Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization

Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai
Purdue University
West Lafayette, IN 47907
{dschuff, milind, vpai}@purdue.edu

ABSTRACT

Reuse distance analysis is a well-established tool for predicting cache performance, driving compiler optimizations, and assisting visualization and manual optimization of programs. Existing reuse distance analysis methods either do not account for the effects of multithreading, or suffer severe performance penalties. This paper presents a sampled, parallelized method of measuring reuse distance profiles for multithreaded programs, modeling private and shared cache configurations. The sampling technique allows it to spend much of its execution in a fast low-overhead mode, and allows the use of a new measurement method since sampled analysis does not need to consider the full state of the reuse stack. This measurement method uses $O(1)$ data structures that may be made thread-private, allowing parallelization to reduce overhead in analysis mode. The performance of the resulting system is analyzed for a diverse set of parallel benchmarks and shown to generate accurate output compared to non-sampled full analysis as well as good results for the common application of locating low-locality code in the benchmarks, all with a performance overhead comparable to the best single-threaded analysis techniques.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]

General Terms

Performance, Measurement

1. INTRODUCTION

The details of locality in multicore processors are architecture-specific, but locality is largely determined by fundamental parallel program characteristics: data reuse and inter-thread interactions. Reuse distance analysis (also known as stack distance analysis) characterizes memory locality by forming a histogram of the number of references to *distinct* data since the last reuse of the same data, or ∞ for data not previously referenced [27]. Reuses are conceptually tracked using a stack, but more efficient implementations also exist [1, 3, 17, 30, 36]. Data granularity can be words, cache blocks, or VM pages, but the metric is independent of the size of storage in the memory hierarchy. Reuse distance directly predicts the hit ratio of a fully-associative LRU cache, since data with reuse distance of less than N would hit in a fully-associative LRU cache of size N. Since a single run of reuse distance analysis models the behavior of of all possible cache sizes, this analysis has been used and validated not only for modeling locality but also for performance optimizations such as generating cache hints or restructuring code and data to improve locality [7, 9, 17, 19, 25, 26, 34, 40].

Recent studies have extended reuse distance analysis to consider the effects of multicore processors, some of which have private coherent caches and some of which have shared caches. Some of these methods combine data sharing characteristics with per-thread measurements of reuse distance to estimate actual reuse distance in a shared reuse stack [16, 20]. In previous work, we measure actual reuse distance in private or shared reuse stacks, keeping private stacks "coherent" by eliminating entries from other stacks when they are written at one stack and modeling shared stacks by interleaving references from different threads onto a single reuse stack [31, 32].

Although faster than running separate tests to simulate the locality-related behavior of all possible cache sizes, full reuse distance analysis is quite slow. Even under the best implementations, analyzing every memory reference of a program costs at least 1–2 orders of magnitude in execution time slowdown, with even slower performance for parallel programs. For maximum utility, analysis should be fast enough to run a whole program with large datasets to completion, allowing, for example, a software developer to see the detailed performance effect of code changes. The popularity of tools like Valgrind has shown that developers are willing to accept an order of magnitude slowdown for a sufficiently useful tool [33].

This paper presents random sampling and parallelized analysis as mechanisms to achieve these performance objectives. Whereas full reuse distance analysis tracks every reference, sampling analysis randomly selects individual references from the dynamic reference stream and yields a sample for each by tracking unique addresses accessed until the reuse of that address. The sampling analyzer can account for multicore characteristics in much the same way as the full analyzer. One important point, though, is that initi-

PACT'10, September 11–15, 2010, Vienna, Austria.

ating a sample collection at any thread requires all threads to record their references in per-thread *distance sets* so that inter-thread interactions can be properly captured through the modeled stacks. For large enough sample counts, unbiased random sampling can closely approximate the full process characteristics [29].

Multicore reuse sampling allows the use of a fast-execution mode when no samples are currently active, but requires mutual exclusion on each distance set update or invalidation to ensure that all inter-thread interactions are captured as expected. Parallelized analysis allows each thread to run analysis at the same time by exploiting the observation from previous work that the exact timing of inter-thread interactions does not significantly impact the measured reuse distances [31]. To model private stacks, each thread separately records its writes, and the model processes the interactions only when the sampled address is reused or a synchronization event (such as a barrier) is reached. To model shared stacks, each thread keeps its own distance set, merging them only when any thread reuses the tracked address.

This paper describes parallelized sampling reuse distance analysis and evaluates its implementation using fork-join, transactional, and pipelined benchmarks from the SpecOMP, NAS, STAMP, and PARSEC benchmark suites [2, 13, 21, 12]. These techniques result in a system with high accuracy that has comparable performance to the best single-thread reuse distance analysis tools, averaging 30x slowdown from native execution, with 96% accuracy compared to a full non-sampled analysis for invalidation-based reuse stacks. Two variants modeling shared caches allow different tradeoffs of speed and accuracy, giving 80–265x average slowdown from native and 74–89% accuracy compared to non-sampled analysis. In addition, all variants are demonstrated effective for an example application of selecting important portions of code which cause cache misses.

2. BACKGROUND ON MULTICORE REUSE DISTANCE

Reuse distance has long been a popular architecture-independent quantitative metric for data reuse in programs, and a single run of reuse distance analysis can model the behavior of all possible memory hierarchy sizes by using a stack to track the depth of accessed memory addresses in LRU order [27]. When considering multicore systems, this analysis must be augmented, as 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. 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 an address between two reuses of that same address by another thread with a different cache, an invalidation will 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. In prior work, we presented a multicore-aware model of reuse distance analysis that accounts for these inter-thread interactions during measurement [32].

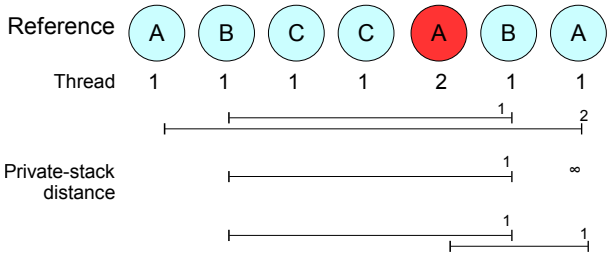

Figure 1: Example reference stream showing reuse distances for single independent stacks, and multicore-aware private and shared stacks

When modeling shared caches, all threads share a single reuse stack in a straightforward manner. When modeling private caches, each thread has its own reuse stack. When a block is invalidated from a real cache, it leaves behind a free slot (a hole) 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. In a general stack this means that blocks should never decrease in depth, other than when they are accessed. However, the hole can be filled without evicting any other blocks; thus, if a hole is filled in, blocks that otherwise would have increased in depth do not. To model this behavior, blocks that are invalidated are left in the stack but marked as invalid. This means an access to a block a deeper in the stack has the same reuse distance as if the invalidation had not occurred. Then when a is brought to the top, the hole is filled in. In the absence of any holes, a block b that was above a before it was accessed would increase in depth. If there is a hole above b, then it is filled in and b does not increase in depth. However, to keep blocks below a's original position from decreasing in depth, the slot at a's original position becomes the hole. If a new address is added to the stack, it effectively comes from infinite depth; thus, if it fills a hole, the hole is simply removed.

