Minimizing Flash I/O Traffic with Explicit I/Os for Efficient Out-of-Core Algorithms

Hiroko Midorikawa

midori@st.seikei.ac.jp,
Seikei University, JST-CREST, Tokyo, Japan

I. Introduction

Today, flash memory has established its position in deep memory hierarchy as a cost-effective, power-efficient, and large-capacity type of memory behind the DRAM layer. Moreover, it is becoming popular as an extension of main memory for out-of-core algorithms. There are several ways in which a flash device can be used for memory extension, e.g., as a swap device, with file-mmap, and with explicit I/O operations [2][3]. The most popular among these methods is the mmap method, where a flash SSD is used as a file system, and a file on the flash drive is mapped to the memory. Thus, mmap enables us to access a flash drive in the same manner as DRAM memory, in byte granularity, without any I/O maintenance between the main memory and flash device. The most important advantage of using mmap is that it requires little or no modification during application programming. The performance of applications using mmap depends on the degree of matching between the operating system (OS) kernel page replacement policy in the page cache and memory access patterns of the application. Unfortunately, an OS page replacement designed for general-purpose usage cannot be easily adapted for each application, even if the memory advice call, madvise(), is used. Hence, performance improvement is limited for applications because of such implicit and general methods of page cache control in mmap. As a result, application-aware tuning is gaining importance in performance–oriented fields.

On the other hand, the most explicit method of carrying out I/O operations to a flash device, namely asynchronous I/O (aio), requires application programs to be drastically restructured. That is memory-semantic accesses to the flash drive in mmap are replaced with explicit I/O operations. However, once the program is rewritten with an aio explicitly, its performance enhances. The amount of data transferred between the flash device and memory is reduced, unlike the redundant data transfer observed in OS-controlled page cache replacement in mmap. The aio method allows applications to maximally utilize the space on DRAM, because there is no need to reserve a page cache area as required by the mmap method. Moreover, users can design efficient I/O scheduling to suit their applications. The use of aio with appropriate I/O parameters allows I/O data traffic to be minimized, in turn maximizing the application performance. A major problem with this approach is that explicit I/O parameter tuning procedures are required when different system hardware settings and/or different application parameters are used. These hardware settings include capacity of each memory layer and number of cores and sockets, and different application parameters like the domain data size and number of time steps.

First, this presentation summarizes the performance of the aio and mmap methods and optimization techniques. Next, the newly developed algorithms, which eliminate the redundant computations of existing algorithms [1], are introduced. Finally, an automatic method is proposed that enables parameter tuning during runtime. The method minimizes the amount of data transferred between the flash device and DRAM, which is the most dominant factor affecting the performance of the out-of-core algorithms using flash. The use of explicit I/O operations to a flash device together with auto-tuning allows users to easily minimize the amount of I/O traffic for achieving maximum performance of different hardware and application settings.

II. MMAP vs. AIO: Implicit and Explicit Methods to Use Flash SSDs for Out-of-Core Algorithms

We have developed out-of-core stencil algorithms for flash SSDs using aio and mmap [1-3], by increasing data access locality using blocking techniques in spatial and temporal spaces as shown in Fig. 1.

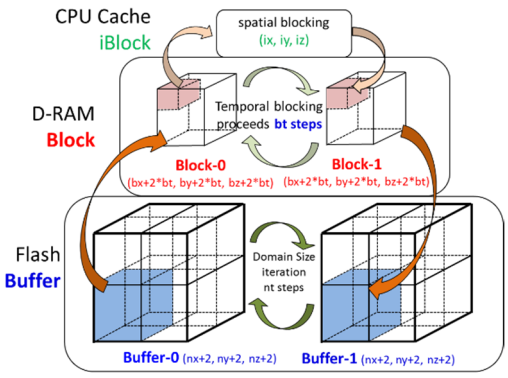

Fig. 1. Three data arrays in memory layers, *buffer arrays* in Flash SSD, *block arrays* in DRAM, and *i-block arrays* in the cache for locality extraction

Several optimization techniques for the algorithms were investigated, in terms of data memory layout and thread work-share scheme, as shown in Fig. 2. These optimizations reduced the execution times of the aio and mmap methods by 55% and 59%, respectively [3]. As a result, despite the large difference in access latency between DRAM and flash of about one thousand times, our algorithm was found capable of supporting the flash device as an extension of the main memory. For the aio method running on the 2-socket system equipped with a 64 GiB of DRAM and 1.2 TiB flash SSD, 7-point stencil computations of a 1 TiB problem (16 times larger than the DRAM amount) exhibited 80% of the performance for the 64 GiB problem (equal to the amount of DRAM) [3].

Fig. 3 shows the execution times of the aio and mmap methods for various size problems, i.e., ranging in 64 GiB-1 TiB. The execution time of the aio method is 50–60% of that of the mmap method. Moreover, during execution using the mmap method, the 19-point 1 TiB problem is terminated by an out-of-memory killer in the OS, because of the lack of memory availability in the large-size file mmap. In contrast, execution of the aio method exhibits a stable behavior. Fig. 4 compares the aio and mmap methods in terms of Mflops for various problem sizes on a system having a fixed DRAM size. The figure shows the problem size in terms of the *buffer array* size for the flash device and *block array* size for the DRAM. As the problem size increases, the DRAM capacity available for the page cache area decreases, thereby causing a larger performance difference between aio and mmap.

