DataStager: Scalable Data Staging Services for Petascale Applications

Hasan Abbasi
College of Computing
Georgia Institute of Technology
Atlanta, GA
habbasi@cc.gatech.edu

Matthew Wolf*
College of Computing
Georgia Institute of Technology
Atlanta, GA
mwolf@cc.gatech.edu

Greg Eisenhauer
College of Computing
Georgia Institute of Technology
Atlanta, GA
eisen@cc.gatech.edu

Scott Klasky
Oak Ridge National Laboratory
Oak Ridge, TN
klasky@ornl.gov

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA
schwan@cc.gatech.edu

Fang Zheng
College of Computing
Georgia Institute of Technology
Atlanta, GA
fzheng@cc.gatech.edu

ABSTRACT

Known challenges for petascale machines are that (1) the costs of I/O for high performance applications can be substantial, especially for output tasks like checkpointing, and (2) noise from I/O actions can inject undesirable delays into the runtimes of such codes on individual compute nodes. This paper introduces the flexible 'DataStager' framework for data staging and alternative services within that jointly address (1) and (2). Data staging services moving output data from compute nodes to staging or I/O nodes prior to storage are used to reduce I/O overheads on applications' total processing times, and explicit management of data staging offers reduced perturbation when extracting output data from a petascale machine's compute partition. Experimental evaluations of DataStager on the Cray XT machine at Oak Ridge National Laboratory establish both the necessity of intelligent data staging and the high performance of our approach, using the GTC fusion modeling code and benchmarks running on 1000+ processors.

Categories and Subject Descriptors

C.5.1 [**COMPUTER SYSTEM IMPLEMENTATION**]: Large and Medium ("Mainframe") Computers

General Terms

Performance, Experimentation, Design

*Joint appointment at Oak Ridge National Laboratory, Oak Ridge TN

HPDC'09, June 11–13, 2009, Munich, Germany.

Keywords

I/O, WARP, GTC, XT3, datatap, XT4, Staging

1. INTRODUCTION

The increase in compute partition sizes on petascale machines (headed toward multi-petascale) accompanied by an increased emphasis on I/O performance is poised to drive the adoption of *application aware data services* for high performance computing. Such services can carry out actions ranging from data capture and mark-up, to reorganization, formatting[24] or compression [19], to filtering and/or preprocessing in preparation for visualization [37] or analysis[36, 1]. This paper describes data staging services that provide scalable, low overhead, and asynchronous data extraction for parallel applications running on petascale machines.

Data staging services use asynchronous methods for data extraction from compute nodes. Therefore, instead of blocking and waiting for data output to complete, they allow the application to overlap data extraction with its computational actions. As shown in Section 3 of this paper, this results in significantly reduced overhead for data output, even when compared to minimal output methods like POSIX I/O. As applications begin to utilize tens of thousands of processing cores the cost of some operations increases to untenable levels due to their lack of scalability. In particular, management of metadata consistency with large I/O can be a performance bottleneck. By delegating to data staging service some of the file semantics (such as a global shared file creation and management), the compute application is not unduly burdened by the synchronization requirements of a normal filesystem operation. As a more complex example, consider data reduction through volume averaging of a large mesh during the output of data from a large simulation, such as the particle-in-cell code GTC [29] which serves both as a motivating application and a key evaluation target (see Section 3).

Data staging services leverage the penetration of RDMA-based network infrastructures such as Infiniband, iWarp,

Quadrics Elan, IBM BlueGene Interconnect, and Cray SeaStar in modern supercomputing infrastructures. Specifically, these provide a new opportunity for the creation of a staging service that shifts the burden of synchronization, aggregation, collective processing, and data validation to a subset of the compute partition – the staging area. Data processing in the staging area can then be used to provide input to a variety of extensible service pipelines such as those needed for formatted data storage on disk [24], for data movement to secondary or even remote machines [19], or for data visualization[37].

Meaningful actions on data in the staging area are made possible by the fact that data staging services recognize and make use of the structure of the data captured on compute nodes. In our work, instead of serving as a simple byte transfer layer from compute to disk storage, data staging services use the FFS data format library to cast output into a self-describing binary form suitable for efficient online manipulation. With such structure information, data staging services can be customized for each type of data produced by the petascale code (e.g., for particle vs. mesh data in the GTC fusion modeling application). Customization is attained through dynamic code generation for unmarshalling and manipulating data, providing low overhead reflection and inspection of data and providing a lightweight infrastructure for stream-based data processing. Using FFS also enables inline data extension, where the data is written to disk with generated metadata attributes that allows for more intelligent reading algorithms.

The specific set of data staging actions and behaviors described and evaluated in this paper are those that pertain to I/O performance and overheads. The DataStager infrastructure follows from a set of developments in server-side I/O[15][32], and it enables the implementation of several of the current dominant extraction patterns from those works. In particular, this paper evaluates and contrasts state-aware with concurrency constrained (i.e., constant slow draining) approaches, as well as with more familiar blocking Posix I/O. Total throughput, the degree of blocking in the application and benchmark codes, and also application perturbation are measured for all of the I/O configurations.

The contributions of this work are its demonstrations of reduced overhead and increased I/O scalability attained from two key innovations: (1) moving from methods for raw byte-transfer to methods for moving data objects meaningful to the application, coupled with (2) using alternative and application phase aware methods for scheduling object movements to attain desired performance properties, such as reduced perturbation on application performance. DataStager differs, therefore, from other approaches that try to hide raw data staging functionality below filesystem APIs [12][23] or at the level of the block I/O subsystem [3]. We justify this new approach not only with improved performance and reduced overheads, but also with increased extensibility in terms of the ability to utilize the ever-increasing computational resources of petascale machines for assisting in the output (processing) needs of high performance applications.

