SLAC-PUB-9098
December 2001

Creating Large Scale Database Servers

Jacek Becla and Andrew Hanushevsky, Stanford Linear Accelerator Center
Stanford University, Stanford, CA, 94309

Abstract

The BaBar experiment at the Stanford Linear Accelerator Center (SLAC) is designed to perform a high precision investigation of the decays of the B-meson produced from electron-positron interactions. The experiment, started in May 1999, will generate approximately 300TB/year of data for 10 years. All of the data will reside in Objectivity databases accessible via the Advanced Multi-threaded Server (AMS). To date, over 70TB of data have been placed in Objectivity/DB, making it one of the largest databases in the world. Providing access to such a large quantity of data through a database server is a daunting task. A full-scale testbed environment had to be developed to tune various software parameters and a fundamental change had to occur in the AMS architecture to allow it to scale past several hundred terabytes of data. Additionally, several protocol extensions had to be implemented to provide practical access to large quantities of data. This paper will describe the design of the database and the changes that we needed to make in the AMS for scalability reasons and how the lessons we learned would be applicable to virtually any kind of database server seeking to operate in the Petabyte region.

1 The *BABAR* project

The High Energy Physics (HEP) community consists of many thousands of physicists and engineers around the world [6]. One of its largest HEP experiments, that has just entered production is called *BABAR*, and is headquartered at the Stanford Linear Accelerator Center (SLAC) in California. A central theme of its research program is detailed study of the difference between matter and antimatter. It was launched in May '99 and is expected to continue running for at least the next 10 years. The project was constructed and is managed by a large international collaboration of physicists and engineers from 10 countries.

The heart of the experiment is the *BABAR* detector attached to the PEP II linear accelerator, both located at SLAC. The detector has been designed to generate data at a rate of about 32MB/sec (or $3*10^8$ *physics events* per year, or 100 events/sec). Other sources of persistent data include simulation processes as well as reconstruction and analysis jobs run by physicists. The estimated size of 1-year's worth of real and simulated data is 300TB, significantly exceeding the data generated by any other HEP project launched so far.

The data coming from the detector is stored persistently, and then "reconstructed" in near real-time: usually within eight hours of the data's collection. Reconstruction runs asynchronously with data taking on multiple computing nodes in a fully controlled environment. The output from the reconstruction process is then passed to physicists, who then analyze the data. Data analysis is performed in a non-controlled way, where physicists are allowed to read the data at any time and generate new persistent data. While real data is coming from the detector, simulated data is generated concurrently by multiple simulation processes and later compared with the outcome from the analysis jobs. All the data is kept locally at SLAC, mostly on tapes; part of the data is copied to several external institutes in Europe.

The software responsible for handling the *BABAR* data is mostly a homegrown system, exceeding 2.5 million of lines of code. Its capability spans many independent tasks, including:

- storing live data (Data Acquisition System),
- capturing conditions and configuration of the detector,
- performing data reconstruction, data analysis and simulation, and
- providing fast access to the data for over a hundred simultaneous users, often working remotely.

All components of the *BABAR* software, often quite independent, have one piece of software in common: the tier providing data persistency. Data access is handled by the *BABAR* Database System. Clearly, handling multiple terabytes of data is non-trivial; the software has to be very robust, well optimized, and it requires a lot of computing power. The *BABAR* software is already in production since the first data was taken in May'99, but it is still not completely optimized. The process of optimizing the system: choosing the right hardware configuration and tuning the software is an ongoing and very lively activity inside the *BABAR* Database Group. Detailed information about the *BABAR* Project can be found in [2].

Work supported by the Department of Energy contract DE-AC03-76SF00515.

Presented at the 9th International Symposium on High Performance Distributed Computing (HPDC-9), Pittsburg, PA, August 1-4, 2000

2 The BABAR database system

The primary goal of the *BABAR* Database System is to provide data persistency for the *BABAR* experiment. In order to fulfil stringent project requirements, including:

- robustness: continuous real-time data taking
- high throughput: >30MB/sec
- concurrency: >100 simultaneous users
- data distribution: 10 countries, 3 continents
- heterogeneous environment: Sun Solaris, OSF, Linux, HP, AIX,
- very complex schema: ~600 persistent classes

the system has no choice but to use cutting-edge, but somewhat uncertain, database technology.