Figure 1 illustrates different approaches to measuring reuse distance with an example reference stream to 3 different blocks; the fifth reference is a write from thread 2 and the rest are reads from thread 1. If reuse stacks are independent, the reuse of block B has a reuse distance of 1 (C is the only block referenced in between) and the reuse of block A has distance 2. With multicore-aware private stacks, the intervening write of A by thread 2 causes the second reference by thread 1 to have infinite distance due to the invalidation, and with a shared stack, its distance is reduced to 1.

This approach effectively accounts for the sharing and invalidation behavior of multithreaded programs, but it drastically reduces the speed of analysis. All reuse distance analysis methods add significant overhead, increasing program runtime many times over. For parallel programs, however, the problem is compounded, because the tight inter-thread interaction used by the analysis negates the benefits of parallel execution. One of the goals of this paper is to rectify this problem. A good solution should analyze execution directly without requiring a trace and should be fast enough to use as part of a program development cycle; preserving a program's parallel execution substantially furthers this goal.

3. SAMPLED PARALLEL MULTICORE REUSE DISTANCE ANALYSIS

3.1 Reuse Distance Sampling

Full reuse distance analysis tracks every reference, maintaining global state that is updated with every reference, and yields a measurement (sample) for every reference. The fastest full reuse distance analysis uses $O(\log M)$ time per reference for accurate analysis or $O(\log \log M)$ time per reference for approximate analysis, where M is the total size of the data accessed by the program [17]. In contrast, the sampling implementation selects individual references from the dynamic reference stream and yields a sample for each. Because reuse distance for a reference is determined by the global state, the sampler must track all references between the use and reuse of the address accessed by the sampled reference. When a reference is selected for sampling, the set of all unique addresses accessed between the use and reuse (the *distance set*) is recorded in a hash table. When the address is reused, the size of the distance set is the reuse distance for the initial reference, and this reuse distance is recorded in the histogram. As the analysis only requires one hash table lookup, this method of measurement requires only amortized $O(1)$ time per reference, but yields many fewer samples than the full analysis. In theory this technique could be used for full analysis by replicating it for every outstanding use/reuse pair in the program, but it would cost up to $O(M)$ time per reference and $O(M^2)$ space in the worst case.

The sampling tool operates in two modes: fast mode, which simply counts down the number of dynamic references until the next sample; and analysis mode, which tracks all references for measurement. References are chosen at random from the dynamic reference stream. At each sample, the number of references before the next sample is chosen from a geometric distribution, ensuring that each dynamic reference has an equal probability of being chosen. The sampling rate can be adjusted by specifying the expected number of references between samples.

The system begins in fast mode and selects the number of references before the first sample. After the selected number of references passes, the sample address is recorded and analysis mode tracks all references until the sample address is reused. The number of references until the next sample is chosen at the beginning of analysis mode rather than the end, to avoid skewing the sample. This means that multiple samples may be tracked simultaneously, if the next sample is reached before the current address is reused. Each active sample has its own distance set that must be tracked independently. When a sampled address is reused, it is removed from the list of tracked addresses; when that list empties, the system returns to fast mode until the next sample.

3.2 Multicore Reuse Distance Measurement

The discussion thus far has not considered multithreading. Data sharing can be accounted for directly by using a shared sampler that sees references from all threads. Accounting for invalidations in private stacks is less straightforward. As discussed in Section 2, an invalidation leaves behind an empty cache block (a hole) in its slot in the reuse stack. A hole is filled in when a block is brought to the top from a position deeper than the hole, or when a new block is added. Along with the distance set, each tracked address also has a count of the holes above it in the reuse stack. On each write of an address a_w by thread t_w, all other threads t_i are checked. For each address a_i currently tracked by t_i, its distance set is checked. If a_w itself is the address being tracked (i.e. $a_w = a_i$), it is recorded as an invalidation with infinite distance and removed from the list of tracked addresses. If a_w is in the distance set of a_i, it is invalidated (removed from the set) and a hole is created. The actual addresses that become holes need not be kept; only the hole count is required. If in the future a new address is added to a_i's distance set, the hole is filled in by decrementing the hole count. When a_i is eventually reused, its reuse distance is the size of the distance set plus the hole count.

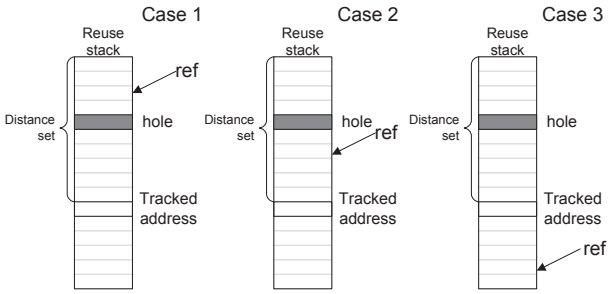

Figure 2: 3 possible stack positioning cases for a hole (invalidated block) and new reference in a distance set

Writing to an address that another thread t_i is tracking causes that address to be sampled as an infinite distance because one of two things will happen: either the address will be accessed by t_i again, which will be a coherence miss, or t_i will never access it again, in which case the sample was a last touch, which is also recorded as an infinite distance. To see that the handling of holes is correct, it is helpful to consider the depth in the per-thread reuse stack of the address a_i being tracked, the address a_r being referenced, and the topmost hole. The distance set of a_i contains all still-valid addresses that are between the top of the stack and a_i, and the hole count is the number of addresses that were in this range but have been invalidated and were not filled in. Therefore the depth of a_i is the size of its distance set plus the hole count, and the depth of all tracked holes is less than the depth of a_i. When an a_r is referenced, there are 4 possibilities:

1. If a_r is above a_i in the stack and above the topmost hole, it has been referenced since the last reference to a_i and is therefore already in the distance set. It is moved to the top of the stack without changing the depth of a_i and without filling in any holes; the distance set and hole count remain the same. This is Case 1 in Figure 2.

