LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-583953

DI-MMAP: A High Performance Memory Map Runtime for Data-Intensive Applications

B. Van Essen, H. Hsieh, S. Ames, M. Gokhale

September 21, 2012

The International Workshop on Data-Intensive Scalable Computing Systems
Salt Lake City, UT, United States
November 16, 2012 through November 16, 2012

Disclaimer

DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications

Brian Van Essen† Henry Hsieh†‡ Sasha Ames† Maya Gokhale†
†*Center for Applied Scientific Computing*
Lawrence Livermore National Laboratory, Livermore, CA 94550
{*vanessen1, hsieh7, ames4, gokhale2*}*@llnl.gov*

‡*Department of Computer Science, University of California, Los Angeles*

***Abstract*—We present DI-MMAP, a high-performance runtime that memory-maps large external data sets into an application's address space and shows significantly better performance than the Linux `mmap` system call. Our implementation is particularly effective when used with high performance locally attached Flash arrays on highly concurrent, latency-tolerant data-intensive HPC applications. We describe the kernel module and show performance results on a benchmark test suite and on a new bioinformatics metagenomic classification application. For the complex metagenomics classification application, DI-MMAP performs up to 4.88× better than standard Linux `mmap`.**

***Keywords*-data-intensive; memory-map runtime; memory architecture; NVRAM;**

I. Introduction

Data-intensive applications form an increasingly important segment of high performance computing workloads. These applications process large external data sets and often require very large working sets that exceed main memory capacity, presenting new challenges for operating systems and runtimes. In this work, we target a data-intensive node architecture with direct I/O-bus-attached Non-Volatile RAM, such as attached Flash arrays today, and STT-RAM, PCM, or memristor in the future. These persistent memory technologies provide new opportunities for extending the memory hierarchy by supporting highly concurrent read and write operations that can be exploited by throughput driven (latency tolerant) algorithms such as parallel graph traversal [1].

In this work, we advocate a memory-mapping approach that maps low latency, random access storage into an application's address space, allowing the application to be oblivious to transitions from dynamic to persistent memory when accessing out-of-core data. However, we, along with many others, have observed that the memory-map runtime in Linux is not suited for memory-mapped out-of-core applications [2] and cannot efficiently support this model. Even with highly optimized massively concurrent algorithms and high bandwidth low latency storage, applications designed to interact with very large working sets in main memory incur significant performance loss if they read and write data structures mapped to external storage as if they were in main memory.

For this reason, most out-of-core algorithms use explicit I/O to load and store data between external store and application-managed data buffers. Optimizing an application for out-of-core execution is an exercise in carefully choreographing data movement, requiring explicit data requests through direct I/O and manual buffering.

The idea of memory-mapping data from storage into main memory is appealing for its simplicity. Additionally, it paves a path for scalable out-of-core computation because buffering and data movement are implicitly handled by the operating system's runtime rather than the application.

In prior work [2] we demonstrated that the standard memory-map runtime in Linux will rapidly lose performance as both concurrency increases and as memory within the system becomes constrained. At the time we speculated that these performance bottlenecks were due to (a) the overhead of dynamic page management, and (b) a page buffering scheme and eviction algorithm ill-suited to many data-intensive applications.

We have developed a new high-performance runtime that can seamlessly integrate NVRAM into the memory hierarchy using the memory-map runtime abstraction. Our new module, a data-intensive memory-map runtime (DI-MMAP) addresses the performance gap in standard Linux memory-map runtime. This paper demonstrates the effectiveness of DI-MMAP for data-intensive applications. We demonstrate that DI-MMAP can consistently achieve significant performance improvement over standard Linux `mmap` on our test suite, including an unstructured read/write access pattern, micro-benchmarks that demonstrate searching several types of data structure, and a bioinformatics application that searches a large (hundreds of GB) "in-memory" metagenomics database. Our memory-map runtime delivers up to 4.88× the performance of standard Linux `mmap` on the bioinformatics application and approaches the peak performance of raw, direct I/O on a random I/O benchmark.