Undoubtedly, commercial products introduce many new features and relieve the burden of writing low-level code. At the same time they always add further restrictions and complications, which rarely can be ignored. Although *BABAR* software is mostly homegrown, the underlying database and mass storage systems are both commercial products. The *BABAR* Database System is based on a combination of:

- an Object Oriented Database System (ODBMS): Objectivity/DB, and
- High Performance Storage System (HPSS).

An Objectivity/DB based solution has been adopted at *BABAR* 5 years ago. During that time, Objectivity/DB appeared (and it still does) to be the only available OO system capable of scaling beyond a petabyte and fulfilling the other, very stringent requirements. Increasingly, many HEP experiments are using, or are going to use Objectivity/DB. However, it is worth mentioning, that *BABAR* was **the first** high-data volume experiment, which used Objectivity/DB in production.

In order to provide sufficient storage to physically store the databases, the *BABAR* System employs the HPSS Mass Storage System. With this system, databases can reside on disk while being read or written, or on inexpensive tape when they are inactive. Unfortunately, at the time HPSS was selected, it was available only on IBM platforms while Sun Solaris was the preferred *BABAR* platform. Since then, SLAC engineers ported the data transfer components of HPSS to Solaris so that the database performance characteristics could be maintained.

3 Structuring the data

3.1 Objectivity/DB storage hierarchy

The highest level in the Objectivity/DB logical storage hierarchy is the *federated database*, physically mapped by a single file containing a schema and a catalog of databases. Each database maps to a physical file. Each file consists of one or more logical structures called *containers*, which in turn contain basic persistent objects. Containers determine the physical clustering of data, and locking granularity. Federation integrity is guaranteed by the Objectivity/DB lock server process, which maintains transaction and lock tables. Usually there is only one lock server per federation, although Objectivity gives a choice of running more (Objectivity/DB Fault Tolerance Option). See [13] for further details.

3.2 Multi-federation environment

Within *BABAR*, the Objectivity/DB System is used for keeping virtually all the experiment's persistent data. Various parts of the experiment have quite different, often almost opposite, sets of requirements:

1. Data Acquisition System (DAQ) demands almost immediate response time, and 24x7 reliability. In return its other requirements like level of concurrency or data volume are not very high.
2. Online Prompt Reconstruction (OPR) requires very high level of concurrency currently reaching as many as 200 client nodes writing simultaneously to the same set of databases. Fortunately it runs in a completely controlled environment, which makes the maintenance much easier.
3. Data Analysis is a world of physicists, who run their jobs in a completely unpredictable way; with no discernable simple pattern. The concurrency level is high, as well as the expected reliability; requiring scheduled database outages to perform necessary maintenance.

In order to provide everybody with the service they expect, the *BABAR* Database System has been broken into several independent pieces, and each major task such as DAQ, OPR or analysis has been assigned an independent federation. Each federation has been assigned a separate set of servers, including data servers and lock servers. Since locking is intrinsic to a federation, such a configuration entirely removes cross-federation dependencies. In addition there are multiple test federation.

In practice, the federations are not completely independent; some data still need to be shared/exchanged between them. Internally within SLAC data distribution strategy takes advantage of the HPSS catalog in minimizing the actual copying of databases between federations. For example, once a database generated by OPR has been migrated to HPSS, the catalog for the downstream federation can be updated, without the necessity for physically copying of the database between the appropriate servers. The staging procedures then support transfer of a database from tape to disk.

3.3 Hardware

Bulk of *BABAR* production runs on Sun machine. Some small external institutes use different platforms, namely Linux or OSF.