DataStager's experimental evaluation uses both a synthetic microbenchmark and a scientific application, the GTC fusion modeling code running on the Oak Ridge National Laboratory's Cray XT Jaguar machine. The DataStager framework is implemented upon Cray portals, which is the native RDMA interface for the XT, and upon uverbs, the native RDMA for Infiniband. DataStager, therefore, operates on both Cray machines and on IB-based cluster engines. This paper reports experimental runs on 1000+ processors on the Cray XT to showcase the scalability and performance expectations on this class of machine and to demonstrate the need for the DataStager approach. In fact, performance results show that proper management of data extraction can maintain I/O overhead at less than 4% even as we scale the application to 2048 cores. For 1024 cores the overhead can be maintain at almost 2%. This is a substantial improvement compared to synchronous I/O methods such as POSIX and MPI-IO which show an overhead of more than 25% for 2048 cores and more than 10% for 1024 cores when configured for identical output periodicity.

In the remainder of this paper, we discuss our goals for the DataStaging service framework in Section 2 and details on the design in Section 2.1. We discuss the implications of an asynchronous data output service and describe in detail some of the scheduling systems we have implemented in Section 2.2. The experimental evaluation is described further in Section 3 where we study the impact of the number of staging nodes on the maximum achievable throughput in Section 3.1 and the impact of multiple data extraction strategies on GTC performance in Section 3.2. Finally we conclude with a discussion of related work in this area and a discussion on the direction of our future research.

2. DATA STAGING SERVICES

Asynchronous methods are known to help in addressing the I/O needs of high performance applications. In [6], for instance, the authors show that when asynchronous capabilities are available, synchronous I/O can be outperformed by up to a factor of 2. The studies performed in our paper use a novel, high performance data transport layer developed by our group, termed DataStager. DataStager is comprised of a library called DataTap and a parallel staging service called DataStager. In order to support easy inclusion of best practice, scalable I/O in high performance codes, others within our research collaboration have implemented the ADIOS I/O portability layer [24], which supports both blocking and asynchronous I/O modules. DataTap interfaces with the ADIOS API in order to keep application level code changes to a minimum and to enable the user to determine the transport of choice at runtime.

One mechanism that has been used to manage asynchronous communication is server-directed I/O[27, 28]. This is particularly useful in high performance architectures where a small partition of *I/O nodes* service a large number of compute nodes. The disparity in the sizes of the partitions, coupled with the bursty behavior of most scientific application I/O [25] , can lead to resource exhaustion on the I/O nodes. In server-directed I/O, the data transfers and hence the resources, are controlled by the I/O nodes, allowing smoother accesses. Such techniques have been used for both large cluster filesystems [27] and for direct to disk I/O [22]. When asynchronous communication in an RDMA environment is added, server control becomes doubly important. Specifically, in addition to managing the resources, the server control of the data transfer allows the application to progress without actively *pushing* the data out. Instead, the server *pulls* the data whenever sufficient resources are available. Under ideal conditions, the rate at which the server pulls the data – the ingress throughput – is equal to the rate

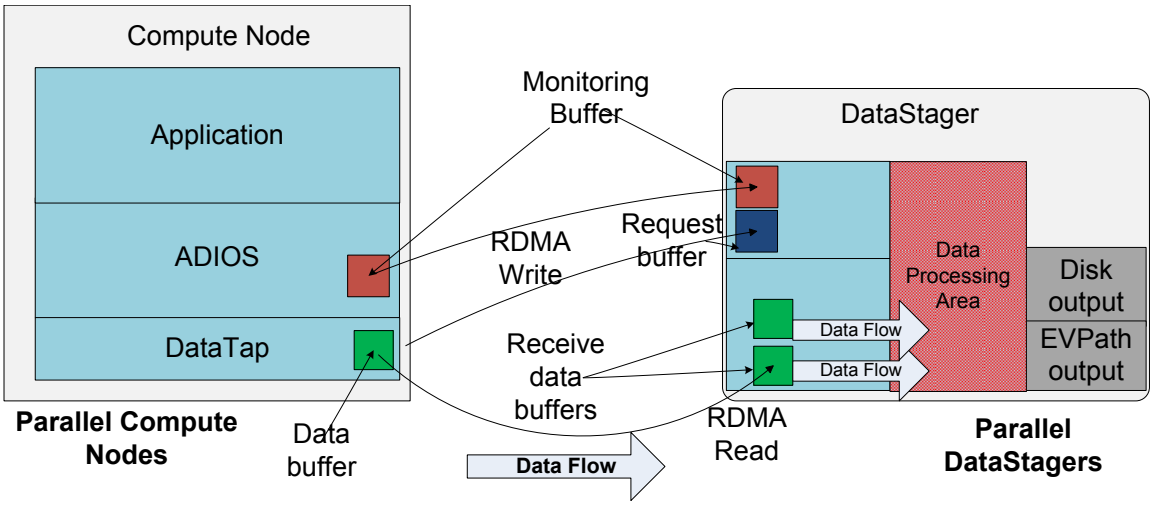


Figure 1: *DataStager Architecture.*

at which the server retires the data – the egress throughput. The ability of the server to satisfy bursty ingress requests will naturally be bound by the interconnect bandwidth between the I/O node and the compute partition.

We address the problems of scaling application I/O to petascale and of the need for runtime understanding and analysis of data with the following:

1. **Minimal Runtime Impact**. To ensure that asynchronous I/O has low runtime impact, the DataTap client library copies data into a fixed and limited amount of buffer space on compute nodes. Minimal bookkeeping is performed to maintain information about buffer status and availability.
2. **Flexibility in I/O via binary data formats**. DataTap marks up its binary data using an efficient, self-describing binary format, FFS [9, 18]. This makes it possible for binary data to be inspected and modified in transit [35], and it enables the association of graph-structured data processing overlays, termed I/OGraphs [1]. With such overlays, I/O can be customized for a rich set of backend uses, including online data visualization, data storage, and transfer to remote sites, including with standard methods like GridFTP [10].
3. **Compatibility with ADIOS**. The ADIOS interface is a platform-customizable mechanism for I/O in scientific applications and provides a uniform I/O interface for application developers. The ADIOS library provides methods for some of the most common interfaces, which allows each run of the application to be optimized for a specific hardware/software architecture [24]. DataTap has been interfaced to ADIOS, thereby providing a novel transport supporting asynchronous managed I/O and allowing the users of an application to select the asynchronous method at runtime.

