Modeling and Predicting Disk I/O Time of HPC Applications

Mitesh R. Meswani, Michael A. Laurenzano, Laura Carrington, and Allan Snavely
San Diego Supercomputer Center, University of California, San Diego, CA, USA
{mitesh, michaell, lcarring, allans}@sdsc.edu

Abstract

Understanding input/output (I/O) performance in high performance computing (HPC) is becoming increasingly important as the gap between the performance of computation and I/O widens. In this paper, we propose a methodology to predict an application's disk I/O time while running on High Performance Computing Modernization Program (HPCMP) systems. Our methodology consists of the following steps: 1) Characterize the I/O operations of an application running on a reference system. 2) Using a configurable I/O benchmark, collect statistics on the reference and target systems about the I/O operations that are relevant to the application on the reference and target systems. 3) Calculate a ratio between the measured I/O performance of the application on the reference system with respect to target systems to predict the application's I/O time on the target systems. Our results show that this methodology can accurately predict the I/O time of relevant HPC applications on HPCMP systems that have reasonably stable I/O performance run to run while systems that have wide variability in I/O performance are more difficult to predict accurately.

1. Introduction

As the gap between the speed of computing elements and the disk subsystem widens, it becomes increasingly important to understand and model disk input/output (I/O). While the speed of computational resources continues to grow, potentially scaling to multiple peta flops and millions of cores, traditionally the growth in the performance of I/O systems has lagged well behind. Data-intensive applications that run on current and future systems will be required to efficiently process very large data sets. As a result, the ability of the disk I/O system to move data to the distributed memories can become a bottleneck for application performance. Additionally, due to the higher risk of component failure that results from larger scales, the frequency of application check-pointing should be expected to grow and put an additional burden on the disk I/O system[7].

To address this problem it is important to understand an application's I/O characteristics and be able to produce models that are capable of predicting I/O performance in current and future systems. In this research, we present such a modeling approach. Our approach consists of the following three steps: 1) Characterize an application's I/O characteristics on a reference system. 2) Using a configurable I/O benchmark, collect statistics on the reference and target systems about the I/O operations that are relevant to the application. 3) Calculate a ratio between the measured I/O performance of the application on the reference system with respect to target systems to predict the application's I/O time on the target systems without actually running the application on the target system. This *cross-platform* prediction can greatly reduce the effort needed to characterize the I/O performance of an application across a wide set of machines, and can be used to predict the I/O performance of the application on systems that have not been built. The cornerstone of this approach is that the I/O operations in the application have to be measured once on the reference system. The target systems then need only to be characterized by how well they can perform certain fundamental I/O operations, from which we can predict the I/O performance of the application on the target system.

We evaluate our methodology by predicting the total I/O time of the MADbench2[1] benchmark (representing the I/O characteristics of a suite of applications) for both the POSIX and MPIIO application programming interfaces (APIs). We make predictions on seven high performance computing (HPC) systems by running MADbench2 on a single reference system, then running Interoperable Object Reference (IOR)[3] in order to characterize the other six target systems. Our results show our methodology has prediction errors that range from 8.17% to 55.74%, with an average of 23.87%. The accuracy of our methodology depends on the system—systems which exhibit a high degree of variability of a single I/O operation are more difficult to predict. The rest of the paper is organized as follows: Section 2 describes our I/O performance prediction methodology in detail;

Section 3 describes the workloads and systems used for evaluation of the methodology; Section 4 presents the results of our evaluation; Section 5 presents conclusions and future work; and Section 6 presents related work.

2. Methodology

This section describes the modeling and prediction methodology used in our research. First, Section 2.1 describes IOR, which is a configurable I/O benchmark that is used to collect statistics about an applications I/O operations on different target systems. Section 2.2 describes the I/O operation tracing tool, which is based on the PEBIL instrumentation framework. Section 2.3 gives an overview of our modeling methodology.

2.1 IOR Benchmark

The IOR benchmark is a configurable benchmark that can be used to simulate the read and write operations of real applications. IOR can be configured for different I/O protocols, modes, and file sizes. The basic operation of IOR is reading and writing data from either a shared or exclusive file. Figure 1 shows how data is organized and transferred to the different processors when operating on a shared file. The files are represented as sequence of segments, each of which can be thought of as a variable of a time-step or different variables. Each segment is divided among N processors in units of blocks. Data from a block for a given processor is transferred in units of the transfer size and thus, represents the size of the data for each I/O call. In case of one file per processor, the blocks for each processor are stored contiguously in separate files.

