

Storage System Design using Finite State Machines

Amol Deshpande Mohana K. Lakhamraju

CS Division, EECS Department
University of California, Berkeley.
{amol, mohan}@cs.berkeley.edu

Abstract

This report addresses the use of alternate programming models in system design. In the context of developing a flexible high performance storage system, we explore the use of finite state machines as building blocks for composable system modules. We use request queues and completions based on upcalls as an abstraction to support asynchrony in the system. Using these design ideas, we have built an asynchronous storage system and a file manager and a web server that use the storage system. We compare the performance of these applications with their corresponding multi-threaded implementations. Our results indicate that the asynchronous design has a much better load sustainability and steady behavior.

1 Introduction

Design decisions in concurrent system development include a choice between multiple processes, multiple user-level threads for concurrency; synchronous versus asynchronous dispatching of events for communication; finite state machines, thread scheduling, continuation-passing for state and control transfer. At a high level, all of these primitives and programming constructs serve the same purpose. In this project, we explore the use of finite state machines (FSMs) as components in developing complex systems. Threaded programming models introduce inefficiencies from context switches and poor scheduling decisions. This is particularly problematic for applications (such as file servers or web servers) that desire high throughput and handle large client loads. Building such systems as compositions of finite state machines allow them to be highly data-flow oriented and overcome many of the scalability limitations of conventional multi-threaded systems. This model has been used in systems desiring high performance and

availability like the Flash web server and the IO-lite IO system. Event based FSM models have also been used in high performance transaction monitors that interact with multiple databases. They typically use persistent queues for event based interaction between multiple servers and guarantee load balancing and fail over.

We explore two design alternatives in the context of a universal storage system being developed under the *Telegraph*[Teleg] project. The goals of this work are two-fold.

- Design of an asynchronous version of the storage system based on the finite state machine model. One of the important aspects of such design is the composition of various subsystems using abstractions different from plain API calls that are used in multi-threaded models.
- Compare the performance of the asynchronous version of the storage system with a multi-threaded implementation in terms of scalability and resource utilization.

To this effect, we have implemented a proof-of-concept version of the storage manager that uses state machines and request queue-completion port communication mechanisms. To evaluate the performance of our implementation and to demonstrate the composition of state machines, we implemented an asynchronous Java file manager on top of the storage manager. Going up one more level in the vertical module stack, we also built a http server application that uses the file manager as its backing store. The performance of the implementations is evaluated at two levels, as a stand-alone file manager and as a web server using the SPECweb96 benchmark.

The rest of the paper is organized as follows. In section 2, we present the abstractions and the framework that we use. Section 3 presents design alternatives for composing different modules. We discuss

the implementation details in Section 4. Section 5 presents results from our performance study. We address related work in section 6 and conclude in section 7.

2 System Design

In this section we present an overview of the system and discuss some of the design aspects. First, we look at a couple of alternative mechanisms that can be used to compose systems in an asynchronous manner. Next, we present an overview of the BUDS system and how our work fits into it. Finally, we briefly introduce the file manager and http server modules that are used in our study.

2.1 Asynchronous communication and module composition

The mechanism we use for asynchronous communication is based on request queues and completion ports. The basic idea is that when a module needs some work to be done, it creates a request and places it on the request queue. It also registers a completion port where it wants the request completion to be posted. The module that services the request dequeues it and proceeds to service it. Once the request has been serviced, a completion event is generated and placed on the registered completion port.

There are two variations of this mechanism depending on the type of the completion port. Completion ports could either be passive or active. In the passive case, the port is implemented as some kind of queue that supports enqueue and dequeue operations. The sender enqueues the event into the queue and the receiver dequeues it. In the other alternative, the completion port is an upcall handler that the requester registers. In this case, when the request is satisfied, the upcall handler is invoked by the servicing module and appropriate actions can be taken in the upcall handler.

Though these two cases are potentially distinct, they can be captured in a unified interface by associating a simple upcall handler function with the passive queue. This upcall handler when invoked by the servicing module simply places the completion on the queue it represents.

In general, for active completion ports, the upcall handlers could be complex and could themselves cause other requests to be generated. A simple ex-

ample is the case where a file read completion causes a network write request to be generated.

The upcall handler is executed in an execution context coming from the request servicing module, which, in the general case may not have any knowledge about the actions taken in the upcall handler. In such a case, it may be expected that the execution context returns back immediately. Hence, one of the desirable properties of upcall handlers is that they take a short duration to execute and return the context back to the caller. The module executing the upcall handler may assume that the upcall handler never blocks. Therefore, it is important to ensure that no blocking activities are done in such upcall handlers. Another property that is important is that of reentrancy and thread safety. Unless explicitly controlled, multiple event completions may happen at the same time resulting in multiple upcall handler executions.

Upcalls are a nice abstraction that can be used in building systems as state machines. The role of upcalls in such systems would be to cause the appropriate state transitions. In later sections, we look at some examples of state machines and the transitions caused using upcalls.