The DataStager-DataTap system was initially developed on the Cray XT class of machines using the Portals programming interface [8, 7]. We have also implemented a version using the Infiniband uverbs interface; performance evaluation of the infiniband version is included in [1]. It is noteworthy to mention that like all asynchronous I/O efforts, the DataStager can only service applications that have sufficient local memory space to buffer the output data.

2.1 Design

DataStager has two different elements: the 'DataTap' client library and the 'DataStager' processes. The DataTap client library is co-located with the compute application. It provides the basic methods required for creating data objects and transporting them, and it may be integrated into higher level I/O interfaces such as the new ADIOS interface used by an increasing number of HPC codes[24]. The DataStager processes are additional compute nodes that provide the data staging service to the application. The actual data output to disk, data aggregation, etc. are performed by the DataStagers. The combined libraries work as a request-read service, allowing the DataStagers to control the scheduling of actual data transfers.

Figure 1 describes the DataStager architecture. Upon application request, the compute node marks up the data in FFS format (see [17] for a detailed description of PBIO, an earlier version of FFS) and issues a request for a data transfer to the server. The server queues the request until sufficient receive buffer space is available. The major costs associated with setting up the transfer are those of allocating the data buffer and copying the data; they are small enough to have little impact on overall application runtime. When the server has sufficient buffer space, an RDMA read request is issued to the client to read the remote data into a local buffer. This data is queued in the DataStager for processing or directly for output.

2.2 Managing data transfers

DataStager's scheduling service, implements server directed I/O. Use of server-side I/O allows us to explore novel methods of scheduling data transfers as part of the service. We have designed four schedulers in order to evaluate their ability to enhance functionality, to provide improved performance, and to reduce perturbation for the application.

1. a constant drain scheduler,
2. a state-aware congestion avoidance scheduler,
3. an attribute-aware in-order scheduler, and
4. a rate limiting scheduler.

DataStager uses resource aware schedulers to select requests for the RDMA service. Selection of a request from the transfer queue is based on the following:

```
Node(R) is the originating node for request R;
Size(R) is the size of the I/O request;
if Node(R) is waiting for completion then
    return TRUE;
else
    t^n_start, t^n_end are the start and end time for compute
    phase n;
    t_current is the current time;
    t_iter is the estimated width of a single iteration for
    Node(R);
    t_request is the time at which the request was issued;
    Δt = t_current - t_request ;
    Δiter = floor(Δt / t_iter) ;
    foreach compute phase, i in Node(R) do
        if t_current is between t^i_start AND t^i_end then
            return TRUE;
        end
    end
    return FALSE;
end
```

Algorithm 1: The Phase aware scheduler determines whether the application is in the compute phase for the DataStager to start the transfer

- Memory Check. A check is performed to determine whether there is sufficient buffer space available to service the request. This check ensures that the DataStager does not issue unbuffered reads and suffer from resource exhaustion.
- Waiting Check. A node may issue an asynchronous I/O request and then block for completion after a period of computation. If a node is currently blocked waiting for a request to complete, the request should be serviced as soon as possible to minimize the performance penalty.
- Scheduler Check. A schedule request is made to the scheduling module for each transfer request. The transfer is only initiated if all the scheduler modules indicate viability. This enables the DataStager to *stack* schedulers in order to fulfill multiple resource allocation policies while also simplifying the development of new schedulers.

Once all schedulers have agreed to issue a transfer request, it is serviced in two parts. First, an RDMA read request is issued to the originating node. Due to the latency of request completion and because available buffer space is usually larger than a single request size, multiple requests may be serviced simultaneously. This overlapping enables DataStager to complete all requests faster. Once the RDMA read is completed, an upcall is made to the staging handler. The handler will then process the message according to the configured policy – direct write to disk, network forwarding, further processing, and so on. The incoming data is in FFS format allowing the use of FFS's reflection interface to query the data block and perform operations such as aggregation and filtering in the data processing area without making a copy. The data can also be published with the EVPath [16] event transport for further processing as part of an application specific I/O pipeline [35].

2.2.1 Continuous Drain scheduler

The Continuous Drain (CD) scheduler is designed to provide maximal usage of the buffering available to the staging area. As soon as buffer space is available, an RDMA read call is issued. The throughput for this scheduler is limited by the ingress and egress throughputs, the rate at which the staging area processes data and the amount of buffer space available in the DataStager. However, it also creates a large impact on the performance of the application (and hence also secondary effects on the ingress throughput). For large cohort sizes (i.e., a large number of clients), the strategy of draining the data as fast as possible can substantially perturb the time taken to complete intra-application communication, particularly large collective calls like MPI_ALLTOALL. In fact, the resulting overhead has an impact on performance that dwarfs the time spent waiting for synchronous data transfers to complete!. Interestingly, despite that level of perturbation, the CD scheduler can yield good performance in cases where an application does not rely on collective global communication or uses asynchronous MPI communication.

2.2.2 Phase-aware congestion avoidance scheduler

As stated earlier, the use of asynchronous methods for data transfer can reduce or eliminate the blocking time experienced by HPC codes using synchronous methods for I/O. A resulting new problem is one of potential perturbation in communication times experienced by tightly coupled and finely tuned parallel application. This is because of the overlap of intra-application communication (e.g., MPI collectives) with the background transfer of output data that uses the same interconnect. Interestingly, this phenomenon is not generally observed for smaller scale parallel codes (e.g., up to a hundred nodes), but as the application scales to larger machines such as the Cray XTs (i.e., to multiple hundreds of nodes and above), it can significantly impact the performance of intra-application communications and thus, of the application itself.

The contention caused by multiple nodes using the interconnect can significantly decrease communication performance. Although the increase in perturbation is not surprising, as we scale to more than 512 nodes we observe that the total perturbation cost is far greater than the that of simply blocking for I/O. Moreover we find that the perturbation caused by the background transfer of data is not limited to the asynchronous DataTap-DataStager method. As can be seen from Figure 5 the function *smooth1* has an overhead with both POSIX and MPI-IO synchronous methods.

