Asynchronous Bulk File I/O in Titanium, a High-Performance SPMD Java Dialect

(CS264 Final Project, Fall 1999)

Dan Bonachea*
EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
bonachea@cs.berkeley.edu

Abstract

Titanium is a high-performance explicitly parallel SPMD dialect of Java. Currently, the runtime support for file I/O is limited to the classes present in Java, which have proven too inefficient to meet the demands of high-performance scientific applications which perform large amounts of file I/O. A new library is presented which adds support for asynchronous file I/O operations to enable masking I/O latency with overlapped computation. The new library also provides support for bulk (array) I/O operations, removing much of the overhead associated with the Java I/O libraries which require the programmer to read arrays a single element at a time. Experimental results are presented which compare the performance of the new library with the existing I/O libraries on a simple external merge sort, and show the performance of the new library to be vastly superior to the existing I/O facilities for this application.

1.0 Introduction

Titanium is a high-performance, explicitly parallel dialect of Java recently developed at UC Berkeley for programming shared-memory and distributed-memory parallel systems. Titanium incorporates the power of Split-C, a low-level SPMD language, into a high-level object-oriented programming language that frees the programmer from much of the tedium associated with writing and debugging parallel programs. Titanium is essentially a superset of Java 1.0, including all the expressiveness and safety features of that language, with a wealth of new features that support high-performance SPMD programming, such as: immutable classes, zone-based memory management, local and global references, flexible and efficient multi-dimensional arrays, unordered loop iteration, and a library of useful parallel primitives including barrier, broadcast, exchange, and various reductions (see [YEL98], [AIK98], [TIT99] for details). The compiler also performs extensive static analysis (with some assistance from programmer annotations) to statically guarantee freedom from deadlock, a notorious problem when writing explicitly parallel programs (see [GAY98]). The primary goals of the language, in order of importance are: performance, safety and expressiveness. Titanium is especially well adapted for writing grid-based scientific parallel applications, and several such major applications have been written and continue to be further developed.

The Titanium compiler performs various optimizations using knowledge of the parallel control flow, and translates programs to C, where they are compiled (and optimized further) by gcc and then linked to the proper Titanium runtime libraries. The Titanium backend has been ported to several platforms, including the Berkeley NOW, Solaris SMP, Cray T3E, Tera MTA, Posix threads, and Solaris and Linux uniprocessors (The Berkeley NOW is a cluster of Ultra-1 SPARCStations, see [NOW99]). The backend on distributed memory machines (e.g. the NOW) utilizes the Split-C runtime library, which is built on top of Active Messages-2 (see [EIC92]). AM-2 is an application-level communications protocol optimized for high-speed networks (multi-gigabit bandwidths and sub-microsecond switch latencies) that allows applications to bypass the operating system and exercise direct control over intelligent network cards. Because AM-2 is a low-latency, application-level protocol, it uses a polling-based interface (rather than an interrupt-driven one) as the communication paradigm, which has been shown to provide a significant reduction in software overheads (see [MAI95]). This means the Titanium Split-C backend must occasionally poll the network to service requests from other processors (for example, remote memory accesses).

The only major Java feature which is not currently supported by Titanium is user-visible multi-threading. That is, the user cannot create arbitrary asynchronous threads of control within a process – the SPMD programming paradigm is central to the design of Titanium, and this programming model doesn't typically allow arbitrary threading. A Titanium program runs on a fixed set of processors (which may be a virtual processors in the case of SMP and uniprocessors, or real processors in the case of distributed systems) – the number of processors is set when the parallel job begins, and remains constant throughout the run of the program. This property is useful in implementing the optimizations and features of Titanium, and crucial to the static prevention of deadlock. Several proposals have been made to integrate a limited form of threads into Titanium (see [BEG98]). All current proposals

* This material is based in part on work supported by DARPA contract No. F30602-95-C-0136, NSF contract No. CDA-9401156 and a SLOAN fellowship.

involve a notion of "thread groups" where programs begin with a single thread group, and all processors in a thread group can simultaneously spawn a new thread, creating a new thread group (essentially doubling the number of processors). Processors within a thread group still behave as SPMD processors, executing and synchronizing in lock-step. It seems no final decision has been made thus far about whether to integrate some form of user-created threads, but the work continues. (Incidentally, Titanium also currently does not support the java.awt (GUI) library, which relies on multi-threading.)