Another use of the request queue, completion port abstraction that has been implicit in the above discussion is that of flexible composition of modules in tiered systems. This abstraction can be used to implement thread barriers. In a tiered system, when no assumptions can be made about the modules that would be build on top a certain module, a thread barrier implemented using a request queue, completion port gives the flexibility of being able to build both kinds (multi-threaded and single-threaded state machine model) of modules on top. If the higher module is multi-threaded, then the threads could block on the completion ports. Instead if it is implemented as a state machine, then it can register an upcall with the completion port which would be invoked on event completion.

We use the request queue, completion port mechanism widely in our system both passively and actively and to implement thread boundaries. Before presenting the implementation details, we briefly introduce the general purpose storage system which is the context of our implementation.

2.2 BUDS Overview

Currently, developers of applications with storage needs do not have an easy solution if neither file systems nor database systems meet their requirements appropriately. Database Management Systems are complex, heavy-weight pieces of software that are highly optimized for specific workloads and are very hard to build and maintain. This is the reason why they are used mainly in mission-critical applications that require high levels of consistency, recovery, and scalability. Most other applications either develop their custom data storage alternatives, or depend on file systems for their storage needs, implementing application specific requirements on top. For instance, search engines like Inktomi and Google have built custom storage managers to meet their data requirements. Other services like email servers, web servers, often build custom data structures over file systems. The Porcupine mail server developed at University of Washington [SHB98], that is capable of handling one billion email messages a day uses a database for user specific data, but reads all the data to redundant memory data structures at boot time. The Jaws high performance web server [HPS99] has a virtual file system in memory that reduces the file system access overhead and supports various caching strategies. These examples demonstrate the need for a unified and customizable storage solution.

The Berkeley Universal Data Storage (BUDS) system is a general purpose storage manager currently under development at Berkeley [LBH99] aimed at addressing the storage needs of a wide range of applications. It exports a flexible and extensible API that can meet the requirements of applications like fully transactional database systems, distributed data structures for scalable and highly available Internet services and traditional file systems. Using this API, an application developer can configure the properties of the storage system like consistency models, access methods and redundancy.

Figure 1 shows the BUDS architecture. It consists of multiple layers of functionality with a fair degree of independence between layers. We refer to the underlying unit of storage on disk as a segment. The lowest layer is a segment-based IO subsystem which is responsible for IO and management of segments on disk. The next layer is the segment management layer which consists of various modules namely, buffer manager, transaction manager, lock manager and recovery manager. Higher up is the access method layer

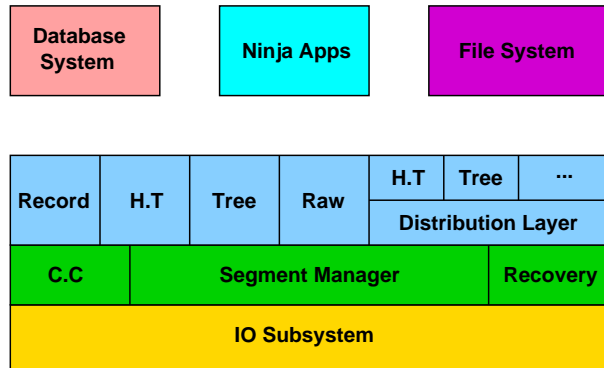


Figure 1: BUDS architecture: In order from bottom - IO layer, segment management layer, segment access layer, applications

which exports various access methods to the storage system including raw file access, record-based access, hash table access and a tree-based access method. In order to support distributed data structures and replication, there is a distribution layer which essentially provides name space management support ¹. The three applications shown at the top of the figure 1 are a sample of the potential targets for BUDS. The Ninja ² applications are essentially scalable persistent data structures.

2.3 IO Subsystem Overview

The storage manager uses an IO subsystem that was developed by Steve Gribble [Ninja]. The IO subsystem exports an asynchronous interface based on request queues and completion ports model that was discussed earlier. The IO subsystem is also implemented in Java. Since Java does not provide asynchronous IO, the IO subsystem simulates it by using a pool of threads at the lowest level. This thread pool is initialized at system initialization and the threads are reused to avoid the cost of thread creation for each request. When an IO thread completes a request, it places a completion event on the completion port that was registered along with the request.

Java does not provide a direct interface to the disk. Hence the IO subsystem using the file system and stores segments in files. One limitation that this causes is that we have no explicit control in the storage manager about flushing data to disk since the

¹The segment access layer has not been implemented yet

²Ninja is a project currently active at UC Berkeley. <http://ninja.cs.berkeley.edu>

underlying files are cached in the file system cache. There are efforts underway (beyond the scope of this project) to address this issue and we hope to have a solution in the near future.

The IO subsystem is integrated with the storage manager using a thread boundary so that none of the IO threads can come into the storage manager. The completion port that is registered when an IO request is made is a queue that enforces this thread boundary. IO completions are posted on this queue and the storage manager dequeues these completions and takes the appropriate actions.

The IO subsystem also includes an asynchronous TCP sockets implementation. Again, the asynchrony is faked by using a pool of threads that themselves do synchronous network IO. The asynchronous network interface is used by the Http server. Since we are not using true asynchronous IO, we have to pay a penalty in certain cases. We will refer to some such cases in the section detailing performance of the system.