In order to prevent application perturbation, the phase-aware scheduling mechanism attempts to predict when each process is involved in either a local computation (*compute phase*) or in an MPI communication (*communicate phase*). Such phase information is provided to the DataStager through a 'performance buffer' that is maintained for each node communicating with the DataStager (see Figure 1). On the compute nodes, the DataTap library updates its local performance statistics at the end of each iteration. If the client detects a significant change (e.g. the current iteration run time exceeds the previously reported one by more than 10%), the client updates a remote performance buffer on the DataStager.

A key requirement for phase-aware scheduling is to accurately estimate the duration of computational and/or communication phases of parallel codes. Specifically, the sched-

uler must estimate when the application transitions to a compute only state, where such a state is defined as an application state with which is associated no more than some small amount of communication (i.e., only isolated send/receives). This is because simple point to point and sub-communicator broadcast communications (or pure computation) are not likely to be perturbed by asynchronous I/O as opposed to global MPI collective operations.

One way to estimate the duration of computational application phases is to solicit input from developers, by asking them to mark the portions of the application code that are suitable vs. not suitable for background I/O. ADIOS provides a uniform API for these types of hints. For simplicity, we currently use this approach, but this can be generalized and automated using known methods for phase detection, including the techniques reported in [14].

An application enters a compute state at time t_{start} and computes for a time Δt, exiting the compute state at time t_{end}. In some scientific applications, the period Δt is regular, i.e., once the application reaches steady state there is very little variation in the time spent in each compute state. For applications where the computational loop is irregular (e.g., optimization applications), a different mechanism needs to be studied for implementing the predictor. For regular applications (such as our motivating application, GTC [29]), a perfect phase-aware scheduler would always start a data transfer after t_{start} and finish the data transfer before t_{end}. Given sufficient iterations between successive I/O calls, such an ideal scheduler would induce no interconnect perturbation and have a minimal performance impact.

In our current implementation, explicitly installed instrumentation is used to inform the I/O library each time the application has entered a computation state, at time t_{start}, and at time t_{end}, the I/O library is informed that the application has left the computation state. Because all of the I/O requests will not be serviced within a single application iteration, the I/O library also tracks the time taken to complete one application iteration (the *main loop*), t_{iter}. The performance tuple for n compute phases in one application iteration, $\{t_{iter}, [\{t_{start}, t_{end}\}]^n\}$, is lazily mirrored on the DataStager, as described previously.

Experimental results attained with this scheduler and shown in Section 3 show that the phase aware scheduler can reduce the performance impact of background I/O from more than 10% with POSIX data output to about 2% even as the application scales to more than 1024 nodes.

2.2.3 Attribute-aware in-order scheduler

One disadvantage of using a state-aware scheduler is the burden placed on the data staging buffer space in order to create an ordered stream of output data for those applications that require one. This can result in reduced performance due to the additional time that data blocks are held in buffer instead of being processed and transmitted (or written to disk). One example of where such a problem arises is writing a snapshot to a shared file. To complete the write of block b_i, we need to know the sizes of blocks $b_j | j < i$. Instead of letting the compute application synchronize itself and exchange sizes, we propose that the data staging service can more simply perform this operation with less overhead.

We address this problem with an attribute-aware in-order scheduler. When a data block is processed for output, an *attribute* is added to the block defining its order in the application-defined attribute space. When the DataStager services its request it guarantees that request i will not be processed before request j if $i > j$. Thus, when a request is processed, the DataStager already knows the sizes of all previous requests and can write a shared file without any additional synchronization. In the case of multiple DataStagers addressing a group of requests, the sizes can be exchanged within this small group of nodes, or multiple shared files can be created and merged in an additional processing step.

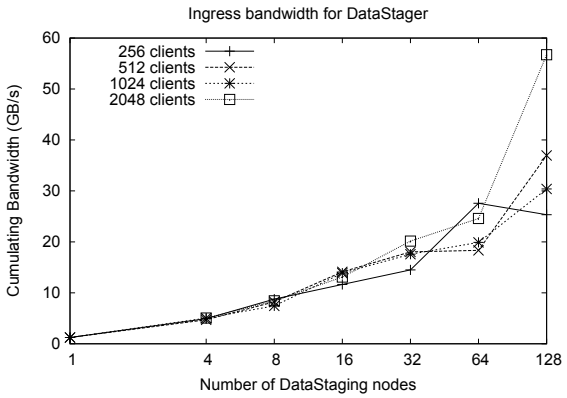


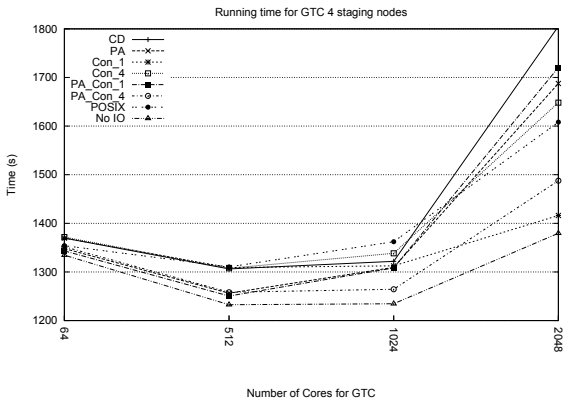
Figure 2: This benchmark compares the cumulative and per DataStager throughput observed when running on 1024 nodes.