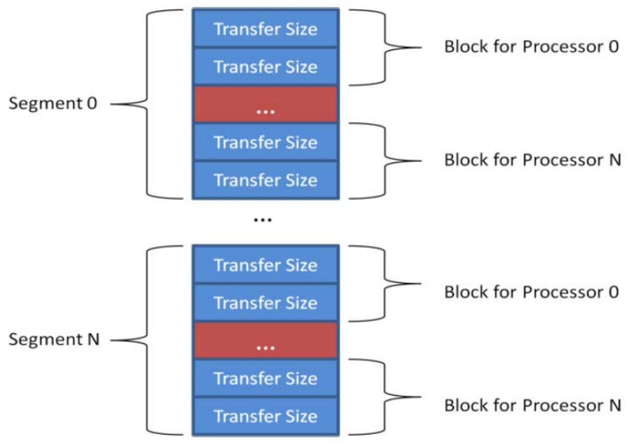


Figure 1. IOR Design for Shared File, a block is transferred in units of Transfer Size

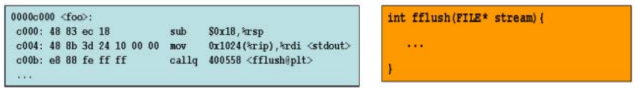
The relevant IOR parameters are shown below:

1. API: POSIX, MPIIO, HDF5, NETCDF
2. NumTasks: number of MPI processes
3. BlockSize: Size in bytes that each processor will operate on
4. FilePerProc: one file-per-processor or shared-file
5. SegCount: number of segments
6. TransferSize: the amount of data for each I/O operation
7. WriteFile/ReadFile: If set to true write/read operation is performed

2.2 PEBIL Tracing Tool

In order to collect information about the I/O activity that occurs in an application, we have developed an I/O tracing tool based on the PEBIL framework[4]. PEBIL is a static binary instrumentation toolkit that provides a mechanism for arbitrarily redirecting any function call to a target supplied by the tool writer. Using this mechanism we have built an I/O tracing tool that redirects the calls of several classes of I/O calls, which currently includes the core I/O functions of the standard C library and of the Message Passing Interface (MPI) standard. The redirection of a function is accomplished by determining which call instructions in the application code refer to that function, then by replacing that instruction to transfer control to some code generated by the instrumentation tool, which in turn can perform some tasks then call the wrapper function, as shown in Figure 2.

These calls in the original application are redirected to wrapper functions that call the original I/O function in an identical fashion to the original application then record some information about was being done in the call. Figure 2b shows an example of a simple wrapper for the `fflush` C library function. Because the I/O tracer uses binary instrumentation that operates on each call-site, as opposed to operating on the targeted function or the interface to that function, it is able to retrieve information about the state of the program, or the *context*, at the time the I/O event occurs. This is one of several advantages to taking the redirecting approach to wrapping functions, rather than taking a traditional approach to function call wrapping that may involve modifying the source code or re-linking the application to use a wrapper library, as can be done with MPI wrapper libraries[5].



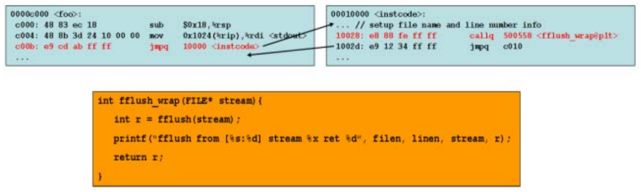
```
0000c000 <foo>:
 c000: 48 83 ec 18             sub    $0x18,%rsp
 c004: 48 8b 3d 24 10 00 00    mov    0x1024(%rip),%rdi <stdout>
 c00b: e8 88 fe ff ff          callq  400558 <fflush@plt>
 ...
```

```
int fflush(FILE* stream){

   ...

}
```