2.4 File System Application

In this project, we have implemented a vertical slice of the BUDS architecture in Java using both the state machine model and the multi-threaded model. Our file manager is built as an application that uses the raw file access provided by BUDS. The following is a brief description of the functionality provided by each of the modules in this vertical slice, top to bottom.

2.4.1 File Manager

This module exports a subset of the Java file access API. One of the functions it manages is the mapping from arbitrary file names to internal segment names. This mapping provides a level of indirection between user specified names and the internal compact representation and is stored internally in a persistent hash table. Though not currently implemented, the other important function of this module is access control.

2.4.2 Storage Manager

The storage manager internally consists of multiple modules of which the two modules most relevant to the file system application are the buffer manager and segment manager. The segment manager deals with the physical layout of data in the segments and updates to this data. It also manages the free space within segments. The buffer manager maintains an in-memory cache of the segments being accessed by

the file manager. It provides functions for both write-back and write-through of updated segments. It keeps track of dirty segments and flushes them when needed. It maintains a map of all the cached segments and their descriptors in an in-memory hash table. The buffer manager supports functions for reading, writing, seeking, flushing, creating and destroying segments on disk using the underlying IO subsystem.

In terms of implementation, we have both multi-threaded and single-threaded state machine based implementations of both these modules. Details of these implementations and their composition are presented in subsequent sections.

2.5 Http Server

Finally, as an application that uses the file manager API, we have developed a simple Http server. The server accepts connections from client machines and waits for their requests for files. Once it gets a file get request, it retrieves the file from the file manager and sends it back to the client. Again, we have two implementations of the web server, one communicating asynchronously with the network and the other communicating synchronously.

3 Design Alternatives for Module Composition

In this section, we describe the three design alternatives available to an application to be built on top of Storage Manager and we also discuss the kind of interface that needs to be exported so that the application programmer has maximum flexibility in implementation.

Given that the Storage Manager itself is built as a single threaded module, an application built on top of the Storage Manager has three choices :

- **It can share a thread with the Storage Manager :** In this case, we integrate the Storage Manager and the application completely. As shown in figure 3, this requires that the application programmer should write an *Event Dispatcher* which receives events directed for both the Storage Manager and the application and directs them to the appropriate location.

Though this scenario is probably the most efficient one, it may not be useful when there are

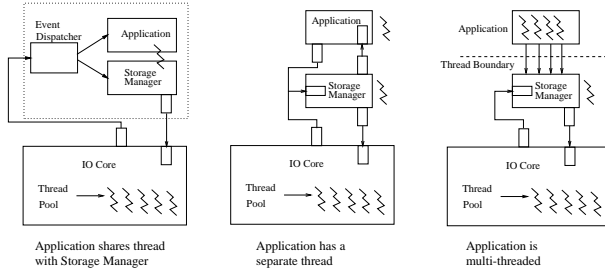


Figure 2: The three alternative ways in which modules can be composed

multiple applications running at the same time. It is conceivable that all these applications share a single thread with the Storage Manager, but that will break the modularity of the design. File sanager is an ideal application for this purpose.

- **Application is single-threaded, but uses a different thread :** In this case, the application uses request queues and completion events mechanism for interacting with the storage manager as described above. This option is most useful when we have multiple application coded by different application programmers which want to use the same storage sanager.
- **The application is multi-threaded :** In this case, there is a thread boundary between the application and the Storage Manager which is implemented using the request queues and completion event mechanism. Since the application is multi-threaded, it should not be required to deal with this mechanism and hence, the Storage Manager exports an *synchronous API* using a wrapper.

4 Implementation Details

In this section, we describe the implementation details of the two versions of the end-to-end system that we have built.

4.1 Synchronous Implementation

In the synchronous implementation, as shown in figure 3, the Http Server is built as a multi-threaded application where a new thread is spawned for servicing each request. The thread travels all the way down to the IO layer and makes a request for the IO.

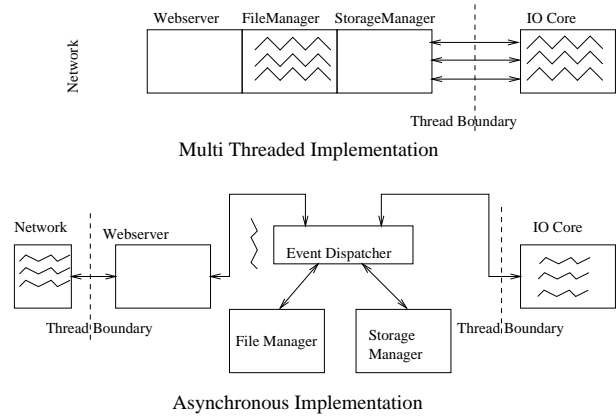


Figure 3: Architecture of the multi-threaded and the asynchronous versions of the end-to-end system

Since we use the same asynchronous IO layer here, we need to create a local queue here on which the calling thread blocks till the IO is finished. Unlike the earlier case where only one thread was executing the Storage Manager code, we need synchronization in Storage Manager and the File Manager to make sure that there are no race conditions.

