Understanding Data Access Patterns Using Object-Differentiated Memory Profiling

Antonio J. Peña
Argonne National Laboratory
Email: apenya@anl.gov

Pavan Balaji
Argonne National Laboratory
Email: balaji@anl.gov

***Abstract*—The information provided by commonly used code-oriented profilers can be complemented by means of data-oriented profiling techniques. Based on a data-oriented approach, in this study we leverage techniques developed in previous papers to analyze the data access characteristics of a range of U.S. Department of Energy applications representative of different application domains. By analyzing object-differentiated memory access profiles, we identify markedly different access patterns across application stages. We find read-only and read-write periods, relatively large periods without accessing particular objects, and a variety of data access rates. This information is useful for devising software optimizations, for software and hardware codesign, and for data distribution and partitioning in heterogeneous memory systems.**

I. Introduction

As processor performance increases faster than memory bandwidth, memory usage optimizations are being widely studied and deployed. Despite the multiple cache hierarchies leveraged in today's computers, processors are often stalled for many cycles waiting for out-of-die data accesses. Clearly, the way applications access memory affects their performance.

Data-oriented object-differentiated profiles may provide application developers with useful additional information that traditional code-oriented profilers do not offer. For example, as illustrated in Figure 1, a single memory object may be the culprit for an overall high cache miss rate while not being visible when looking at the per-source-line statistics offered by traditional approaches. Another example of use is on data distribution and partitioning tasks for heterogeneous memory systems. Emerging extreme-scale computers are likely to be equipped with multiple memory subsystems based on a variety of technologies, such as NVRAM, scratchpad memory, vector-optimized memory, or different types of DDR. Identifying per-object access patterns helps determine the most appropriate memory technology to host them [1].

In this study we leverage techniques developed in previous papers [1], [2] to analyze the data access characteristics of a set of DOE applications from the CESAR [3] and Mantevo [4] codesign projects. We employ a data-oriented profiling tool [2] developed on top of the Valgrind instrumentation framework. Analyzing the information gathered with this profiling tool, we are able to identify per-memory-object access patterns that are markedly different across application stages, finding large periods without accesses, read-only and read-write periods, and widely different access rates. Our analyses and methodology are intended to help application developers better adapt their code to the underlying memory architecture and to be useful in the codesign of emerging hardware and software.

```
a[i] = b[j] * c[k]; ← 5%
b[l] = d[m] * 2;    ← 5%
c[n] += b[o];       ← 5%
```

```
a ← 0% / b ← 15% / c ← 0%
```

Fig. 1. Code-oriented (left) versus data-oriented (right) profiling.

II. Access Pattern Analysis

For our analysis we have selected three miniapplications representative of different domains. We discuss access patterns of single MPI processes.

A. MiniMD

MiniMD [5] is a cut-down molecular dynamics simulator. It simulates the atoms of a cubic space, splitting it among the different processes. We use the reference implementation of MiniMD version 1.2.

Before the simulation starts, the application sets up the initial status of the system. Next, the simulation is run, iteratively computing the forces, positions, and velocities of the atoms in successive timesteps. Every n number of timesteps (defined by the user), a reneighboring is performed, in order to recompute each atom's neighbors.

We ran a simulation of 10 timesteps of the Lennard-Jones interaction of 864,000 atoms in 27 single-threaded MPI processes (this is 32,000 atoms per process) to form a $3 \times 3 \times 3$ periodic mesh, so that every process has different left and right neighbors, in order to replicate production conditions for 3D-box stencil communications. We set the reneighboring frequency to 5 timesteps.

Table I shows the amount of data read and written for the most-accessed memory objects in our simulation: the atoms' positions (`atom.x`) and forces (`atom.f`), and a copy of the atoms' positions (`atom.x_c`), used alternately after each out-of-place sorting of the atoms. The other two most-accessed memory objects, `neighbor` and `force`, are instances of small C++ objects of fixed size (i.e., independent of the problem size) easily fitting in cache. Hence, these should not represent any performance penalty regardless of the underlying memory architecture on which they reside, so we will not consider them in our subsequent analysis. The remaining