Figure 2a. Original application code example, which shows a function call to `fflush`



```
0000c000 <foo>:
 c000: 48 83 ec 18            sub    $0x18,%rsp
 c004: 48 8b 3d 24 10 00 00   mov    0x1024(%rip),%rdi <stdout>
 c00b: e9 cd ab ff ff         jmpq   10000 <instcode>
 ...
```

```
00010000 <instcode>:
... // setup file name and line number info
 10028: e8 88 fe ff ff        callq  500558 <fflush_wrap@plt>
 1002d: e9 12 34 ff ff        jmpq   c010
 ...
```

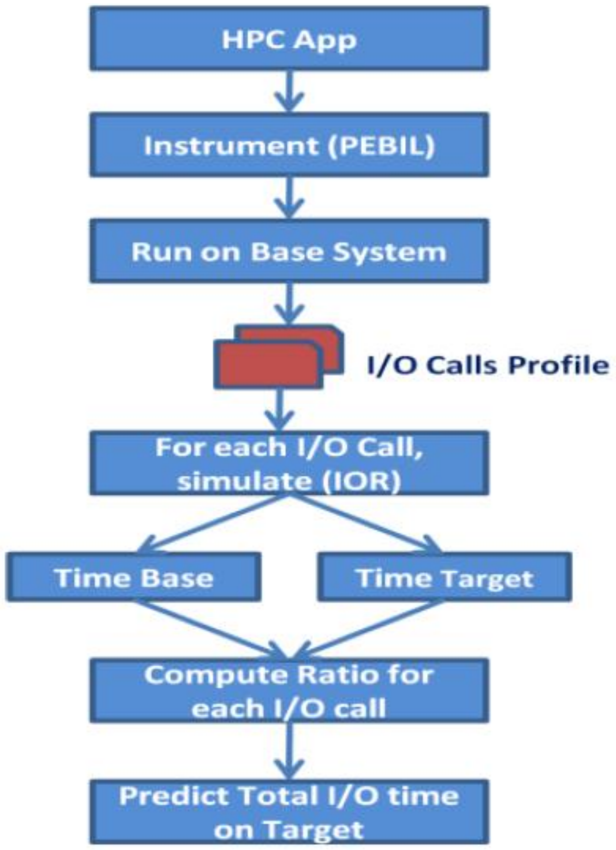
```
int fflush_wrap(FILE* stream){
   int r = fflush(stream);
   printf("fflush from [%s:%d] stream %x ret %d", filen, linen, stream, r);
   return r;
}
```

Figure 2b. Application code that has been instrumented to redirect the call from `fflush` to instead call `wrap_fflush`

Currently the I/O tracing tool presents the file and line number from whence the call originated, but in principle is limited only by the limitations of the underlying instrumentation tool. It is therefore possible to provide much more detailed information about the context of any I/O event, such as the functions that comprise the call stack or whether control is in a loop when the event occurs. By recording detailed information about the I/O calls in the application, this I/O tracing tool allows us to capture all of the information that may be necessary both for the modeling efforts discussed in this work and for future models that can utilize more information about what occurred in the application.

2.3 Modeling and Prediction

Given a reference system and target systems for which prediction is required, Figure 3 shows the modeling and prediction workflow used in our experiments. As shown in this figure, using PEBIL we first instrument all I/O calls in the application. The instrumented application is then executed on the reference system and the application's I/O profile is stored for further analysis. The profile contains the time spent by each MPI task in all I/O calls. Additionally, we also collect data that pertains to each call; for example, we collect data size for read/write calls, and for seeks we collect the seek distance. At this time; however, we have not incorporated seek distance into the modeling. Next, for each I/O call we simulate its execution using IOR. We collect the time spent by IOR running on the reference system and target systems. An I/O ratio is calculated as shown in Equation 1. This ratio is our prediction for the predicted speedup or slowdown of the application's I/O on the target system relative to the reference system. We then use these ratios, as shown in Equation 2, to predict the application's total I/O time on target systems. To calculate accuracy of our predictions, we run the application on the target systems and compare the predicted times with the actual time spent in I/O.