4.2 Asynchronous Implementation

As shown in figure 3, the File Manager is implemented so that it shares a thread with the Storage Manager. The Http Server, on the other hand, is built as a separate thread which listens on the network to accept connections. The Http Server uses the asynchronous network IO mechanism described earlier. When it receives request for a new connection, it creates an Http Session for servicing that request. Http session is built as a passive object whose code is executed by different threads at different points of time. When a GET request is received, the network reader thread reading it passes the control to the Http Session object which makes an asynchronous request for opening the corresponding file. When the open completes, the File Manager thread, again using the upcall handler mechanism passes the control to the Http Session object which then sends the data over the network asynchronously. Finally, when the network writes are completed, the corresponding http session object is destroyed.

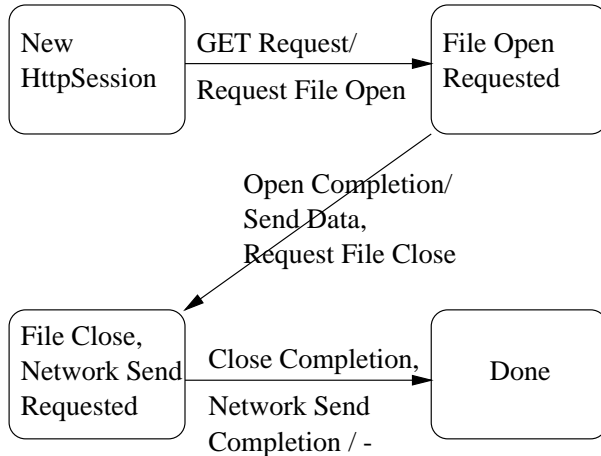


Figure 4: Http Session Finite State Machine

4.3 Examples of state machines

We present two state machines that we use to serve as an examples of how the a state machine based system can be architected.

4.3.1 Http Session FSM

The figure shows a simplified version of the HttpSessionFSM. An HttpSessionFSM is created in response to a request by a client by the HttpServer thread. The HttpSession created is registered as the Upcall Handler for the thread reading on the corresponding socket. When the read is finished, HttpSession is evoked. It sends an asynchronous request for opening of the file and when the open if finished, it sends the data to the requesting client asynchronously and also makes an asynchronous request for closing the file. It then waits for the network send completion event and also the file close completion event and then exits. Note that, the HttpSession FSM is handled by various thread at different times. The state transition from **New HttpSessionFSM** to **File Open Requested** is handled by the thread which reads from the network. The next transition is effected by the File Manager thread, whereas the handling of network send completion event is done by the threads doing the network writes.

4.3.2 Segment Close FSM

A SegmentCloseFSM is created in response to a request from the file manager to close the corresponding file. If the segment is not dirty, then the FSM creates

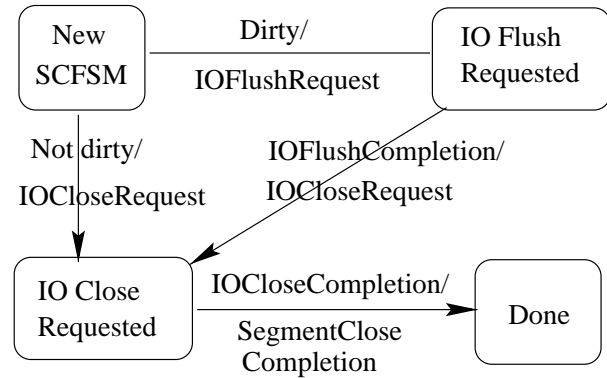


Figure 5: Segment Close Finite State Machine

a segment close request with the unified event dispatcher queue of the storage manager registered as the upcall handler and enqueues the request with the IO Manager. The IO Manager closes the underlying file and enqueues the completion in the event dispatcher queue of the storage manager by calling the upcall handler. When this event gets dispatched, the FSM's upcall handler is invoked by the storage manager thread. This causes a segment close completion event to be created and sent up to the file manager using its upcall handler. If the segment were dirty in the first place, then a segment flush request would be created instead of the close request. This would go through the same process and return to the FSM, when then creates a segment close request and the same process as above repeats. The state transitions of this machine are shown in figure 5.

4.4 Implementation Complexity Analysis

In this subsection, we qualitatively compare the multi-threaded and state machine implementations and discuss our experiences in developing the asynchronous storage manager and the file manager application.

- **Control flow:** In the normal threaded model that most of us are used to programming in, the execution context is local and is easily comprehensible. In the state machine model, the execution context is distributed and harder to comprehend, especially during development. If the system has to be extended, changes have to be made at many different places since there is no code locality like in the case of the threaded model.

On the other hand, the state machine model is highly data flow oriented and has better data locality since data is acted upon through a sequence of state transitions. This has the potential for exploitation of processor cache locality.

