Characterizing Active Data Sharing in Threaded Applications Using Shared Footprint

Hao Luo
Department of Computer Science
University of Rochester
Rochester, NY 14627
hluo@cs.rochester.edu

Xiaoya Xiang
Department of Computer Science
University of Rochester
Rochester, NY 14627
xiang@cs.rochester.edu

Chen Ding
Department of Computer Science
University of Rochester
Rochester, NY 14627
cding@cs.rochester.edu

ABSTRACT

Footprint is one of the most intuitive metrics in measuring locality and the memory behavior of programs. Much of the footprint research has been done on workloads consisting of independent sequential applications. For threaded applications, however, the traditional metric of footprint is inadequate since it does not directly measure data sharing.

This paper proposes two new metrics called *shared footprint* and *sharing ratio* to capture the amount of active data sharing in a threaded execution. It also presents an empirical characterization of the data sharing behaviors using these metrics on the PARSEC Benchmark Suite. Based on the initial results, this paper discusses possible uses of the new metrics in program tuning and optimization for threaded applications on multicore processors.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, performance

Keywords

Data footprint, multithreaded program, data sharing

1. MOTIVATION

Locality theory is the foundation for understanding and managing how programs use hierarchical memory and storage, which have become ubiquitous on modern machines. Programs can be characterized by their data access features based on a set of locality metrics derived from locality theory.

In the 1970s, program locality was studied in a series of research in working set theory by Denning et al. Different metrics were proposed for quantifying program locality, including cache miss rate, which is widely used in the evaluation of program performances. Footprint is another well-studied metric that also quantifies a workload's locality. Performance models based on footprint are composable and advantageous over cache miss rate. Time- and space-efficient algorithms for computing footprint have been developed recently. Previous research efforts using footprint to predict cache miss rate for sequential programs have proved its power in exploiting a program's memory usage pattern.

With recent trends in utilizing multi-core processors in high throughput computing, parallel applications have become pervasive nowadays. Performances of such workloads heavily rely on use of memory components such as the shared last level cache (LLC). For parallel programs, when and how much data will be shared during the course of execution directly determines the optimal LLC allocation strategy. Data sharing behavior is therefore one of the key features that characterizes modern parallel workloads. Although recent studies on the footprint model have demonstrated success in composing multiple execution flows to predict cache miss rate, they shed little light on how data is shared. This is due to the fact that the footprint model only reveals the features of $OVERALL$ data accesses interleaved by ALL threads. In order to utilize footprint theory to tackle the problems for multithreaded workloads, the theory needs to be extended.

Therefore we propose two new locality measurements to describe data sharing of multhreaded applications. These new notions essentially represent the amount of shared data and its proportion in footprint over a time period. They are natural extensions of the footprint metric for sequential programs in the world of parallel workloads.

This paper makes following contributions:

- We define a new metric called *shared footprint* (*sfp*) for multithreaded applications. As in footprint model, for efficient computation and representation, we use average shared footprint ($\overline{sfp}$), the average amount of shared data within the windows of equal size in memory trace. Furthermore, we denote the ratio of average shared footprint and footprint as *sharing ratio*. It can be seen that shared footprint and sharing ratio are powerful tools to exploit the inherent data sharing patterns of parallel programs.

- We perform a preliminary analysis on the data sharing behaviors of the PARSEC Benchmark Suite [2], a suite of emerging parallel workloads which are representative of next-generation shared-memory programs for chip multiprocessors (CMP). As an initial analysis of parallel workloads using shared footprint and sharing ratio, we collected statistics from PARSEC at runtime and characterized benchmarks based on these metrics.

- We also sketch a few applications of the shared footprint model to show its flexibility and extensibility. These applications include quantification of false sharing and suggestion of shared cache allocation strategy. It is also helpful in providing guidance for thread scheduling.