The Veritas File System (VFS) is used on all data servers and some metadata servers, depending on access patterns and the quantity of data. VFS allows to build a large file system (e.g. 500 or 800TB) from many single disks. Maintenance of several large file systems is significantly simpler then hundreds of small ones. Additionally, the VFS is a journaled file system so that system reboot time is minimized after an outage; critical when using large file systems.

4 Tuning the system - achieving scalability

4.1 Unit of transfer

Tuning the system in order to achieve design scalability began long before the project officially started. One of the first important decisions impacting the performance was choosing a correct page size, the unit of disk and network transfer in Objectivity/DB. It is one of the very few parameters, which cannot be altered after a federation has been created. Currently Objectivity/DB restricts that value to be within 512 bytes and 64KB range.

Unfortunately, HPSS performs most efficiently with much larger transfer sizes – typically in the MB range. This is at conflict with many database systems, which prefer much smaller transfers, usually below 64KB. The solution that has been adopted is to "*disk cache*" files between the Objectivity/DB servers and HPSS. Transfers in and out of HPSS are performed in large chunks and thus we are able to get the expected performance out of the HPSS, and still use small transfer units for Objectivity/DB, which sees only the disk-resident files and is unaware of the HPSS mechanism behind.[1]

Clearly a too small page size results in many transfers, while a too large page size might increase the load on the network and the amount of unnecessary transferred data. After many tests, which were simulating behavior of real applications, it has been shown that on a Sun Solaris platform (the major *BABAR* production platform) a 16KB page size outperforms other choices almost by the factor of two. These tests were run before the start of the experiment, but after many months of running large-scale tests, we are starting to consider using a larger page size. We believe it might give some benefits we did not foresee in the past (see also discussion about the Veritas File System below). There are discussions in progress with Objectivity/DB, whether it would be possible to remove the 64K restriction. We probably might still be able to convert our federations to a larger page size by copying all of the data -- a very expensive process. Most likely we would try using a size in the range 256KB – 1MB. Of course, the conversion may not be feasible once we will have too much data, e.g. more then 1PB.

4.2 Testbed

Intensive work on improving performance began shortly after the project officially started. After more then half a year of testing and tuning the system, it is still an on-going and very live activity within *BABAR* Database Group.

The part of the system, which required tuning the most, was Online Prompt Reconstruction. According to the design, the ultimate goal was to reconstruct 100 events/sec. When the *BABAR* project was launched, in May '99, the OPR software was able to achieve only 7-8 events/sec, using 50 computing nodes. All attempts to increase the throughput by increasing the number of processing nodes were ending up with an even slower processing rate, mostly due to lock congestion, and other problems, waiting to be uncovered. The OPR was also the most convenient part of the system to focus the tests on, since it runs in a fully controlled environment. It was clear from the beginning, that once we learn where the problems were within OPR and understood the major bottlenecks, it would be relatively easy to apply that knowledge to the remaining parts of the system.

Two month after the project had officially started, a dedicated test bed was established. The goal was to improve the performance, and speed up the system as much as possible. A dedicated batch system consisting of 100 processing nodes and several server nodes made up the testbed.[2] All machines were of the same class as those used in production. Another 100-240 nodes were occasionally "borrowed" from the production farm augmenting the test farm to 340 nodes, whenever needed for an intensive test.

4.3 Software optimization

Multiple optimization improvements were made inside our software. Running the tests and profiling the applications helped us to learn where most of the time is spent, and which pieces of the software need improvement. For instance we learned to avoid naming containers (whenever we can, of course). Naming a container involves an extra lock on a shared resource, and therefore can lead to significant performance loss, especially when thousands of containers are created in a relatively short period of time.

Another noticeable improvement was observed when we started to transiently cache database identifiers within a job, rather then keep referring to them by name. Referring by name involves a name lookup in the large database catalog. In Objectivity, the client performs the look-up so that the whole catalog needs to be transferred to the client before the look-up can be done. Avoiding the

[1] Objectivity/DB clients are always writing to the disk cache, and are not aware of HPSS. A specialized server, written at SLAC, controls transfers between the disk cache and the HPSS tapes.