- **Overhead:** Multi-threaded code has a high synchronization overhead that can be avoided by using the state machine model. As the system scales and the number of threads has to be increased, this overhead becomes increasingly unacceptable. In our experiments, we observed up to 50% of response time being spent on synchronization. A concrete example occurs when the main file manager thread tries to submit data to be sent over the network. Since asynchrony was being faked using a thread pool, there was synchronization required which was blocking the important thread. Interferences like this could lead a tremendous performance degradation.
- **Code Complexity:** Code size, specifically in Java, and complexity is much more in the state machine model compared to the multi-threaded model. The reason is that separate classes are required for all the requests and completion events and the different state machines. For our implementations, the threaded one took 2000 lines of java code while the state machine model required 5000 lines. The fact that the code is not organized linearly through function calls causes additional complexity.
- **Debugging:** Quite surprisingly, debugging was easier with the state machine model than with the threading model. One important reason for this is that the language being used is Java. The other reason is that debugging multi-threaded applications is fundamentally hard. It is hard to reproduce deadlocks and race conditions and these bugs can potentially surface any time. It is easier to be confident about bug-freeness of single-threaded code comparatively.

5 Performance Study

In this section, we present results comparing the performances of the multi-threaded and the asynchronous models. We will first describe the experimental setup, then compare the performance of the file manager for reads and finally, compare the per-

formance of a simple HTTP server built on top of the filemanager.

5.1 Experimental setup

The experiments were run on 500 MHz Pentium II with 256MB of memory running Linux. All the experiments were done using Sun JDK 1.2.2 with *green threads* (unless specified otherwise) and *just-in-time* compilation enabled. The number of threads in the IO core threadpool was fixed at 60 threads, whereas the number of threads in the network core threadpool was fixed at 50.

5.1.1 Metrics and Methodology

We compared the performance of both the File Manager and the Web Server using the *average response time* and *throughput* that they provide. Throughput for the file manager is defined as the number of file read or write operations it supports in unit time. For the web server, it is defined as the number of *HTTP GET* requests per second that it can support. To eliminate the effects of startup time and shutdown time, we ran the experiments for large number of requests and then took the measurements over 5000 thousand requests in the the middle portion of the request stream.

5.1.2 Spec96 Benchmark

All the experiments that we report here were performed against the SPECweb96 benchmark. The data set that we use is also defined by this benchmark. This benchmark has been designed for testing the performance of web servers. It divides the requests made according to the size in four categories : 100 bytes to 1kb : 35%, 1kb to 10kb : 50% 10kb to 100kb : 14%, 100kb to 1MB : 1%.

Each category consists of 9 different file sizes and the distribution within each category is according to Poisson distribution about the mean of the sizes in that category. Thus there are total 36 files in directory. We used this benchmark with the number of directories fixed at 100.

5.2 File Manager Performance

We compared the performance of the file read requests on the distribution as described above for both the models. We also did benchmarking to figure out exactly where the time goes for both the models.

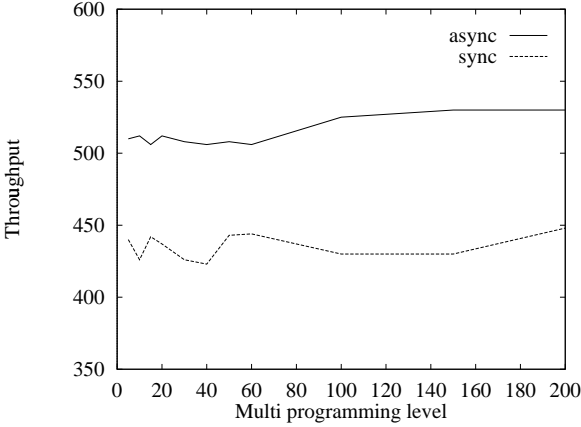


Figure 6: Throughput

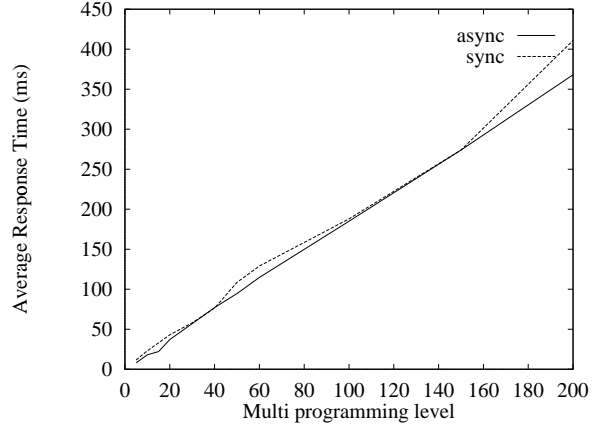


Figure 7: Average Response Times

- Asynchronous Model :** The maximum number of requests that can be in progress at any time is limited by the application. The application runs as a single thread which starts off with issuing these many asynchronous *open* requests. When the application receives the *open completion* event, it reads the data (this is done by the application thread synchronously) and issues an asynchronous *close* request. When the application receives the completion event for the *close*, it issues another *open* request thus ensuring that there are always same number of requests in progress inside the system. As we increase this number, we expect that the average response time will increase because of interference and increased queue waits, whereas the throughput will also increase.
- Synchronous Model :** In this case, the maximum number of requests in progress is limited by the number of threads the application has. Each thread synchronously performs *open*, *read* and *close* operations continuously. Here also we expect that both the response time and throughput will increase with increasing number of threads, though the throughput should start decreasing after a certain stage.