The primary motivations for adding some form of multi-threading to Titanium are to provide support for multiple dimensions of SPMD parallelism (which seems more natural than a single dimension in some applications), and to enable asynchronous (non-blocking) file I/O to hide disk latency. The current file I/O support in Titanium is entirely blocking, which means that processors performing I/O can be suspended for tens of milliseconds waiting for disk accesses. The situation is especially bad in the context of Active Messages, because suspended processors don't currently poll for requests, so blocking I/O may also hold up other processors that need to access the memory of the blocked processor. Unfortunately, one of the defining characteristics of high-performance scientific applications is that they frequently involve massive amounts of I/O. For example, [ROS93] reports that typical modern supercomputer applications involve anywhere from 1GB to 4TB of I/O per run and I/O rates of up to 40MB/s. The primary techniques for maintaining high performance with this level of disk activity are tuning the data distribution across nodes and providing implicitly or explicitly overlapped I/O. Data distribution is an important and well-studied topic, but is beyond the scope of this paper – the interested reader is referred to [ACH95], [CHO93] and [KEN94].

The contribution of this project is the design and prototype implementation of an asynchronous file I/O interface[1] for Titanium. The goal was to provide the Titanium programmer with efficient, non-blocking I/O routines that would permit overlapping I/O with continued computation and network activity. The prototype is written primarily in Java, with crucial functions written as native C methods. The native method code is specific to the Titanium runtime system, but the library could easily be ported to standard Java and most likely to other Java-based languages; the concepts are definitely applicable there as well.

2.0 Existing Asynchronous I/O Implementations

Historically, asynchronous I/O has often been sited as an important optimization in masking disk latency, but it seems to rarely be implemented at the level of the application or even the runtime system, and there is no widely accepted standard interface. The reason seems to be that the implementation of synchronization and threading varies widely between various languages and operating systems. Most modern operating systems implement some form of file-system buffering (which can be seen as a limited form of asynchronous I/O) that does effectively overlap some disk latency with ongoing computation. However, this buffering tends to work best with sequential reads and writes (where sequential pre-fetching and write-behind caching help out), but in any case cannot effectively remove the latency and resource overhead associated with copying the data to and from the OS buffers (which may be too small to handle the requests being issued – these are limited to the physical memory of the machine for obvious reasons). For details, see [JAC91], [SHI96], [WOR94].

Implementations of asynchronous I/O seem to cluster about 2 major points in the design space. The first type is multi-threading of blocking I/O – where the application programmer writes code to explicitly create a new thread that calls a standardized blocking I/O interface, and to synchronize that thread with the computation thread once the I/O completes. This approach places the most burden on the applications programmer and can quickly lead to very messy code or deadlocks because the programmer has to manage the I/O synchronization explicitly. However, this approach does have the advantage of being the most portable of the three – although even this is not very portable, because the multi-threading interface can vary widely between platforms. This approach also seems to work best when programmed to be explicitly aware of the virtual memory system – first, to prevent buffers from being swapped out to disk, and second, to reduce the number of intermediate in-memory data copies to one or zero. This generally makes implementations even more platform-specific.

The second option relegates the bulk of the complexity to the kernel or an I/O library, which manages the various I/O threads internally, providing the programmer with a relatively simple interface for making requests and getting the results. There are variations in how these implementations communicate results to the application – some use an interrupt-based approach, where the application receives a signal or a call to a handler routine when the operation completes. Others use explicit synchronization, where the application must explicitly poll the status of or wait for the results of an ongoing asynchronous I/O operation. Still others are some hybrid combination of these approaches.

[1] Throughout this paper, references to the word "interface" mean it in the sense of the Application Programming Interface (API) of the class library that was designed, not in the sense of a Java interface, which is an object-oriented language construct.

All such interfaces decree that the application is responsible for maintaining the memory buffer associated with the request allocated and untouched throughout the duration of the operation to prevent the overhead of making an additional copy solely to ensure this property.

