2011 IEEE International Conference on Cluster Computing

Performance optimization of data structures using memory access characterization

Ashay Rane
Texas Advanced Computing Center
The University of Texas at Austin
ashay.rane@tacc.utexas.edu

James Browne
Department of Computer Science
The University of Texas at Austin
browne@cs.utexas.edu

***Abstract*—Program performance optimization is generally based on measurements of execution behavior of code segments. However, an equally important task for performance optimizations is understanding memory access behaviors and thus, data structure access patterns and properties. Because memory-related problems in multi-core applications can have a significant impact on overall performance, optimizations in data access patterns will likely give a big boost to application performance. But effective diagnosis of performance bottlenecks requires that the memory measurements be related to high-level data structures (C, C++ arrays, structures, etc.). In this work, we present a low-overhead tool that captures memory traces and computes several metrics for performance characteristics of source-level data structures. Explicit consideration is given to measurement and diagnosis for multicore chips. Case studies which include (manual) use of the data structure memory access metrics to select and implement optimizations are given.**

***Keywords*-performance; optimization; memory; data structures;**

I. Introduction

Performance optimization requires detailed characterization of the causes of and remedies for performance bottlenecks. Out of the four subtasks of performance optimization (measurement, diagnosis of bottlenecks, selection of effective optimizations, and implementation of optimizations), the measurement and diagnosis subtasks have, to date, largely focused on code-centric measurements, which associates execution costs with code segments such as functions or loops. Association of execution properties such as memory latency with data structures is not often done. Because memory is much slower than processors, a full understanding of the memory access patterns of important data structures is critical to accurate diagnosis of performance bottlenecks. Understanding memory access patterns becomes more important for multi-core chips which share memory access paths such as L3 caches across processing cores.

Memory access patterns have been studied and analyzed using a variety of tools in the past. The approach common to most of these tools has been to gather program-wide statistics related to memory accesses. Such program-wide information (or even statement-level information) does not support performance optimization. Application of memory access characteristics to performance optimization requires that the measurements and diagnoses be associated with source code (C, C++) data structures, usually with code-segment local scope. This paper describes an approach to and implementation of a tool for measurement and association of execution costs to data structures and identification of opportunities for data reuse. The tool measures and associates such metrics as average latency per access with a data structure.

There are several previous or existing tools which focus on measurement and analysis of memory behaviors but all have some limitations with respect to support for effective and efficient performance optimization. In [1], the authors present an approximation to reuse distance analysis that takes $O(log\ log\ M)$ time for processing every memory reference, where M is the number of unique memory references made by the program. However, their solution applies to single-core reuse distance calculation only. We extend their reuse distance calculation to allow modelling of multi-core reuse distances. A probabilistic model for determining the concurrent reuse distance based on the data locality of standalone programs is described in [2]. This approach does not take into consideration interactions between threads arising in a non-simulated environment. Schuff et al. demonstrate a technique in [3], [4] for modeling multi-core reuse distances however due to a restricted cache model, their method does not take into account multiple levels of caches. Their approach suffers from over-estimated penalties from multi-core interactions and it does not appear to scale as the number of threads grows. Chameleon [5] incorporates memory trace collection using the PIN dynamic binary instrumentation tool [6], as does [4]. Chameleon is built with the objective of generating and replaying synthetic traces to evaluate cache architectures. TAXI [7] is an X86 simulation environment that collects traces using the Bochs [8] simulator but it does not include the high-level trace analyses we describe in our work. All of the above mentioned approaches, with the exception of [1] for which there is little information regarding the implementation, use either simulators or PIN. Using simulators has the problem that they may not precisely model real-life scenarios like thrashing and prefetching. The use of simulators often greatly increases the time to obtain the results because of the software emulation of instructions. PIN has a wide user base