The rest of the paper is organized in the following manner. Section 2 gives a formal definition of shared footprint and sharing ratio. An example to further explain our metrics is given in Section 3. Section 4 provides a characterization of PARSEC Benchmark Suite using the sharing ratio metric. Section 5 discusses applications of the shared footprint metric that analysts could use to gain insights into multithreaded workloads. Section 6 presents related work and Section 7 is future work and conclusions.

2. DEFINITION

Essentially the footprint is defined as the amount of distinct data accessed in a time period while shared footprint refers to the amount of data that are accessed by at least two threads. The proportion of shared data in all data accessed in that period is called sharing ratio.

During the course of a program's execution, a binary-rewriting tool such as Pin[7] can instrument the program to record the target address of every memory operation and construct a trace of memory accesses. A performance tool can utilize the trace or its statistics obtained from intrumentation to compute locality metrics such as footprint. Reseachers usually refer to a subtrace as a window and the amount of distinct data within a window is called the *footprint* of that window. The notion of window is an abstraction of time period in our locality theory. The length of a window roughly corresponds to the duration of execution. For multithreaded programs, all concurrent threads, executions can be serialized to produce one whole execution trace for convenience of processing. An element or data in the trace is said to be *actively shared* if it is accessed by at least two threads in an execution window. The number of distinct data so accessed in the window is called its *shared footprint* which measures the amount of active data sharing by different threads.

The distinction of being actively shared (or not) differs from the quality of being shared (or private). A datum is *shared* if it may be accessed by multiple threads. A datum is *actively shared* if the accesses from multiple threads happen in a given execution window. If we take the window to include the whole execution, the set of shared data is also the volume of active data sharing. If we take a shorter window, some of the shared data may not be actively shared. In the following, we use the term footprint, which represents active data usage; and shared footprint, which represents active data sharing.

More formally, we define a memory access trace of an execution of workload as a sequence, $s_1 s_2 ... s_N$, where each s_i is a data unit, representing a memory region of unit size (a byte, a word, a cache block, or a page) and each s_i is associated with a thread id t_i, which identifies which thread made access to s_i. In this paper, we use $w_{i,j}$ to denote subtrace (or window) $s_i s_{i+1} ... s_j$ and the size of it is $j - i + 1$. $fp(w)$ is a function of window that maps window w to the number of distinct data units within it. Data unit d in a window $w_{i,j}$ is *shared* if there exist p and q such that $d = s_p = s_q (i \leq p \neq q \leq j)$ but $t_p \neq t_q$. $sfp(w)$ is a function of window. It maps a window w to the number of distinct shared data units in w.

Given a memory access trace, computing the shared footprint for all windows will involve quadratically many windows. Therefore we use the average shared footprint over all windows of one size instead. We denote the *average shared footprint* over all windows of size l as $\overline{sfp}(l)$, which is

$$\overline{sfp}(l) = \sum_{i=1}^{N-l+1} \frac{sfp(w_{i,i+l-1})}{N-l+1}$$

Note that an element may be actively shared in window w but not so in another window w'. This is because it is accessed by multiple threads in w, but by just one thread in w'. Also note that we could define *thread-exclusive footprint* in terms of data exclusively accessed by only one thread, the traditional metric of footprint is sum of thread-exclusive footprint and shared footprint. Let us denote average footprint over windows of size l as $\overline{fp}(l)$.

$$\overline{fp}(l) = \sum_{i=1}^{N-l+1} \frac{fp(w_{i,i+l-1})}{N-l+1}$$

It is helpful to know in a time period how many data were shared among those accessed. Therefore we define the notion of sharing ratio as the ratio of *sfp* and *fp* of a window size. Parallel applications can be characterized according to their sharing ratios under different configurations of runs, as we will see later.