II. The DI-MMAP runtime

The data-intensive memory-map runtime (DI-MMAP) is a high performance runtime that provides a custom memory-map fault handler and page buffering. It is a loadable Linux character device driver and it works outside of the standard Linux page caching system. It is derived from the PerMA simulator outlined in [2], sharing a common core codebase, and source code is available at [3]. It has been developed and tested for the 2.6.32 kernels in RHEL6.

The key features of the runtime are:

- a fixed size page buffer
- minimal dynamic memory allocation
- a simple FIFO buffer replacement policy
- preferential caching for frequently accessed pages

The combination of these features allows DI-MMAP to provide exceptional performance at high levels of concurrency compared to standard `mmap`, as shown in Section V. The DI-MMAP device driver is loaded into a running Linux kernel. As it is loaded, the device driver allocates a fixed amount of main memory for page buffering. Once the device driver is active, it creates a control interface file in the `/dev` filesystem. The control file is then used to create additional pseudo-files in the `/dev` filesystem that link (*i.e.* redirect) to block devices in the system. When a pseudo-file is accessed all requests are redirected to the linked block device.

DI-MMAP uses a simple FIFO buffering system with preferential storage of frequently accessed pages. Figure 1 shows a logical diagram of the DI-MMAP buffer and page management queues. The buffer contains enough pages to fill all of the queues plus one spare page. When a page fault occurs, the page location table is checked to see if another process (or thread) has already faulted the page into the buffer. If the page is in the buffer, the page is added to the page table of that process and the fault is completed. Otherwise a free page is allocated from a pool of empty pages. Data is then read from the block device into the fresh page, and the page is queued into the series of FIFOs.

In the steady state, a page fault will displace the oldest page in the primary FIFO. If the displaced page has been faulted more frequently than the buffer's average it will be placed into the hot page FIFO, otherwise it will be placed into the eviction queue. When an newly displaced page is inserted into the hot page FIFO, it will displace a formally hot page, which is then placed in the eviction queue. Once a page is in the eviction queue, it will eventually be flushed to storage if dirty, cleaned and returned to the free page list. One important aspect to maintaining performance is to properly manage TLB occupancy and eviction. Examples of the performance loss that can occur due to excessive TLB thrash have been noted by other research projects, such as Wu et al.'s [4] work on storage class memory. To address these problems, DI-MMAP removes pages from the page table of every process (it was mapped in to) as they are scheduled for eviction, but the translation look-aside buffers (TLBs) are flushed in bulk (only when an eviction buffer is full). Another optimization is page recovery. When a page fault occurs for a page that is in the eviction buffer, it is not flushed out. Instead, it is put into the primary FIFO, and a fault counter is incremented to indicate that it has some temporal locality (thus it might be a hot page).

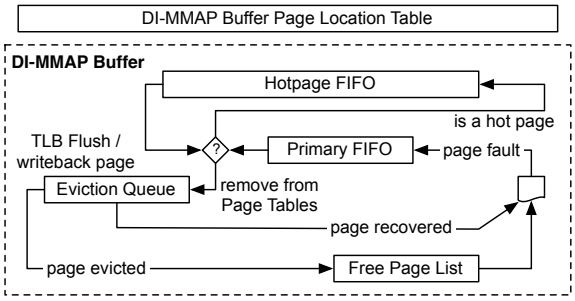

Figure 1. DI-MMAP page buffer

III. Related Work

Providing more control, and application specific-control, over memory page management is not a new idea. Previously, there were several research efforts focused on the virtual memory management system in the Mach 3.0 micro-kernel that have yet to be revisited for modern HPC operating systems. They studied the effects of different page eviction policies, application-specific pools of pages, and even application defined replacement policies. Examples include the HiPEC project by Lee et al. [5] and efforts by Park et al. [6]. Qureshi et al. [7] studied adaptive cache insertion policies, with an online voting mechanism, for CPU's.

