FINAL PROJECT REPORT

Project Title: An Optimizing Compiler for Petascale I/O on Leadership Class Architectures

PI: MahmutKandemir (Penn State); Co-PIs: AlokChoudhary (Northwestern); Rajeev Thakur (ANL)

Award Number: DE-SC0002156

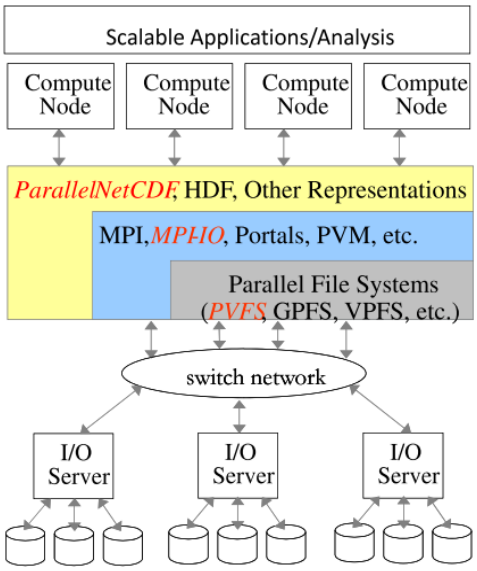

Figure 1: I/O stack involving interaction across many layers (applications, runtime, libraries, parallel and native file systems, and disk storage).

In high-performance computing (HPC), parallel I/O architectures usually have very complex hierarchies with multiple layers that collectively constitute an I/O stack, including high-level I/O libraries such as PnetCDF and HDF5, I/O middleware such as MPI-IO, and parallel file systems such as PVFS and Lustre (see Figure 1). Our DOE project explored automated instrumentation and compiler support for I/O intensive applications. Our project made significant progress towards understanding the complex I/O hierarchies of high-performance storage systems (including storage caches, HDDs, and SSDs), and designing and implementing state-of-the-art compiler/runtime system technology that targets I/O intensive HPC applications that target leadership class machine. This final report summarizes the major achievements of the project and also points out promising future directions Two new sections in this report compared to the previous report are IOGenie (Section 3) and SSD/NVM-specific optimizations (Section 6).

1. IOTRACE: A Parallel I/O Profiling and Visualization Framework for High-Performance Storage Systems

To optimize I/O behavior of large-scale applications, the first step is to understand how I/O is performed across the software stack and identify potential bottlenecks with the stack. However, the complex hierarchy makes it difficult and nearly impossible to track I/O behavior manually at each layer for petascale applications and beyond. As a part of our DoE project, we implemented IOTRACE, an automated I/O tracing tool, which provides insights on I/O flows across the I/O stack automatically for any given parallel I/O application. The tool supports both automated instrumentation and automated data synthesis based on user queries. The high-level metrics of interests include I/O latency, I/O throughput, and disk power consumption in the stack, with various operations such as “breakdown”, “max”, “min”, “average”, “list”, etc. The statistics returned by our tool can be used to investigate not only the bottleneck of individual applications, but also the behavior of a group of applications when running together on a cluster of computing nodes.

Figure 2 below shows the high-level view of IOTRACE, which consists of three major components, namely, *instrumentation engine, execution engine,* and *data processing engine*. The instrumentation engine is responsible for probe selection and insertion. That is, based on user setup and configuration information, it selectively inserts probes from the probe library to the application and I/O libraries in the stack (to collect requested I/O statistics), and invokes the execution engine to compile and run the instrumented code. The data collected at runtime are recorded in the log files, which are then passed to the data processing engine for various user queries. Since the data processing engine is separate from the execution engine, our tool can be used to query collected statistics from

multiple angles. More specifically, we can look at the statistics from the server side, client side, or from any layer in the storage stack (an important capability to pinpoint I/O bottlenecks and tune I/O behavior and storage system performance). Also, since the "build" step is separate from the "run" step in the execution engine, user can reconfigure the I/O architecture (topology) and run the application multiple times without recompiling the I/O stack. Using the well-defined query format and the powerful data synthesizer, IOTRACE supports complex I/O flow analysis across different layers and provides a way of obtaining both top-down and bottom-up views for the connections in the I/O hierarchy, e.g., which server nodes that client node X contacted and which client nodes that server node Y served, for a particular group of I/O calls.

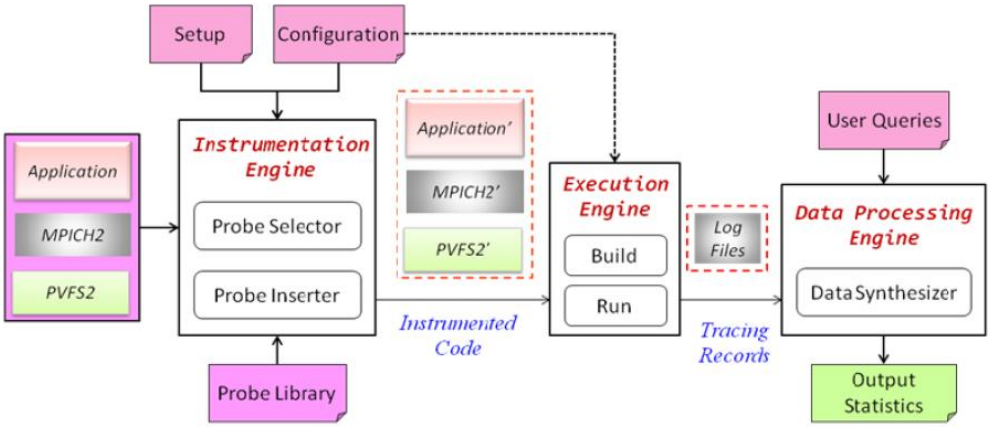

Figure 2: High-level view of IOTRACE.

In the following, we present some snapshots to illustrate the working of IOTRACE:

1) Input the "Setup" information, as shown in Figure 3, including (a) the locations of I/O libraries, application program and log files; and (b) compiling flag and tracing options, e.g., what kind of I/O operations to trace (read or write) and which segment of code to trace (e.g., between lines 30-5000).
2) Input the "Configuration" information, as shown in Figure 4, which includes the definition of the I/O stack. By changing the information here, our tool can target any I/O stack.
3) Click on the "Build" button, to compile and build the application program and related I/O libraries.
4) Click on the "Run" button, to execute the application program.
5) Input the "Query", as shown in Figure 5. The queries supported include latency, throughput, power consumption, disk access pattern, and I/O call information (e.g., how many I/O calls are issued to server X?). The operations on latency can be further divided into (a) inclusive breakdown (the timing spent across all levels below a certain layer) and exclusive breakdown (the latency spent at a certain layer) for a particular I/O call; and (b) "list", "max", "min", "avg" operations for a group of I/O calls.
6) Output Statistics, as illustrated by Figure 6 for the query of "inclusive latency of client 0".
7) User may input new "Queries", or change "Configuration" to redefine the I/O stack.