3. EXAMPLE

In this section, we give an example to explain the notions above. To represent memory trace, we use letters to represent data units and numbers above them as thread id. Figure 1 shows a trace generated by a 2-threaded execution. Thread 1 and 2 both accessed private data before accessing shared data to synchronize. In this trace, the data units "a", "c" are private to thread 1 and "b", "d" are private to thread 2 respectively. Data "e" is a global data shared by both threads. Let us consider the windows of size 2. According to the definitions of our new metrics, the average footprint $\overline{fp}$ is $\frac{2+2+1+2+2+2+1}{7} = \frac{12}{7}$ and the average shared footprint $\overline{sfp}$ is $\frac{0+0+1+0+0+0+1}{7} = \frac{2}{7}$. In two windows, the data "e" is accessed twice, thus only one distinct data is contained in them and their footprints are 1. The rest windows contain two different elements therefore their footprints are 2. On the other hand, for a datum to be shared, there has to be at least two accesses to it by different threads. In exactly those windows accessing "e" twice, "e" is accessed by both threads. Therefore only these two windows have non-zero shared footprint.

Even if the data "e" in this example is shared by two threads throughout the whole trace, it is private within some windows. For instance, considering the window $w_{4,7}$(the window starting at 4th element and ending at 7th element), "e" is accessed solely by thread 2. In this window "e" is not actively shared, therefore the shared footprint of this window, according to our definition, is 0.

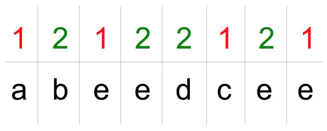

Figure 1: Example of memory trace for concurrent programs

4. CHARACTERIZATION RESULTS

In this section, we characterize multithreaded programs according to their sharing ratios and present their computation pattern. We could use the definition of $\overline{sfp}$ and the algorithm described in [16] to collect footprint (fp) and shared footprint (sfp) statistics. We did experiments on PARSEC v2.1 Benchmark suite testing programs using the three provided input sizes: simsmall, simmedium and simlarge. Upon each memory operation, our instrument tool tries to obtain a global lock provided by Pin to synchronize with other threads and record the intended memory address and its thread id. The data unit in the obtained memory trace is of size 64 bytes, the cache block size on our experiment platform. Every workload is benchmarked under thread count configuration 1, 2, 4, 8, 16, 32, 64 except *swaptions* which does not allow 64-thread configuration for simmedium and simsmall input. For the sake of brevity, we only present the memory trace size (N) and memory footprint (M) statistics for simlarge input under 1, 4, 16, 64 thread configurations in table 1. We used Pintool v2.11 [7] to implement instrumentation algorithm. The applications were profiled on a Linux cluster where each node has two 4-core Intel Xeon 3.2GHz processors. The operating system kernel is Linux 2.6.43.8-1.fc15.x86_64.

We analyzed the sharing ratio of each benchmark under simlarge input. By comparing this metric, multithreaded programs can be categorized based on two criteria, sensitivity to thread count and sharing ratio change over window size.

4.1 Sensitivity to thread count

This criterion measures a workload's sensivitity to thread count variation. If increasing multithreading does not have a noticeable impact on a workload's sharing ratio, we call this workload insensitive to thread count. Otherwise it is sensitive. This characterization relates the degree of data sharing to thread count.

- **Insensitive Applications** Programs having this characteristic have consistent sharing ratios under different thread count configurations. A typical example is *blackscholes* (Figure 2). Blackscholes is data parallel, where a global array is initialized and partitioned by the main thread. Worker threads handle their local parts of the array with little interaction with each other. Since thread interaction is quite low, increasing threads has limited impact on sharing ratio. Similar applications in our experiment are *x264*, *canneal* and *raytrace*.

