Unimem: Runtime Data Management on NVM-based Heterogeneous Main Memory for High Performance Computing

Kai Wu
kwu42@ucmerced.edu
University of California, Merced

Dong Li
dli35@ucmerced.edu
University of California, Merced

1 INTRODUCTION

Non-volatile memory (NVM) is a promising technique to build future high performance computing (HPC) systems. NVM can provide a scalable and power-efficient solution as main memory, alternative to DRAM. However, comparing with DRAM, NVM as main memory can be challenging, since there is a big performance gap between NVM-based and traditional DRAM-based systems for HPC applications. Our initial performance evaluation with representative numerial kernels shows that there is 1.09x-8.4x slowdown on NVM-only systems, when NVM is configured with 1/2-1/8 of DRAM bandwith or 2x-8x DRAM latency. Our results are different from those of the existing work that claims that inferior performance of NVM has limited impact on the performance of *HPC applications*. To make NVM feasible for future HPC, NVM must be paired with a fraction of DRAM to form a heterogeneous memory system (HMS). By selectively placing frequently accessed data in DRAM, we are able to exploit cost and scaling benefits of NVM while minimizing the limitation of NVM with DRAM.

To selectively place data in DRAM, it is important to characterize memory access patterns associated with data objects. Figure 1 summarizes the results of our characterization study on a numerical kernel SP. The results reveal that some data objects (e.g., in_buffer+out_buffer), after being moved from NVM with less memory bandwidth to DRAM, there is big performance improvement. However, we do not have such improvement after moving them from NVM with longer access latency to DRAM. We claim such data objects are sensitive to memory bandwidth. Similarly, we find some data object which is only sensitive to memory latency (e.g., lhs), or sensitive to both bandwidth and latency (e.g., rhs).

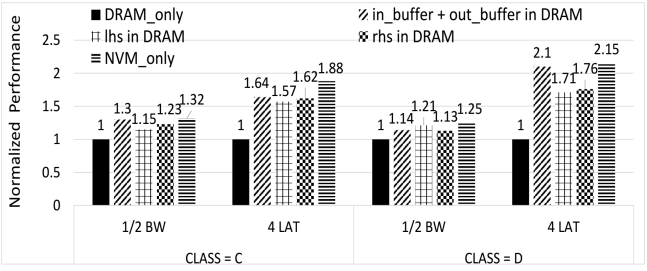

Figure 1: The impact of data placement on performance (execution time) of NVM-based main memory. The performance is normalized to DRAM-only systems. The legend entries "in_buffer+out_buffer", "lhs", and "rhs" are the data objects placed in DRAM in the DRAM+NVM system. The x axis shows the configuration of NVM (4x DRAM latency or 1/2 DRAM bandwidth).

*The full paper of this abstract was published on the 29th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC) https://doi.org/10.1145/3126908.3126923

Based on the above performance study, we introduce a software-based solution, named "Unimem", to automatically and transparently decide and implement data placement on NVM-based HMS. Different from some existing work, Unimem is a lightweight runtime system to manage data placement for *MPI programs*, without hardware modifications and disruptive changes to applications and system software. Also, Unimem employs online profiling based on performance counters to capture memory access patterns for application execution phases, based on which we characterize the sensitivity of data objects to memory bandwidth and latency. Furthermore, Unimeme introduces lightweight performance models, based on which we predict performance benefit and cost if moving data objects between NVM and DRAM.

Our performance evaluation results show that using Unimem, the performance difference between DRAM-only and HMS is only 6.2% on average (16% at most) for six common numerial kernels and a large HPC production code. Unimem greatly narrows the performance gap between NVM- and DRAM-based systems, and demonstrates better performance than a state-of-the-art software-based solution.

2 UNIMEM DESIGN

We target on MPI-based HPC applications with an iterative structure (i.e., typically such application has a main computation loop). For such an application, we decompose it into phases. A phase can be a computation phase delineated by MPI operations; A phase can also be an MPI communication phase.

2.1 Design

The workflow of Unimem includes three steps: phase profiling, performance modeling, and data placement decision and enforcement. The phase profiling happens in the first iteration of the main computation loop of the application. At the end of the first iteration, we build performance models and make data placement decision. After the first iteration, we enforce the data placement decision for each phase. We describe the three steps in details as follows.

2.1.1 Phase Profiling. This step collects memory access information for each phase. This information is leveraged by the second and third steps to decide data placement for each phase.

We rely on hardware performance counters widely deployed in modern processors. In particular, we collect the number of last level cache miss event, and then map the event information to data objects. Leveraging the common sampling mode in performance counters, we collect memory addresses whose associated memory references cause last level cache misses. Those memory addresses help us identify target data objects that have frequent memory accesses in main memory.

2.1.2 Performance Modeling. Performance modeling estimates performance benefit and data movement cost. We trigger data

movement only when the benefit outweighs the cost. To calculate the performance benefit, we must decide if the data object is bandwidth sensitive or latency sensitive.