HPC App
Instrument (PEBIL)
Run on Base System
I/O Calls Profile
For each I/O Call, simulate (IOR)
Time Base
Time Target
Compute Ratio for each I/O call
Predict Total I/O time on Target

Figure 3. Methodology Overview

For each I/O call *i,* target system *x*, calculate ratios as follows:

$$Ratio_{i,x} = \frac{IORTime_{i,x}}{IORTime_{i,reference}} \tag{1}$$

For each target system *x*, calculate predicted total time spent in I/O as follows:

$$PredictedTime_{i,x} = \sum_{i=0}^{n} Ratio_{i,x} * ApplicationTime_{i.reference} \tag{2}$$

3. Workloads and Systems

This section describes the workloads used in our experiments in Section 3.1 and experimental systems in Section 3.2.

3.1 Workload

MADbench2 is a benchmark that is derived from Microwave Anisotropy Dataset Computational Analysis Package (MADCAP) Cosmic Microwave Background (CMB) power spectrum estimation code[2]. CMB data is a snapshot of the universe 400,000 years after the big bang. MADbench2 retains the most computationally challenging aspect of MADCAP, to calculate the spectra from the sky map. MADbench2 retains the full complexity of computation, communication, and I/O, while removing the redundant details of MADCAP. Hence, MADbench2 permits performance studies of HPC systems under a real application's workload. MADbench2s' I/O characteristics are similar to those found in applications such as FLASH, NWCHEM, MESKIT[6], and thus the modeling approach for MADbench2 may be largely applicable to those applications as well.

MADbench2 consists of four steps:

1. For each bin, recursively build a Legendre polynomial-based CMB signal pixel-pixel matrix, writing it to disk *(loop over calculate, write)*.
2. Form and invert the full CMB signal+noise correlation matrix *(calculate/communicate)*.
3. From the disk, read each CMB signal correlation component matrix, multiply it by the inverse CMB data correlation matrix (using PDGEMM), and write the resulting matrix to disk *(loop over read, calculate/communicate, write)*.
4. Read each of the result matrix and then calculate the trace of their product *(loop over read, calculate, communicate)*.

Note that step 2 does not involve any I/O, while only step 3 requires both reading and writing. The nature of the large calculations required for CMB data means that the large matrices used do not fit in memory; hence the benchmark uses an out-of-core algorithm. Each processor requires enough memory to fit five matrices at a given time. MADbench2 stores the matrices to disk when they are first calculated and reads them when required.

We configured MADbench2 to run in I/O mode. In this mode, calculations are replaced by busy-work that can be configured by the user. The relevant configuration parameters available are shown below:

1. NPIX: The dimension of the matrix; size of the matrix is NPIX*NPIX
2. NBIN: Number of matrices
3. IOMETHOD: POSIX or MPIIO
4. IMODE: Synchronous or Asynchronous
5. BWEXP: the component of busy-work α, this translates to O(NPIX$^{2\alpha}$) operations, tuning this value to less than 0.1, lets MADbench2 to spend most of its time in I/O
6. NOGANG: the number of gangs
7. FBLOCKSIZE: each file read/write happens at an integer number of these blocks into a file
8. RMOD/WMOD: the number of concurrent readers/writers

The most critical parameter that determines disk I/O performance is the file size. Typically UNIX systems have a memory space called the buffer cache, which can be used to store the most recently accessed pages of a file in memory. This allows the system to achieve high-bandwidth while performing disk I/O and significantly improves application execution time. However, CMB applications process large data sets and it's expected that their data sets will not fit in memory, and thus the need for out-of-core algorithms. Hence, we choose NBIN and NPIX so that the data does not fit in memory for any of the nodes for the systems we used in this research and described in Section 3.2. To avoid unusual speed-ups some systems may experience from the use of local disk, we chose to use a single shared-file.

The configuration parameters used in this research are shown in Table 1. The total file size is NPIX * NPIX * NBIN * size of real=50,000*50,000*8*8=160GB. We ran this benchmark using 64 processors, and hence, each processor reads/writes 2.4GB data, which is more than the available memory to an application for any processor core on our experimental systems. We set the value of BWEXP, busy-work, low so that I/O dominates run-time. For simplicity, we chose to experiment only with Synchronous I/O to avoid complications with calculating computational overlap; however, we may do so in the future. Similarly, again for simplicity, we set the number of concurrent readers/writers and number of gangs to 1; however, in the future we shall try different values.