Our implementation of IOTRACE has been tested on the breadboard cluster in Argonne National Laboratory (ANL), and can be downloaded from *http://www.cse.psu.edu/~seokim/iotrace.*

2. IOPin: Runtime Profiling Framework for Parallel I/O in HPC Systems

While automated static I/O instrumentation described above is extremely useful in identifying I/O bottlenecks and thus improving application programmer's productivity, it may not be the best option for I/O-intensive applications whose behavior changes dynamically during the course of execution. Motivated by this, we also developed in our project a dynamic performance analysis and visualization tool for parallel I/O. In this part of our project, we leverage a lightweight binary instrumentation using Pin to implement our current prototype of the tool. That is, our tool performs the instrumentation in the binary code of the I/O stack at runtime. Therefore, our tool provides the language-independent instrumentation working with large-scale scientific applications written in C/C++ or Fortran. Furthermore, our tool requires neither source code modification nor recompilation of the applications and I/O stack. We observe that the overhead incurred by our proposed dynamic instrumentation is about 7%, on average, as plotted in Figure 7.

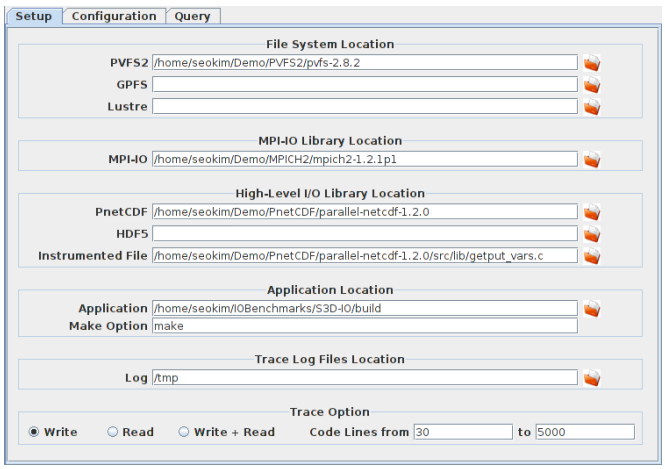

Figure 3: Setup GUI.

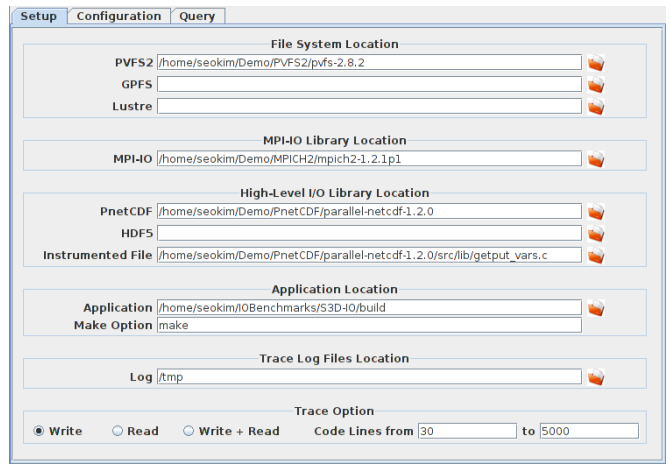

Figure 4: Configuration GUI.

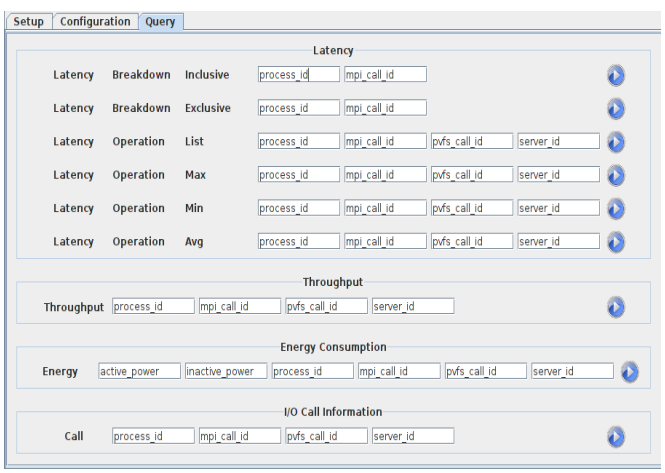

Figure 5: Query GUI.

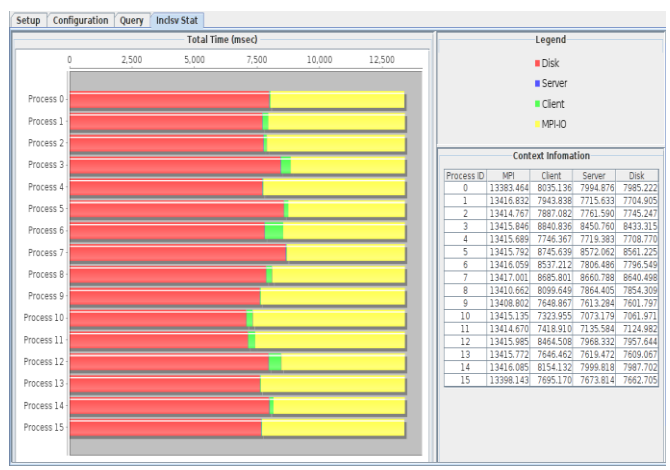

Figure 6: Returned statistics.

Figure 8 illustrates how our framework carries out dynamic instrumentation. When the MPI I/O function call is issued from the high-level I/O libraries or applications, the Pin process on the client side generates trace information, including rank, mpi_call_id, pvfs_call_id, I/O types, and timestamp in the MPI library. By definition, MPI I/O function call is replaced with PVFS_sys_io() function with additional arguments to be issued to PVFS client. Here, the Pin process packs the trace information into a PVFS_hints structure and replaces the last argument, PVFS_HINT_NULL with the customized hint in the PVFS_sys_io(). In PVFS client, the Pin process extracts the trace information from the hint and stores the trace information in the buffer. The Pin-defined hint is encapsulated into state machine control block and passed to PVFS server.

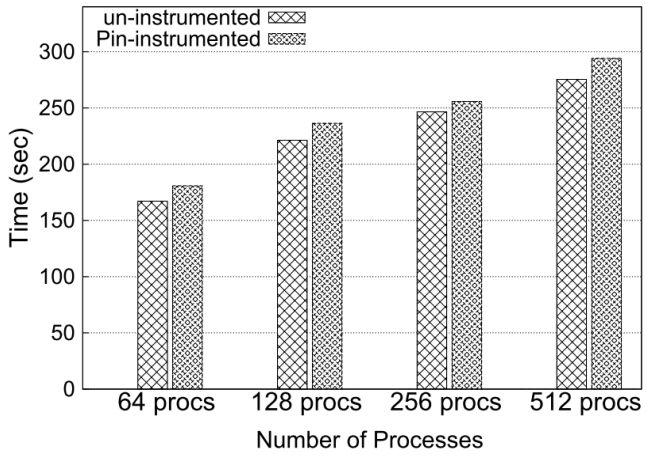

Figure 7: Execution time in un-instrumented I/O stack and Pin-instrumented.