All of these previous research projects have demonstrated that customized memory management and paging policies can dramatically improve a system's performance. They demonstrated that scalable performance is possible as applications shift from in-memory to out-of-core computations.

IV. Experimental Methodology

The DI-MMAP runtime is designed to provide high performance on highly-concurrent, data-intensive workloads. To test DI-MMAP we use three types of benchmarks: a synthetic random I/O workload, a small set of three microbenchmarks, and a metagenomics classification application. The synthetic random I/O workload was chosen because it is a good approximation for the unstructured access patterns found in many data-intensive applications. The micro-benchmarks are three commonly used data traversal and search algorithms. Finally, the metagenomics classification application is a new data-intensive bioinformatics application developed at LLNL to identify pathogens in samples containing an unknown variety of biological material.

The common approach to testing DI-MMAP was to load data onto a PCIe-attached Flash storage card, and have the

DI-MMAP runtime create a pseudo-device that linked to the raw Flash card. Each benchmark then memory-maps the DI-MMAP pseudo-device(s), enabling all page faults for the mapped address range to be serviced and buffered by the DI-MMAP runtime. These results are then compared to the existing Linux memory-map runtime and to direct (unbuffered) I/O as appropriate.

A. LRIOT

The Livermore Random I/O Testbench (LRIOT) is a synthetic benchmark that is designed to test I/O to high-performance storage devices. We have developed LRIOT to augment the industry standard FIO benchmark for testing high data rate memory-mapped I/O with different process/thread combinations. LRIOT can generate tests that combine multiple processes and multiple threads per process to simulate the highly concurrent access patterns of latency tolerant data-intensive applications. Furthermore LRIOT can generate uniform random I/O patterns that mimic the unstructured access patterns of algorithms such as breadth-first search graph analysis [2]. LRIOT can also do standard and direct I/O in addition to memory-mapped I/O, and thus provides a common testing framework. Finally, the LRIOT benchmark has been validated against the FIO benchmark and provides comparable results for direct I/O.

B. Micro-benchmarks

To complement the LRIOT experiments, we tested three micro-benchmarks that reproduce memory access patterns common to data-intensive applications. The micro-benchmarks are: binary search on a sorted vector, lookup on a ordered map structure that is implemented as a red-black tree, and lookup on an unordered map structure that is implemented as a hash map. The micro-benchmarks use the C++ STL and Boost library implementations of these algorithms.

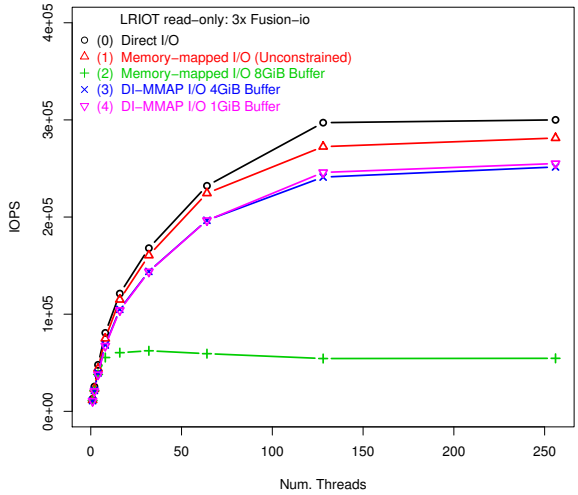

Figure 2. Read-only random I/O benchmark with uniform distribution.

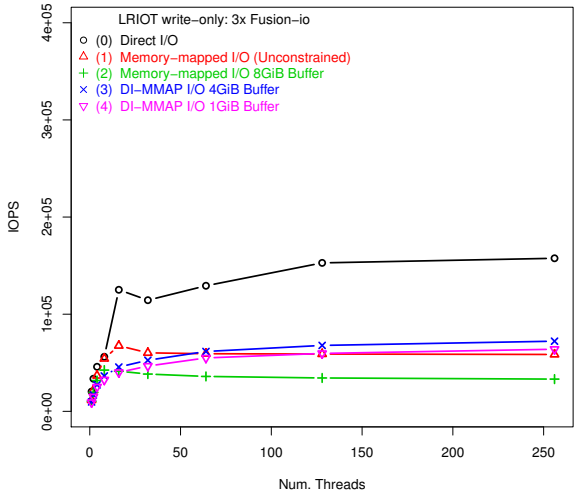

Figure 3. Write-only random I/O benchmark with uniform distribution.