Table 1. MADbench2 configuration

MADbench2 Parameter Name	Value
NPIX	50,000
NBIN	8
IOMETHOD	POSIX, MPIIO
IOMODE	Synchronous
BWEXP	0.01
NOGANG	1
FBLOCKSIZE	1MB
RMOD/WMOD	1

Table 2. Experimental systems architecture summary

Machine Name	File System	Interconnect Compute to I/O nodes	Processor	Total Disk *(TB)*
Jade – reference	Lustre	Seastar2	Opteron	379
Sapphire	Lustre	Seastar2	Opteron	372
Diamond	Lustre	Infiniband	Xeon Nehalem	830
Pingo	Lustre	Seastar2	Opteron	100
Mana	Lustre	Inifiniband	Nehalem	400
Einstein	Lustre	Not known	Opteron	516
Jaguar	Lustre	Seastart2	Opteron	600

3.2 Experimental Systems

In general, the disk I/O architecture of these systems is quite similar. All systems have the Lustre parallel file system, and have separate sets of I/O and compute nodes. All the nodes are connected over the network, and all disk I/O requests are serviced by the I/O nodes. The I/O nodes may have a combination of locally attached disks or remote storage devices connected on the backend of the I/O nodes over a different dedicated network. Lustre separates bulk I/O requests and metadata requests, (open, close, permissions) and sends bulk I/O requests to I/O nodes and metadata requests to metadata servers. Table 2 highlights the systems used for our experiments, Jade is the reference system in our modeling, and the remaining are target systems. A brief overview of each system is provided in the sub-sections below.

3.2.1 Jade – Reference System

Jade is a Cray XT4 system, and has a total of 2,152 compute nodes. Each node runs Compute Node Linux (CNL) and has one quad-core AMD Opteron processor and 8GB of main memory. All nodes are connected in a three-dimensional (3D) torus using a HyperTransport link to a Cray Seastar2 engine. The system has a total of 379TB fiber-channel redundant array of inexpensive disks (RAID) disk space that is managed by a Lustre file system.

3.2.2 Sapphire

Sapphire is a Cray XT3 system, and has a total of 4,160 nodes. Each node has one dual-core AMD Opteron processor and 4GB of main memory. There are a total of 4,096 compute nodes available for computation, and each node runs CNL. All nodes are connected in a 3D torus using a HyperTransport link to a Cray Seastar engine. The system has a total of 372TB fiber-channel RAID disk space that is managed by a Lustre file system.

3.2.3 Diamond

Diamond is an SGI Altix ICE 8200LX, and is a single-plane DDR 4X InfiniBand hypercube super-cluster. Diamond has 1,920 compute nodes, each of which runs CNL. Each node has two quad-core Intel Xeon Nehalem processors and 24GB of main memory. All nodes are connected via a single-rail DDR 4X InfiniBand (IB) interconnect. This system has a total of 830TB of Infiniband-connected RAID disk space that is managed by a Lustre file system.

3.2.4 Pingo

Pingo is a Cray XT5 system with 432 compute nodes. Each node has two quad-core AMD Opteron processors and 32GB of main memory. All nodes are connected by Cray Seastar2 compute engine. This system has a total of 100TB of Infiniband-connected RAID disk space that is managed by a Lustre file system.

3.2.5 Mana

Mana is based on Dell's M610 series with 1,152 compute nodes. Each node has two quad-core Intel Nehalem processors with 24GB of main memory. All nodes are connected by DDR Infiniband. This system has 400TB of disk space managed by a Lustre file system.

3.2.6 Einstein

Einstein is a Cray XT5 system, and has 1,592 compute nodes. Each node runs CNL and has two quad-core AMD Opteron processors and 16GB of main memory. This system has a total of 516TB of disk space that is managed by a Lustre file system.

3.2.7 Jaguar

Jaguar has both a Cray XT5 and an XT4 partition. In this research we use the Cray XT4 partition that has 7,832 compute nodes. Each node runs CNL and has a quad-core AMD Opteron processors and 8GB of main memory. The nodes are connected by a SeaStar2 router and they use the same network to access the file system. The XT4 partition has a total of 600TB of disk space that is managed by a Lustre file system.