[1]n in the legend is the minimum number of threads to use

Benchmark		4-thread	16-thread	64-thread
blackscholes	N	16.69	16.69	16.69
	M	6.8	6.9	7
bodytrack	N	37.11	37.11	37.13
	M	24.18	24.52	25.42
facesim	N	124.45	126.87	132.32
	M	537.92	535.26	556.37
vips	N	85.21	85.22	85.26
	M	27.96	57	161.2
x264	N	46.69	46.81	46.85
	M	38.58	92.38	291.02
raytrace	N	186.75	186.75	144.14
	M	294.85	294.99	281.92
ferret	N	85.8	85.8	85.78
	M	103.83	130.81	201.77
freqmine	N	159.61	162.75	160.1
	M	239.2	287.39	418.68
swaptions	N	53.88	53.92	53.95
	M	3.07	3.4	4.03
fluidanimate	N	36.7	40.63	47.74
	M	90.01	110.48	145.62
streamcluster	N	89.69	89.77	119.42
	M	14.96	15	15.26
canneal	N	27.55	27.54	27.54
	M	257.69	257.88	258.43
dedup	N	133.78	133.8	133.81
	M	1697.06	2160.51	1922.81

Table 1: For each benchmark, N is the memory trace size of whole execution(10^8 times), M is the number of distinct data blocks ($10^4 \times 64$ bytes) accessed during the course of execution.

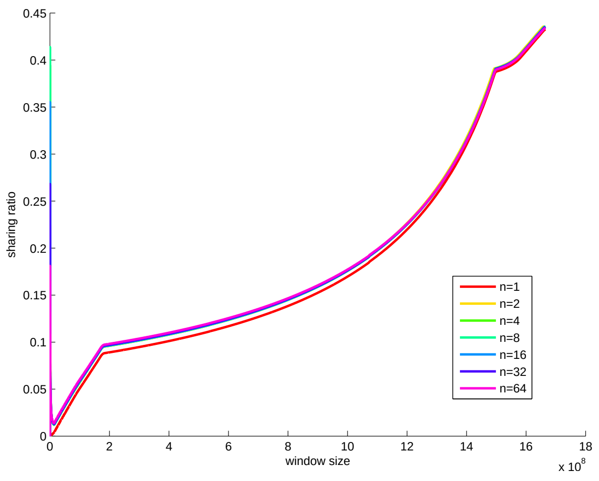

Figure 2: Blakcscholes: sharing ratio under different window size[1]

- **Sensitive Applications** Programs in this category exhibited sensivity to thread count variation. With varying number of threads, their $\overline{sfp}$ have varied portion in $\overline{fp}$. One such program is *vips* (Figure 3). *Vips* is a pipeline-style multithreaded image processing application, it has 18 stages, which are grouped into 4 kernels [3]. Similar applications with high thread contention like *fluidanimate* and *facesim* are also in this category. *Fluidanimate* is data parallel but its worker threads have communication with each other. It partitions a global 2D array based on the thread count. Each thread not only processes its local part but also interacts with its neighbors to handle the data on the boundary. Therefore more threads incur more shared data and varied sharing ratio.

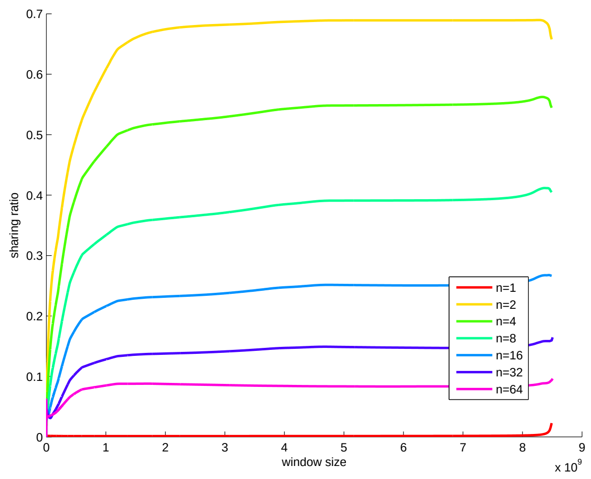

Figure 3: Vips: sharing ratio under different window size

4.2 Sharing ratio change over window size

This criterion categorizes applications according to their sharing ratio changes as window lengthens. Different applications may exhibit different sharing behavior in differently sized windows. The amount of shared data in windows of different sizes indicates the level of thread contention within different time periods.