2.2.4 Rate limiting scheduler

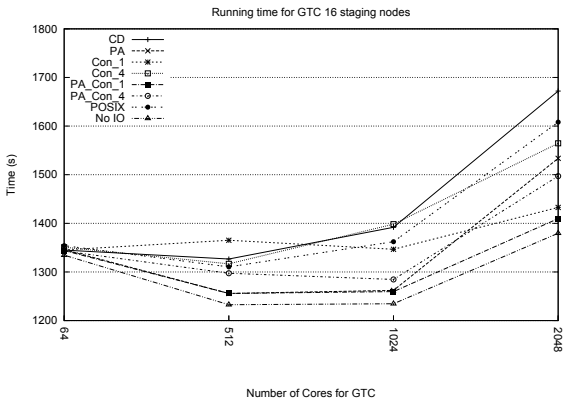
Phase aware scheduling provides solutions for applications that follow regular predictable patterns for data output. Using these predictable patterns we can, with a degree of confidence, avoid the resource usage conflict between DataStager and intra-application communication. In the case of applications with irregular patterns, however, such as AMR applications [33], the state-aware scheduler cannot predict the phases of the application with any reasonable degree of accuracy. In such cases, a different strategy must be employed.

A rate limiting strategy for extracting data from a large cohort of application nodes can be considered if the periodicity of data output is sufficiently large and if the data is not required to be processed immediately. By managing the number of concurrent requests made to the application nodes, the DataStager can greatly reduce the impact of perturbation on intra-application communication. Limiting the rate can have performance implications in terms of reduced ingress throughput and slower time to completion for the requests. This impact can be avoided by appropriately varying the level of concurrency to achieve a proper balance of throughput and perturbation.

Consider an application that writes out data of size 200 GB every 5 minutes from 1024 cores. In order to reduce perturbation and maintain a consistent drain rate from the application, we need to manage the level of concurrency of requests. As seen in Section 3.1 by varying the number of staging nodes we can control the ingress throughput from the application. Using 128 compute processes per DataStager nodes, we see ingress throughput of approximately 8GB/s.

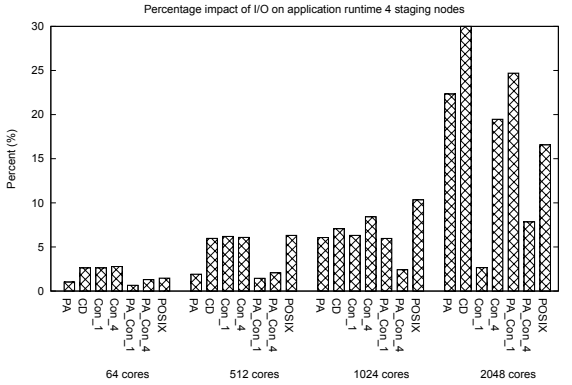


(a) Using 4 staging nodes with GTC

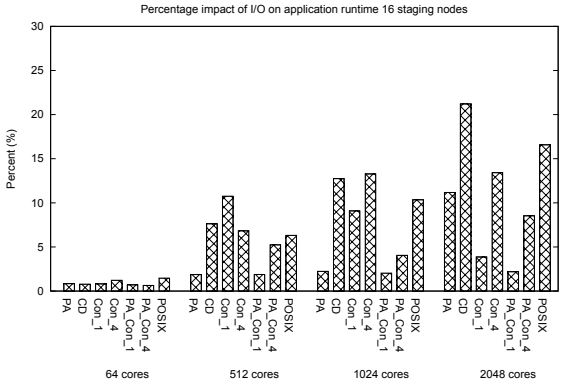


(b) Using 16 staging nodes with GTC

Figure 3: GTC run time with multiple data extraction strategies



(a) Using 4 staging nodes with GTC



(b) Using 16 staging nodes with GTC

Figure 4: Percentage overhead of data extraction on application runtime

Thus draining at the best possible speed we can complete the data transfer in 25 seconds. However this may result in an unacceptable level of perturbation on the source application. By reducing the number of concurrent data transfer requests being serviced to 1 per DataStager we would increase the time to completely move the data from the compute nodes to the DataStager, but we also could reduce the impact on performance caused by background I/O.

One aspect of the rate limiting scheduler is the determination of an appropriate concurrency rate for each type of data output by an application. Currently we do not modify the rate autonomically but we are investigating policies that will enable the DataStager to determine the optimal rate at which data is extracted.

3. EVALUATION

We developed and evaluated the DataStager on National Center for Computational Sciences (NCCS) Jaguar Cray XT at ORNL. Currently each Jaguar node is a 2.1 GHz quad-core AMD Opteron processor and 8 GB of memory, connected to the Cray SeaStar2 interconnect. The interconnection topology is a 3-D torus with each SeaStar2 link enabling a maximum sustained throughput of $6.5GB/s$. The compute node operating system is Compute Node Linux (CNL), and all applications were compiled with the pre-installed PGI compilers. Cray uses the low level Portals API for network programming and provides high level interfaces for MPI and Shmem.

As mentioned before, we used the Gyrokinteic Turbulence Code, GTC [29] as an experimental testbed for DataStager. GTC is a particle-in-cell code for simulating fusion within tokamaks, and it is able to scale to multiple thousands of processors. In its default I/O pattern, the dominant I/O cost is each processor's output of the local particle array into a file. Asynchronous I/O potentially reduces this cost to just a local memory copy, thereby reducing the overhead of I/O in the application. No effort was made to optimize the location in the interconnect mesh of the compute processes with regards to the DataStagers.

We also performed micro-benchmarks to evaluate the maximum throughput for data extraction. For all tests we used the NCCS Jaguar platform with the number of client nodes varying from 64 to 2048. The DataStager nodes used the entire physical node - 4 cores and 8 GB of memory.

We evaluated 6 different data extraction scenarios.

- **CD** is the *continuous drain* method for data extraction.
- **PA** is the *phase aware* method to manage the timing of data extraction.
- **Con_X.** is the *rate limiting* scheduler which limits the number of outstanding concurrent requests to X. We explored two values for X, 1 and 4.
- **PA_Con_X.** is a stacked combination of the *rate limiting* scheduler and the *phase aware* scheduler. In this scenario the number of concurrent requests are limited to X and a new request is only issued if the application is the compute phase. As above we explored two values for X, 1 and 4.

For the remainder of this paper we use the above notation to reference the data extraction strategies.