4. Results

For each target system, ratios and predictions were calculated using Equations 1 and 2 respectively. Since wall-clock time of an application may differ from run to run, each application was executed five times on the machine and an average run-time was used in these two equations. Thus, the average run-times of IOR were used to predict the run-time of MADbench2. To calculate accuracy, the predicted time is compared against the average run-time of five executions of MADbench2 on the target system. Accuracy for each target system *x* is calculated using Equation 3.

$$Accuracy_x = 100 * \frac{PredictedTime_x - ActualTime_x}{ActualTime_x} \qquad (3)$$

As described in Section 3.1, MADbench2 has four main I/O operations: two reads and two writes. During an

I/O call, 312MB of data are accessed from the disk. A total of 160GB of data are evenly divided among 64 MPI tasks, and thus each MPI tasks uses 2.4GB of data. From the I/O trace of MADbench2 we observe that a total of 16 read calls are made and 16 write calls are made. As described in Section 3.1, MADbench2 in step 1 writes 8 matrices, and then in step 3 reads them back and again writes them to disk, and finally reads them again in step 4. Thus, as expected, the I/O trace confirms that each task performs 16 reads and 16 writes. Additionally, the I/O trace also confirms that each read/write call operates on 312MB of data. In I/O mode MADbench2 spends most of its time doing I/O, and computation time was configured to be negligible. Hence, the time spent in steps 1, 3, and 4 correspond to time spent in the two read and two write calls, which are reported by the benchmark. In absence of such information, we can easily calculate this information using timers that are written into the I/O call wrappers; however, for this benchmark that was not needed.

Table 3 shows the prediction accuracy, which is calculated using Equation 3. In this table, the predicted and actual times used in Equation 3 are calculated using an average of 5 runs each of IOR and MADbench2 on reference and target systems. A negative value indicates that actual I/O time was greater than the predicted time, and a positive value indicates that the actual I/O time was less than the predicted time. For example, consider prediction for Sapphire for POSIX API, errors are 18.27%, 1.61%, and 12.75% for reads, writes, and total I/O time respectively. The highest prediction accuracy for reads was 3.76% error for Jaguar using MPIIO, for writes was 0.00% error for Jaguar using POSIX, and for total I/O time was 8.17% error for Jaguar using POSIX. On an average the absolute prediction error is 41.16% error for reads, 12.99% error for writes, and 23.87% error for total I/O.

Table 3. Accuracy for MADbench2

Machine Name	API	Accuracy of Read Time %	Accuracy of Read Time %	Accuracy of Read Time %
Diamond	POSIX	−68.68	17.81	−45.38
Sapphire	POSIX	18.27	1.61	12.75
Pingo	POSIX	−62.82	2.38	−16.85
Einstein	POSIX	37.60	11.46	13.69
Mana	POSIX	41.23	24.94	30.06
Jaguar	POSIX	16.46	***0.00***	***8.17***
Diamond	MPIIO	−64.55	8.42	−42.87
Sapphire	MPIIO	16.36	5.07	12.55
Pingo	MPIIO	−70.61	−1.99	−22.08
Einstein	MPIIO	39.02	5.06	17.59
Mana	MPIIO	54.55	56.34	55.74
Jaguar	MPIIO	***3.76***	−20.83	−8.65

To understand why our predictions may be more accurate for systems such as Jaguar, and relatively poorer for systems such as Diamond and Mana, we calculated the relative standard deviations (RSD) of 5 run times each of IOR and MADbench2. RSD is calculated as ratio of standard deviation divided by mean and expressed as a percentage. This statistic allows us to meaningfully compare different systems, which identify those that have significant variability in disk I/O times from run to run.

The results of RSD analysis are shown in Table 4. For example, the first row shows RSD on Jade using POSIX and shows that: 1) for MADbench2 RSDs are 4.32%, 6.15%, and 4.78% for reads, writes, and total I/O respectively; and 2) for IOR RSDs are 5.94%, 9.52%, and 6.48% for reads, writes, and total I/O respectively. Hence, for Jade, the base system, both IOR and MADbench2 have small percentage deviation in I/O time across different runs. Next, consider Sapphire for both MPIIO and POSIX APIs, the RSDs for reads of MADbench2 and IOR are nearly the same and very small, and in fact they are less than 10% each. Consequently our predictions for Sapphire are quite accurate. In contrast consider Diamond using MPIIO, RSDs of read for MADbench2 and IOR are very different, and especially note the large RSD of Diamond IOR read and, hence our predictions have more error. A similar analysis applies to Mana which also has high variability in I/O performance.