C. Metagenomic Classification

Metagenomics involves the sequencing of heterogenous genetic fragments taken from the environment, in which the fragments (also called "reads") may be derived from many organisms. This application queries a database of genetic markers called k-mers, which are length k sequences out of a DNA, RNA, or protein alphabet. We place these large (hundreds of GiB) k-mer databases in Flash storage and memory-map the database files to access the indexed data sets. The access patterns to the datasets are extremely random.

We perform two types of experiments to evaluate DI-MMAP using this metagenomic database. First, we report the performance of a raw k-mer lookup benchmark. Second, we report the performance of the entire metagenomic classification application. In both scenarios, we compare the performance of standard Linux `mmap` of a file with DI-MMAP.[1] For the two experiments, we use the following input sets: first a synthetic metagenome derived from a human gut sample (HC1) and second, three real-world collections of metagenomic samples.

The metagenomic database contains k-mer markers referring to genomes from within a reference database (set of collected genomes) along with additional data associating the k-mer with a genome and the genome's position in the taxonomy tree of organisms. For our tests, $k = 18$, the k-mer is encoded in a 64-bit integer [8], and our database size is 635 GiB. Our implementation uses the *gnu* hash map with the k-mer as key, and pointers to the associated genomes and taxonomy information as value. A lookup retrieves the associated data, which can range from from a few hundred bytes to several thousand.

The metagenomic classification application uses k-mer

[1]We have also conducted tests comparing `mmap` of a raw device (without DI-MMAP) containing a database with one stored on a file system and have not observed any significant difference.

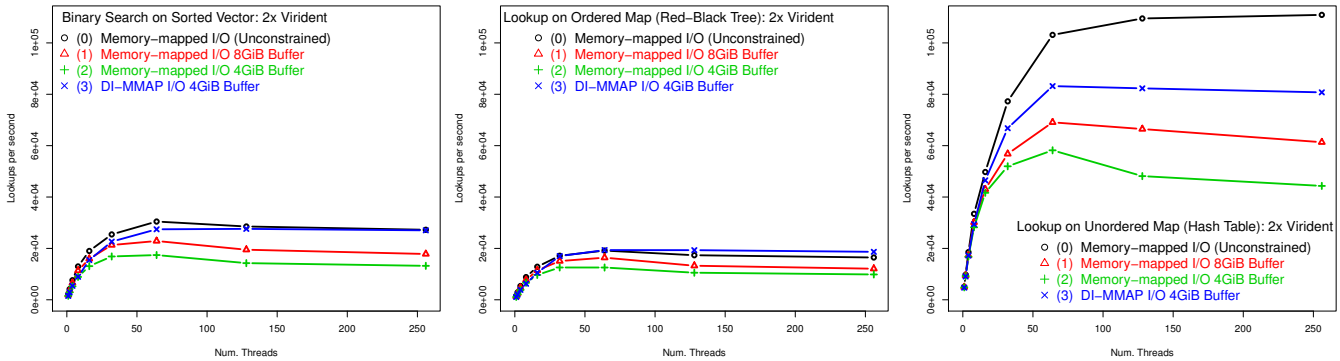

Figure 4. Micro-benchmarks: Binary Search on Sorted Vector, Lookup on an Ordered Map, and Lookup on an Unordered Map, respectively.

lookup as a frequent kernel operation. The application processes input files containing reads from the metagenomic samples. Once the application has queried the index using the extracted k-mers, it uses those results — the presence of particular taxonomic identifiers — to assign a label to each read. Input data is trivial to partition for processing in parallel; thus, many classification procedures are run concurrently using OpenMP threads.