978-0-7695-4516-5 2011

U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/CLUSTER.2011.77
IEEE computer society

because of its ease of use. However, relating source-level data structures with trace analysis requires that the binary contain debug information (-g compilation flag), which may introduce additional perturbation during program execution. ThreadSpotter [9] is a commercial tool that runs various analyses on memory traces without requiring source-level instrumentation. However, to be able to relate measurements back to source code, the binary has to be compiled with debug information. Instead we instrument our code using LLVM [10] in such a way that the debug information is not required to be present in the binary. Since the instrumented code is compiled to native code, the binary can be run directly on the target platform. SLO, as described in [11], [12] uses a modified version of GCC to instrument programs to collect information about runtime reuse paths and analyze them for locality. Based on the analysis, SLO suggests optimizations in source code. However, SLO restricts itself to reuse distance analysis. In our tool, apart from reuse distance analysis, we compute several other metrics such as average latency for source code data structures.

One of the key motivations behind writing this tool was the growing difference between CPU and memory speeds. This has led to memory becoming a bottleneck for a large class of applications, which in turn requires memory analysis to be the focus of performance debugging.

There are two additional factors that prompt the need to perform detailed memory analysis. Multi-threaded codes are more prone to memory bottlenecks in comparison to codes that are single-threaded. With threading becoming commonplace, detailed memory analysis in the context of threads can reveal several opportunities for optimization. Issues such as memory bandwidth, Non-Uniform Memory Access (NUMA) latencies, number of available DRAM banks [13], etc. often limit scalability of applications due to the increased contention. Furthermore, both multi-threading on a given core and multiple threads due to multiple processors per chip are sources of complications. In fact, sharing patterns due to multiple cores are more subtle since the multi-threading on a single core only has one active thread at a time.

The other issue that forces one to consider memory traces is the limited information available from profiling of code alone. Various profilers [14], [15] among others show information at the level of functions or loops, or sometimes at the level of individual statements. Performance counter-based profiling of code can provide sophisticated information such as cycles per accesses by each level of cache [16]. However, combining data structure analyses provides additional insight into bad cache behaviour by identifying whether the problem is arising due to cache line invalidation, capacity misses, remote fetch latencies or other reasons. Another area where diagnosing a problem using memory analysis can help is when data structures are distributed throughout the code. For such cases, relating "use" and "reuse" of data structures can become very difficult with just code profiling.

The rest of the paper is organized as follows. Section 2 describes our approach and lists the memory analyses we have built into our tool. Section 3 lists the key innovations this work. We present the results of using the tool to optimize parts of the Rodinia benchmark suite [17] in section 4. Finally, section 5 presents our conclusions and lists our plan for further development of this tool.

II. Approach

In our tool, collection of memory traces for a given source code is done by instrumenting the code with calls to a statically linked library, this using the LLVM compiler infrastructure. The memory traces are then analyzed offline with respect to a particular cache configuration. The various analyses that we have currently built into our tool are listed below. We relate each of the following metrics to source-level data structures.

A. Multi-core reuse distance analysis

Reuse distance analysis has been a long standing metric of the locality of data accesses made by a program. Essentially it is a measure of the number of unique memory accesses made by a program between accesses to a particular location. The definition of reuse distance changes slightly in the context of multiple cores to account for shared or private references to data items.

B. Cycles per access

Reuse distance represents program behavior that does not take into account all architectural nuances that might cause a program to slow down. The most notable case is cache thrashing, where the reuse distance could be low but the penalty of accesses is high. We therefore decided to model latencies separately from reuse distances.

C. Non-Uniform Memory Access (NUMA) hit ratios

On machines that have memory affinity enabled, there can be a significant penalty arising from naïve patterns of allocation and reference of data structures. Threads from a different processor package may incur a high latency from remote fetches while accessing the same memory. NUMA hit ratios help to identify such problems.

D. Determining access strides

Programs that have unit strides or strides with small values generally execute faster than programs that don't. This is because the hardware prefetchers can recognize patterns in data accesses and bring data into caches before it is referenced, thus reducing data access penalty.