[2] 100 client nodes: Sun Ultra 5 machines, with 256MB real memory, running Sun Solaris 2.6, 2 main server nodes: Sun 4500, with 1GB real memory, also running Sun Solaris 2.6

whole operation multiple times led to a significant improvement.

4.4 The tuning process

The list of knobs we tried to twiddle is very long. Practically, we tried to estimate the influence of every adjustment, which could have an influence on performance/scalability. As one could imagine, the total number of all possible permutations is too large to be fully explored. Instead, we attempted to understand how each change influenced the throughput, and applied one improvement on top of the previous one, making additional adjustments.

Below are the major changes that gave visible performance improvements.

1. Increase hard and soft file descriptor limit on all machines running Objectivity/DB Data Servers. Objectivity/DB opens multiple TCP connections between each client and a data server. If hundreds of clients start to connect to a server, it quickly runs out of default system resources (the default limit of open file descriptors on Unix is 1K, we are currently using 8K).
2. Veritas read/write performance. Due to large number of database files, which are read/written simultaneously, disk access is essentially random. Random write performance of a single VFS seems to be limited to ~8MB/sec in the configuration we are using[3]. Random read performance is close to 4MB/sec. Both quoted numbers were confirmed by running many independent tests (for example with and without Objectivity/DB). Multiple attempts to tune the VFS did not succeed. Probably part of the problem is connected with the small unit transfer (we are using a 16K page size).
3. Increase number of data servers / file systems. Initially, while we started to run tests on 50 nodes, two data servers with one file system on each were able to serve all the clients. While we were increasing the number of nodes to 100, and then to 200, we had to add more file systems in order to be able to keep up with the data.
4. Running multiple Objectivity/DB Data Servers per host. Until recently, the Objectivity/DB data server was a single-threaded process. Starting 4 servers instead of one gave a very significant performance boost. A major recent improvement was the release of a multi-threaded data server.
5. Balancing data across multiple servers. It is important to put relatively the same load on each file system. By sorting data across file systems per type, we were able to improve both read and write performance. Because we are constantly changing configurations, we found this model a bit difficult to maintain (e.g. different data types have different sizes, and the ratio between then is changing). We developed a new model, which allows us to balance the load based on clients, e.g. if data is written to 6 file systems, we are redirecting each 1/6 of clients to a separate file system. The new model simplifies the process of re-balancing the load significantly, especially when a new file system or a new set of clients is added to the system.
6. Pre-sizing containers. Objectivity/DB gives multiple choices during the creation of containers, for example a user can specify the initial number of pages and/or container percentage growth. Creating containers close to their final size proved to be much better then starting off with a small container and increasing its size while it is being filled. The behavior is connected with the way Objectivity/DB extends containers and locks their internal structures.
7. Cache size. Each client writes persistent data into databases, and occasionally commits transaction. Data is usually not written to disk immediately when the write occurs. Instead it is cached in client memory (Objectivity/DB cache) until the transaction is committed/aborted. If there is not enough space in the cache, some data has to be pushed to disk before the end of a transaction. If we create the cache large enough to hold all of the data written between transactions then actual physical writes occur only during a commit. We discovered this is a worthwhile optimization. Writing more data in one chunk is more efficient in terms of overall throughput of the system then writing data constantly in small pieces. On the other hand, if the cache is too large then cache look-up speed decreases, degrading overall performance. In effect, cache size depends on transaction length, described in the next point.
8. Tuning & randomizing transaction length. Since we start all of the clients at the same time, the jobs tend to synchronize, and try to commit at the same time. Each commit is associated with a lot of data being transferred at one time and increased lock traffic. The only solution to that problem is to force jobs to do the commit asynchronously[4]. We are currently randomizing transaction granularities per client, each client is allowed to use value within +/-50% of preferred length.
9. Dependencies on NFS/AFS. Although it is not directly a database issue, it is still worth mentioning that reducing dependencies on AFS or NFS brought a lot of benefits. Whenever one starts hundreds of clients, neither NFS nor AFS perform well. Using local file system is always preferred (if possible).