III. Temporal Blocking Stencil Algorithm without Redundant Computations for Flash Memories

A detailed description of the new algorithm is omitted here because of space limitations. The algorithm is based on locally sliding a spatial-block calculation window in *block arrays* on DRAM in temporal block time steps. Fig. 5 shows the relative execution times of several versions of the algorithm. The *mmap1*, i.e., the standard method designed for memory and cache systems without a flash device, does not have the *block arrays* shown in Fig. 1, where cache-DRAM maintenance is implicitly performed by hardware. When using *mmap1* as an out-of-core algorithm, its DRAM-Flash maintenance is implicitly carried out as page cache maintenance by the OS kernel for mmap. Fig. 5 compares *(1) mmap1+opt*, which is *mmap1* with our optimization technique shown in Fig. 2; *(2)* our new algorithms *mmap2*, which introduces *block arrays* on DRAM; and two versions of advanced algorithms *(3)mmap3y* (or *aio3y*) and *(4) mmap5y* (or *aio5y*), which minimize the total amount of data transferred to the flash device. The original algorithms, *mmap1* and *mmap1+opt*, are fast when a sufficient amount of memory (128 GiB) is available for use. However, their performance deteriorates when using 32-GiB DRAM, which is half the size of the problem (64 GiB). This situation is attributed to implicit page caching in mmap as a result of the flat data structure that only uses *buffer arrays* with mmap, thereby causing inefficient data transfer to the flash device. In comparison, the use of 2-layer algorithms to introduce intermediate *block arrays* in DRAM improves performance considerably when 32 GiB DRAM is used.

IV. Runtime Auto-Tuning to Fit Underlying Hardware

The retrieval of information regarding the underlying hardware during runtime and the use of this information to calculate the total amount of I/O data for a flash device, enables the selection of an optimal combination of spatial and temporal blocking sizes to suit the capacity of each memory layer (flash, DRAM, L3-cache, and L2-cache). This auto-tuning mechanism allows users to easily minimize the amount of I/O traffic and gain maximum performance for particular hardware and application settings.

References

[1] H. Midorikawa, et al. "An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations," Proc. of the 2014 International Conference on High Performance Computing and Simulation IEEE-HPCS2014, pp. 268-277, (2014.7)

[2] H. Midorikawa, "Using Flash SSDs as Main Memory Extension with a Locality-aware Algorithm," Non-Volatile Memories Workshop 2015

[3] H. Midorikawa, H. Tan "Locality-Aware Stencil Computations using Flash SSDs as Main Memory Extension," Proc. of IEEE/ACM International Symp. on Cluster, Cloud and the Grid Computing CCGrid2015, pp. 1163-1168, (2015.5)

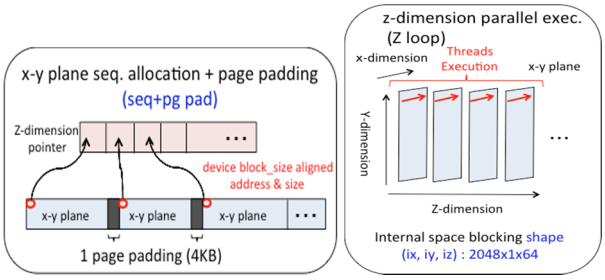

Fig. 2. *Block array* memory layout for block-aligned access of the aio method (left), and work-share among threads in an *i-block array* (right)

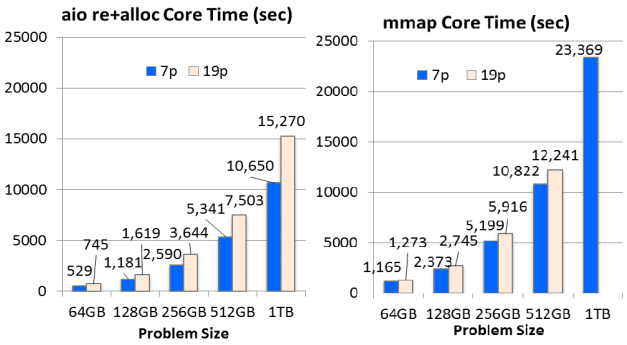

Fig. 3. Execution times required by the aio and mmap methods for problems of various sizes

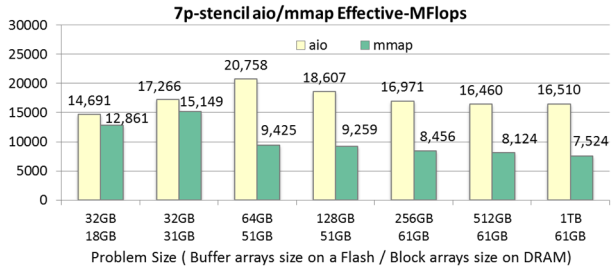

Fig. 4. Effective MFlops on fixed-size memory (64 GiB) in a 2-socket system for aio and mmap methods

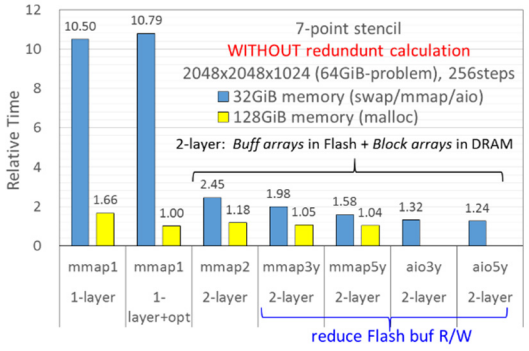

Fig. 5. Relative execution times of various algorithms with the aio and mmap methods