3.1 Ingress Throughput Evaluation

To measure the ingress throughput we use a parallel test application writing out 128 MB per node per output. Each client process issues an output request and waits for completion immediately. The time taken to complete the data transfer for all client processes is used as the measure for maximum ingress throughput to the DataStagers. In order to maximize the ingress throughput we only utilize the continuous drain scheduler and retire the data buffers from the DataStager staging area immediately.

As can be seen Figure 2 the ingress throughput increases as we increase the number of staging nodes. For a single staging node we see an ingress throughput of 1.2GB/s. As we increase the number of staging nodes the available data extraction throughput increases to more than 55GB/s for 128 DataStager nodes with 2048 client processes. Note that the ingress throughput does not scale arbitrarily with addition of more DataStagers for a small compute partition size. We believe that as the number of clients is increased the overall efficiency of the DataStager's ingress bandwidth will also increase.

3.2 GTC benchmarks

We have extended I/O in the Gyrokinetic Turbulence Code GTC [29] using the ADIOS application interface. The flexibility of the ADIOS interface allows us to run experiments using blocking binary I/O, DataTap with multiple scheduling strategies, and even no I/O without modifying the application binary. For all of the runs, the total configuration size was adjusted so that the amount of data per compute node was a consistent 6,471,800 ions/core. GTC is used for evaluation due to the size of the data output as well as the ability for the code to scale to more than 30,000 cores. We have used a version of the GTC source tree with support added for ADIOS as its I/O library. Using ADIOS has provided us with the opportunity to perform exact comparison tests with the application by simply switching a parameter in the *config.xml* file. To allow better understanding of the performance of different scheduling parameters we disabled all outputs from GTC except the particle output. Also to keep the comparison to multiple run sizes as close as possible we used *weak scaling* (i.e., maintain a fixed per process problem size) instead of *strong scaling* (i.e., maintain a fixed global problem size and only change the number of processes) to maintain a consistent size for the output data per core. Thus the total size of the problem increases but the size per core remains constant. The data size from each output is 188MB/core. The total data volume varies with the number of parallel cores from 12 GB/output to 3.8 TB/output. The output had a periodicity of 10 iterations (approx. 3 minutes wall clock time) and the application ran for 100 iterations. In order to avoid the variations at startup we only measure the time from the 20th iteration onwards.

3.2.1 Runtime impact from background staged I/O

We compare the GTC run time for each of the different data extraction methods described earlier. In order to get an accurate understanding of the cost associated with I/O we also evaluate the default POSIX data output (through the POSIX transport method in ADIOS) as well as a no-op transport method, NO-IO. When required we also use the default implementation of the MPI-IO transport method as a second example of synchronous I/O.

Consider Figure 3(a) shows how application run time is impacted by different DataStaging schedulers. For a small number of compute cores (e.g. 64) there is very little impact on the overall performance from I/O. Con_1 is the only strategy that shows appreciable overhead and even then it is less than 5%. At 512 compute cores the different schedulers start to differentiate. 512 is the minimum size at which we see significant impact from perturbation. Below 512 compute cores, the runtime difference between synchronous and asynchronous I/O is minor. As we move to 512 and 1024 cores we see statistically significant differences in run times. PA, PA_Con_1 and PA_Con_4 continue to show very little overhead from I/O extraction. Con_1 and Con_4 perform at the same level as synchronous output with POSIX. The low impact of all asynchronous strategies is also evident with 2048 application cores. The performance impact of POSIX increases to over 20%. Con_4 and PA_Con_4 maintain an acceptable level of performance impact even at this scale. The same pattern is observed with 16 staging nodes, but the performance impact of background I/O using continuous drain increase greatly.

In Figure 4 we quantify the percentage cost of using the DataStager for performing non-blocking I/O for 4 and 16 staging nodes. For a small number of client cores (64) the synchronous POSIX method offers superior performance. However as we scale and the total size of the data increases, the time spent in synchronous I/O increases more than the overhead of the DataStager method. The percent impact of the POSIX method increases from less than 2% with 64 cores to 10% with 1024 cores, increasing to over 25% with 2048 cores. In contrast the impact of the DataStager depends greatly on the type of scheduling mechanism used, as well the number of staging nodes used stacking the rate limiting scheduler with the Phase aware scheduler PA_Con_4 and PA_Con_1 provides the best performance as we scale the number of compute cores. The number of staging nodes also has an impact on the perturbation of the compute application, with 4 staging nodes showing lower perturbation in general than 16 staging nodes.

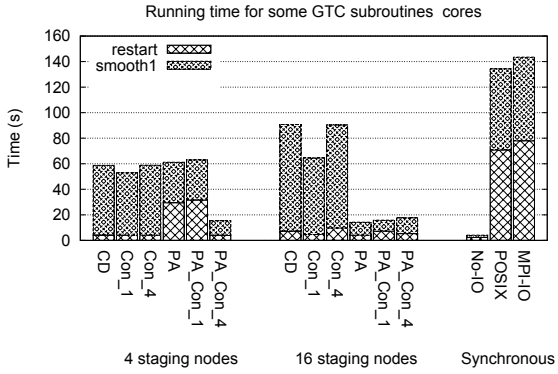


Figure 5: Total Runtime (8 iteration) for functions impacted by background I/O with 1024 cores

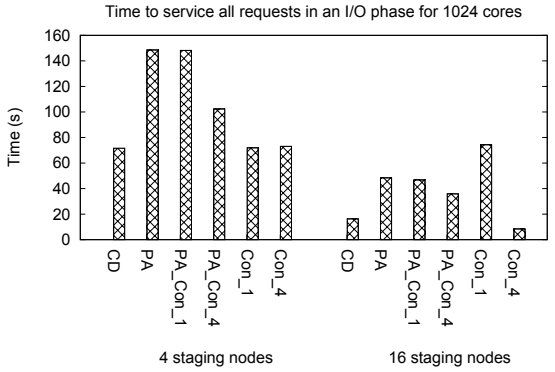


Figure 6: The time taken to complete all pending I/O requests from all processors

3.2.2 *Breakdown of impacted subroutines in GTC*

To further our understanding of the performance characteristics for the DataStager we analyzed the runtime for the *smooth* subroutine in the GTC code path. For clarity we are not displaying the impact on the rest of the subroutines. Smooth immediately follows the data output and hence in cases of improperly managed data transfers shows the greatest level of perturbation. Consider Figure 5 which shows the runtime for the smooth and restart function with 1024 cores for all schedulers as well the synchronous methods. For both POSIX and MPI-IO we see a large increase in the time for restart, signifying the I/O blocking time. We also see a significant increase in smooth due to the partial buffering of output data by the Lustre client and subsequent background evacuation of the buffer to OSTs. In contrast the DataStager shows very low runtime overhead for the restart subroutine. However the performance of smooth is negatively impacted by non phase aware data extraction strategies. such as CD, Con_1 and Con_4.