III. Innovations

We differentiate our tool from the past tools based on the following key points:

A. Source-level data structure information with approximate cost of access

Our tool relates trace information collected about the program back to source-level (C, C++) data structures without requiring the binary to be compiled with debug information (-g compilation flag). This makes it possible to understand the exact program variable that needs to be changed along with its location in the source code (file name and line number). Source-level information is obtained during the instrumentation process. While collecting this information, we are able to keep the overhead of instrumentation an order of magnitude low as compared to most other tools that use instrumentation [4], [5], [11]. We employ various techniques such as periodically disabling the instrumentation and spacing out the "windows" of instrumentation asymmetrically, whenever possible using atomic data structures instead of locks, etc. This results in an overhead of about 5-12x for our tool (as compared to the average slowdown of 30x for the fastest existing tool [4] according to our survey). We believe that this overhead can be further reduced by employing non-blocking IO operations.

B. Leveraging architectural information for reuse distance analysis

From our survey of existing tools, we found that most reuse distance techniques modeled cache configurations that are generally too simplistic. A wide variety of information can be included into the reuse distance and memory trace analyses to make the results of the analysis closer to being accurate. Reuse distance tools are inherently error-prone because they almost universally assume that caches are fully associative. Such an assumption is often a necessity to due to complexity in modeling the non-determinism in choosing the cache line to evict in set-associative caches. There exists other information like the levels of caches and sharing of caches among processor cores that can be exploited. Incorporating latencies of the individual caches allows a more accurate analysis. As an example, using information about sharing of caches and latencies makes it possible to include the effects of cache thrashing into the memory analysis. This information also makes it possible to answer questions like what percentage of memory accesses were likely served from the individual caches or how many memory requests were for local memory and how many for remote memory, which can be very helpful for tuning programs on NUMA configurations.

C. Multi-core reuse distance calculation with low bookkeeping overhead

Modeling the reuse distance in our tool is performed using a technique similar to that described in [1]. However, the idea is adapted for multi-core cases by maintaining timestamps for each cache. This results in $O(M)$ storage for every cache but maintains $O(N\ log\ M)$ time for reuse distance calculation. Depending on the cache that is referenced, the appropriate counter is chosen for finding the reuse distance. As an example consider a stream of references made by two processing cores as shown in Table I.

Table I
MEMORY ACCESS STREAM WITH NO CONFLICTING ACCESSES

Data	**a**	b	c	b	d	e	**a**
Core#	**1**	2	1	2	2	1	**1**

Table II
MEMORY ACCESS STREAM WITH CONFLICTING ACCESSES

Data	**a**	b	c	b	d	e	**a**
Core#	**1**	2	1	2	2	1	**2**

Here, since neither core references any common data, the reuse distance for data *a* for core #1 would be based on all references made by core #1 only. Thus the reuse distance would be based on the number of data references seen by the L1 cache of core #1. Consequently, the reuse distance for *a* is 2. However, if both cores performed writes to any common data, then the reuse distance calculation cannot consider the streams of each core in isolation. Consider the case in Table II, that differs in the core that accessed *a* for the second time. In this case, the reference to data *a* made by core #2 will cause the fetch to be performed from the cache that is common to both core #1 and core #2. Consequently, the reuse distance for *a* is now 4.

When compared with code profiling, although the information from profiling of code is essential to diagnosing bottlenecks and may indicate areas of code that are a problem, they do not always pinpoint the cause of the bottleneck. Adding the analyses of memory traces, makes it possible to distinguish between the individual manifestations of the problem. For example, it becomes possible to differentiate between first-time accesses (data never previously referenced), cache thrashing (low reuse distance but high access penalty), capacity misses (high distance and thus high penalty) and memory-affinity related problems such as remote memory accesses. When profiling code, all of the above mentioned cases would be seen as high cache misses in the last level cache but it would be almost impossible to easily ascertain the cause of the issue.

IV. RESULTS

We present a subset of the programs from the Rodinia benchmark suite [17] that were optimized based on measurements using our memory analyses. For this study, we have used the Rodinia OpenMP benchmarks in only.

We ran the original and the optimized benchmarks on Longhorn which is comprised of dual quad-core Intel Nehalem chips. Each node has 48 GB of main memory. Each of the 8 cores have a 32 KB level 1 data and instruction caches. A unified level 2 cache of size 256 KB is shared

between two cores while a unified level 3 cache of size 8 MB is shared by four cores.