The Pin process on the server searches a PVFS_hints passed from the upper layer and extracts the trace information. In PVFS server, a flow_descriptor structure is maintained for the I/O operations and flows until the end of the I/O operations. Since the customized hint containing the trace information exists in flow_descriptor, the Pin process can extracts it from the hint in flow_descriptor at any point in the server. At the exit point of the server, the Pin process produces the log file with necessary information, e.g., rank, mpi_call_id, pvfs_call_id, I/O type, bytes processed for each MPI I/O operation, the number of accesses to disk, latency spent in the server, and disk bandwidth. Again, the Pin process on the client side generates a log file at the exit point of the layer. Figure 9 shows the latency spent in each layer when S3D-IO ran on 256 processes. The number on the bar representing server is the number of disk accesses for the MPI I/O call. We notice

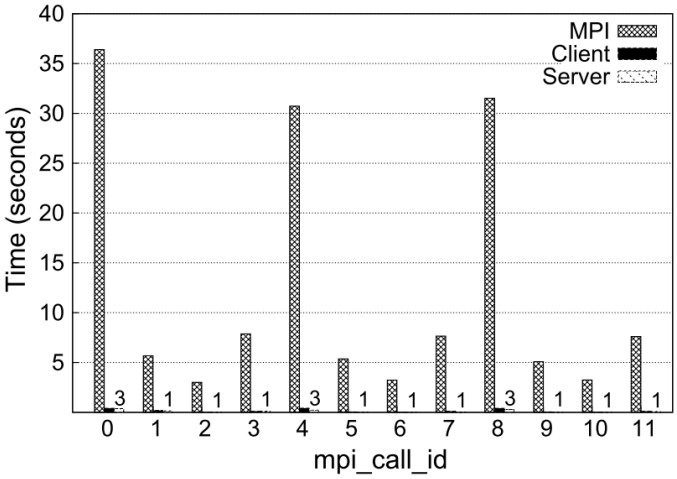

Figure 9: Latency in each layer.

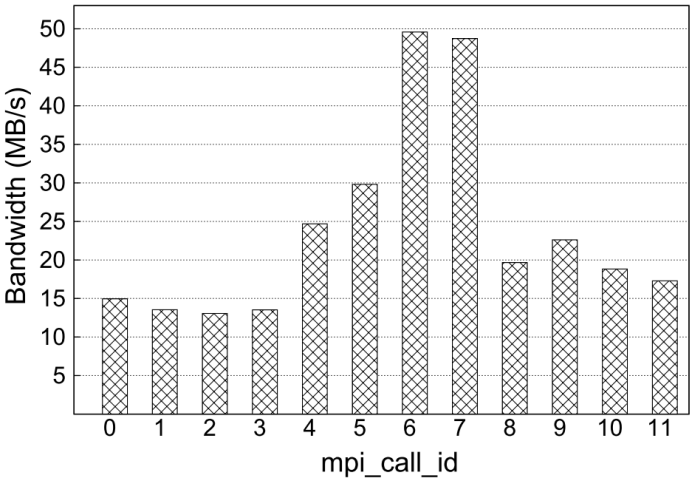

Figure 10: Disk throughput.

that most of the execution time is spent in the MPI library to exchange data for communication and optimization and server operation time is less than 1 second. Disk throughput for the corresponding I/O operations is plotted in Figure 10.

Our IOPin implementationhas been tested on the machines at ANL, and can be downloaded from *http://www.cse.psu.edu/~seokim/iopin.*

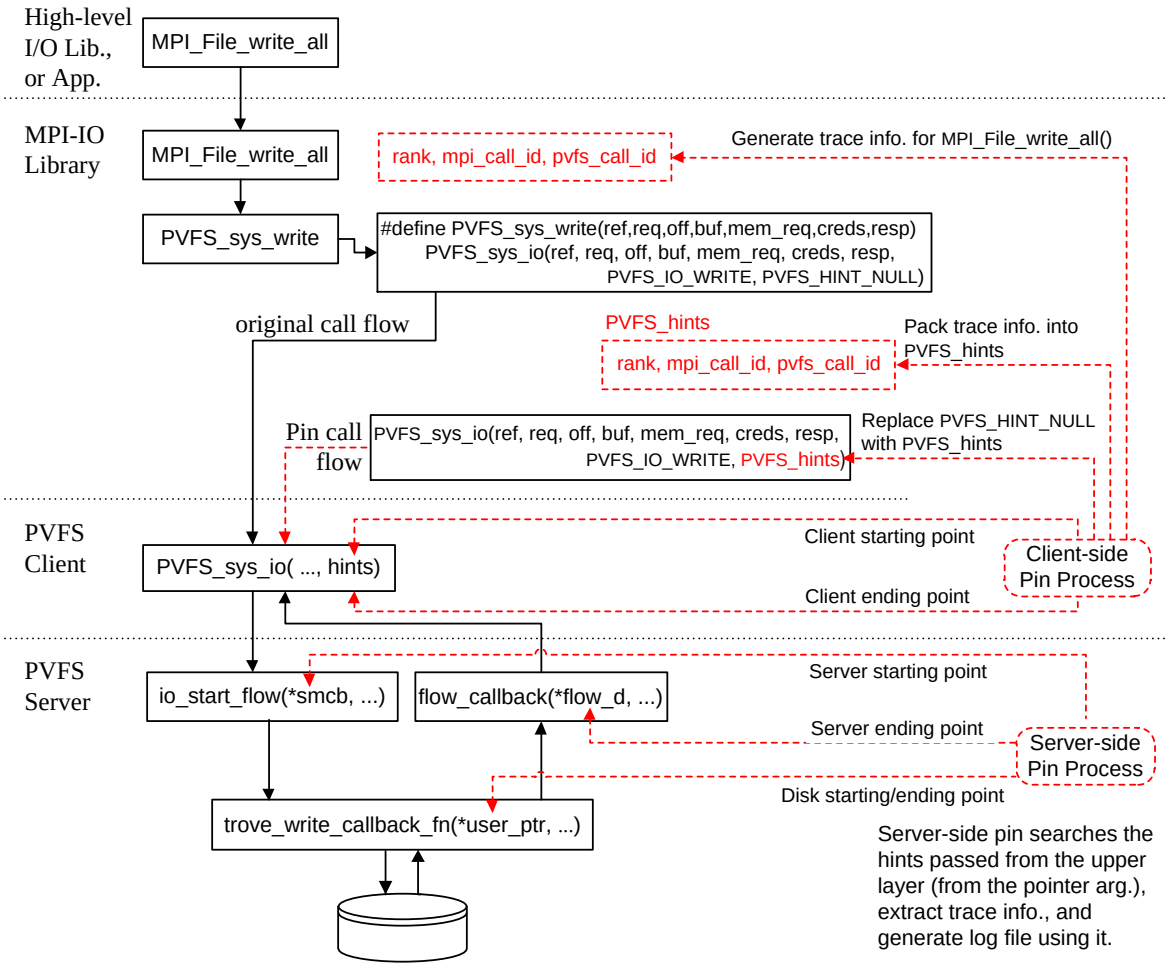

Figure 8: Structural description of IOPin.

3. IOGenie: An Automatic Parallel I/O Code Generation and Optimization framework for HPC Applications

Efficient execution of parallel applications highly depends on I/O performance in HPC systems, leading scientists to observe and understand the complex I/O interactions between applications and the underlying I/O software. Although significant progress to improve I/O performance has been made in high-level I/O libraries, middleware, and parallel file systems, without effective use of them, it will be challenging for users to make scientific discoveries. Motivated by this, we proposed an automatic parallel I/O code generation framework, called *IOGenie*, which generates the efficient and optimized code for the scientific parallel applications written in C/C++ and Fortran. Using a graphical user interface, our tool takes high-level annotations for I/O as input, analyzes the given options, and generates optimized I/O code that effectively exercises the underlying I/O stack. This tool helps users write data-intensive applications easily and effectively. It also enhances the quality of tool-generated code by exploiting various optimizations provided by the underlying I/O software. Our experience indicates that the overhead of running tool-generated applications is negligible.