- **Gradually increasing sharing ratio** These programs have gradually increasing sharing as window size increases. Representative workloads include *blackscholes* (Figure 2), *canneal*, *facesim* and *raytrace*. These workloads have a nontrivial amount of shared data and they keep accessing new shared data as program runs. In other words, their shared data do not have much reuse.

- **Consistently high sharing ratio over all windows** The defining characteristic is high sharing ratio in both short and long windows. The high sharing ratio at short windows implies that threads compete for shared data frequently. Benchmarks in this category include *x264* (Figure 4), *bodytrack*, *ferret*, *freqmine* and *vips*.

- **Sudden rise of sharing ratio** Two programs in our experiment had sudden increase in sharing ratio. They are *fluidanimate* and *streamcluster* (Figre 5). These two programs have low and consistent data sharing in short and medium windows but data sharing rises dramatically at one time point. *Streamcluster* is a coarse-grained, data-parallel program, where threads have low interactions with each other (This can be seen from the invariance of sharing ratios under different thread count). The sudden rise is likely to be

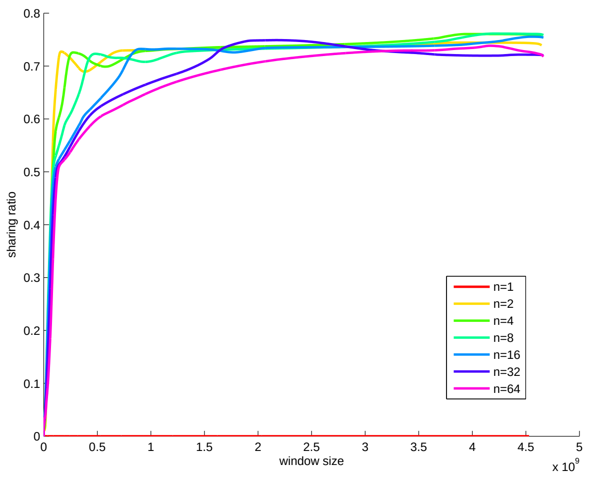

Figure 4: X264: sharing ratio under different window size

caused by threads joining after parallel regions and postprocessing performed by main thread.

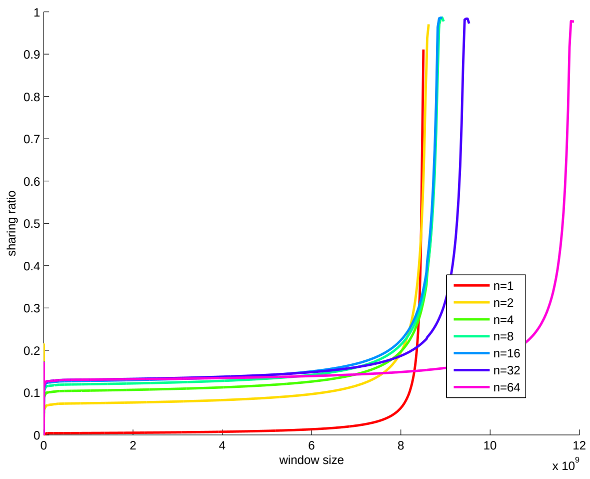

Figure 5: Streamcluster: sharing ratio under different window size

- **Sharing ratio decreases** The only benchmark in PARSEC of this type is *swaptions* (Figure 6). The sharing ratio decreases as the window size increases. For workloads in this category, there is a fixed set of shared data but a much larger set of private data. The shard data are repeatedly reused while private data do not have much reuse. So, as the window grows, the overall memory footprint grows but the shared footprint stays the same. *Swaptions* is a coarse-grained data-parallel program where little data is shared. It is worth noting the curve of 64-thread run. We believe this high sharing ratio is caused by false sharing. *Swaptions* under this configuration allocated a global array of 64 swaptions, each of which is of size 104 bytes, less than 2 cache blocks. Each swaption is assigned to handle one thread, therefore almost *60%* of cache blocks among the 104 cache blocks occupied by swaptions array is shared.