TABLE I
DATA ACCESS STATISTICS FOR THE DIFFERENT MINIAPPLICATIONS

Object	Max. Size (Var./Fix.)	Loads		Stores	
		Abs.	Rel.	Absol.	Rel.
MiniMD					
atom.x	1.4 MB (V)	624 MB	24.1%	9 MB	2.5%
atom.f	1.4 MB (V)	272 MB	10.5%	266 MB	74.9%
neighbor	232 B (F)	464 MB	17.9%	20 KB	0.0%
atom.x_c	1.4 MB (V)	359 MB	13.8%	7 MB	2.0%
force	120 B (F)	292 MB	11.3%	576 B	0.0%
MCCK					
ptrs	64 B (F)	5.8 GB	25.9%	4.1 GB	34.4%
particle	69 MB (V)	2.5 GB	11.1%	0.7 GB	5.6%
p_match	8 B (F)	0.8 GB	3.6%	0.7 GB	6.0%
mc*	8 B (F)	0.9 GB	4.1%	8 B	0.0%
mc	176 B (F)	0.5 GB	2.1%	1 KB	0.0%
XSBench					
xs_vector	80 B (F)	177 MB	19.2%	163 MB	49.5%
NGP	319 KB (V)	244 MB	26.4%	1 MB	0.3%

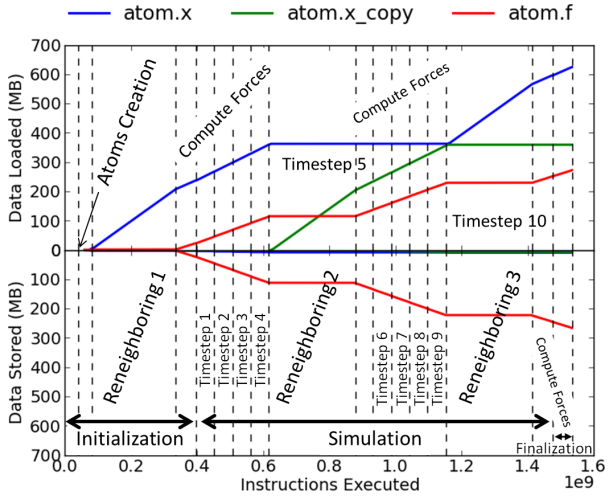

Fig. 2. MiniMD's cumulative data accessed for our three objects of interest.

accesses are spread over different objects, being negligible for the overall execution.

Figure 2 shows the access patterns of our objects of interest, covering almost 50% of all the read accesses of the execution and almost 80% of the overall writes. We can observe that `atom.x` and `atom.x_c` are mostly read variables, whereas `atom.f` is both read and written, although it does not present periods of being mostly read or mostly written. All objects experience large periods of not being accessed, comprising over 200 million instructions. In addition, we observe only two data-intensive stages, *reneighboring* and *compute forces*, which we describe next.

Reneighboring: As we can see in Table II, reneighboring is a relatively long operation reading over 200 MB from the atoms' positions. This corresponds to the process of building the atoms' neighbors list following Newton's third law. After separating the atoms in bins by position—a task requiring only a pass through all the atoms of the process reading their positions—the per-atom neighbor list is built. This process involves checking for every atom the position of the atoms

TABLE II
ACCESSES PER TASK FOR MINIMD

Object		Reneighboring		Compute Forces	
atom.x (_c)	L	202 MB	0.8 B/ins	29 MB	0.6 B/ins
	S	0 B	0.0 B/ins	0 B	0.0 B/ins
atom.f	L	0 B	0.0 B/ins	20 MB	0.4 B/ins
	S	0 B	0.0 B/ins	22 MB	0.4 B/ins
Instructions		$250 \cdot 10^6$		$53 \cdot 10^6$	

within its bin and those in its nearest bins according to the user-specified cutoff distance, which involves a relatively large number of iterations. In our setup we experience up to 14 atoms per bin and 41 neighbor bins, which is up to 574 iterations per atom reading atoms' double-precision 3D positions, while only a few of these atoms are found to be neighbors (39 on average in our test case). Hence, only a few 4-byte writes to the neighbors list are performed, giving a ratio of around 50 bytes read per each written. Although larger cutoff distances or higher densities may increase the number of neighbors per atom, the total number of atoms is expected to be several orders of magnitude larger than the number of neighbors per atom in production runs, and hence the neighbors list is likely to remain featuring a small number of accesses.