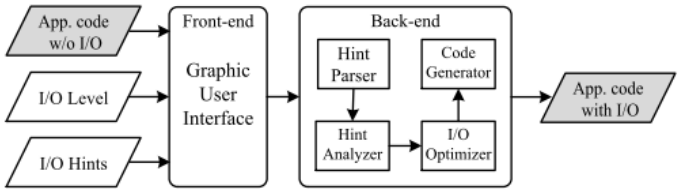

Figure 11. High-level view of IOGenie.

Figure 11 presents the high-level view of IOGenie. It comprises two main components: *a front-end graphical user interface (GUI)* and *a back-end code generation engine.* The GUI accepts as input application code *without I/O*, I/O level, and I/O hints. Here, the application code is in the form of a template that only contains computation code, *not* including I/O operations. The I/O level indicates the layer and the library that should perform I/O in the software stack (Figure 1). Lastly, I/O hints specify the high-level I/O information required to generate optimized code, e.g., data set definition and data access pattern. This input information is utilized by IOGenie to guide the I/O code generation.

More specifically, these input parameters are passed to the code generation engine that accommodates *a hint parser, a hint analyzer, an I/O optimizer,* and a *code generator*. First, the hint parser breaks the given hints down. Then, the hint analyzer resolves them. The I/O optimizer tries to improve the quality of the I/O operations by employing the optimization techniques available in the selected I/O library. Finally, the code generator outputs the resulting code augmented with parallel I/O operations. Note that the GUI interface is developed to simplify and facilitate the use of our tool.

```
int main(int argc, char** argv)
{
  int rank, size, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int ncid, lonid, latid, timeid, Tid, Tdimids[3], len;
  size_t counts[3], starts[3];
  char *Tname = "temperature", *Tunits = "K";
  double Trange[] = { 0.0, 500.0 }, *temp, *alltemps;
  char *title = "Global surface temperature data", history[256], buf[26];
  time_t tm;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Get_processor_name(processor_name, &namelen);
  printf("Rank %d of %d started on %s\n", rank, size, processor_name);

#begin_dimensions
lon: NLON;
lat: NLAT;
time: unlimited;
#end_dimensions

#begin_variables
T: double, 3, time, lon, lat;
W/Wid: double, 3, time, lon, lat;
#end_variables

#begin_attributes
long_name: T, text, Tname;
units: T, text, Tunits;
valid_range: global, double, Trange;
history: global, text, history;
#end_attributes

  tm = time((time_t *)NULL);
  (void)ctime_r(&tm, buf);
  if (buf[24] = '\n') buf[24] = '\0';
  sprintf(history, "Created %s", buf);
```

```
//////////////////////////////////////////////////////////////////
//I/O code automatically generated by Tool1.0
//////////////////////////////////////////////////////////////////

/* Definition of ids. */
int ncid, dim_ids[3], dim_starts[3], dim_counts[3];
int lonid, latid, timeid;
int Tid, Wid;

/* Creation of nc file. */
ncmpi_create(MPI_COMM_WORLD, "test.nc",  0, MPI_INFO_NULL,  &ncid);

/* Definition of dimensions. */
ncmpi_def_dim(ncid, "lon", (MPI_OFFSET)(NLON), &lonid);
ncmpi_def_dim(ncid, "lat", (MPI_OFFSET)(NLAT), &latid);
ncmpi_def_dim(ncid, "time", (MPI_OFFSET)(NC_UNLIMITED), &timeid);

/* Definition of variables. */
dim_ids[0] = timeid;
dim_ids[1] = lonid;
dim_ids[2] = latid;
ncmpi_def_var(ncid, "T", NC_DOUBLE, 3, dim_ids, &Tid);
dim_ids[0] = timeid;
dim_ids[1] = lonid;
dim_ids[2] = latid;
ncmpi_def_var(ncid, W, NC_DOUBLE, 3, dim_ids, &Wid);

/* Definition of attributes. */
ncmpi_put_att_text(ncid, Tid, "long_name", strlen(Tname), Tname);
ncmpi_put_att_text(ncid, Tid, "units", strlen(Tunits), Tunits);
ncmpi_put_att_double(ncid, NC_GLOBAL, NC_DOUBLE, 1, Trange);
ncmpi_put_att_text(ncid, NC_GLOBAL, "history", strlen(history), history);

/* End of definition mode. */
ncmpi_enddef(ncid);

//////////////////////////////////////////////////////////////////
//End of I/O code automatically generated by Tool1.0
//////////////////////////////////////////////////////////////////
```

Figure 12. Sample I/O code automatically generated by IOGenie.

Figure 12 shows a sample I/O code generation performed by our framework. In this example, the application is written in C and utilizes PnetCDF as the high-level I/O library, on top of PVFS. The necessary I/O hints such as

dimension, variables, and attributes are taken in the left window (starting with "#"). Then, our tool transforms the hints internally and generates the corresponding I/O code on the right. The goal of IOGenie is not only to improve the productivity of code generation, but also to ensure the quality of the generated I/O code. The latter one is achieved via the I/O optimizer component in the code generation engine. An important point to note is that the optimizations in our optimizer are accomplished for user level applications, not the I/O library. Thus, our tool will never modify the I/O libraries under any circumstance.

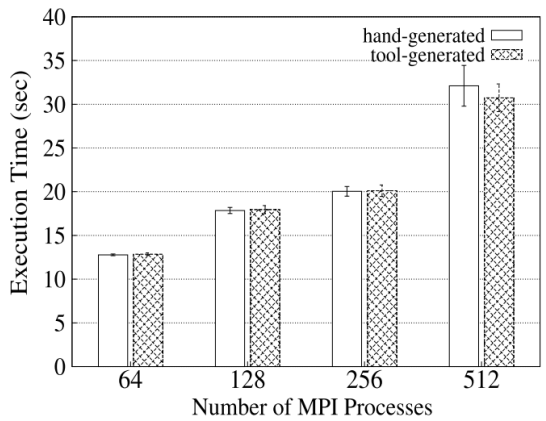

Figure 13. FLASH I/O average execution time.

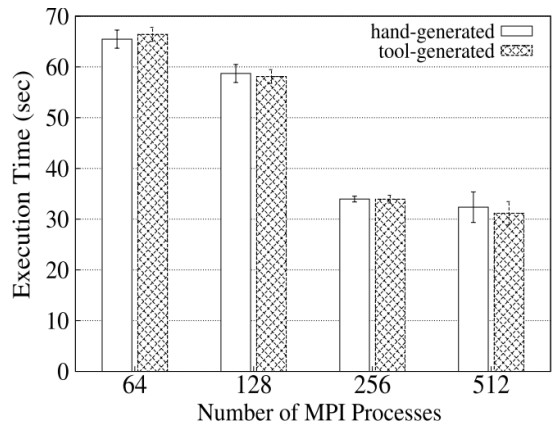

Figure 14. S3D I/O average execution time.

In our evaluation, using two scientific data-intensive applications, FLASH I/O and S3D I/O, written in C and Fortran, respectively, our tool generates optimized I/O operations, depending on data access patterns of the applications. For the FLASH I/O benchmark whose data access pattern is noncontiguous, our tool generates I/O code to be suitable for data sieving, while it generates I/O operations to be appropriate to two-phase I/O for the contiguous data access in the S3D I/O benchmark. Figure 13 and Figure 14 compare the average execution times with the hand-generated I/O code and the IOGenie-generated I/O code for FLASH I/O and S3D I/O. Our experiments indicate that the overhead of tool-generated code is negligible. IOGenie will be placed into the public domain soon.