2. If a_r is above a_i in the stack and below the topmost hole, it moves to the top and fills in the hole, but the hole moves down to a_r's original position, which is still above a_i. Therefore the distance set and hole count remain the same. This is Case 2 in Figure 2.

3. If a_r is below a_i (and therefore below the topmost hole) it will go to the top of the stack, and fill in the topmost hole. It will be be added to the distance set and the hole count will be decremented, so the depth of a_i to remains the same. This is Case 3 in Figure 2.

4. If a_r is a new address that is not in the stack at all, the situation is the same as case 3.

3.3 Parallel Reuse Distance Sampling

The section so far has described a sampled method of measuring multithreaded reuse distance. However, it shares with full reuse distance the requirement that access to the measurement structures be serialized, or at least heavily synchronized. For shared reuse distance stacks, each thread must update the shared hash table on every reference, which requires locking at least part of the table. For private stacks, a write reference requires the writing thread to check every other thread's distance sets and, if the written address is in the distance set, to remove it and increment the hole count. This again requires synchronized data structure accesses on every reference, potentially many per reference. Because atomic operations are many times slower than normal memory references, and because having data updated frequently by many processors causes severe performance penalties, the analysis would ideally require neither of these things in the common case. Parallel reuse distance sampling is a modification of the sampling method that achieves this goal, at the possible cost of some accuracy.

Private stacks. The method so far described requires that when a thread writes to an address, this potential invalidation is immediately propagated to all other threads, updating their distance sets and hole counts. Previous work showed that in data-race-free programs, data overwritten by one thread is not usually referenced immediately by another thread; there must be some synchronization controlling the access. Therefore the exact timing of the propagation of these writes between threads does not significantly impact the reuse distance measurement [31]. The parallel method takes advantage of this fact. Instead of immediately checking the other threads' distance sets on every write, each thread keeps track of the set of unique addresses written since the sample was created; this set is called the *write set*. For every active sample, the thread that made the initial reference tracks the distance set, updating it on every reference. All other threads track a write set for this sample, updating it on every write. This split allows each thread to have its own private analysis data. At a synchronization event such as a barrier, any invalidations caused by the writes are propagated back to the sample's distance set. This process is the same as above; each written address is checked against the distance set, and if the address is found in the distance set, it is removed and the hole count is incremented. When this synchronization happens, all threads are suspended and wait until the writes are all propagated before continuing. This synchronization also happens when the sample's address is reused. If there are no recognizable synchronization events, invalidations are not propagated until reuse, potentially causing overestimation of reuse distances (a new block added to the distance set after an unpropagated invalidation occurred will increase the distance set size rather than filling a hole).

Shared stacks. The parallel method for shared stacks is similar, but simpler. In a shared reuse distance stack, the reuse distance of a reference is the total amount of unique data referenced by all threads between the use and reuse. Once a sample is activated, all threads keep their own distance set for the sample, until one of them encounters a reuse of the address. Then they all pause and the distance sets are merged; the combined size of the distance sets is the sample's reuse distance.

Pruning. The parallel sampling method has the benefits of being completely parallel and free of any atomic synchronization operations for most references. This is one source of speedup compared to the full analysis. The second source is the fast mode; when there are no active samples, no operations need to be performed on a per-reference basis. Therefore, the fraction of time the sampler spends in fast vs. analysis mode significantly impacts the performance. This fraction in turn is determined by the locality of the program itself; the longer the references go between use and reuse, the longer the sampler will have to remain in the slow analysis mode. This behavior is in fact controlled not by the reuse *distance* (in space) but by the reuse *time* (or total number of references between use and reuse, whether to unique or repeated data elements). Suppose for example an "unlucky" sample is chosen at the beginning of the execution, such as an address that is used during initialization and not touched again until the end. This outstanding sample will force the sampler to operate in analysis mode for the entire execution, significantly limiting the speed benefits of sampling. To keep these stray samples from having a disproportionate effect on the analysis speed, a heuristic called pruning is employed. Periodically (e.g. whenever a new sample is created) the pruning procedure checks the oldest outstanding sample. If this sample's current reuse distance is sufficiently large (e.g., if it is in the top 1% of reuse distances seen so far), it is pruned and recorded as if it were a cold miss. This scheme prevents long-standing samples (especially those from the very beginning of execution) from keeping the sampler in analysis mode for too long. Making the threshold based on distance rather than time allows better control over the resulting inaccuracy (pruning will only occur at large distances), but this can have a performance cost. If, for example, the program executes a long-running loop with a small memory footprint while a sample is active, it will consume a lot of time in slow analysis mode without significantly increasing the reuse distance of the sample (and therefore will not trigger pruning). Making the pruning threshold relative to references seen so far allows it to handle any size input, but requires that pruning not start until enough samples have been recorded to have meaningful statistics.

4. EXPERIMENTAL METHODOLOGY

Binary Instrumentation. Both the full reuse distance analyzer and the sampled analyzer are implemented as plugins for the Pin binary instrumentation system [24]. For the full analysis, the reuse distance stack implementation is based on Sugumar and Abraham's splay tree code as distributed with SimpleScalar 4.0 [23, 36], modified as described in previous multithreaded reuse distance work [32]. It is invoked by Pin on every memory reference; because it is not parallel, access is serialized using Pin's built-in locking facility. For the sampled implementation, the operation mode is determined by the global count of the number of active sampled addresses. If it is greater than 0, the system is in analysis mode and the analysis code is invoked on every reference. Regardless of the current mode, the reference counting code is called at every basic block rather than every reference (for increased efficiency).

Workloads. This system is evaluated with twelve benchmarks: six from SPEC OMP2001 [2], three from the OpenMP version of the NAS Parallel Benchmarks version 3.3 [21], one from the STAMP transactional memory benchmark suite [13], and two from the PARSEC benchmark suite [12]. 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; `genome` from STAMP; and `canneal` and `ferret` from PARSEC. The OpenMP benchmarks are data-parallel, `genome` uses transaction-based parallelism, `ferret` uses a pipelined model, and `canneal` uses unstructured parallelism with lock-free synchronization.

All benchmarks were run with 4 worker threads except `ferret`, which requires at least 5 for its pipeline. The "train" data input was used for SPEC benchmarks and the "A" input size for NAS benchmarks. The "native" input parameters were used for `genome` with the TL2-x86 STM system distributed with STAMP [15]. The "simlarge" input was used for the PARSEC benchmarks. Time comparisons were taken for the region of interest as defined by the benchmark. Due to the randomness inherent in sampling, sampling results were averaged over 3 runs. The expected average sampling rate was one per million references for all benchmarks except `canneal`, which was sampled at one per 500,000 to get sufficient samples. After 100 samples were collected, further samples were pruned if their distance became greater than 99% of all previous samples.