Table 4. Relative Standard Deviations (RSD) for reads/writes for IOR and MADbench2

		MADbench2			IOR		
Machine Name	API	RSD[1] Read %	RSD Write %	RSD Total IO %	RSD Read %	RSD Write %	RSD Total IO %
Jade - reference	POSIX	4.32	6.15	4.78	5.94	9.52	6.48
Diamond	POSIX	5.06	12.34	3.03	30.12	4.90	9.80
Sapphire	POSIX	9.41	8.48	8.48	4.64	6.23	4.90
Pingo	POSIX	6.55	4.01	4.50	14.62	12.50	12.23
Einstein	POSIX	6.89	4.34	5.19	6.12	4.33	4.84
Mana	POSIX	1.61	4.37	2.77	77.01	19.70	21.26
Jaguar	POSIX	7.61	15.90	11.25	23.34	32.51	20.30
Jade - reference	MPIIO	3.79	5.56	4.19	7.21	9.52	7.88
Diamond	MPIIO	5.60	16.96	2.05	38.54	5.44	10.75
Sapphire	MPIIO	4.26	5.23	4.55	7.49	6.39	6.84
Pingo	MPIIO	5.44	3.22	3.27	15.40	3.32	1.89
Einstein	MPIIO	10.27	12.05	11.09	3.73	6.56	4.33
Mana	MPIIO	2.02	4.67	3.13	101.37	45.99	41.11
Jaguar	MPIIO	6.07	16.15	9.44	25.17	5.93	13.36

[1]RSD – Relative Standard Deviation = (standard_deviation/mean)*100

The RSD difference between IOR and MADbench2 indicate the variation experienced by IOR and MADbench2 when running on the system. If the variation is similar and small then our predictions are close and vice-versa. This is not surprising, the I/O time of an application is dependent on a number factors such as available memory to cache files, contention for the disk I/O system, and synchronization delays which can change the I/O events rate. These factors may need to be modeled to accurately capture variation and are not currently part of IOR, but in the future work we discuss these and other factors. At the same time, one can argue that repeatability and stability of performance of I/O run to run is a nice attribute of a well-provisioned file system, so in a sense a measure of predictability is also a measure of quality of the I/O system.

5. Conclusions and Future Work

In this paper we presented a methodology to predict disk I/O time of HPC applications. Our method used a configurable I/O benchmark to measure speed-up ratios of each I/O operation of an application, and used them to predict an applications total I/O time. Our evaluation showed that reasonable accuracy maybe obtained by using this simple model, and in the best case our prediction error is only 8.17%.

However, since our model and real run-times experience variability in the inputs (IOR) as well as in the target (Madbench2), our prediction error for total I/O time in the worst case is as high as 55.74%. To make our predictions more accurate, we would like to consider other factors that effect an applications I/O time. Some of the factors that we would like to model are as follows: 1) File caching: The ability to cache files in the memory sub-system significantly speeds up disk I/O. 2) Seek distance: the distance between two consecutive calls affects the distance that disk head must travel before reading/writing the data. Consequently, the greater the distance the more the time to service the I/O call. 3) Concurrent reads/writes: In parallel applications, multiple readers/writers could be accessing a single file, thus the mix of I/O calls may determine the service time for each call. 4) Contention: The contention for disk systems by other applications affects the I/O time each application receives, and thus is important to model. 5) Synchronization delays: In parallel applications, barrier synchronization and other data dependencies may change the rate at which I/O calls are made, and thus may affect disk I/O time.

At the same time we want to suggest that if the I/O performance of the same test or application varies a lot from run to run on the same system, then in a sense high-error in prediction is to be expected—in other words, predictability is a nice feature of a well-designed I/O system, while those that are not predicted accurately by simple models are probably also insufficiently engineered to meet the workload demands being placed upon them.

6. Related Work

Related work has pursued I/O modeling and prediction by using script-based benchmarks to replay an applications causal I/O behavior[8,9], or using parameterized I/O benchmarks[10] to predict run-time on the target system. Our modeling approach differs from the related work by using parameterized benchmarks to compute *speed-up ratios* on target systems for each call and use that to predict an applications I/O time.