4. Compiler-Directed File Layout Optimization for Hierarchical Storage Systems

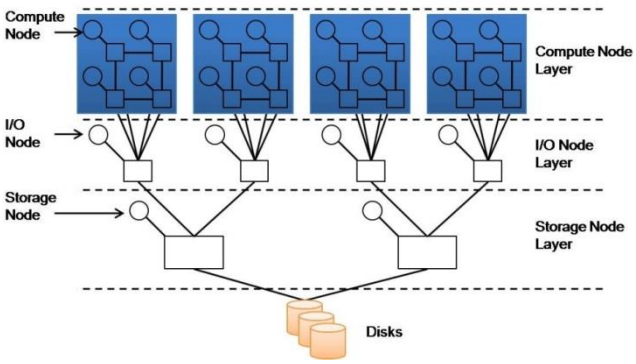

Figure 15: A sample storage system with three-layer storage cache hierarchy.

Compiler support for I/O-intensive applications has been investigated in the past, which can be roughly divided into two categories: *code transformations* and *file layout optimizations.* Most of previous code transformation work deals with iteration space tiling and distribution of loop iterations across multiple nodes. File layout optimizations represent a complementary approach and prior efforts in this direction target sequential applications. To our knowledge, there is *no* prior compiler-based work that optimizes file layouts targeting parallel computing platforms with hierarchical storage caches. In our study, we assume that nodes in all layers can be equipped with certain amount of storage caches (though in our experiments we allocate caches only in I/O and storage nodes, due to the high demand of compute node memory). In Figure 15, each rectangle represents storage

caches at different layers. In this particular configuration, every four compute nodes are connected to one dedicated I/O node, all of which are then connected to two storage servers. We note that the ratio of I/O nodes to compute nodes can vary from system to system, and depending on the specific configuration, each I/O node services I/O request from different number of computer nodes. Since multi-layered architectures are prevalent, it is crucial to explore cache management and optimization policies that operate in a hierarchy-aware manner.

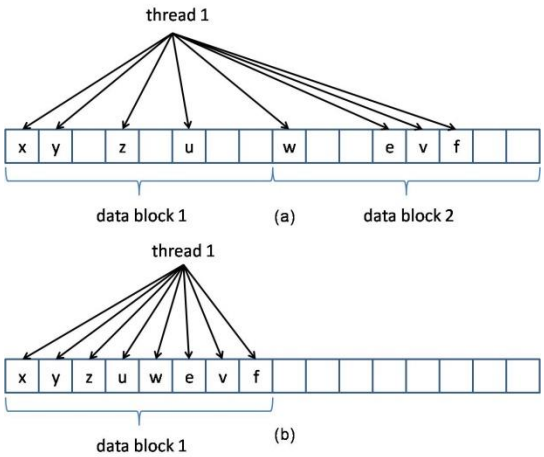

Figure 16: Importance of optimized file layout in minimizing the number of blocks needed to store the accessed data.

We now discuss what role file layout optimization can play in improving performance of hierarchical storage caches. From a data access perspective, one can talk about three different "layouts" for data. "Array Layout" represents how a multi-dimensional data structure is viewed by the program. "File layout", on the other hand, is a result of the mapping between array elements (in the multi-dimensional space) and file locations. While traditional file layout, such as "row-major" and "column-major" are probably sufficient for sequential applications with good spatial reuse, they may not be the best option for multi-threaded applications that manipulate disk-resident datasets. To illustrate this, consider Figure 16(a) which shows a highly simplified example where a thread (of a multi-threaded application) makes data access to a file under a default (e.g., row-major) file layout. It can be observed that these data accesses are scattered all over the file space, which tends to increase the number of "*data blocks*" (our I/O cache management unit) occupied. Higher number of data blocks required to store the requested data elements in turn increases the pressure on shared storage caches. More specifically, since each thread brings to the cache more data elements than it needs, effective cache capacity is reduced, which in turn affects application performance. Further, this also hurts performance of other threads that share the same storage cache with this thread. Consider now Figure 16(b) which shows the same data accesses under an alternate file layout optimized using our approach. In this case, the data elements accessed by our thread are stored in consecutive file locations, which helps us minimize the number of data blocks occupied. Based on this observation, we propose an "inter-node file layout optimization" scheme that can reduce the "block footprint" of a node at *each* I/O cache level in the target hierarchical cache system.

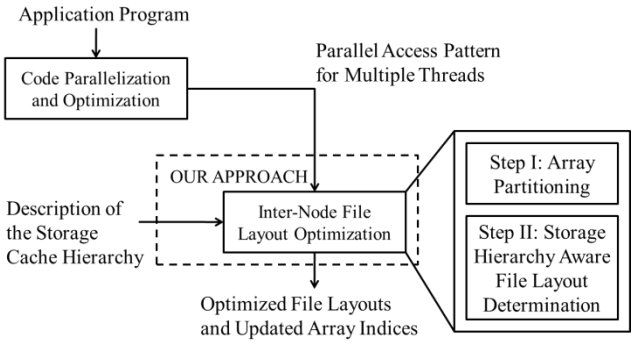

Figure 17: High level overview of our approach.

Figure 17 shows where our scheme falls in the compilation flow. Our optimization is applied following the code parallelization and optimization phase. Whether input code parallelization is performed using an optimizing compiler or by the user is orthogonal to our approach. Basically, the input to our approach is a multi-threaded application that accesses one or more disk-resident array data and a description of the target storage cache topology (e.g., the number of layers, number of caches at each layer, their capacities, and block sizes). Each array is assumed to be stored in a separate file. The output of our approach is modified array accesses (array index functions are updated) and optimized file layout for each array based on parallel accesses issued to it by different threads.

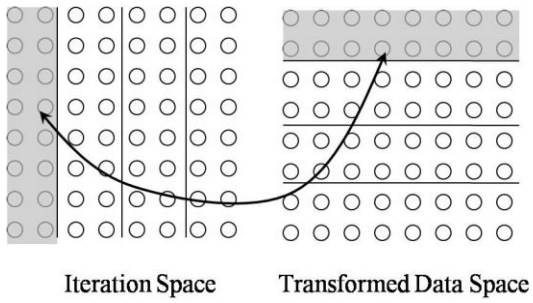

Figure 18: Array partitioning to track the data regions accessed by each thread.

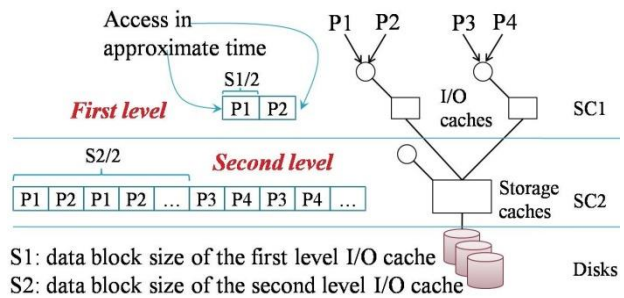

Figure 19: two layout patterns correspond to the sample architecture, where the rectangles denote caches and circles represent processors.