V. Results

A. LRIOT: Uniform random I/O distribution

The first experiment compares the performance of DI-MMAP, standard `mmap`, and direct I/O. LRIOT generated a random sequence of 6.4 million read operations to a 128GB file that was striped across three 80 GiB SLC NAND Flash Fusion-io ioDrive PCIe 1.1 x4 in a RAID 0 configuration. The input read sequence is constructed so that it is repeatable, has one address per page, and is unique per process. Therefore each test will fetch 6.4 million unique pages, about 24 GiB of data. The data transfer size for all I/O (direct and memory-mapped) was 4KB pages. The host system was a 16 core AMD 8356 2.3GHz Opteron system with 64 GiB of DRAM and running RHEL 6 2.6.32.

Figure 2 shows the number of I/O per second (IOPs) that LRIOT achieved for the different I/O methods as concurrency increased. Note that each test used one process and the x-axis shows the number of concurrent threads. There are 5 specific test configurations shown here. The first line is for direct I/O and is typically the upper bound on achievable performance for a set of devices. The second and third lines are for the standard Linux memory-map handler when there is sufficient memory to hold all pages that are accessed in memory, *i.e.* `mmap` buffering is unconstrained, and when the page cache is constrained to hold only 8 GiB of pages. Finally, curves four and five are for DI-MMAP with a fixed buffer size of 4 GiB and 1 GiB, respectively. Figure 2 shows that the performance of DI-MMAP is very close to the performance of direct I/O and `mmap` when unconstrained, even with a very small buffer size of 1GiB. Furthermore, Figure 2 shows that standard Linux `mmap` performs well when memory is unconstrained, but performance drops significantly when system memory is constrained and the requested data exceeds the capacity of main memory. Overall, we see that DI-MMAP is able to deliver near peak performance with limited buffering resources, with only a 15% loss in IOPs due to overheads.

Figure 3 shows the number of IOPs that LRIOT is able to achieve with a similar write-only working set. As with the read-only working set, DI-MMAP offers a bit more than double the performance of Linux memory-map when it is constrained and similar performance to the unconstrained Linux mmap. However, unlike the read-only test the performance of DI-MMAP does not match that of direct I/O and is the subject of further investigation.

B. Micro-benchmarks

The three micro-benchmarks were all performed on an 8 core AMD 2378 2.4GHz Opteron system with 16 GiB of DRAM and two 200 GiB SLC NAND Flash Virident tachIOn Drive PCIe 1.1 x8. The database size for the vector and maps ranged from $\sim$ 112GiB to $\sim$ 135GiB and each micro-benchmark issued 2^{20} queries. For each of graphs in Figure 4 performance is measured in lookups per second and the x-axis is the number of concurrent threads. In each figure, line one is the performance of Linux `mmap` with unconstrained memory, lines two and three is the performance of Linux `mmap` with 8 and 4 GiB of available buffering (respectively), and line four is the performance of DI-MMAP with 4GiB of available buffering. These figures show that the performance of DI-MMAP significantly exceeds the performance of Linux `mmap` when each is constrained to an equal amount of buffering, and in some cases the performance with DI-MMAP is able to approach the performance of mmap with no memory constraints.

C. Metagenomics Search & Classification

We have performed these experiments on a 4 socket, Intel E7 4850 @ 2 GHz, running Linux kernel 2.6.32-279.5.2 (Red Hat Enterprise 6). For storage we use a software RAID over two Fusion-io 1.2 TB ioDrive2 cards, formatted with block sizes of 4 KiB, and 16 GiB DRAM available for buffer cache.