Computing of Forces: Computing of forces requires two passes through the atoms. In the first pass the forces are cleared. In the second pass these are iteratively computed. This process leads to a ratio of around 1.4 bytes loaded from `atom.x` per load and store of `atom.f` in our runs, as detailed in Table II. The slightly higher writes reflected on `atom.f` correspond to the initial zero-out process.

B. MCCK

MCCK [6] is focused on evaluating the memory implications on communications of the particle-tracking algorithm leveraged by many nuclear reactor physics analysis codes.

MCCK runs *stages* repeatedly until all particles have been absorbed by a physical domain. The leakage rate is user-specified, and at each stage the nonabsorbed particles move to a new processor's domain.

In our experiments we use MCCK version 1.0, leveraging the original communications approach. We ran 8 MPI processes with 1 million particles per processor and a global leakage of 0.8, leading to 66 stages before all the atoms were absorbed.

Table I depicts the data accesses for the most-accessed memory objects in our run: the particles' data (`particle`); four particle pointers used in two comparison functions (`ptrs`), two integer variables also used in the comparison functions (`p_match`), the pointer to a structure containing the configuration parameters (`mc*`), and the structure itself (`mc`). Of these, only the size of the `particle` object depends on the problem size; the remaining are accessed mostly by a high number of calls to particle comparison functions during sorting tasks. The next most-accessed memory object represents only 0.4% of the overall accesses. We focus the remainder of our

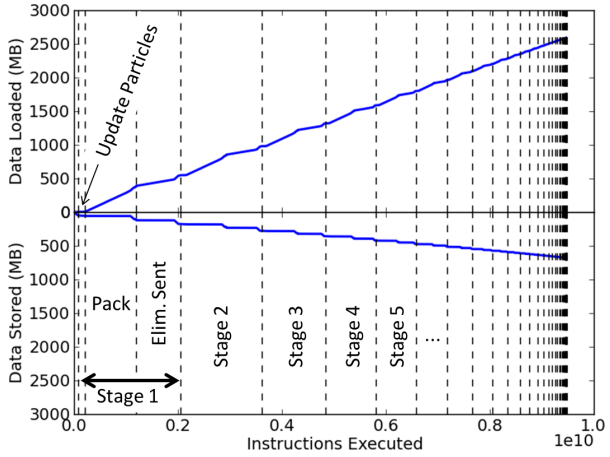

Fig. 3. MCCK access pattern for the main data memory object.

analysis on `particle`, which represents 9.3% of the overall data accesses of our simulation.

Figure 3 shows the access pattern for the `particle` object along the execution of the simulation. We can see that this object intermittently presents relatively large periods without store accesses at the beginning (up to around 10^9 instructions), becoming shorter and shorter as the execution progresses. It also presents relatively short intermittent periods of not being accessed at all. Overall, this object is read and written during the entire execution, with a ratio of 3.8 of data loaded to data stored. As can be seen in the plot, each successive stage is shorter because approximately 20% fewer atoms remain. Next, we briefly discuss the four main tasks of each cycle, summarized in Table III.

Update Particles: Iterate through the nonabsorbed particles to determine which of them are going to be absorbed in the current stage, leave, or stay within the domain. This task involves write accesses to the absorbed flag and optionally the process identifier if the particle is not absorbed, leading to a write-only stage with a reduced store-per-instruction rate.

Pack: Remove the absorbed particles from the array, and compress it. This task translates into an array sorting, placing first the nonabsorbed particles and sorting all of them by the new process to hold them. The sorting is performed by means of the GNU C library's "qsort," whose current implementation is actually a merge-sort algorithm that employs auxiliary buffers for the intermediate sorting stages and hence performs only reads from the original buffer until reaching the last stage, where both reads and writes are required to merge the two last subarrays. A final traversal of the nonabsorbed particles is performed in order to count them. This leads to an overall ratio of 8 bytes read per each written.

Exchange: Exchange leaving and arriving particles. This task involves $O(nonabsorbed)$ accesses to the particles array. Since most of this task consists of data transfers, it is a short task in terms of instructions, comprising a high ratio of read and write accesses.