Figures 6 and 7 compare the performance of the two models under warm cache conditions. The requests were generated according to the Spec96 benchmark distribution as describe above. As we can see, with increasing number of maximum requests in the system, the response times and the throughput increases, though the increase in throughput is not as

much as we expected. In fact, it remains more or less independent of the number of threads or the multi-programming level. It can be seen though, that the performance of the asynchronous case is approximately 20% better than the performance for the synchronous case. The main reason there is not much difference in performance is because, as we will see soon, the time for IO is very small. Also, we are using *green* threads, which do not have much scheduling overhead.

5.2.1 Microbenchmarking

To explain the results that we reported above, we decided to break down the time taken by a request like *open* (which under our implementation results in reading the segment into memory) or *close*. Here we present those results.

5.2.2 Asynchronous Model

The time spent by *open* request can be roughly divided into intervals as shown in the table :

| MPL | 5 | 50 |
|---|-------|-----------|
| Time till IO request is generated | 0-1ms | 0-1ms |
| Time till IO starts | 0-1ms | 0-1ms |
| Time till IO finishes | 0-2ms | 0-2ms |
| Time till the Storage Manager thread posts the completion event | 0-2ms | 0-2ms |
| Time till application thread gets this event | 4-6ms | 100-200ms |

The table shows the approximate amount of time spent by the request in each of these stages for multi-programming levels 5 and 50. As we can see, almost all the time is spent by the completion event to wait for the application thread to get scheduled and service it. This time depends almost linearly on the maximum number of requests allowed in the system. This is because the Storage Manager thread runs at the highest priority and services all requests it can before yielding the CPU. Since the average number of requests it has to service at any time will be linearly dependent on the maximum number of requests, we get that the average response time increases linearly with the number of threads.

5.2.3 Synchronous Model

In this case as well, we divided the time spent by on an *open* call as shown in the table :

| No of Threads | 5 | 50 |
|-----------------------------------|-------|-----------|
| Time till IO request is generated | 0-1ms | 0-1ms |
| Time till IO finishes | 5-6ms | 100-200ms |
| Time for open call to finish | 0-1ms | 0-1ms |

The time for IO completion can be further broken down into the actual time for IO and the waiting time, but since the IO time is really small, we decided to ignore it. The table shows the microbenchmarking results for 5 threads and 50 threads. As we expect, an application thread spends most of its time waiting to regain the control of the CPU. Again, since the average number of threads waiting for IO to finish is proportional to the total number of application threads, the average time to finish is proportional to the total number of threads.

5.3 Webserver Performance

In this section, we present the results from running the SPEC96 web server benchmark on the two version of web versions that we have implemented. The setup for these set of experiments includes, in addition to a web server, a set of clients running on linux machines different from the server. Each client makes HTTP GET requests to the server based on a reference stream generated using the Spec96 benchmark described at the beginning of this section.

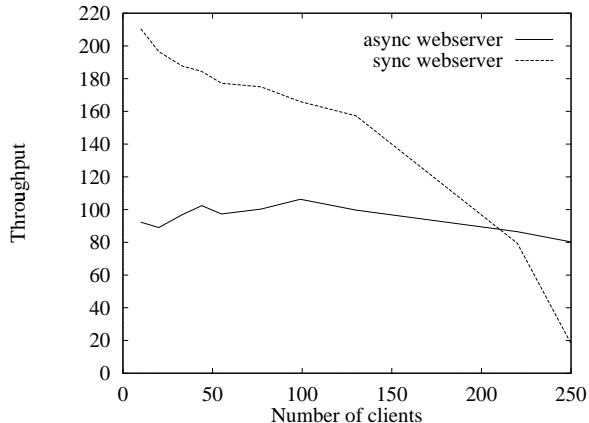


Figure 8: Throughput comparison with clients using native threads

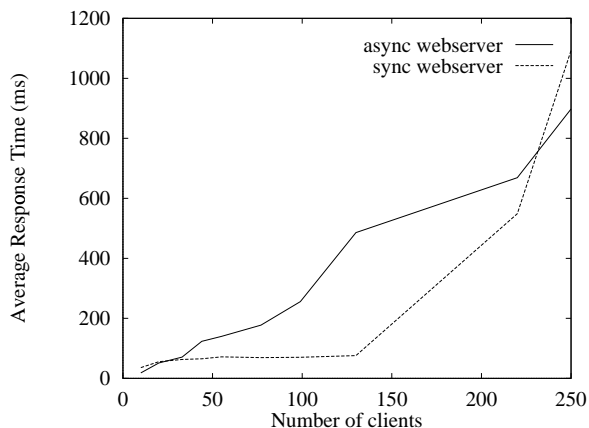


Figure 9: Average response time comparison with clients using native threads

The web server was always run using Java green threads since native threads do not scale beyond a small limit. When native threads were used, the thread pools used to simulate asynchronous disk and network IO were proving to be too expensive. On the other hand, the clients were run using both green threads and native threads. We observed that the performance in case of native threads was much better. This is explained by the fact that that machines we used to run the clients were 8-way SMPs. Hence native threads were able to use more number of processors and improve the performance of the clients when the server was not the bottleneck.