5. EXPERIMENTAL RESULTS

5.1 Private stacks

Figure 3 shows the reuse distance results reported by the the full and sampled analyzers for all benchmarks analyzed using the private per-thread stack model. The figure shows the reuse distance histograms as CDFs; the X axis is reuse distance in 64-byte blocks, and the Y axis is the fraction of references with a reuse distance less than or equal to x. In general the results of the full analysis and the sampled analysis correspond closely, and the working sets of the programs are clearly identifiable using either analysis.

The major exception is `genome`. In the fully-measured histogram, 87% of its references have a reuse distance of 0. This is an artifact of the full analysis code's perturbing the analyzed program's behavior, especially during locking in the STM system. Because the instrumented program spends so much time in the analysis code while the analyzed program holds the locks, the other threads spin more often while waiting to acquire them. Because the sampled analysis has much lower overhead than the full analysis, it does not suffer from this problem. As reuses due to spinning have no intervening accesses, this perturbation largely affects the 0-distance bin. Ignoring this bin when making comparisons can mostly compensate for the interference; the reported accuracy results adopt this approach.

There are several possible ways to determine if sampling produces accurate results for reuse distance measurement. The simplest is to compare the overall reuse distance histograms. A straightforward way to perform this comparison is to normalize the histograms such that each bin represents the fraction of references with that range of distances, and compare each bin. If f_i and s_i are the values in bin i of the fully-measured and sampled histograms, then the cumulative error $E = \sum_i(|f_i - s_i|)$ for all i. E has a maximum value of 200% and the overlap of the histograms (i.e. the accuracy of the sampled histogram on a scale of 0% to 100%) is given as $1 - E/2$, as in [17]. The histogram bins are distributed logarithmically, such that there are 10 bins per power of 2. Appendix A discusses the error bounds for accuracy measurements on sampled reuse distance histograms.

Table 1 shows the 12 benchmarks, with the accuracy of parallel sampled reuse distance compared to the full analysis, the slowdown compared to native execution, and the analyzed fraction (the percent of all memory references spent in analysis mode). On average, for the private stacks, the accuracy is 95.6% with a slowdown of 29.6x. This is quite accurate for a very general definition of accuracy; exactly what accuracy metric is appropriate and what accuracy level is sufficient depends on the application. For example, an application such as PC selection requires enough samples to get meaningful distributions for many PCs, the longer distances are more important than the shorter distances, and the relative distributions of the PCs are more important than the overall average.

The base performance overhead of Pin in the fast mode is is 5.3x on average, which includes the basic JITing and instrumentation, and decrementing the references remaining until the next sample on each basic block. Each memory reference in the instrumented program is converted to check if there are any active samples and if so, call the analysis code (which at minimum doubles the number of memory references, with additional instructions in each basic block for reference counting). Comparison with other reuse distance and other memory access analyzers is difficult, particularly since this paper specifically targets multithreaded programs, which pose challenges both for instrumentation platforms and analysis. Zhong and Chang report base slowdowns (i.e. with just instrumentation but no analysis) of 2–4x for compiler-based instrumentation and 4–10x for the Valgrind framework [28] for single-threaded benchmarks [39]. In contrast Valgrind with no analysis showed 4–75x slowdown for our parallel benchmarks, averaging 23x and failing to run 3 of them. The same paper reports an additional 10–90x slowdown with sampled analysis. Other slowdowns have been reported as 15–25x for compiler-based single-threaded non-random sampling with a lower-resolution reuse distance approximation [8]. Valgrind's popular memory checker [33], which also analyzes every memory reference, averages 97x slowdown for our benchmarks. So, although there are no direct comparisons, the slowdowns reported here are broadly similar to existing tools, while extending reuse distance analysis to multithreaded programs.

The overall performance is dependent on the *analyzed fraction*, the percent of all program references spent in analysis mode. The overall analyzed fraction is 19.6%, indicating that the sampler processes most references in fast mode. This in turn is dependent on the reuse behavior of the program. The pathological case is `swim`, which is forced to analyze nearly all its references because pruning is ineffective, leading to a slowdown of 143x. One possible way to improve this could be to reduce the sample rate and try a longer-running input.

The sampled analyzer is on average 177 times faster overall than the full analyzer. Factoring out the speed gains from using the fast mode, the sampler's analysis mode in isolation is 29x faster than the full analysis. This is the speed gain from the parallelism (and the removal of most atomic