Our approach has two steps: *array partitioning* and *storage hierarchy aware file layout determination*. The first step is to partition the data space of each (multi-dimensional) array, with respect to the most significant loop nest, i.e., the loop nest that accesses this array most heavily. This helps to track the data regions touched by different threads created after parallelization. Since the parallelization of a loop nest divides the (multi-dimensional) loop iteration space into computation blocks, each of which is to be executed on a different threads, as depicted in Figure 18, our task is essentially to find a data space partitioning for each disk-resident array based on this iteration space partitioning, such that each data region is locally accessed by a particular computation block as much as possible. We call this data space partitioning the "best-effort partitioning" for an array, since depending on access patterns, there still might be data shared by multiple computation blocks to some degree.

Our second step is to create a linear layout for each array based on array partitioning. An important feature of this step is that it explicitly considers the storage-cache hierarchy in the target I/O architecture and forms a linear file layout that minimizes the number of file blocks used by all threads at any given time as well as conflicts between threads in shared storage caches, thus reducing expensive I/O cache misses. As shown in Figure 19, in the target architecture, we first construct a SC1 pattern <P1, P2> of size S1. This pattern has two contiguous file chunks, each of which has size S1/2. The first file chunk contains S1/2 data elements accessed by thread P1, and the second file chunk contains S1/2 data elements accessed by thread P2. Clearly, if we place the data elements accessed by P1 and P2(into the file) in this way, each of them utilizes half of the shared SC1 cache space for this array access, which helps reduce the conflict misses between P1 and P2 at runtime. Similarly, we can also construct a SC1 pattern <P3, P4> for P3 and P4. Next, we construct a SC2 pattern <P1, P2, P3, P4> of size S2, by first repeating the SC1 patterns <P1, P2> until size S2/2, and then repeating the SC1 pattern <P3, P4> for the rest. Clearly, if we place the data elements accessed by P1, P2, P3 and P4 in this way, the space contention among these threads in the shared SC2 cache can be alleviated. This SC2 pattern is the layout pattern we are looking for, and it will be applied repeatedly to create a linear file layout for the entire array.

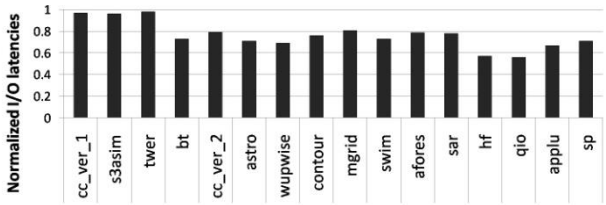

Figure 20: Execution time results.

Figure 20 shows normalized I/O latencies for a set of HPC applications when using 128 client nodes. Each bar is *normalized* with respect to the original version of application (i.e., when using default storage layouts). It can be seen from these results that, our applications can be divided into three groups. In the first group, which contains cc-ver-1, s3asim and twer, are applications that do not benefit from inter-node file layout optimization. The underlying reason for this can change from one application to another. For example, cc-ver-1 and s3asim already have very good storage cache hit rates in their default execution; there is simply no scope for additional performance improvement. In comparison, in twer, overly-conflicting requests from different threads at different points in execution prevent the compiler from choosing a good file layout. The applications in the second group, which contains bt, cc-ver-2, astro, wupwise, contour and mgrid, benefit reasonable well from the inter-node layout optimization (with improvements ranging between 8% and 13%). In the third group, which includes applications swim, afores, sar, hf, qio, applu and sp, the inter-node layout optimization brings significant benefits (between 21% and 26%). When all applications are considered our compiler-based file layout optimization scheme brings an average execution time improvement of 23.7%.

5. Cashing in on Hints for Better Prefetching and Caching in PVFS and MPI-IO

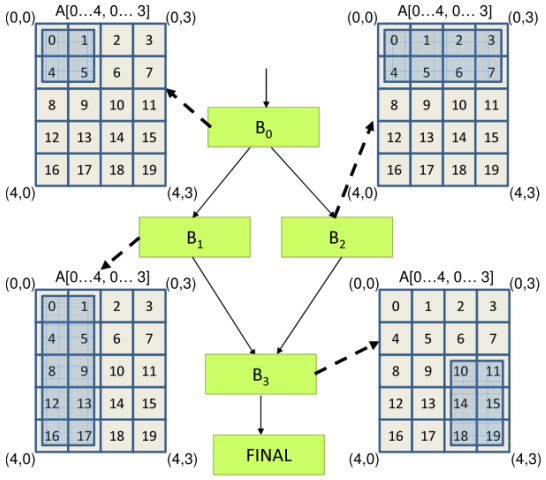

Figure 21: Illustration of a hint flow graph

We also investigated a novel approach to the management of parallel I/O in the context of large scale systems targeting applications from scientific and engineering domains. Our proposed approach is built upon three complementary ideas: (i) allowing users to place *hints* into the application code indicating high-level data access patterns, (ii) enabling an optimizing compiler to process these hints and develop I/O optimization strategies, and (iii) enhancing the I/O stack to accept these optimizations and process them across different layers in the stack. In this way, the proposed approach combines the advantages of user level and compiler level information about the program I/O patterns and runtime information available within the I/O stack during execution. We describe a general framework that accommodates this approach and demonstrate its potential by applying it to two important I/O optimizations: (i) *shared storage cache management* and (ii) *I/O prefetching*. In the former, our approach decides, at each program point of interest, the ideal set of data blocks to keep in shared storage caches in the I/O stack, and in the latter, the high level data access pattern is propagated from application layer to the parallel file system layer for prefetching data from disks.

Our approach consists of three major components, namely, *programming language* (PL) *support*, *compiler support*, and *runtime system support,* as described below.

5.1. Programming Language (PL) Support

The PL support includes *programmer directives* (which are also called *user hints* in this paper) that help us specify high-level parallel data access patterns. It is in the form of language directives (user hints) and can be used within different languages such as C, C++, C#, Fortran and UPC as well as new HPCS languages such as X10, Fortress, and Chapel. The main goal behind programmer-inserted directives (user hints) is to convey the high-level data access patterns to the underlying compiler and I/O stack for optimization purposes.

5.2 Compiler Support

The compiler support automatically analyzes of the directives (user hints) mentioned above, as well as the program source code, and then decide the I/O optimizations to employ. These optimizations are expressed as *system hints* and passed to the I/O stack. Specifically, it has two complementary components, which may, based on the application code and I/O optimizations considered, iterate multiple times until an optimization strategy is fully developed (e.g., each iteration can generate a system hint to be passed to the I/O stack). The first of these components, called *Selector,* analyzes the application level I/O pattern and, taking into account the storage layout and architectural details as well as user directives, develops an I/O optimization strategy. The second component, *Executor*, implements these optimization decisions by inserting appropriate system hints in the program code and by restructuring the code if necessary. Recall that the I/O optimization decisions taken by our compiler are expressed as system hints that are passed to the I/O stack. These hints are later taken care of by our proposed runtime system enhancements. An important benefit of this approach is that we can take advantage of runtime conditions (since hints are primarily runtime actors), which may not be possible with a conventional compiler support that restructures application code based on static (compile time) information alone. Also, by considering the hints coming from different concurrently-running applications, the I/O stack has the flexibility to make global (inter-application) optimization decisions. In a typical implementation, Selector carries out a hint flow analysis (explained below) and Executor inserts a hint into the program code.