Figures 8 and 9 compare the performance of the asynchronous and synchronous implementations of the web server when the clients use native threads in

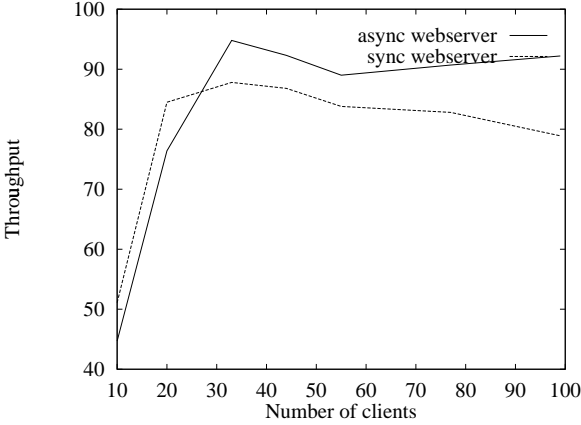


Figure 10: Throughput comparison with clients using green threads

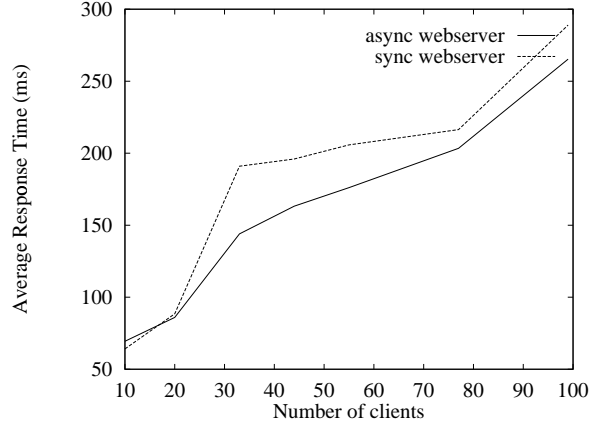


Figure 11: Average response time comparison with clients using green threads

Java. As can be seen from the throughput comparison graph, at low client numbers the multi-threaded server performs much better. The reason for this is that the number of threads active at the server at any point in time is pretty low here causing the synchronization overhead to be low. As the number of clients is increased, we see the throughput of this server falls drastically to the point that at around 250 clients, it is almost at a standstill. On the other hand, the asynchronous version of the web server is more stable with increasing load. Part of the explanation for this behavior is that the request queues in the asynchronous implementation impose some degree of admission control. We give a higher priority to the main storage manager thread so that when it looks at the request queue it dequeues and services all the outstanding requests. While this is being done, new requests get queued up in the request queue. Thus, there is explicit batching of requests and request completions that we have confirmed by looking at the server logs. This batching behavior also explains the increasing average response time. Another observation based on the server logs is that in the synchronous implementation, at higher client loads, the requests and their responses basically get serialized explaining the sharp increase in average response time at the very high client numbers. Overall, the win for the asynchronous web server comes from the steady throughput sustained assuming that response time is not the critical metric.

Figures 10 and 11 present the same comparison as above for the case when the clients use Java green threads. Here we observe that there is very little

difference between the performance of the two implementations. One reason for this is that since most of the IO requests are satisfied by the cache under the Spec96 workload, there is very little synchronization interference between the different threads.

Though this performance study gives a reasonable flavor of the merits of the asynchronous model, true asynchronous disk and network IO is required to get the complete picture. Some of our results could be misleading because of the fake implementations of the IO modules using thread pools.

6 Related Work

The merits and demerits of various programming models have been widely debated in literature [ABLL92, DBR91, Cla98, Ren98, HPS97]. In [ABLL92], the authors discuss the merits of user-level threads arising from their light-weightedness and propose kernel support for user-level threads. At the same time, programming with threads is error-prone, hard to debug and often non-portable [Ren98]. Moreover, it is extremely hard to compose packages with different thread abstractions. In [DBR91], the authors claim that using continuations as the basis for control transfer in the thread and IPC facilities of Mach 3.0 operating system saved 85% on space per thread and greatly improves performance. In [HPS97], the authors examine web server design and present a comparison of synchronous and asynchronous event dispatching and various concurrency strategies like thread-per-request,

thread-pool, thread-per-connection. They conclude that a asynchronous event dispatching mechanism using a common thread-pool fits the context well. All these mechanisms differ in the abstractions they provide and are not easily composable.

Another design that is being explored is to implement the IO subsystem and the transaction subsystem as finite state machines with a single thread of control passing through them. It has been shown previously that finite state machine based design could be very light-weight and efficient [PDZ99, PDZ98]. Such an implementation obviates the need for explicit synchronization since there is a single thread of control active in the state machine at any time. While it is possible to structure the storage system as a composition of state machines it is restricting to assume that applications using the storage machine would also be state machines.