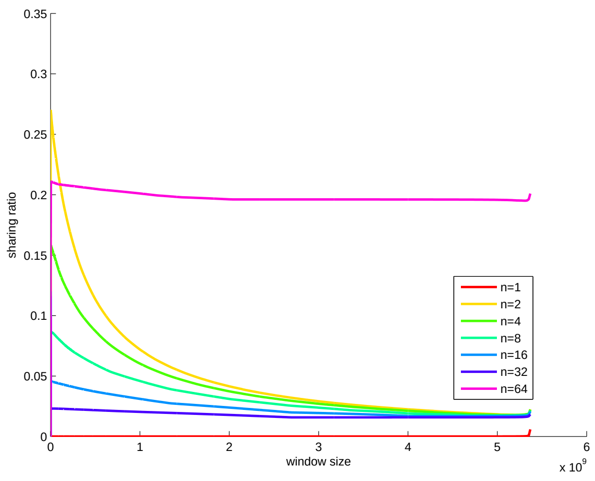

Figure 6: Swaptions: sharing ratio under different window size

4.3 Discussion

Our methodology in recording memory trace from multiple threads' execution is strongly dependent on thread scheduling and Pin's internal locking facilities. It is admitted that the exact memory trace is very likely to vary at different runs. However, such dependences are inherent to profiling of multithreaded program's execution. Our experimental results do exhibit impact from thread scheduling, but such impact is bounded. What remains unchanged among different runs is the basic shapes of sharing ratio curves, on which our characterization is based. Therefore we argue that our work is valid and our metrics are inherent to the programs. We are currently restructuring our memory trace model and incorporating probabilistic techniques into our analysis to enable more statistically sound results.

5. APPLICATIONS

In this section, we briefly sketch three potential uses of shared footprint and sharing ratio in parallel program analysis.

- **Hints on Shared Cache Allocation** When multiple threads run on shared cache, it is unclear whether partitioning the cache or leaving the whole cache as shared is a good strategy. And partitioning part of the cache for each thread as private while sharing the rest is also permissible and may produce better cache usage. The optimal allocation strategy is not easy to obtain in practice and is dependent on specific workloads. The shared footprint model tells the amount of shared data at any given cache size. Given a program's shared footprint and sharing ratio curves, analysts can read out how much data is shared and make decisions on shared cache allocation strategy to optimize cache usage.

- **Quantifying Level of False Sharing** False sharing is a performance degradation when two threads attempt to periodically access seperate data that share the same cache block [4]. With help of shared footprint modeling, we have a new representation of false sharing measurement. Note that the size of data unit in memory traces we obtained from instrumentation is typically a cache block size for the purpose of modeling cache. By adjusting data unit size, we can manipulate the granularity of sharing. The difference of two shared footprint curves of different data unit size can be thought of as a quantification of false sharing level.

- **Thread Scheduling** Another usage of shared footprint metric is to determine thread affinity. High shared footprint indicates high level of data sharing. For any given two threads, if memory trace generated by them exhibits a high sharing ratio, scheduling them together will yield highly efficient shared cache usage. Therefore the metric of shared footprint could also be useful in designing thread scheduling policy.

6. RELATED WORK

Modeling of program memory behavior has been investigated since 1970s. Denning et al. have used memory access trace to analyze programs' working set [5]. Their work focused on sequential workloads. Memory footprint is proved to be a successful indicator of application's locality. An efficient algorithm in computing footprint was well develped recently in [16, 15]. In [16], a footprint-based composable model to predict memory performance was proposed and evaluated.