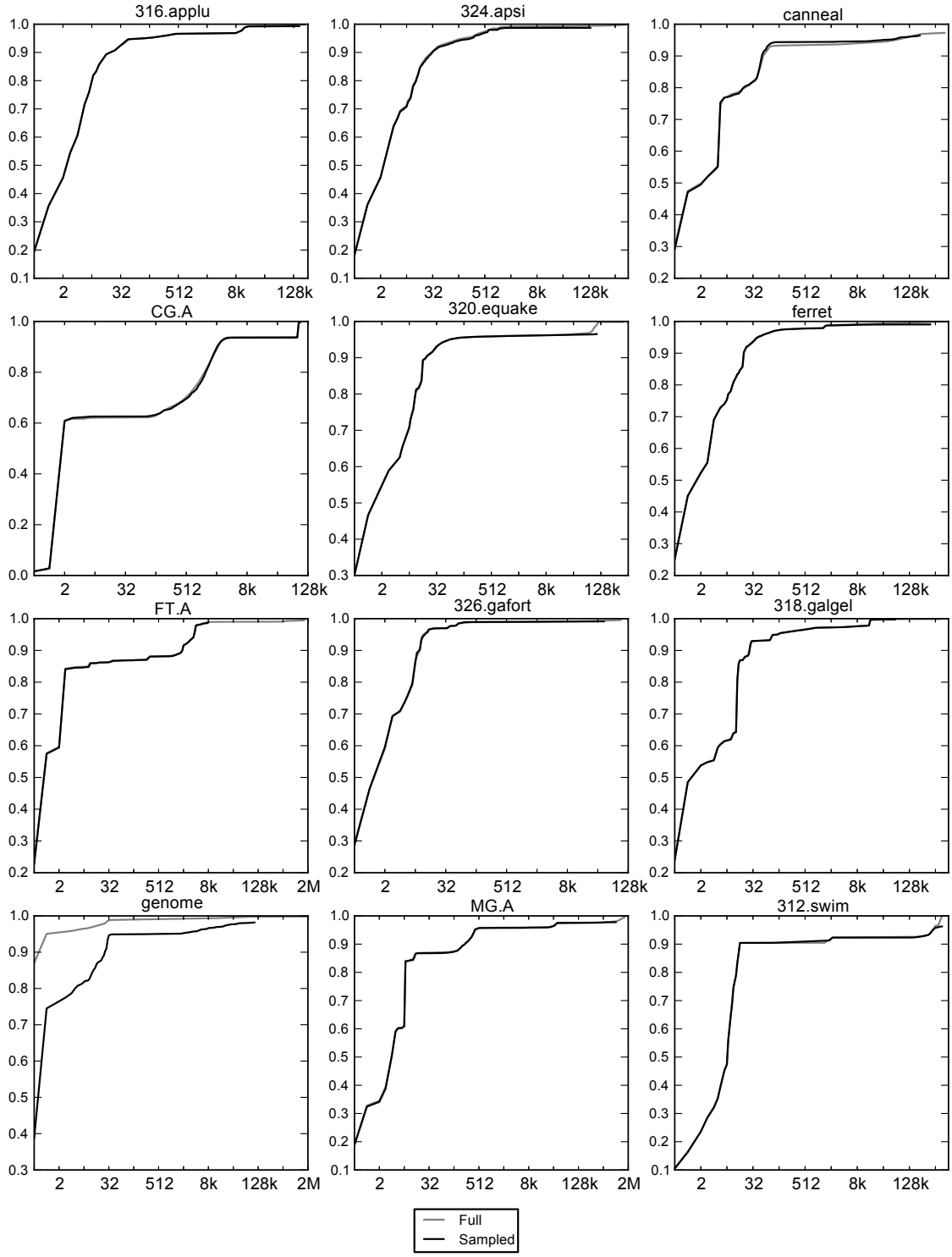

Figure 3: CDFs of reuse distance using private per-thread stacks. X axis is reuse distance in 64-byte blocks, Y axis is fraction of references with reuse distance less than or equal x.

operations), the use of private data structures (which do not suffer remote cache misses), and the fact that the sampled analyzer needs only $O(1)$ time per active sample on each reference, rather than $O(\log M)$ time (where M is the total program data) for the full analysis.

5.2 Shared stacks

With shared reuse stacks, the reuse distance seen by a reference in one thread is the sum of the unique data accessed by all threads that share the stack. Therefore that thread's measurements are dependent on references made by other

Table 1: Accuracy, slowdown, and fraction of references analyzed for sampled histograms

		Private			Shared			Shared, sequential samp.		
Benchmark	Samples	Acc.	Slwdn.	%Refs	Acc.	Slwdn.	%Refs	Acc.	Slwdn.	%Refs
applu	141814	98.6%	20.2	8.1%	86.6%	43.2	17.5%	93.6%	281.8	16.8%
apsi	148647	97.5%	7.8	4.5%	85.4%	9.6	4.1%	93.0%	62.2	7.0%
canneal	1098	91.4%	12.6	15.4%	75.2%	25.3	20.8%	88.7%	94.6	17.7%
CG	2335	95.9%	26.8	22.9%	45.5%	90.3	19.8%	94.8%	354.6	19.4%
equake	46718	95.3%	16.1	15.5%	87.1%	33.6	15.6%	93.1%	130.7	11.9%
ferret	9804	97.1%	24.5	6.8%	89.2%	28.5	6.2%	93.7%	154.2	11.6%
FT	8050	98.1%	21.9	7.4%	57.9%	42.2	12.0%	84.2%	225.5	10.4%
gafort	27135	95.6%	7.1	3.6%	83.3%	7.7	3.1%	93.0%	37.2	2.9%
galgel	167527	96.4%	10.2	8.2%	74.5%	13.7	9.1%	81.4%	13.2	1.8%
genome	1488	90.7%	7.7	22.6%	74.6%	12.3	21.8%	84.7%	22.2	4.8%
MG	4252	94.8%	57.0	28.3%	60.0%	157.2	32.9%	71.6%	720.8	36.9%
swim	67589	95.8%	143.0	92.1%	70.0%	496.3	93.8%	94.6%	1084.0	91.7%
Average		95.6%	29.6	19.6%	74.1%	80.0	21.4%	88.9%	265.1	19.4%

threads, regardless of whether they are sharing data with or synchronizing relative to the measuring thread. Consequently the measured reuse distance depends on the relative rate of execution of the different threads. This relative rate can vary between runs of the same program, either slightly or drastically (e.g., a thread may be suspended by the operating system and access no data at all) and can potentially be altered by analysis. This is an undesirable characteristic in a metric that is supposed to be hardware-independent; however, it does not mean that reuse distance becomes meaningless. In particular, threads in data-parallel programs with regular structure will tend to execute at the same rate on average; likewise threads in some pipelined programs may execute at the same relative rate, although for others it may depend on the input. This means that reuse distance is still an inherent property of the program and has meaning independent of the particular execution or hardware environment; however, there will be more variation both in measured reuse distance results and in cache performance on real hardware.

As a baseline to investigate this effect, the full analyzer (which serializes all reuse distance measurement) is modified to use a ticket-based queueing lock, ensuring fair ordered access to the waiting threads. It does not completely assure the same execution rate for all threads because they can still be put to sleep or get delayed executing code other than the analysis code (e.g., in Pin itself or in long sequences of non-memory instructions in the benchmark); however, it does come close, and it is significantly more fair than the default Pin lock implementation, which can cause waiting threads to sleep and often results in runs of many consecutive accesses by one thread. Figure 4 shows part of the histogram and CDF for the `FT` benchmark. In the baseline full analysis (black line), the working sets of the program are clearly visible; the well-defined knees in the CDF curve are the result of many references falling in the same histogram bucket, which can be seen as spikes in the histogram.

To compare against the baseline, two different sampled implementations are considered. First is the parallel sampler, implemented as described in Section 3. The result of the variation in relative speed of the threads is variation in the distances, which causes some of the samples to fall into neighboring bins in the histogram. Thus the spikes in the histogram and the knees in the curve of the CDF are not as well-defined, and the average accuracy over all benchmarks falls to 74%, as shown in Table 1.

