 2003.

[3] B. T. Bennett and V. J. Kruskal, "LRU stack processing," *IBM Journal of Research and Development*, vol. 19, pp. 353–357, 1975.

[4] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2003, pp. 245–257.

[5] F. Olken, "Efficient methods for calculating the success function of fixed space replacement policies," Lawrence Berkeley Laboratory, Tech. Rep. LBL-12370, 1981.

[6] R. A. Sugumar and S. G. Abraham, "Multi-configuration simulation algorithms for the evaluation of computer architecture designs," University of Michigan, Tech. Rep., 1993.

[7] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *IASTED conference on Parallel and Distributed Computing and Systems 2001 (PDCS01)*, 2001, pp. 617–662.

[8] K. Beyls and E. D'Hollander, "Generating cache hints for improved program efficiency," *Journal of Systems Architecture*, vol. 51, no. 4, pp. 223–250, 4 2005.

[9] C. Fang, S. Carr, S. Önder, and Z. Wang, "Reuse-distance-based miss-rate prediction on a per instruction basis," in *In Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, 2004, pp. 60–68.

[10] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2004, pp. 2–13.

[11] G. Marin and J. M. Mellor-Crummey, "Pinpointing and exploiting opportunities for enhancing data reuse," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2008*, Apr. 2008, pp. 115–126.

[12] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2004, pp. 165–176.

[13] Y. Zhong, S. G. Dropsho, and C. Ding, "Miss rate prediction across all program inputs," in *In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2003, pp. 91–101.

[14] D. Bhandarkar and J. Ding, "Performance characterization of the pentium®pro processor," in *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 1997, p. 288.

[15] Intel Corporation, "Intel core 2 extreme quad-core processor qx6000 sequence and intel core 2 quad processor q6000 sequence," Datasheet, Aug. 2007.

[16] E. Larson and S. Chatterjee, "Mase: A novel infrastructure for detailed microarchitectural modeling," in *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, 2001, pp. 1–9.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[18] V. Aslot, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "Specomp: A new benchmark suite for measuring parallel computer performance," in *Workshop on OpenMP Applications and Tools*, 2001, pp. 1–10.

[19] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multiprocessing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[20] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance." [Online]. Available: http://citeseer.ist.psu.edu/408248.html

[21] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *20th International Symposium on Distributed Computing*, 2006, pp. 194–208. [Online]. Available: http://dx.doi.org/10.1007/11864219_14

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

[23] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model," *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pp. 89–99, March 2006.

[24] X. Shi, F. Su, J.-K. Peir, Y. Xia, and Z. Yang, "Modeling and single-pass simulation of cmp cache capacity and accessibility," in *Proc. IEEE International Symposium on Performance Analysis of Systems & Software ISPASS 2007*, 2007, pp. 126–135.

[25] A. Agarwal and A. Gupta, "Memory-reference characteristics of multiprocessor applications under mach," in *SIGMETRICS '88: Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1988, pp. 215–225.