Analysis on parallel workloads has become a new focus of research today. Barroso et al. used performance counters and simulation tools to characterize the memory system behavior of many commercial workloads [1]. SPLASH-2 Benchmark Suite was characterized by Woo et al [12]. Their methodology was using execution-driven simulation. Jaleel et al. analyzed data sharing for a set of parallel bioinformatic workloads [6] using binary instrumentation. Another focus of research is co-scheduling of threaded code. Scheduling policies based on locality metrics have gained its success in recent years. Pusukuri et al. developed a faithful thread scheduling policy using a simple metric called scaling factor[8]. Transformations of multithreaded code could also yield high benefit in performance. Zhang et al. discovered that currently standard compilation techniques still lack support in cache sharing, therefore placement of threads on cores does not impact performance much [18]. Sarkar and Tullsen showed that by simply optimizing data placement in awareness of cache sharing feature, noticeable improvements will be achieved for a variety of parallel workloads in SPEC2000 suite [9]. They utilized a *Temporal Relationship Graph* to model thread relation and guide data placement strategy.

Reuse distance analyses for multithreaded programs have also been studied in recent years. Schuff et al. described a private/shared stack model to extend reuse distance analysis for parallel programs running on multi-core platforms [10, 11]. Wu and Yeung focused their work on loop-based programs and have achieved high accuracy in predicting last-level cache performance [13]. In [14], they built a theotical model to exploit an optimal hierarchical cache configuration for loop-based parallel applications. Their works are based on reuse distance analysis. Our work is also on program locality but uses techniques borrowed from footprint theory.

For sequential programs, footprint is proved to be more informative than reuse distance [17]. So we expect that the shared footprint model can provide deeper insights into parallel program understanding than concurrent reuse distance anaysis.

In this paper, we adapted the well established footprint theory to multithreaded workloads, which has never been proposed before to our knowledge. The traditional footprint model does not capture data sharing behavior for parallel workloads. Shared footprint and sharing ratio can be a fitting complement for multithreaded applications. This is the first study that introduces a powerful locality analyzing tool for parallel applications.

7. CONCLUSION AND FUTURE WORK

This paper proposed new locality metrics, shared footprint and sharing ratio, for multithreaded workloads. They are valuable for looking deep into the nature of parallel programs and preliminary analysis has proved its worth. We conclude that our new metrics are amenable to extension and practical use.

Sharing ratio and shared footprint are foundation of a new set of locality thoeries for parallel applications. In future work, we will focus on lowering the overhead of computing shared footprint and constructing other relevant notions like lifetime, sensitivity and flow rate for parallel workloads.

8. ACKNOWLEDGMENTS

We would like to thank Jacob Brock, Weixing Ji, Li Lu and other colleagues for their helpful suggestions. We also thank the reviewers of WODA for their careful review and feedback.

9. REFERENCES

[1] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, Washington, DC, 1998.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[4] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, Sept. 1993.

[5] P. J. Denning and S. C. Schwartz. Properties of the working set model. *Communications of ACM*, 15(3):191–198, 1972.

[6] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In *Proceedings of HPCA*, 2006.

[7] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200, 2005.

[8] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. No more backstabbing... a faithful scheduling policy for multithreaded programs. In *Proceedings of PACT*, pages 12–21, 2011.

[9] S. Sarkar and D. M. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *Proceedings of HiPEAC*, pages 353–368, 2008.

[10] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.

[11] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. Technical Report TR-09-07, Purdue University School of Electrical and Computer Engineering, 2009.

[12] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *The 22nd annual international symposium on Computer architecture (ISCA '95)*, pages 24–36, 1995.

[13] M.-J. Wu and D. Yeung. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of PACT*, pages 264–275, 2011.

[14] M.-J. Wu and D. Yeung. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, pages 2–11, 2012.

[15] X. Xiang, B. Bao, and C. Ding. Program locality sampling in shared cache: A theory and a real-time solution. Technical Report URCS #972, Department of Computer Science, University of Rochester, December 2011.

[16] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of PACT*, pages 350–360, 2011.

[17] X. Xiang, C. Ding, B. Bao, and H. Luo. A higher order theory of cache locality. In *Proceedings of ASPLOS*, 2013.

[18] E. Z. Zhang, Y. Jiang, and X. Shen. The significance of CMP cache sharing on contemporary multithreaded applications. *IEEE TPDS*, 23(2):367–374, 2012.