We used PerfExpert [16], [18] to identify code segments that had high cycles per access. We were able to identify most, but not all, code sections that needed changes for improvement. We then used these memory analyses to find out other problem areas or possible optimizations. An interesting case was the Particle Filter program, which included a bad algorithm in one of its phases (unnecessary repeat passes over a large array). Using NUMA hit ratios and the estimated cycles per access calculated using the trace analysis, we were able to discover this problem. However, code profiling based on PerfExpert did not highlight that this was an issue.

For the sake of brevity, we include the output from our analysis tool for only one of the programs (Particle Filter) in Figures 1 and 2.

A. Particle Filter

The Particle Filter code tracks an object across video frames. For this code, as shown in Figure 1, we noticed that two data structures (`CDF` and `matrix`) had very high estimated cycles per access (78, 95 respectively). For `matrix`, it was reported that a majority of the accesses were not being fetched from the L1 cache and that this data structure did not use unit strides. For the other data structure (`CDF`), we saw the NUMA hit ratio to be about 50% (indicating that remote fetches probably contributed to the latency). The code in question has all threads performing a linear search over a single large sorted array. To make each core operate on separate portions of the array without significant refactoring of the code, the code was changed to keep track of the search index from the previous iteration. The resulting analysis (Figure 2) showed that cycles per accesses for `matrix` dropped to 13 while the number of accesses to `CDF` dropped to an insignificant number. On large inputs (40K particles), this resulted in the entire program taking about a quarter of its original running time.

B. Needleman-Wunsch

This code is a parallel implementation of a dynamic programing solution. In this code, iterations (size of the diagonal) are distributed equally among the cores by the OpenMP runtime. Memory analysis showed that access to the `input_itemsets` array was becoming a problem (reuse distance as high as 13 cache lines, i.e. about 832 bytes). Considering that all threads access the same array in a loop, one can infer that the reuse distance is affected by simultaneous use by multiple cores. To reduce this distance we focused on increasing the affinity between each core and its data. With the objective of ensuring that shuffling of data among cores does not occur too often, we changed the OpenMP chunk size of the iteration distribution to 32. This results in at least 32 consecutive iterations being assigned to a single core, thus increasing reuse within a single cache. The optimized code shows about 4% improvement and the reuse distance for the data structure is decreased from 13 cache lines to 9 cache lines.

C. Back Propagation

This machine learning code is used to determine weights in a layered neural network. Trace analysis on this code showed high cycles per access (64) and relatively high reuse distances (5 cache lines) for two data arrays (`w`, `oldw`). Inspection of the code showed that the arrays were being accessed in a strided manner. A simple reordering of loops reduced the cycles per access to 5, reuse distance to 2 cache lines and the code showed 17% improvement in total time.

D. SRAD (Speckle Reducing Anisotropic Diffusion)

SRAD uses Partial Difference Equations for ultrasound imaging. For this code, we observed that the `image` array had a high reuse distance (21 cache lines) and a high access latency (20 cycles). Moreover the stride analysis showed that this array was accessed with an average stride of 32 cache lines. From the line number information produced from the analysis, we could verify that the loops were inverted which resulted in strided accesses of the data structures. Changing the loop order reduced the reuse distance to 1 cache line and access penalty to 13 cycles, ultimately reducing the total time by 38%,

V. Conclusions and Future Work

We have demonstrated that analyzing memory traces can reveal a wide variety of opportunities for optimizing multi-core programs. More specifically, relating information from these analyses to program source code can be helpful in quick diagnosis of the problems and their resolution. Including architecture information such as cache sizes and their sharing makes it possible to diagnose situations such as thrashing, thus making it possible to model reuse distances and access penalties for data structures more accurately.

For novice programmers, selecting the correct tile size is often a process of trial and error. We plan to improve our tool is by computing blocking factors for regular loops, thus eliminating the ad-hoc tile size selection process. Another direction we are pursuing is to use these memory analyses, in conjunction with profilers like PerfExpert, to determine code segments that can benefit from mapping to GPUs. Specifically, we intend to incorporate sophisticated data flow analysis to recommend code transformations that result in fewer data movements, thus contributing to improved memory behavior.