[3] For more details about the VFS configuration please refer to [17]

[4] The formula used to pseudo-randomize transaction granularity: X +/-50% of (process ID%X), where X is the transaction granularity (in sec).

4.5 Physics analysis

Similar improvements are gradually being applied to physics analysis. Of particular importance is the combination of the number of database servers, the number of CPUs per server, and the number of file systems per server. Significant tune-ups have also been applied to the code. Performance optimizations in the analysis programs themselves have resulted in improvements from 35 events/sec to 2k events/sec for one particular benchmark where events are selected on the basis of so-called tag filters.

5 Database internals

Objectivity/DB is an object-oriented database system. Data in C++ objects are grouped together on pages of up to 64K in size and written to disk into logical containers (the locking granularity) that are segments of a standard file. A file may contain up to 32,768 containers. Files are grouped together to form a federation. There can be up to 32,7578 files in a single federation. A federation is essentially a single database in Objectivity/DB and applications, given the appropriate access permissions, can access any object in any file within a federation.

Access to files is via the operating system's file system interface (i.e., native access), through the Network File System (NFS), or through a specialized file server called the Advanced Multi-threaded Server (AMS). A combination of methods may be used as long as any particular file is accessed through only one interface.

Each file is limited by the constraints imposed by the host operating system. For most systems, a database file can be 2^{64} bytes in length. Thus, a complete database (i.e., single federation) can hold up to 2^{80} bytes of directly addressable data, a prodigious amount of information. Because of the various supported access modes, such a large amount of information should be efficiently handled.

Unfortunately, while it is possible to create databases of such size, it is extremely difficult to access that amount of data using standard methods. First, file systems, or even combinations of file systems, cannot scale to 2^{80} bytes. Secondly, spreading the data amongst many servers and using NFS to access the data is problematic since large-scale NFS access does not perform sufficiently well. This leaves AMS access as the only viable alternative for such large databases.

The AMS is a specialized file server akin to an NFS server. A client makes a TCP connection to the AMS and then reads and writes database pages that can be up to 64K bytes in length. Unlike NFS, the AMS provides additionally functionality such as database replication and partitioning. While such functionality is not strictly necessary to support large databases; the protocol that the AMS uses is necessary. Unfortunately, the AMS suffers the same limitations as any file server; it is bound by the limits imposed by the underlying file system. If the file system cannot scale, the AMS cannot make up for it.

With this in mind, we undertook a significant restructuring of the AMS to allow us to enhance it to the point that any constraints were due to the underlying operating system. Originally, the AMS was a single monolithic server, like many database servers today. It was clear that unless we were able to independently focus on the various aspects of a database server we would not be able to make scalability modifications. Thus, the first major change was to re-architect the server into three separate components:

1. The AMS protocol layer,
2. The Objectivity Open File System (OOFS) layer (the logical file system),
3. The Objectivity Open Storage System (OOSS) (the physical file system).

The protocol layer is responsible for appropriately responding to requests using a highly efficient network protocol. The vast majority of requests require the server to read, write, or manipulate files. This is done through the OOFS that presents a logical file system to the protocol layer and is responsible for creating and deleting directory and file objects. The OOFS is a virtual file system and depends on the OOSS layer to actually implement a physical file system. From the OOFS perspective, the OOSS simply creates interface objects to directories and files.

Having split the server into these three replaceable components allowed us to independently optimize each layer as well as try various file systems with different scalability and performance characteristics. The following sections explain the types of optimizations we performed in order to scale the system to be able to handle petabytes of data and hundreds of simultaneous users.

5.1 Physical layer optimizations

The most significant undertaking in the OOSS was the use of a Mass Storage System (MSS) to back-end a high performance file system. The implementation at SLAC used the High Performance Storage System (HPSS) as the MSS and the Veritas file system as the disk cache.