Pianola[8] is a script-based I/O benchmark that captures causal information of I/O calls made by a sequential application. The information is captured by a binary instrumenter that, for each call, captures wall clock time of the call, the time spent servicing the call, and arguments passed to the call. Using this information, a script is constructed which has sufficient information to replay an application's I/O calls and time between two successive calls. Additionally, the script is also augmented to simulate the memory used by an application between calls. A replay engine can then use this script to replay an application's I/O behavior on any platform.

Like Pianola[8], TRACE[9] is a script-based I/O benchmark that simulates an I/O behavior of an application using causal information about the I/O calls. Unlike Pianola, TRACE used interposed I/O calls to capture information regarding I/O calls. TRACE is targeted for parallel applications, and thus captures I/O events for each parallel task. In addition to I/O events, for each task, TRACE includes information related to synchronization delays and computation time. Using this information a replayer simulates the I/O characteristic of each task of the original application.

In Reference 10, IOR was used to simulate the I/O behavior of HPC applications. In this research, an application's I/O behavior is first obtained by code and algorithm analysis, and then this information is used to prepare inputs for the IOR benchmark. Next, IOR is run on the target system to predict the *actual* I/O time of an application. Unlike in Reference 10, we use IOR to capture speed up ratios for prediction.

While IOR provided the widest range of parameters, there are other I/O benchmarks[11–13] that provide a subset of the parameters. Similarly there are a number of binary instrumentation toolkits besides PEBIL that can be used to instrument binaries. For example, DyninstAPI[14] provides both dynamic and static instrumentation, while PIN[15] provides dynamic instrumentation only. We chose to use PEBIL to instrument binaries because of the low overhead that it adds onto an application's execution time as compared to other tools[4].

Acknowledgements

This work was supported in part by the DoD High Performance Computing Modernization Program and by National Science Foundation (NSF) under NSF OCI award #0951583 entitled "I/O Modeling EAGER".

References

1. Borrill, J., L. Oliker, J. Shalf, H. Shan, and A. Uselton, "HPC global file system performance analysis using a scientific-application derived benchmark." *Parallel Computing*, 35, 6, pp. 358–373, 2009.

2. Borrill, J., "MADCAP: The microwave anisotropy dataset computational analysis package." *Fifth European SGI/Cray MPP Workshop*, 1999.

3. IOR, http://sourceforge.net/projects/ior-sio/.

4. Laurenzano, M.A., M.M. Tikir, L.C. Carrington, and A.E. Snavely, "PEBIL: Efficient Static Binary Instrumentation for Linux." *Proceedings of the IEEE Symposium on Performance Analysis of Systems and Software*, White Plains, NY, March 2010.

5. Tikir, M.M., M.A. Laurenzano, L.C. Carrington, and A.E. Snavely, "PSiNS: An Open Source Event Tracer and Execution Simulator for MPI Applications." *Proceedings of Euro-par*, August 2009.

6. Seelam, R.S., A. Kerstens, and P.J. Teller, "Throttling I/O Streams to Accelerate File-IO Performance." *Proceedings of HPCC*, 2007.

7. Oldfield, R.A., S. Arunagiri, P.J. Teller, S. Seelam, M.R. Varela, R., Riesen, and P.C., Roth, "Modeling the Impact of Checkpoints on Next-Generation Systems." *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, 2007.

8. May, J., "Pianola: A script-based i/o benchmark." *Proceedings of the 2008 ACM Petascale Data Storage Workshop (PDSW 08)*, November 2008.

9. Mesnier, M.P., M. Wachs, R.R. Sambasivan, J. Lopez, J. Hendricks, G.R. Ganger, and D. O'Hallaron, "Trace: Parallel Trace Replay with Approximate Causal Events." *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, February 2007.

10. Shan, H., K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark." *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC)*, 2008.

11. IOzone file system benchmark, http://www.iozone.org.

12. High performance I/O, http://cholera.ece.northwestern.edu/~aching/research webpage/hpio.html.

13. SPIOBENCH: Streaming Parallel I/O Benchmark, http://www.nsf.gov/pubs/2006/nsf0605/spiobench.tar.gz, 2005.

14. Buck, B. and J.K. Hollingsworth, "An API for runtime code patching." *International Journal of High Performance Computing Applications*, 2000.

15. Luk, C.K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.