3.2.3 *Time to complete data extraction to DataStager*

One important factor to consider for asynchronous I/O is the total time taken to service all of the application's I/O requests. This time determines how often an application can issue an asynchronous I/O request without requiring additional buffer space. We compared the completion time for the I/O phase at 1024 application cores (total data size of 180 GB) and the results are shown in Figure 6. As CD tries to extract the data as fast as possible it is not surprising that the time taken by CD is the lowest for 4 staging nodes, and next to lowest for 16 staging nodes. However both Con_1 and Con_4 also show very low completion time. This is because by limiting the concurrency of the inflight requests the ingress throughput for a single request is maximized. The phase aware strategies show much higher completion time with PA and PA_Con_1 performing almost the same. This is because phase aware strategies can only initiate a transfer during a small window in order to avoid interfering with the application.

4. RELATED WORK

There has been significant prior research into studying improvements to the I/O performance for scientific applications. Highly scalable file systems like PVFS [11, 23], Lustre [12], and GPFS [31] are examples of efforts to improve I/O performance for a wide range of applications. Although file systems such as GPFS do offer asynchronous primitives, there has been no effort to study and eliminate interference of asynchronous I/O and intra-application communications in these file systems. LWFS [27, 28] is a high performance lightweight file system designed to eliminate metadata bottlenecks from traditional cluster file systems. The current implementation of LWFS is very close to that of the DataStager, offering an asynchronous RPC and a server-directed data transfer protocol. The scheduling mechanisms described in this paper are orthogonal to the functionality of LWFS and can be used in order to further improve application performance.

Recently there has been an effort to consider data staging in the compute partition in order to improve performance. [26] is an effort to improve I/O performance by using the additional nodes as a global cache. Since I/O delegates are implemented as part of MPI-IO the advantage of this approach is generality. However the performance impact of this approach is limited for large data outputs where the I/O delegates exhaust the available caching space.

PDIO [34] and DART [15] provide a bridge between the compute partition and a WAN. Similar in design to our data staging services, both platforms would potentially suffer from similar problems with interference. The idea behind managed data transfers in DataStager could be utilized by both projects to reduce the impact of asynchronous I/O on application performance. As part of our future work, we are also addressing the connection to WANs through the EVPath [16] messaging middleware.

Hot spot detection and avoidance in packet switched interconnects [21, 20] and in shared memory multiprocessors [13, 5] are related to our efforts to reduce interference with communication in scientific application. Solutions to the problems in those domains are still significant on the highly scalable MPP hardware, we are targeting, where state-aware scheduling provides a software-only solution to the problem of contention.

In [6] the authors evaluate MPI non-blocking I/O performance on several leading HEC platforms. They found only two machines actually support non-blocking I/O and

benchmark results showed substantial benefit by overlapping I/O and computation. The characteristics of the benchmark used, however, does not allow the authors to study the impact of the overlap of I/O and computation for asynchronous I/O. In our work we have discovered that the real performance penalties for asynchronous I/O are from interference with communication.

[30] studies the impact of different overlapping strategies for MPI-IO. The authors consider different strategies for the overlap of I/O with computation and communication showing the performance benefits of asynchronous I/O. However the results are limited to a small number of processors and as we show there is limited interference at these sizes. The innovative benchmarking tool used can be an aid to our own effort in developing better strategies for data extraction.

Overlapping I/O with application processing has been shown to dramatically improve performance. SEMPLAR [2, 4], built on the SDSC Storage Resource Broker, supports asynchronous primitives allowing asynchronous remote I/O in the Grid environment. Because SEMPLAR uses a separate thread to implement a push-based model, there is a smaller likelihood of interference with application communication. However, the authors observed a performance decrease in some scenarios where resource contention penalized performance. Due to their push based model, the solution involved a reorganization of the application code to remove overlap between I/O and MPI. The DataStager provides a server-based mechanism for accomplishing the same task, while keeping application modifications to a minimum.

5. CONCLUSION AND FUTURE WORK

Measurements conducted on large-scale machines and reported in this paper demonstrate that asynchronous I/O can offer high levels of performance to end user applications. At the same time, an interesting issue with such I/O methods is the need for careful and coordinated use of machine resources, to avoid the contention issues commonly occurring in high end machines. A specific issue addressed in this paper is jitter induced in the parallel application's execution by overly aggressive data transfers performed for I/O purposes. The state-aware approach to I/O scheduling developed and evaluated in this paper constitutes one way to avoid such jitter, substantially reducing or eliminating the unintended performance impact of asynchronous I/O while at the same time, offering significant improvements in I/O performance

We have not yet implemented the end-to-end notions of I/O performance desired by more complex multi-scale or multi-model HPC codes. A first step toward that goal is the fact that we already support multiple types of I/O for each single application, but we have not yet implemented priorities for different I/O groups. Our current plans are to add multiple priority levels so that high priority I/O can complete faster than lower priority I/O, without any adverse performance impact. Next, we will add application-specific scheduling mechanisms to allow data to be streamed from the DataStager to an I/OGraph in some specific order. Our longer term goal is to provide a framework for realizing domain-specific high performance I/O scheduling algorithms and services that go beyond high performance output to providing end-to-end I/O services and guarantees for large-scale applications.