For some purposes this variation in reuse distances is not a problem; for example, it does not significantly affect the problem of low-locality PC selection discussed in Section 6 (those results are presented using the parallel sampler). However for other applications (*e.g.*, a working set analysis) it can be a problem. One simple way to keep the threads running at closer to the same rate is to serialize the references during analysis mode, in the same way as the full analyzer, using the fair lock (the two-mode operation and stack-merging behavior are kept the same). This approach could be called "sequential sampling". The results of this technique can be seen in Figure 4 (dotted line), where the spikes in the histogram and corresponding knees in the CDF are much clearer than with parallel sampling (grey line). And because the general histogram accuracy metric depends on samples falling into the correct bin, the accuracy is much improved as well, reaching 89% on average.

Performance. The parallel shared sampling analysis has reduced performance compared to the private stacks, as Table 1 also shows. This is due to two effects: the first is the increased cost of merging the reuse stacks when a sample is reused. With invalidation, the synchronizing operation requires traversing the write set and removing any items found there from the distance set. With merging, all threads' distance sets must be merged into a single hash table to filter out duplicates, possibly substantially increasing its size. This extra overhead occurs during synchronization and penalizes all threads if any of them have slow merges. The second is the resulting increased load on Pin's memory allocator (which has its own synchronization), which reduces the performance. This reduction in performance could be mitigated by tuning the implementation of the distance sets and potentially using more intelligent merging algorithms, such as parallel reduction (which would give even more benefit as the number of threads scales).

The cost of the additional accuracy gained by sequential sampling is also reduced performance; the average slowdown of analysis mode compared to the base shared analysis is 4.1x, (almost exactly equal to the number of threads), indicating that aside from the serialization the technique introduces little extra overhead. Note that fast mode remains parallel, even in the sequential sampling implementation. Accuracy can be traded off for performance fairly smoothly by synchronizing the threads periodically (but without merging the stacks), which would allow limited skew between the threads' execution rates. Decreasing the frequency of synchronization will improve performance, but

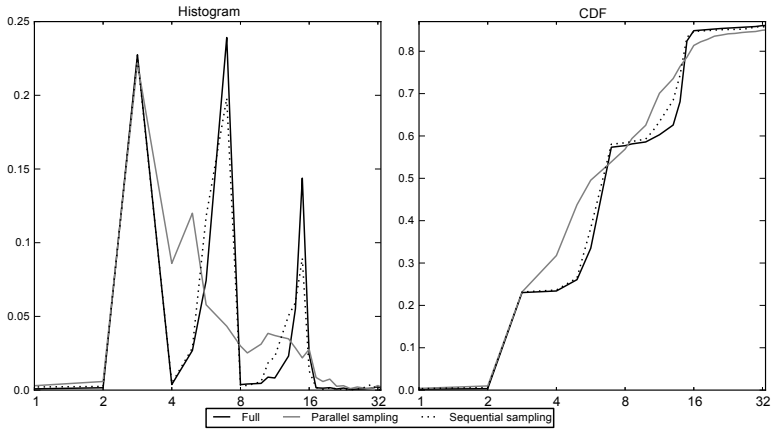

Figure 4: Portion of the reuse distance histogram (left) and corresponding cumulative distribution function (right) for full analysis, parallel and sequential sampling on the FT benchmark using the shared stack model

degrade accuracy; in the limit, the analysis will behave like the full parallel analysis.

6. APPLICATION: LOW LOCALITY PC IDENTIFICATION

One major application of reuse distance analysis is determining which portions of a program contribute most to poor locality and assisting the programmer in making improvements [8, 11, 26]. The chief challenge in this application is selecting PCs that contribute a significant proportion of a program's misses.

Suppose we would like to concentrate our optimization effort on a small number of PCs. For example, the top N PCs that are responsible for 75% of the misses at a particular cache size C. If we use the sampled parallel reuse distance analysis to select these PCs, how effective will this selection be as an optimization target? To measure the accuracy of low locality PC selection using sampled reuse distance analysis, we use Wall's *weight-matching* approach [38], which measures how well an estimated profile (generated by the sampled analysis) selects entries of interest compared to a reference profile (generated by full analysis). Intuitively, weight-matching answers the question, if we select N PCs using the sampling run, how important are those PCs compared to the N most important PCs according to the reference run?

Each PC i in the reference run is assigned weight $w_{r,i}$ (*i.e.*, the number of misses suffered at PC i in the reference run). We compute W_r, the total weight of the top N PCs (by weight) in the reference run. Next, we perform the optimizing run using the sampled, parallel reuse distance analysis. Each PC is assigned a weight $w_{s,i}$ according to the number of misses suffered as predicted by the sampled run. We find the top N PCs according to their weights in the sampled run, and compute their total weight, W_s, *according to their weights in the reference run.* Note that the identities of these top N PCs are determined by the weights $w_{s,i}$, but the total weight W_s is determined by the weights $w_{r,i}$. The accuracy, α, of low locality PC selection in the sampled run is given by $\alpha = \frac{W_s}{W_r}$.

Note that the accuracy measurement is dependent solely on the weights of the selected PCs, not the identity of the PCs. This measures the *effectiveness* of targeting PCs using the sampled analysis, not whether the exact same PCs are selected as in full analysis (the latter is captured by Wall's *key-matching* metric [38]).

The remaining question is how to select N. We cannot simply choose a fixed N; in some benchmarks, the few most important PCs suffer the bulk of the misses, while in others, they contribute only a small fraction. Furthermore, determining which reuses cause misses requires choosing a target cache size, C. As the working set of every benchmark is different, there is no single, suitable C. What is needed is a benchmark-independent method for choosing N and C. We choose C by examining the CDF of reuse distances (in the full analysis), and setting C such that a fixed percentage, x, of reuses would be predicted as misses; in our experiments, we set x to 10%. We then find the set of PCs responsible for those high-reuse distance accesses, and choose N to account for a given percentage, y, of the total misses in the program; Table 2 shows results for $y \in \{75\%, 80\%, 90\%, 95\%\}$, for both private and shared stack configurations. The average accuracy over the benchmarks is 91%–92% across both stack types and all different sets of PCs.

These accuracy results demonstrate that sampled reuse distance analysis can provide accurate, and hence useful, PC selection information. There are other formulations of the problem that use additional criteria, both to guide PC selection and to further assist programmers. For example, methods have been proposed that include information about which code is executed between a use and reuse, to suggest refactorings [10]. Parallel sampled reuse distance can be extended to incorporate such additional information.

7. RELATED WORK

7.1 Multicore-aware reuse distance

Jiang et al. discuss the issues related to shared-stack reuse distance, such as its potentially hardware-dependent nature [20]. They present a probabilistic model based on per-thread reuse distance that takes data sharing information into account to predict reuse distance for shared caches (it does not consider private invalidating caches). Ding and Chilimbi also predict multithreaded reuse signatures and cache performance by collecting information on data sharing and thread interleaving and combining this information with per-thread reuse distance analysis [16]. However, their approach is limited by its statistical nature—it does not directly track the interactions between threads.