Eliminate Sent: Eliminate the sent particles from the array. This task is performed by another sort operation with a comparison function that ignores the absorbed status. It requires a final pass through the particles in order to count them. In this case, the overall ratio of bytes read per each written is only 3, because fewer fields need to be considered when comparing elements if compared with the sorting procedure in *Pack*.

TABLE III
ACCESSES PER TASK FOR MCCK AND XSBENCH

Action	Loads		Stores		Instruct.
MCCK's `particle` object (1st stage)					
`Update`	0 MB	0.0 B/ins	7 MB	0.1 B/ins	$129 \cdot 10^6$
`Pack`	366 MB	0.4 B/ins	46 MB	0.0 B/ins	$968 \cdot 10^6$
`Exchange`	21 MB	0.9 B/ins	18 MB	0.8 B/ins	$25 \cdot 10^6$
`Elimin.`	158 MB	0.2 B/ins	55 MB	0.1 B/ins	$848 \cdot 10^6$
XSBench's `NGP` object					
`Initial`	1 MB	0.2 B/ins	1 MB	0.1 B/ins	$6 \cdot 10^6$
`Generate`	31 MB	0.4 B/ins	0 MB	0.0 B/ins	$89 \cdot 10^6$
`Simulat.`	212 MB	0.8 B/ins	0 MB	0.0 B/ins	$267 \cdot 10^6$

C. XSBench

XSBench [7] targets the typically most computationally intensive piece of Monte Carlo transport algorithms: the computation of macroscopic neutron cross-sections. According to its authors, in a typical simulation this poses around 85% of the total run time.

The input of XSBench includes the number of nuclides to simulate, the number of grid points per nuclide, and the number of lookups of cross-section data to perform.

We run our analysis on XSBench version 11 performing 150,000 lookups on a reactor of 68 nuclides with 100 grid points per nuclide, using eight MPI processes.

Table I shows the two most-accessed memory objects in our execution, representing over 45% of the accesses. The most accessed object—`xs_vector`—is an array of five double-sized floating-point elements used to store the vector of the cross section. `NGP` is an array of data structures comprising six double-sized floating-point elements representing the energy and the cross-section attributes of each nuclide grid point; it holds the matrix of nuclide grid points. The next most-accessed object represents only 3.6% of the total data accesses of our simulation. As in the rest of the cases, we drop the fixed-size memory object (`xs_vector`) from the rest of our analysis.

Figure 4 depicts the single-process access pattern of the `NGP` memory object. As we can see in the plot, this object is used extensively on read accesses, while no write accesses are performed during most of the execution. This is a good example of a variable that could benefit from its placement in a memory technology optimized for read-only accesses. In addition, we can observe that its access rate varies markedly between the two main stages of the simulation: the generation of the unionized energy grid and the lookup iterations.

We next describe the main stages of this miniapplication, whose access data is summarized in Table III.

Initial Tasks: The two major initial tasks comprise the generation of the nuclide energy grids and their subsequent sorting. These do not involve a large number of instructions executed nor a high rate of data accesses.

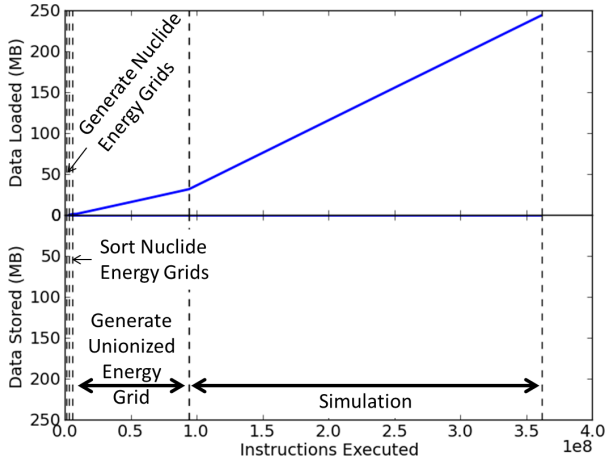

Fig. 4. XSBench access pattern for the most-accessed memory objects.