Bandwidth sensitivity vs. latency sensitivity. To decide if a target data object in a phase is bandwidth sensitive or latency sensitive, we use main memory bandwidth consumption as an indicator.

Calculation of data movement benefit. To calculate the performance benefits (after data movement from NVM to DRAM) for data objects, we estimate the performance difference between running the application on NVM and on DRAM. For bandwidth sensitive data objects, the performance difference is estimated by memory access time in NVM minus memory access time in DRAM. For latency sensitive data objects, we follow the similar idea but consider the latency difference instead of bandwidth difference between NVM and DRAM.

Calculation of data movement cost. The data movement cost can be simply calculated as $\frac{data\_size}{mem\_copy\_bw}$. However, to reduce the data movement cost, we overlap data movement with application execution. In particular, we introduce a helper thread that runs in parallel with the application and triggers data movement before the application accesses data. This helper thread-based approach is an asynchronous data movement.

2.1.3 Data Placement Decision and Enforcement. Based on the above performance modeling to calculate benefit and cost of every possible data movement, we determine data placement by two strategies ("phase local search" and "cross-phase global search"). Phase local search determines data placement at the granularity of individual phases. This strategy leads to optimal data placement for each phase, but can result in frequent data movement across phases. Cross-phase global search treats all phases as a combined single phase: Once an optimal data placement is determined within the combination of all phases, there is no data movement within the combination. Hence, this strategy has less data movement than phase local search. Unimem compares the potential benefits of two strategies and choose the best one.

2.2 Optimization

To improve runtime performance, we introduce a couple of optimization techniques as follows.

Handling workload variation across iterations. In some HPC applications, the computation and memory access patterns do not remain stable across iterations of the main loop. To accommodate workload variation across iterations, Unimem monitors the performance of each phase after data movement. If there is obvious performance variation, then Unimem will activate phase profiling again and adjust the data placement decision.

Initial data placement. Even though we use the asynchronous data movement to overlap data movement with application execution, data movement can still be expensive, especially for large data objects. To reduce the data movement cost, we place some data objects with a large number of memory references into DRAM at the beginning of application execution. To calculate the number of memory reference for each data object, we employ compiler analysis.

Handling large data objects. We move data between DRAM and NVM at the granularity of data object. This means that a data object larger than the DRAM space cannot be migrated. A method to address this problem is to partition the large data object into multiple chunks, with each chunk smaller than the DRAM size. At runtime, we profile memory access for each chunk instead of the whole data object, and move data chunk if the benefit overweight the cost of data chunk movement. In Unimem, we use compiler techniques to partition one-dimensional arrays with regular memory references into chunks.

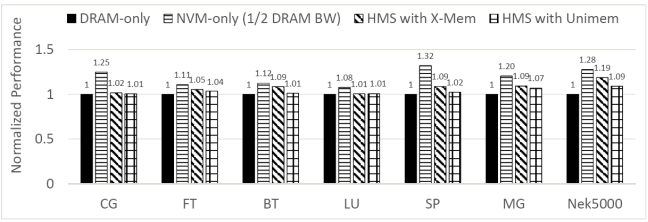

Figure 2: The performance (execution time) comparison between DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 1/2 DRAM bandwidth.

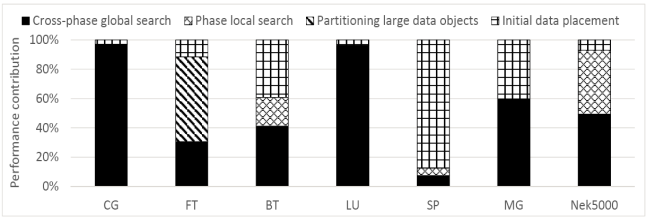

Figure 3: Quantifying the contributions of our four major techniques to performance improvement.

3 EVALUATION

We use Quartz emulator [2] to enable efficient emulation of a range of NVM latency and bandwidth characteristics. We compare the basic performance (execution time) of DRAM-only, NVM-only, HMS with X-Mem [1] (a state-of-art software solution for data manangement) and HMS with Unimem (See Figure 2). The result shows that Unimem greatly narrows the performance gap and makes performance of HMS very close to that of DRAM-only cases: the average performance difference between DRAM-only and HMS is only 3%. Also, Unimem performs similarly to X-Mem, but performs 10% better than X-Mem for Nek5000 (a production HPC code). We also quantify the contributions of our performance optimization techniques to performance improvement on HMS. Figure 3 shows that cross-phase global search can be very effective (contributing 90 % in CG and LU); Using phase local search can complement cross-phase global search (contributing 19% and 44% in BT and Nek5000 respectively); Initial data placement takes effect on all benchmarks (improving SP by 87%); Partitioning large data objects does not take effect except FT, because the partitioning of data objects introduces very frequent data movement which loses performance.

REFERENCES

[1] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[2] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Annual Middleware Conference (Middleware)*, 2015.