Table 2: Accuracy of low locality PC selection (percent)

Benchmark	Private Accuracy				Shared Accuracy			
Miss Coverage	75%	80%	90%	95%	75%	80%	90%	95%
applu	95.96	97.02	96.48	96.74	96.18	95.3	94.64	94.65
apsi	97.82	97.46	96.98	96.97	95.86	94.78	93.44	92.98
canneal	86.08	87.73	87.47	87.95	98.61	96.81	96.38	92.62
CG.A	100	100	100	100	100	100	100	100
equake	85.68	85.53	93.08	93.29	89.22	87.01	84.82	87.34
ferret	85.46	87.48	88.88	90.72	93.34	93.02	90.94	88.76
FT.A	94.45	94.72	89.45	89.89	96.88	97.17	89.39	88.5
gafort	91.67	96.54	96.04	97.67	93.44	90.37	91.2	89.03
galgel	99.93	99.99	99.87	99.65	76.94	84.45	96.71	95.65
genome	98.28	98.11	96.63	95.56	87.7	86.63	87.6	84.21
MG.A	73.55	72.01	75.99	76.19	76.68	75.61	81.65	85.77
swim	84.15	82.87	80.4	79.58	99.69	99.71	99.94	100
Average	91.09	91.62	91.77	92.02	92.05	91.74	92.23	91.63

Berg et al. present a statistical model of multiprocessor caches with full associativity and random replacement [6]. Unlike a reuse distance analysis, which tracks the number of *unique* references between two accesses to a specific address, this approach counts *all* intervening references, whether distinct or not. Like the Ding and Chilimbi approach, this technique does not directly model coherence effects, but instead approximates them statistically. Eklov and Hagersten show how to estimate stack distances from these reference counts [18], but do not cover multiprocessors.

Several studies have investigated using reuse distance-based models to study the cache performance of multiprogrammed workloads (*i.e.*, multiple, distinct programs running simultaneously) [14, 22, 37]. Because these approaches focus on multiprogramming, they need only deal with destructive interference between separate processes contending for limited cache resources; they do not address the effects of constructive sharing. Furthermore, all these techniques profile the different programs separately and then use a probabilistic model to combine the separate reference streams; this is less accurate than directly tracking the interaction between threads.

Shi et al. present an analytical model of data replication and a method to simulate multiple caches in a CMP in a single pass [35]. 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.

7.2 Sampling-based reuse distance analysis

The statistical cache model of Berg et al. uses SPARC hardware performance counters and watch-points to count instructions and references and to monitor accesses to specified addresses in their single-thread implementation [4, 5, 6]. Similarly to the fast mode/analysis mode used here, their system randomly selects a number of instructions and configures a counter to interrupt after that many have executed. It chooses the next reference as a sample and uses a hardware watch-point to interrupt when it its address is reused; then it uses the performance counters to determine the number of references that have passed. Note that this approach measures *time* distance, rather than stack distance.

Zhong and Chang use sampling to reduce the overhead of traditional reuse distance measurement [39]. They use a 2-mode system where accesses that occur in a sampling period are marked as "sampled" (meaning they are the start of a reuse pair and will be measured) and accesses that appear in the "hibernation period" are not. During sampling periods, their method uses the well-known tree-based approximate analysis [17]. During the hibernation period the tree is not updated, but all references are checked and the count of unique data during the period is tracked. Then the entire period is added as one node in the tree. In addition, for every access in both modes, the analyzer checks to see if the address is marked as sampled, and if so, it uses the tree to determine its reuse distance. This results in good accuracy and a 5.7x speedup of the analysis over the non-sampled version.

Beyls and D'Hollander used a form of sampling in their reuse distance visualization, turning the analyzer on for 20 million references and then off for 180 million references, reporting a slowdown of 15-25x [8]. However, no discussion is given on the accuracy of this method or its impact on the results or speed of execution. They later present the use of reservoir sampling to speed up the measurement of time distance for single-threaded programs [10]. It uses a target number of samples and dynamically adds and discards samples in the pool to get desired statistical properties. The analysis also tracks other program properties that are more heavyweight than time distance, so the sampling allows most references to only record time distance, while sampled references do more. This approach takes advantage of the fact that measuring time distance is simple; if it were used to track stack distance, the reservoir sampling technique would require running in analysis mode all the time.

8. CONCLUSIONS

This paper explores and validates the use of random sampling and parallelization as techniques to accelerate multicore-aware reuse distance analysis. Sampling accelerates performance by allowing a fast-execution mode during periods when no monitored references are active. Parallelization allows the analysis to simultaneously utilize as many threads as the base application, adding synchronization only at application synchronization points or when actually recording a measurement. Additionally, the new methods use measurement techniques that do not model the details of the reuse distance stack, allowing even the analyzed portions to proceed much faster than in full reuse distance analysis.

The combination of these techniques allow multicore-aware methods to achieve performance comparable to the best single-thread reuse distance analysis tools while still maintaining high fidelity to the reuse distance histograms generated by full multicore reuse distance analysis. In addition, the analysis with parallelization and sampling is highly accurate at identifying the most performance-critical PCs in terms of their impact on cache miss count. These techniques thus facilitate the use of multicore-aware reuse distance analysis for both performance modeling and targeted optimization.

9. REFERENCES

[1] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. *SIGPLAN Not.*, 38(2 supplement):37–43, 2003.

[2] 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*, pages 1–10, 2001.

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

[4] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Proc. of 2004 IEEE ISPASS*, pages 20–27, 2004.

[5] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proc. of SIGMETRICS '05*, pages 169–180, New York, NY, USA, 2005. ACM.

[6] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. *Proc. of 2006 IEEE ISPASS*, pages 89–99, March 2006.

[7] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proc. of IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.

[8] K. Beyls and E. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *Proc. 4th Intl. Conf. on Computational Science*, volume 3038, pages 448–455, 6 2004.

[9] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 4 2005.

[10] K. Beyls and E. D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. In *Proc. 2nd Intl. Conf. on High Performance Computing and Communications (HPCC)*, 2006.

[11] K. Beyls, E. D'Hollander, and F. Vandeputte. Rdvis: A tool that visualizes the causes of low locality and hints program optimizations. In *Proc. 5th Intl. Conf. on Computational Science*, 2005.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proc. of PACT '08*, pages 72–81, New York, NY, USA, 2008. ACM.