Generation of Unionized Energy Grid: This stage comprises the allocation of the unionized energy grid and the assignment of its relationship with the energy levels of the nuclide grids. After allocating the energy grid, a sorted nuclide grid point matrix is created, the original nuclide grid is copied into it, and the energies are subsequently sorted by means of the "qsort" interface. Next, the energies from the sorted grid of nuclides are assigned to the energy grid. Last, the energy grids are related to the nuclide energy grids (by means of assigning pointers to them). This last task comprises the major part of this operation, involving $O(gridpoints \times isotopes^2)$ binary searches. The overall stage leads to an average of 0.4 bytes read from the `NGP` memory object per instruction executed.

Simulation: The simulation stage consists of performing the requested number of cross-section lookups. A single lookup is a negligible operation in terms of both instructions executed and data accessed (fewer than 2,000 instructions and 1.5 KB per lookup), since it involves generating a pair of random numbers to represent the energy and the material, as well as a binary search on the unionized energy grid looking for that energy level. Next, for every input material (12 by default in XSBench) a lookup (of asymptotic constant time) is performed in the nuclide grid. This leads to an asymptotic logarithmic time. Since the nuclide grid is extensively accessed, the ratio of accesses per instruction doubles that of the previous stage.

III. Related Work

The potential benefits of object-differentiated profiling were presented by optimization case studies based on the triangular system solve phase of the incomplete Cholesky conjugate gradient algorithm [8] and blocked-matrix multiply code [9]. The recently emerged Gleipnir memory analysis tool—which includes object-differentiated capabilities—was presented along with some examples of its use in optimizing data access performance [10]. A performance study of the MCF benchmark from the SPEC CPU 2000 benchmark suite including object-differentiated profiling data based on hardware counters was presented for the UltraSPARC-III family of processors by extending the functionality of the Sun ONE Studio compilers and performance tools [11].

Our work differs from those previous efforts in that we present and extensively analyze *access patterns* of kernels from different domains of wide interest to the high-performance computing community.

IV. Conclusions

The different memory objects leveraged by the codes we analyzed present markedly different access patterns among the different execution stages. For instance, we have seen that the object storing the atoms' forces in MiniMD is not accessed during the reneighboring stage for 250 million instructions, while being actively accessed for both reading and writing during the computation of the forces. In the case of MCCK, the object storing the particles information is accessed mostly by an out-of-place sorting algorithm that performs only writes to this buffer during the last merge stage. On the other hand, the memory object of interest in our XSBench execution revealed a read-only access pattern.

Memory usage and bandwidth requirements are a major concern of system designers. Using miniapplications for analyzing access patterns of individual memory objects helps researchers and developers understand the way applications access data and provides insight into unexpected behaviors.

Acknowledgments

This work was supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

References

[1] A. J. Peña and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *IEEE Cluster*, Sep. 2014.

[2] ——, "A framework for tracking memory accesses in scientific applications," in *43nd International Conference on Parallel Processing Workshops (ICPP-W)*, Sep. 2014.

[3] Argonne National Laboratory, "Center for Exascale Simulation of Advanced Reactors," http://cesar.anl.gov, 2015.

[4] Sandia National Laboratories, "Home of the Mantevo project," http://mantevo.org, 2015.

[5] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., 2009.

[6] K. Felker, A. R. Siegel, and S. F. Siegel, "Optimizing memory constrained environments in Monte Carlo nuclear reactor simulations," *International Journal of High Performance Computing Applications*, vol. 27, no. 2, pp. 210–216, 2013.

[7] J. R. Tramm and A. R. Siegel, "XSBench — the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR — The Role of Reactor Physics toward a Sustainable Future*, 2014.

[8] M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing memory system bottlenecks in programs," *ACM SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1, pp. 1–12, 1992.

[9] M. Martonosi, A. Gupta, and T. E. Anderson, "Tuning memory performance of sequential and parallel programs," *Computer*, vol. 28, no. 4, pp. 32–40, 1995.

[10] T. Janjusic, K. Kavi, and B. Potter, "Gleipnir: A memory analysis tool," in *International Conference on Computational Science (ICCS)*, 2011.

[11] M. Itzkowitz, B. J. Wylie, C. Aoki, and N. Kosche, "Memory profiling using hardware counters," in *Supercomputing Conference (SC)*, 2003.