The main structure used by our compiler for implementing hint based optimizations, is called the *hint flow graph* (Figure 21). A hint flow graph, G(V,E), is built using (i) the hints inserted by the user into the code and (ii) control V is a *hint insertion point* in the program, chosen by the programmer. There is an edge between nodes (hint insertion points) v and v′ if there is a direct path - a path that does not go over another hint - in the underlying control flow graph from the hint location represented by v to the hint location represented by v′. Therefore, a hint flow graph is a complete representation of the hints inserted by the programmer into the program code. On this graph, we can use traversals similar to those employed by conventional compilers during data-flow analysis of control-flow graphs to capture the interactions among different hints. The exact nature of the traversal, which is referred to as the *hint flow analysis* in this paper, depends on the I/O optimization(s) targeted.

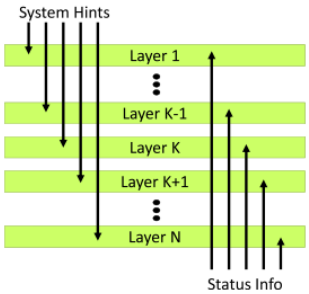

Figure 22. Illustration of the generic hint and its status flow through an I/O stack which consists of N layers.

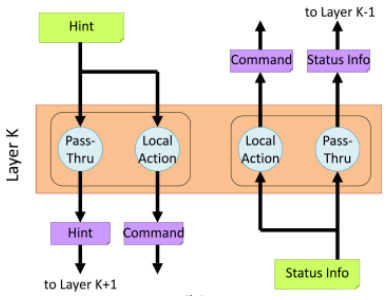

Figure 23. Action on a system hint inserted by our compiler at a specific layer (K).

5.3 Runtime System Support

Finally, the runtime support is responsible for management of system hints during execution. It will not be possible to optimize parallel I/O in the petascale era by just enhancing compilers and/or language level support. It is critical that compiler cooperates with runtime environment and I/O stack through hints. While many current software layers in the I/O stack accommodate different types of hints, current systems rarely use these hints. A system hint inserted by our compiler can flow through the layers in the I/O stack and can be acted upon at different layers. The details of a specific layer (K) are presented in Figure 23. When a hint from layer (K-1) comes to layer K, it is fed into two modules: *pass-thru* and *local action.* The pass-thru module re-expresses (translates) the hint or a part of it (if necessary) in a form that can be understood by the following layer (K+1) and passes it to layer (K+1). The local action module on the other hand converts the hint or a part of it (if necessary) to an I/O command and sends it to a layer L where L > K. The important point here is that, for a given hint, there are three possibilities: either it is re-expressed and passed to layer K + 1, or it is converted to an I/O command and passed to layer L, or both. This last possibility typically means re-expressing part of the hint or converting part of the hint to an I/O command, and thus represents the most general case. In our approach, a hint (or a part of it) will be converted at some point to an I/O command and executed. An implemented hint will typically generate a response (which we call *status information*) which will be propagated in the reverse direction (bottom-up) of the hint (which is top-down). As in the case of top-down flow, for typical status information, three possibilities exist at layer K: it can either be passed to layer (K-1) as it is; or it can be acted upon at this layer (K); or both. In our scheme, system hints are processed within the I/O stack.

Note that our approach clearly combines the advantages of static and dynamic optimization strategies. Through user-inserted hints, we capture what is known to the programmer at various program points (i.e, local information). The compiler analyzes this information in a systematic fashion, derives (using local information inserted at different program points) global optimization strategies, and passes them to the runtime system. In this way, we extract all the information available before the execution starts (from both user and compiler). This information is passed to the I/O stack, which processes it for improving parallel I/O behavior, considering data layout and current state of the storage system. Therefore, directives and compiler analysis constitute the static support in our system, whereas runtime hint management provides our dynamic support.

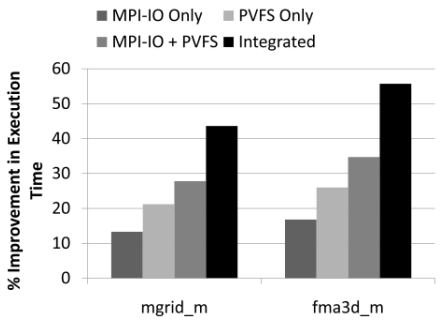

Figure 24: Caching results using 32 client nodes.

5.4 Results

To evaluate our storage caching strategy, we used two I/O intensive multi-threaded applications that can benefit from storage caching when working with disk-resident data sets: mgrid_m and fma3d_m. The former is a program that implements multi-grid solver in 3D potential field, whereas the latter is a finite element crash simulation code. Each of these applications manipulates about 28GB of disk resident data. For each of these two benchmarks, we experimented with four different versions: MPI-IO Only, PVFS Only, MPI-IO+PVFS, and Integrated (our proposed scheme). In MPI-IO Only, caching is performed only at the MPI-IO layer, and similarly, in PVFS Only, it is implemented only in the PVFS layer. MPI-IO + PVFS on the other hand implements caching at both MPI-IO and PVFS layers but these cachings are *not* coordinated. In contrast, in our scheme, the PVFS layer caches only the frequently-referenced data blocks that could not be cached by the MPI-IO layer. The storage caching results collected with the four versions described above (MPI-IO Only, PVFS Only, MPI-IO+PVFS, and our scheme) are presented in Figure 24. Each bar in this plot represents percentage improvement in overall application execution

latency over an execution that does not employ any storage caching. As expected, both our scheme (Integrated) and MPI-IO + PVFS generate better results than MPI-IO Only and PVFS Only schemes. What is more important however is the significant performance difference between our scheme and MPI-IO + PVFS. For mgrid_m and fma3d_m, our approach performs, respectively, 21.8% and 32.1% better than the MPI-IO + PVFS scheme. This is mainly due to double-caching that occurs in the MPI-IO + PVFS scheme, i.e., the same data block can be cached (under the MPI-IO + PVFS scheme) in both MPI-IO and PVFS layers, since these two layers function independently under this scheme. This in turn reduces the effective (cumulative) cache size (capacity), and eventually, impacts overall performance negatively. We also made experiments with larger cache sizes. Our results with 2GB MPI-IO layer cache and 4GB PVFS cache showed that the percentage improvements brought by our scheme over MPI-IO + PVFS are 24.4% and 35% for mgrid_m and fma3d_m, respectively. These results indicate that our scheme takes better advantage of additional storage cache capacity than the MPI-IO + PVFS scheme. Overall, these results underline the importance of coordinated caching in the I/O stack.

The source files of our compiler implementation can be obtained from *http://www.cse.psu.edu/~wzd109/IOC.7z.*

6. SSD/NVM-Specific Optimizations

Our project also investigated SSD-specific issues, focusing in particular on the runtime system layer, as well as the interactions between the system I/O stack and SSD firmware stack. The specific contributions of our project can be summarized as follows:

Revising Widely-Held SSD Expectations in the Context of HPC: Storage applications leveraging Solid State Disk (SSD) technology are being widely deployed in diverse HPC systems. These applications accelerate system performance by exploiting several SSD-specific characteristics. However, modern SSDs have undergone a dramatic technology and architecture shift in the past few years, which makes widely held assumptions and expectations regarding them highly questionable. The main goal of this part of our project was to question popular assumptions and expectations regarding SSDs through an extensive experimental analysis using 6 state-of-the-art SSDs from different vendors. Our analysis led to several conclusions which are either not reported in prior SSD literature, or contradict to current conceptions. For example, we found that SSDs are not biased toward read-intensive workloads in terms of performance and reliability. Specifically, random read performance of SSDs is worse than sequential and random write performance by 40% and 39% on average, and more importantly, the performance of sequential reads gets significantly worse over time. Further, we found that, in the context of HPC applications, reads can shorten SSD lifetime more than writes, which is very unfortunate, given the fact that many existing systems/platforms already employ SSDs as read caches or in applications that are highly read intensive. We also performed a comprehensive study to understand the worst-case performance characteristics of our SSDs, and investigated the viability of recently proposed enhancements that are geared towards alleviating the worst-case performance challenges, such as TRIM commands and background-tasks. Lastly, we uncovered the hidden overheads of these enhancements and their limits, and discuss system-level implications.