Acknowledgments

This project is funded in part by the National Science Foundation under OCI award #0622780. The authors would like to thank Andrew Lenharth for many useful inputs on using the LLVM compiler infrastructure in our work.

variable: CDF (**avg_dist**: 3.85, **count**: 524174, **avg_cpa**: 78.53, **numa_hit_ratio**: 52.38)

dist	cpa	use ln#	reuse ln#	count	file:
0	150	0	290	17010	ex_particle_OPENMP_seq.c
1	250	290	290	129964	ex_particle_OPENMP_seq.c
1	4	290	290	328165	ex_particle_OPENMP_seq.c

variable: matrix (**avg_dist**: 0.37, **count**: 50996, **avg_cpa**: 95.25, **numa_hit_ratio**: 100.00)

dist	cpa	use ln#	reuse ln#	count	file:
1	4	185	185	19122	ex_particle_OPENMP_seq.c
0	150	0	185	31874	ex_particle_OPENMP_seq.c

Figure 1: Excerpt of trace analysis output for naïve version of Particle Filter code

variable: matrix (**avg_dist**: 0.94, **count**: 50996, **avg_cpa**: 13.13, **numa_hit_ratio**: 100.00)

dist	cpa	use ln#	reuse ln#	count	file:
0	150	0	185	3189	ex_particle_OPENMP_seq.c
1	4	185	185	47807	ex_particle_OPENMP_seq.c

Figure 2: Excerpt of trace analysis output for optimized version of Particle Filter code

REFERENCES

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

[2] Y. Jiang, E. Zhang, K. Tian, and X. Shen, "Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?" in *Compiler Construction*. Springer, 2010, pp. 264–282.

[3] D. L. Schuff, B. S. Parsons, and V. S. Pai, "Multicore-aware reuse distance analysis," *Measurement*, p. 8 pp., 2010.

[4] D. L. Schuff, M. Kulkarni, V. S. Pai, and W. Lafayette, "Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization," *Measurement*, pp. 53–63, 2010.

[5] J. Weinberg and A. Snavely, "Chameleon: A framework for observing, understanding, and imitating the memory behavior of applications," in *PARA08: Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway*. Citeseer, 2008.

[6] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: a binary instrumentation tool for computer architecture research and education," 2004.

[7] S. Vlaovic and E. S. Davidson, "TAXI: Trace Analysis for X86 Interpretation," Ph.D. dissertation, University of Michigan, 2002.

[8] K. Lawton, "Bochs IA-32 Emulator Project," 2004.

[9] "ThreadSpotter," http://www.roguewave.com/.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization 2004 CGO 2004*, no. c, pp. 75–86, 2004.

[11] K. Beyls and E. DHollander, "Discovery of locality-improving refactorings by reuse path analysis," *High Performance Computing and Communications*, pp. 220–229, 2006.

[12] K. Beyls and E. H. D'Hollander, "Refactoring for Data Locality," *Computer*, vol. 42, no. 2, pp. 62–71, 2009.

[13] J. Diamond, M. Burtscher, J. McCalpin, B. Kim, S. Keckler, and J. Browne, "Evaluation and optimization of multicore performance bottlenecks in supercomputing applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 32–43.

[14] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation Practice and Experience*, vol. 22, no. 6, pp. n/a–n/a, 2009.

[15] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[16] M. Burtscher, B. D. Kim, J. Diamond, J. Mccalpin, L. Koesterke, and J. Browne, "PerfExpert : An Easy-to-Use Performance Diagnosis Tool for HPC Applications," in *Computer*. IEEE, 2010, pp. 1–11.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *2009 IEEE International Symposium on Workload Characterization IISWC*, vol. 2009, no. c, pp. 44–54, 2009.

[18] O. A. Sopeju, M. Burtscher, A. Rane, and J. Browne, "AutoSCOPE : Automatic Suggestions for Code Optimizations using PerfExpert," *Evaluation*.