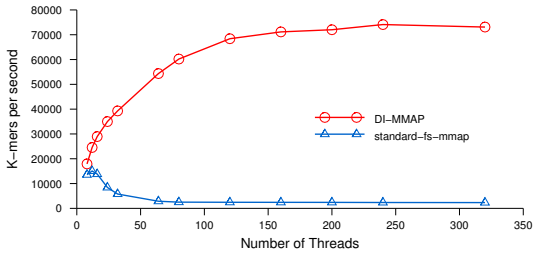

Figure 5. Performance of raw kmer lookup using k-mer identifiers extracted from the HC1 input set.

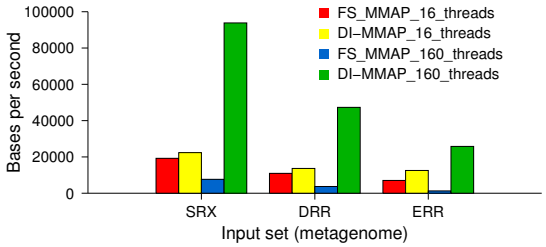

Figure 6. Performance of the metagenomic classification application using three real metagenomic input sets.

Figure 5 shows the performance of the raw k-mer lookup benchmark using the HC1 input set. The x-axis denotes increasing numbers of threads used for each trial and the y-axis shows k-mers per second. When using 8 threads, k-mer lookup performs better using DI-MMAP than standard `mmap` with a file system, and the performance gap increases with additional concurrency. Notably the performance with standard `mmap` peaks at 16 threads and then degrades. The peak performance for DI-MMAP with 240 threads is 4.92× better than the peak performance for standard `mmap` with 16 threads.

Figure 6 shows performance for metagenomic sample classification. We have selected thread counts that offer near peak performance, 16 and 160, for standard `mmap` and DI-MMAP, respectively. The difference in overall performance between the input sets varies due to percentages of redundant k-mers and the diversity of the metagenome. The peak performance of DI-MMAP vs standard `mmap` ranged from 4.88× for the SRX input set down to 3.66× for the ERR input set, which exhibited greater diversity, and thus, required additional computation for the classification.

VI. Conclusions

The goal of the data-intensive memory-map (DI-MMAP) runtime is to provide scalable, out-of-core performance for data-intensive applications. We show that the performance of algorithms using DI-MMAP scales up with increased concurrency, and does not significantly degrade with smaller memory footprints. As such, DI-MMAP provides a viable solution for scalable out-of-core algorithms. DI-MMAP offloads the explicit buffering requirements from the application to the runtime, allowing the application to access its external data through a simple load/store interface that hides much of the complexity of the data movement.

We demonstrate the performance of DI-MMAP over Linux's existing memory-map runtime with a simple random I/O workload, three micro-benchmarks, and a metagenomics classification application. Our results show that as the tests increase in complexity the performance of DI-MMAP can be 3.66× to 4.88× better than standard Linux `mmap` for the metagenomics sample classification application. Furthermore, the use of DI-MMAP alleviates the need to implement a custom, user-level buffer caching algorithm and infrastructure to achieve high performance.

Acknowledgments This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-583953). Funding partially provided by LDRD 11-ERD-008, LDRD 12-ERD-033, and the ASCR DAMASC project. The metagenomic classification algorithm was developed by Jonathan Allen, David Hysom, and Sasha Ames, all of LLNL. Portions of experiments were performed at the Livermore Computing facility resources, with special thanks to Dave Fox and Ramon Newton.

References

[1] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[2] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of NVRAM in data intensive HPC architectures: an evaluation," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012, pp. 703–714.

[3] "Data-centric Computing Architectures Research Group," https://computation.llnl.gov/casc/dcca-pub/dcca/Data-centric_architecture.html.

[4] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 39:1–39:11.

[5] C.-H. Lee, M. C. Chen, and R.-C. Chang, "HiPEC: high performance external virtual memory caching," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, ser. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994.

[6] Y. Park, R. Scott, and S. Sechrest, "Virtual memory versus file interface for large, memory-intensive scientific applications," in *Proc. ACM/IEEE Conf. Supercomputing*, 1996.

[7] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391.

[8] G. Marcais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, Jan. 2011.