Automatic Management of Large-Scale Storage-Class RRAM: Resistive Random Access Memory (RRAM) is a promising next generation non-volatile memory (NVM) technology, thanks to its performance potential, endurance and ease-of-integration with standard silicon CMOS processes. While prior work has evaluated RRAM as a replacement for DRAM or even cache memory, to our knowledge there is *no* prior study that has investigated whether RRAM could be a viable NAND flash replacement in building large-scale storage-class memory systems in the HPC domain. Motivated by this observation, our project quantified the main bottlenecks associated with RRAM that prevent it from replacing NAND flash. The main solution we proposed, "slab-based memory access with local/global bitlines," enabled dense RRAM islands but can also cause performance related problems. To compensate for the latter, we

also proposed exploiting internal resource parallelism in RRAM using runtime techniques and employing optimized data movement interfaces. Our extensive experimental evaluation indicated that the proposed architecture can provide 2.95 ~ 8.28 times better bandwidth and 66% ~ 88% shorter latency as compared to the conventional NAND flash, and improve the system-level performance of our workloads by 5x, with a storage capacity similar to that of the state-of-the-art NAND flash.

Runtime System Based Page Allocation: Exploiting internal parallelism over hundreds NAND flash memory is becoming a key design issue in high-speed Solid State Disks (SSDs). In this part of our project, we simulated a cycle-accurate SSD platform with twenty four page allocation strategies, geared toward exploiting both system-level parallelism and flash-level parallelism with a variety of design parameters. Our extensive experimental analysis revealed that 1) the previously-proposed channel-and-way striping based page allocation scheme is not the best from a performance perspective, 2) As opposed to the current perception that system and flash-level concurrency mechanisms are largely orthogonal, flash-level parallelism are interfered by the system-level concurrency mechanism employed, and 3) With most of the current parallel data access methods used in commercial SSDs employed in HPC systems, internal resources are significantly underutilized, pointing to future compiler optimizations that maximize storage space utilization.

Optimizing SSD Space Management in HPC applications: Garbage collection (GC) in SSD introduces significant latencies and large performance variations, which renders widespread adoption of SSDs difficult. To address this issue, we investigated a novel garbage collection strategy, consisting of two components, called *Advanced Garbage Collection* (AGC) and *Delayed Garbage Collection* (DGC), which cooperate to migrate GC operations from busy periods to idle periods. More specifically, AGC is employed to defer GC operations to idle periods in advance, based on the type of the idle periods and on-demand GC needs, whereas DGC complements AGC by handling the collections that could not be handled by AGC. Our comprehensive experimental analysis using different HPC workloads revealed that the proposed strategies provide stable SSD performance by significantly reducing GC overheads. Compared to the state-of-the-art GC strategies, P-FTL, L-FTL and H-FTL, our AGC+DGC scheme reduced GC overheads, on average, by about 66.7%, 96.7% and 98.2%, respectively.

The NAND Flash memory simulator developed during our project can be found in *http://www.utdallas.edu/~jung/nfs/pmwiki.php*.

7. Potential Future Research Directions

One of the main contributions of this project is the demonstration of the need for compilers and runtime systems to cooperate in a synergistic manner to exploit current and future high-performance I/O systems. We found static and dynamic code/data profilers to be vital for successfully establishing this cooperation. We also found it necessary to take into account unique features of emerging SSDs/NVMs (storage-class memory) to maximize the benefits. At this point, the potential future research directions include:

(1) Code and data optimizations for hybrid memory/storage systems that can contain DRAM, disks, SSDs, PCMs, etc.
(2) Design of strategies for enabling quick evaluation of future hardware and software enhancements to storage systems. An example would be quick evaluation of potential impact of adding NVMs to an existing storage system, for a given HPC workload.
(3) Interfacing issues between storage architecture and runtime system, which motivates for research in providing storage system-specific hardware counters to collect runtime data.

8. Publications

APP: Minimizing Interference Using Aggressive Pipelined Prefetching In Multi-Level Buffer Caches. Christina M Patrick, Nicholas Voshell and Mahmut Kandemir. Proceedings of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)

Minimizing Interference Through Application Mapping in Multi-Level Buffer Caches. Christina M Patrick, Nicholas Voshell and Mahmut Kandemir. Proceedings of 11th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2011)

Cashing in on Hints for Better Prefetching and Caching in PVFS and MPI-IO. Christina M Patrick, Mahmut Kandemir, Mustafa Karaköy, Seung Woo Son and Alok Choudhary. Proceedings of 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010)

Compiler-Directed File Layout Optimization for Hierarchical Storage Systems. Wei Ding, Yuanrui Zhang, Mahmut Kandemir, Seung Woo Son. Proceedings of International Conference on SuperComputing. (SC 2012, Best Student Paper Nomination).

Automated Tracing of I/O Stack. Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Ramya Prabhakar, Mahmut Kandemir, Christina Patrick, Wei-keng Liao, Alok Choudhary. Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface (EuroMPI 2010).

IOPro: A Parallel I/O Profiling and Optimization Framework in HPC Systems. Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Mahmut Kandemir, Wei-keng Liao, Rajeev Thakur, Alok Choudhary (submitted Journal of Supercomputing).

Taking Garbage Collection Overheads off the Critical Path in SSDs. Myoungsoo Jung, Ramya Prabhakar, Mahmut Kandemir. Proceedings of USENIX Middleware, 2012.

Revisiting Widely-held Expectations of SSD and Rethinking Implications for Systems. Myoungsoo Jung, Mahmut Kandemir. Proceedings of ACM SIGMETRICS, 2013.

Disk-Cache and Parallelism Aware I/O Scheduling to Improve Storage System Performance. Ramya Prabhakar, Mahmut Kandemir, Myoungsoo Jung. Proceedings of IEEE IPDPS, 2013.

Design of a Large-Scale Storage-Class RRAM System. Myoungsoo Jung, John Shalf, Mahmut Kandemir. Proceedings of ACM ICS, 2013.

Provisioning a Multi-Tiered Data Staging Area for Extreme-Scale Machines. Ramya Prabhakar, Sudharshan Vazhkudai, Youngjae Kim, Ali R. Butt, Min Li, Mahmut Kandemir. Proceedings of ICDCS, 2011.

IOPin: Dynamic Profiling of Parallel I/O in HPC Applications. Seong Jo Kim, Seung Woo Son, Mahmut Kandemir, Wei-keng Liao, Rajeev Thakur, Alok Choudhary. Proceedings of the 7th international workshop on parallel data storage (PDSW 2012)

IOGenie: An Automatic Parallel I/O Code Generation and Optimization framework for HPC Applications Seong Jo Kim, Jun Liu, Mahmut Kandemir (in progress)