The use of an MSS allowed us practically an unlimited amount of storage since any less-used databases would be automatically migrated to tape. Databases on tape would be migrated back into the file system, as needed. Thus, while the amount of online space was limited, the total amount of accessible space was practically unlimited. Since HPSS is capable of handling 2^{64} bytes per file and over 2^{32} files, it easily matched the limits imposed by Objectivity/DB,

The Veritas file system allowed us to have multiple terabyte RAID caches so that we could always keep a sufficient amount of highly used data (i.e., the current database working set) online. Furthermore, Veritas has various performance options such as linear space preallocation to significantly speed access. A crucial aspect of the Veritas file system is its journaling feature, an absolute necessity when dealing with extremely large file systems. After an operating system failure, the file system need only scan a short log of a few megabytes to recover file system information; minimizing reboot time.

We also implemented another critical optimization we call file descriptor partitioning. While subtle in nature it significantly reduces CPU utilization. In order to understand why this is the case it is necessary to consider what happens during the course of system operations.

When a client makes a connection to the AMS, the server opens a network socket that causes the operating system to allocate a file descriptor in order to handle the socket. Practically all Unix operating systems allocate the lowest numbered file descriptor, and this is where the problem occurs. Typically, a client request will cause a file to be opened with the consequence of yet another file descriptor to be allocated for the file. The file remains open while the socket connection is used. This means that as the system runs, socket file descriptors are interleaved with socket file descriptors; at worst, the interleaving is one to one. When the server needs to wait for socket activity, only half of the file descriptors are eligible to be waited on. Thus, the system performs needless work of masking out interleaved non-socket file descriptors. This happens several hundred times a second as the server fields new network requests.

File descriptor partitioning simply moves descriptors allocated to files to the top of the file descriptor numbering space. Then, all socket file descriptors are compacted in a sequential range at the low end of the numbering space. This greatly reduces the amount of processing needed to wait for network activity. The saved CPU time can be devoted to handling additional clients.

5.2 Logical layer optimizations

The logical layer is responsible for presenting a virtual file system to the protocol layer. This layer encompasses all file processing that does not involve the physical storage of data (e.g., performance monitoring and security). The layer provided many opportunities to enhance scalability without impacting the physical handling of the files nor the semantics of the database protocol.

The most significant enhancement was file interface reuse. When a client opens a file, the OOFS requests that the OOSS supply an interface object to the file. In many cases, another client is already using the same file and, consequently, has an interface to it. With this in mind, the OOFS searches all allocated interfaces to see if an existing interface can be used for the new request. If a suitable one is found, it is reused. In practice, up to two interfaces may be allocated to a file: one for read and another for update. The appropriate interface is chosen based on the open mode requested by the client. An interface object is deleted only after all uses of the interface cease.

File interface reuse allows the handling of a significant additional number of client requests because the memory load on the system is substantially decreased. Furthermore, it is less likely that the operating system's file descriptor limit will be reached, thus enhancing scalability.

Another enhancement is called redundant sync elimination. This is another subtle optimization that requires the consideration of a running system. When a large number of users are using the database, a large portion of those users may be updating a single file. These updates can occur in parallel because a single file may be composed of multiple containers and the locking granularity is container-based, parallel updates are allowed. When a client commits an update, the AMS is requested to perform a file synchronization to make sure that all data is written to disk before the transaction completes. In order to reduce system overhead, the OOFS tracks whether or not there has been an intervening write to the file since the last synchronization. If a write has been performed, the synchronization occurs. However, if no intervening write has been performed, which is likely when many users are updating the same file, the synchronization is skipped since it is not necessary. This optimization substantially reduces system overhead under heavy loads.

The final optimization is called file interface time-out. This optimization has been used by many other databases to conserve system resources. Simply put, whenever an allocated interface object has not been used for a substantial amount of time (e.g., a user has stopped a program for a long time with the debugger), the OOFS will delete the interface object. The interface object will be automatically re-created when the next request for a file associated with a deleted interface object is received. This optimization is most useful for highly interactive clients in test-mode scenarios.