Integrated Layer Processing is an implementation concept which permits the implementor the option of performing all the data manipulation steps in one or two integrated processing loops. It has been used in networking protocol implementations [BD95]. Finite State machine based design of the storage system lends itself well to Integrated Layer Processing (ILP). In the case of the storage manager, this corresponds to a data-driven approach where data is processed through a sequence of state machines before other data is processed. Such a data-driven approach would have good data locality though it potentially compromises code locality. In comparison, a threaded implementation has good code locality as it is more control driven and not data driven. It should be pointed out here that the ILP approach is not different from that of a iterator-based query plan processing [Gra90].

7 Conclusions and Future Work

The subject of programming models for complex system design has and will always be hotly debated. In this report, we contribute our due share to this discussion by exploring alternatives for such system design in the context of building a storage system. We have described our experience with building a the storage manager using Finite State Machines and compared its performance with a system built using a thread-per-request model. We found that the asynchronous implementation usually performs better than the thread-per-request model, though the

performance benefit is not high unless the time for servicing requests is a significant portion of the total response time.

In terms of design and code complexity, we found that using FSM's has both advantages and disadvantages. There is a definite performance improvement, but the code is more distributed and lacks a linear flow like in threaded implementations. We believe that this is not a serious limitation and once that can definitely be overcome with some practice. In retrospect, using Java as the programming language helped tremendously in reducing the debugging effort. On the other hand, though the code is much easier to write and understand with threads, it is harder to debug and guarantee bug-freedom since race conditions are hard to reproduce.

Future work includes the enhancement of the file manager API so that it can be released for general use. We are also in the process of implementing more functionality into the BUDS system using the state machine model including transaction management, recovery and concurrency. Another aspect that needs more work is the validation of our claims on a system that supports true asynchronous IO.

References

- [LBH99] Mohana K. Lakhamraju, Rob von Behren, J. Hellerstein The Design of the Telegraph Storage Manager UCB Technical Report, Oct 1999.
- [Teleg] The Telegraph Project, UCB. <http://db.cs.berkeley.edu/telegraph>
- [Ninja] The Ninja Project, UCB. <http://ninja.cs.berkeley.edu>
- [ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism
- [HPS97] James C. Hu, Irfan Pyarali and Douglas C. Schmidt Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks In Proceedings of the 2nd Global Internet mini-conference, GLOBE-COM '97
- [HPS99] James Hu, Irfan Pyarali, and Douglas C. Schmidt The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks The Parallel and Distributed Computing Practices journal, special issue on Distributed Object-Oriented Systems, 1999

- [WC99] Matt Welsh and David Culler Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, Special Issue on Java for High-Performance Applications, December, 1999.
- [GC96] M. Greenwald, D.R. Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure. *Proceedings of the Second Symposium on Operating System Design and Implementation*. USENIX, Seattle, October, 1996, pp 123-136.
- [Cla98] David D. Clark Structuring Operating Systems using Upcalls. *ACM symposium on Operating Systems Principles '91*, Pacific Grove, CA, USA, Oct. 1991.
- [PDZ98] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, Flash: An Efficient and Portable Web Server. *USENIX Annual Tech Conference*, Monterey, CA, USA, June 1999
- [Ren98] R. van Renesse Goal-Oriented Programming, or Composition Using Events, or Threads Considered Harmful. *Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. Sept 1998.
- [SHB98] Y. Saito, E. Hoffman, B. Bershad, H. Levy, D. Becker. The Porcupine Scalable Mail Server. *Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. Sept 1998.
- [BD95] Torsten Braunl and Christopher Diet Protocol Implementation Using Integrated Layer Processing. *Proceedings of the ACM SIGCOMM '95*, Cambridge MA, p161-151.
- [Se93] Margo I. Seltzer. Transaction Support in a Log-Structured File System. *ICDE 1993*, p.503-510.
- [Bat86] D. S. Batory GENESIS: A Project to develop an Extensible Database Management System. *Proceedings of Int'l Workshop on Object-Oriented Database Systems*, Sept 1986.
- [St87] Micheal Stonebraker. The design of the POSTGRES Storage System. *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB) 1987*, p289-300.
- [PDZ99] Vivek S. Pai, Peter Druschel, Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *Proceedings of the 3rd Usenix Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Feb 1999.
- [MHL⁺92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transaction on Database Systems*, Vol 17, No. 1, March 1992, p.94-162.
- [FGC⁺97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, Paul Gauthier. Cluster-Based Scalable Network Services. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint-Malo, France, Oct 1997.
- [CHS99] Micheal J. Carey, Joseph M. Hellerstein, Micheal Stonebraker. *Designing the B-1: A Universal System for Information*. CS Division, UC Berkeley 1999. (submitted for publication).
- [Ca86] Carey M.J, et al. The architecture of the EXODUS Extensible DBMS. *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, 1986.
- [DBR91] Richard P.Draves, Brian N.Bershad, Richard F.Rashid, and Randall W.Dean Using continuations to implement thread management and communication in operating systems. *Proceedings of the thirteenth ACM symposium on Operating systems principles* October 13 - 16, 1991, Pages 122 - 136
- [Gra90] Goetz Graefe Encapsulation of Parallelism in the Volcano Query Processing System. *Proceedings of the ACM SIGMOD '90*