6. REFERENCES

[1] H. Abbasi, M. Wolf, and K. Schwan. LIVE Data Workspace: A Flexible, Dynamic and Extensible Platform for Petascale Applications. *Cluster Computing, 2007. IEEE International*, Sept. 2007.

[2] N. Ali and M. Lauria. Improving the performance of remote i/o using asynchronous primitives. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 218–228, 0-0 0.

[3] P. Beckman and S. Coghlan. ZeptoOS: the small Linux for big computers, 2005.

[4] K. Bell, A. Chien, and M. Lauria. A high-performance cluster storage server. *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 311–320, 2002.

[5] R. Bianchini, M. Crovella, L. Kontothanassis, and T. LeBlanc. Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors. Technical report, DTIC, 1993.

[6] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation Of Leading HPC I/O Performance Using A Scientific-Application Derived Benchmark. In *Proceedings of the 2007 Conference on SuperComputing, SC07*, 2007.

[7] R. Brightwell, T. Hudson, R. Riesen, and A. B. Maccabe. The Portals 3.0 message passing interface. Technical report SAND99-2959, Sandia National Laboratories, December 1999.

[8] R. Brightwell, B. Lawry, A. B. MacCabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 268, Washington, DC, USA, 2002. IEEE Computer Society.

[9] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 39. IEEE Computer Society, 2000.

[10] Z. Cai, G. Eisenhauer, Q. He, V. Kumar, K. Schwan, and M. Wolf. Iq-services: network-aware middleware for interactive large-data applications. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 11–16, New York, NY, USA, 2004. ACM Press.

[11] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[12] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[13] S. Dandamudi. Reducing hot-spot contention in shared-memory multiprocessor systems. *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, 7(1):48–59, Jan-Mar 1999.

[14] C. Ding, S. Dwarkadas, M. Huang, K. Shen, and J. Carter. Program phase detection and exploitation. *Parallel and Distributed Processing Symposium, 2006.*

IPDPS 2006. 20th International, pages 8 pp.–, 25-29 April 2006.

[15] C. Docan, M. Parashar, and S. Klasky. High speed asynchronous data transfers on the cray xt3. In *Cray User Group Conference*, 2007.

[16] G. Eisenhauer. The evpath library. `http://www.cc.gatech.edu/systems/projects/EVPath`.

[17] G. Eisenhauer. Portable binary input/output. `http://www.cc.gatech.edu/systems/projects/PBIO`.

[18] G. Eisenhauer, F. Bustamente, and K. Schwan. Event services for high performance computing. In *Proceedings of High Performance Distributed Computing (HPDC-2000)*, 2000.

[19] M. K. Gardner, W.-c. Feng, J. S. Archuleta, H. Lin, and X. Ma. Parallel Genomic Sequence-Searching on an Ad-Hoc Grid: Experiences, Lessons Learned, and Implications. In *ACM/IEEE SC|06: The International Conference on High-Performance Computing, Networking, Storage, and Analysis*, Tampa, FL, November 2006. Best Paper Nominee.

[20] S. Golestani. A stop-and-go queueing framework for congestion management. In *SIGCOMM'90 Symposium*, pages 8–18. ACM, September 1990.

[21] R. Jain, K. K. Ramakrishnan, and D. M. Chiu. Congestion avoidance in computer networks with a connectionless network layer. Technical Report DEC-TR-506, Digital Equipment Corporation, MA, Aug. 1987.

[22] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–47, February 1997.

[23] R. Latham, N. Miller, R. Ross, and P. Carns. A next-generation parallel file system for linux clusters. *LinuxWorld*, 2(1), January 2004.

[24] J. Lofstead, K. Schwan, S. Klasky, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Challenges of Large Applications in Distributed Environments (CLADE)*, 2008.

[25] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 567–576, New York, NY, USA, 1991. ACM.

[26] A. Nisar, W. keng Liao, and A. Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[27] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. R. sen, L. Ward, and P. Widener. Lightweight I/O for Scientific Applications. In *Proc. 2006 IEEE Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[28] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock. Efficient Data Movement for Lightweight I/O. In *Proc. 2006 Workshop on high-performance I/O techniques and deployment of Very-Large Scale I/ O Systems (HiPerI/O 2006)*, Barcelona, Spain, September 2006.

[29] L. Oliker, J. Carter, michael Wehner, A. Canning, S. Ethier, A. Mirin, G. Bala, D. Parks, Patrick Worley Shigemune Kitawaki, and Y. Tsuda. Leading computational methods on scalar and vector hec platforms. In *Proceedings of SuperComputing 2005*, 2005.

[30] C. M. Patrick, S. Son, and M. Kandemir. Comparative evaluation of overlap strategies with study of i/o overlap in mpi-io. *SIGOPS Oper. Syst. Rev.*, 42(6):43–49, 2008.

[31] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[32] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1995. ACM.

[33] S. Sinha and M. Parashar. Adaptive system sensitive partitioning of amr applications on heterogeneous clusters. *Cluster Computing*, 5(4):343–352, 2002.

[34] N. Stone, D. Balog, B. Gill, B. Johan-SON, J. Marsteller, P. Nowoczynski, D. Porter, R. Reddy, J. Scott, D. Simmel, et al. PDIO: High-performance remote file I/O for Portals enabled compute nodes. *Proceedings of the 2006 Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June*, 2006.

[35] P. M. Widener, M. Wolf, H. Abbasi, M. Barrick, J. Lofstead, J. Pullikottil, G. Eisenhauer, A. Gavrilovska, S. Klasky, R. Oldfield, P. G. Bridges, A. B. Maccabe, and K. Schwan. Structured streams: Data services for petascale science environments. Technical Report TR-CS-2007-17, University of New Mexico, Albuquerque, NM, November 2007.

[36] M. Wolf, H. Abbasi, B. Collins, D. Spain, and K. Schwan. Service augmentation for high end interactive data services. In *IEEE International Conference on Cluster Computing (Cluster 2005)*, September 2005.

[37] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smartpointers: Personalized scientific data portals in your hand. In *Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page 20. IEEE Computer Society, 2002.