[13] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE International Symposium on Workload Characterization*, 2008.

[14] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. of HPCA '05*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of 20th Intl. Symp. on Distributed Computing*, pages 194–208, 2006.

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

[17] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proc. of ACM SIGPLAN PLDI '03*, 2003.

[18] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *Proc. of 2010 IEEE ISPASS*, pages 55 –65, 28-30 2010.

[19] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proc. of 1st Workshop on Memory System Performance*, 2004.

[20] Y. Jiang, E. Zhang, and K. Tian. Is reuse distance applicable to data locality analysis on chip multiprocessors. In *Proc. of Intl. Conf. on Compiler Construction*, Paphos, Cyprus, March 2010.

[21] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance.

[22] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. of PACT '04*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[23] E. Larson and S. Chatterjee. Mase: A novel infrastructure for detailed microarchitectural modeling. In *Proc. of 2001 IEEE ISPASS*, 2001.

[24] 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 *Proc. of ACM SIGPLAN PLDI '05*, 2005.

[25] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc. of SIGMETRICS '04/Performance '04*, pages 2–13, New York, NY, USA, 2004. ACM.

[26] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *Proc. of 2008 IEEE ISPASS*, 2008.

[27] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of ACM SIGPLAN PLDI '07*, pages 89–100, New York, NY, USA, 2007. ACM.

[29] NIST/SEMATECH. *e-Handbook of Statistical Methods*, June 2003.

[30] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

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

[32] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. In *Workshop on Performance Modeling, Evaluation, and Optimisation of Ubiquitous Computing and Networked Systems*, 2010.

[33] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of 2005 Usenix Annual Technical Conference*, Apr. 2005.

[34] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proc. of ASPLOS-XI*, 2004.

[35] 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. of 2007 IEEE ISPASS*, pages 126–135, 2007.

[36] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, 1993.

[37] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proc. of 15th Intl. Conf. on Supercomputing*, 2001.

[38] D. W. Wall. Predicting program behavior using real or estimated profiles. *SIGPLAN Not.*, 39(4):429–441, 2004.

[39] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proc. of 7th Intl. Symp. on Memory Management*, pages 91–100, New York, NY, USA, 2008. ACM.

[40] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *In Proc. of PACT '03*, 2003.

APPENDIX

A. REUSE DISTANCE SAMPLING ACCURACY

The accuracy of sampling can be determined by comparing the the histogram produced by the sampled analysis against the one produced by the full analysis. Each bucket in the histogram counts the number of analyzed events that produced a reuse distance outcome in a given range. As a result, the histograms can be considered proportion statistics, where p_h represents the fraction of the n samples that produce the outcome corresponding bucket h. According to the central limit theorem of statistics, the observed sample proportion for a large enough sample size is normally distributed around the actual population proportion π_h with variance $\frac{\pi_h(1-\pi_h)}{n}$ [29]. The confidence interval range for the sample proportion is thus $\pi_h \pm z\sqrt{\frac{\pi_h(1-\pi_h)}{n}}$, where the area under the standard normal function curve between $-z$ and z is the desired confidence level ($z \approx 1.96$ for 95% confidence).

Different options exist to state with the desired confidence level that the error in the histogram introduced by sampling is within a certain bound:

- Limit the absolute error of the sample proportion reported by each bucket h to some value ϵ. In this case, $\epsilon = z\sqrt{\frac{\pi_h(1-\pi_h)}{n}}$, which sees its maximum value at $\pi_h = 0.5$. This implies that the number of samples n must be at least $\frac{z^2}{\epsilon^2}(0.5)^2 \approx \frac{1}{\epsilon^2}$ for 95% confidence. Since these are absolute errors, they should be quite small to avoid causing problems for small measurements; an e value of 0.001 would thus require a million samples.
- Limit the relative error of the sample proportion reported by each bucket to a given fraction of the population proportion. Here, ϵ represents the relative error, with a range limit of $\frac{z\sqrt{\frac{\pi_h(1-\pi_h)}{n}}}{\pi_h} = z\sqrt{\frac{1-\pi_h}{n\pi_h}}$. This implies that the sample size n must be at least $\frac{z^2(1-\pi_h)}{\epsilon^2\pi_h}$ Both the error bound and the number of samples required reach their maximum value at the smallest π_h value of interest. If we wish to track π_h as low as 0.01% and maintain per-bucket relative errors of no more than 10% with a 95% confidence level, then n would be nearly 4 million, as suggested by previous work on sampling for reference-count reuse distance (not stack reuse distance) [10].
- Limit the total error across all the H histogram buckets. This requires us to formulate the aggregate absolute error E across the buckets. This is maximized when each bucket sees its maximum error:

$$E = \sum_{h=1}^{H} z\sqrt{\frac{\pi_h(1-\pi_h)}{n}} = \frac{z}{\sqrt{n}}\sum_{h=1}^{H}\sqrt{\pi_h(1-\pi_h)}$$

This is clearly not a straight-forward summation; however, an upper bound is $\frac{z}{\sqrt{n}}\sum_{h=1}^{H}\sqrt{\pi_h}$. Using the generalized-mean inequality, followed by algebraic operations:

$$\begin{aligned}(\frac{1}{H}\sum_{h=1}^{H}\sqrt{\pi_h})^2 &\leq \frac{1}{H}\sum_{h=1}^{H}\pi_h \\ \sum_{h=1}^{H}\sqrt{\pi_h} &\leq \sqrt{H}\sqrt{\sum_{h=1}^{H}\pi_h}\end{aligned}$$

Since all samples must be in some histogram bucket, $\sum_{h=1}^{H}\pi_h = 1$. Thus, the aggregate absolute error limit is $z\sqrt{\frac{H}{n}}$, implying a sample requirement of at least $\frac{z^2H}{E^2}$. If our goal is to have at least 95% confidence that the aggregate error across 200 histogram buckets is no more than 10%, then we can make do with just 76,832 samples. Although the paper uses this accuracy metric, most of our tests have far fewer samples but still have about this level of accuracy. This is because the actual error caused by sampling is more likely to be close to 0 than to the bound shown here, as it is typically normally distributed for unbiased random sampling.

Note that the above only considers errors induced by sampling, not by other deviations in the modeling (such as those sometimes caused by parallelization in the shared stack models). The exact choice of accuracy metric clearly depends on our application. If we are simply trying to approximate overall application-locality and cache performance, we can most probably use the aggregate error metric. If we need to focus on specific bucket entries, though, we will need to use one of the single-bucket error metrics.