5.3 Protocol enhancements

There were three critical enhancements that we needed to make to the database protocol in order to ensure scalability with adequate performance. The first was the addition of an Opaque Information Protocol (OIP). Using OIP, a client can transfer OOFS and OOSS specific information to the AMS. The AMS does not inspect the information (i.e., it is opaque) but merely forwards it to the OOFS layer that can inspect it and, in turn, forward it to the OOSS layer.

OIP allows a client to relay application specific information that may be implementation dependent. For instance, an application can inform the OOSS on access patterns (i.e., sequential or random) as well as anticipated file size when creating or adding to a database. Such information is important when trying to optimize the placement of a file. While one can argue that such information can be inferred by the system, it not feasible to do so with any great assurance in a very large-scale database environment. In short, the best information is known by the application.

Another enhancement was the addition of Deferred Request Protocol (DRP). This enhancement is crucial to accommodating high latency file systems, such as Mass Storage Systems. Consider the case where a client opens a database that resides on tape. When this happens, the tape must be mounted and the database copied to the disk cache before the client can use the database. Such an operation can be quite lengthy, taking anywhere from a minute or so to almost half an hour, depending on the size of the database. Under normal conditions, such a delay would cause the client to abandon the request and try another database server under the assumption that the original server had failed. Of course, the easiest bypass is to not use time-outs on requests. However, this prevents the detection of true failures.

With DRP, the OOSS layer can relay the anticipated time to request completion to the OOFS layer which, in turn, tunnels the information back to the client via the AMS protocol. The client can then wait the specified amount of time and retry the request. The solution elegantly handles high latency requests without impacting server failure detection.

Finally, when considering massively large databases and distributed clients, it is unlikely that any one server can handle the load. Indeed, such databases require multiple servers to adequately handle the amount of data as well as the potential number of users. The normal method of handling this situation is to statically partition or fully replicate the database among many servers. In practice, neither solution works well when accessing a massively large database. Therefore, we chose to implement a Request Redirect Protocol (RRP).

RRP allows a server to redirect a client request to another server that can better handle the request. The decision is made by the server and can include any number of criteria such as server load, database availability, number of users, or even the best network routing relative to the client's location. Redirection is not a new concept. Many web servers use redirection for load balancing. However, to our knowledge this is the first time server-mediated redirection will be used for load balancing a commercial database system.

Another interesting aspect of RRP is that it allows for asynchronous database replication. In this scenario, only currently "hot" portions of a database are dynamically replicated. Coupled with a Mass Storage System, such as HPSS, that has point-to-point data transfer operations, dynamic asynchronous replication can achieve massive scalability while maintaining high levels of performance. In fact, dynamic asynchronous replication is likely the only practical replication strategy for petabyte-sized databases.

5.4 Concluding the scalability tests

Creation of the testbed started to pay off practically from the first day. After 1 month of running the tests and tuning the system, we were able to double the throughput,; and the improvements were immediately fed into the production system. Currently the improvements, compared to the point we started, exceed 600%.

Currently the only real limitation we observe in OPR is the lock server CPU saturation. The Objectivity/DB lock server is a single-threaded process and whenever we run >200 clients it uses almost 100% CPU. Attempts to run the lock server on a faster CPU or on a multi-CPU machine[5] did not solve the problem. It is expected, that the next two Objectivity/DB releases, (first due in a few weeks) will take care of this problem.

On the analysis site, we do not see problems with the lock server. Instead, jobs seem to be more I/O bound. Most of the problems come from the fact, that the access to the data is purely random. Even if each job accesses the data sequentially, given the total number of users and jobs, the access quickly becomes random. The main focus in the near future is going to be on the access de-randomization and pre-fetching the data combined with client-side caching.

5.5 Future improvements

Within several months after the experiment started the *BABAR* detector was able to generate data with a speed significantly exceeding its design. In the near future the efficiency of the detector is going to be doubled, though it was not initially anticipated. In the current situation, further optimizations will be necessary for the software to keep up with the incoming data.

We already can identify several changes that could be easily made and improve the performance of the system. They include:

- Increase the number of file systems and data servers.
- An improved lock server.
- Introduction of read-only databases. It should reduce lock server traffic since no locks would be required to read read-only databases; thus improving reference database performance.

We expect it will be possible to at least double the throughput relative to current values within next several months.

6 Conclusions

The *BABAR* Database System is responsible for persistent storage of, and access to the data. Given the volume of data, which is likely to enter the petabyte region in three years, the system is intrinsically very complex and requires a lot of computing power. Currently, the major

[5] second CPU can other tasks, e.g. network transfers.

focus is on improving the performance and scalability of the system.

Tuning the system is an on-going process. After the first 5 months of work, we were able to increase the throughput from ~8Hz up to 130Hz. At the same time scalability has been improved from ~50 nodes to over 200 nodes. Most likely the processing farm will be expanded up to 300-350 nodes in the near future, with a corresponding increase in throughput requirements. The performance tests will continue for at least the next few months.

Additionally, our ability to independently optimize and enhance specific portions of the AMS was a direct result of our chosen layered architecture. It's a true "plug and play" architecture where one can mix and match various critical internal database services in order to achieve the required scalability and performance. In retrospect, we were very pleased with the result. Not only did it speed the development of new processing algorithms, but it also trivialized the deployment of such algorithms.

We were also pleased with our optimization in the OOFS and OOSS layers. A fully optimized AMS uses 25% less CPU time than an AMS that has no such optimizations -- significant savings in resource utilization. This directly translates into our ability to handle a 40TB database with over 3,000 active database sessions. We expect that with additional improvements we can achieve a 50% increase in overall performance.

Of course, neither the architecture nor the optimization would have been of practical use without the protocol enhancements. While we could have devised bypasses for Mass Storage System latency, request optimization, load balancing and replication, the solutions would have been awkward, at best. We feel that the key to smooth integration of scalability optimizations in the AMS relied on the protocol enhancements.

We feel that none of the optimizations and enhancements presented here is particularly tied to Objectivity/DB. Any type of database server can benefit by simply choosing a layered architecture, optimizing each layer relative to its function, and allowing generic processing information (e.g., time delays, server location, etc.) to be passed through the protocol either for action by the client or by the server. While the advice sounds simple, our experience shows that the implementation is not. For very large databases however, the benefits far exceed the required effort.

Objectivity/DB, one of the very few commercial systems used within the *BABAR* software, proved to be a very robust and reliable system, which can be effectively deployed in a large production system, able to handle many terabytes of data along with hundreds of simultaneous clients.

7 Bibliography

1. J. Becla, *Data clustering and placement for the BABAR database*, CHEP'98, Chicago, Summer 1998
2. *BABAR* Webpage: http://www.slac.stanford.edu/BFROOT
3. CERN Webpage: http://www.cern.ch
4. R. Cowan, G. Grosdidier, *Visualization Tools for Monitoring and Evaluation of Distributed Computing Systems*, CHEP'00, Padova, Winter 2000
5. A. Hanushevsky, *Pursuit of Scalable High Performance Multi-Petabyte Database*, 16th IEEE Symposium on Mass Storage Systems, San Diego, CA, Spring 1999
6. High Energy Physics Information Center: http://www.hep.net/
7. HPSS, http://www.sdsc.edu/hpss/
8. LCB Status Report/RD45, CERN/LHCC 99-28, September 1999
9. LCB Status Report/RD45, CERN/LHCC 98-11, April 1998
10. LCB Staus Report/RD45, CERN/LHCC 98-6, February 1997
11. *Object Database Features and HEP Data Management*, CERN/LHCC 97-8, February 1997
12. *Object Databases in Practice*, M. Loomis, A. Chaudhri, 1998
13. Objectivity, Inc. Webpage: http://www.objectivity.com
14. D. Quarrie, *Operational Experience with the BABAR Database*, CHEP'00, Padova, Winter 2000
15. RD45 Webpage: http://wwwinfo.cern.ch/asd/cernlib/rd45/index.html
16. SLAC Webpage: http://www.slac.stanford.edu
17. Veritas: http://www.veritas.com