The Microsoft Windows NT operating system has direct support for "Overlapped Files" within the Win32 API [MS99]. In this interface, the application initiates an I/O operation and specifies either a handle for the operation or a completion routine callback. The application can check the status of ongoing operations, stop and wait for a particular operation to complete, or poll for any queued completion routines (callback routines execute synchronously when the application explicitly indicates it is ready to receive them). The interface allows the application to specify at file open time whether the I/O should be buffered (which implies an additional copy to/from the OS buffers) or unbuffered (zero copies – the buffer passed by the application is used directly by the disk I/O driver). The important caveat associated with unbuffered files is that all I/O must made in sizes that are an even multiple of the disk volume's sector size, and the I/O requests must be sector aligned in memory and on-disk. This is a significant limitation, but is fundamental to any zero-copy scheme because all modern disks perform I/O in aligned sector units ([RUE94],[RUE91]). The advantage to having such a low-level interface available is the application can directly control its caching behavior using application-specific knowledge, a technique which is especially effective in applications such as database management systems, which are designed from the ground up to exploit this level of control.

The Solaris operating system has primitive support for buffered asynchronous I/O through the aio library [SOL99]. The functions aioread() and aiowrite() are used to initiate operations, and synchronization is achieved by catching the SIGAIO UNIX signal sent on completion or calling the aiowait() function to poll or wait for an operation to finish (note that unlike other synchronization implementations, the aiowait() interface is not request specific – that is, it waits only for the next asynchronous I/O spawned by the current process to complete without allowing one to specify which requests are of interest). This is an inconvenience that was remedied in the new Solaris Posix4 library, which offers request-specific polling and waiting, a more flexible signaling interface, application-directed request prioritization, and support for asynchronous buffer flushing requests. Unfortunately, this new interface is only available for Solaris 2.6 and higher. The prototype discussed in this paper was implemented using the basic aio library in the interests of maximizing portability, but future incarnations may move to the newer interface in order to simplify the control logic.

The Message Passing Interface (MPI) Standard, a somewhat influential standard in the parallel computing industry, has included buffered asynchronous I/O primitives in its new MPI 2.0 Specification through the MPI_FILE_IREAD_* and MPI_FILE_IWRITE_* initiation functions, using the MPI_TEST and MPI_WAIT functions for synchronization [MPI97]. Implementations of the specification are still free to use blocking semantics for these calls if the "hardware is unsuitable", but the fact they were included in the specification is a clear sign that OS-managed, asynchronous I/O has been recognized as an important point in the design space for high-performance computing. Hopefully this will prompt more OS vendors to incorporate such primitives into their system call or standard library interfaces.

3.0 Existing I/O Primitives in Java and Titanium

The I/O primitives previously supported by Titanium were limited to those present in Java 1.0. The Java I/O primitives come in two basic forms. The java.io.RandomAccessFile class provides unbuffered random file access using a seekable file-pointer similar to the read()/fread() functions in the C runtime library. The java.io.FileInputStream and java.io.FileOutputStream classes act as a bottom-level stream interface for sequential, unbuffered, stream-based file I/O – the java.io library provides a rich set of stream operators which can be composed to provide various filtering and buffering characteristics. For example, a common composition for input is:

```
DataInputStream dis = new DataInputStream(new BufferedInputStream(new FileInputStream(<filename>)))
```

where the BufferedInputStream provides buffering, and the DataInputStream performs translation from the abstract stream to useful Java primitive types. Both interfaces are rather elegant and make for writing clear, well-abstracted code – however, they're both very poorly suited to meet the performance demands of scientific applications that involve large amounts of file I/O.

The semantics for both interfaces are completely blocking – the BufferedInputStream provides implicit prefetching and write-behind caching in a private buffer (of a user-specified size), but all buffer reads and writes are synchronous and block the thread of control issuing the request that triggers a buffer flush/refill. There may be some opportunity for adding asynchronous buffer management at this level of the abstraction; however, because users are free to compose streams in any way they wish and even define streams of their own, this could produce

unpredictable and often undesirable results (especially when the bottom level is something other than a file, such as a communication pipe to a different process).

The second problem shared by both interfaces is they provide essentially no support for bulk-I/O transfers between the library and the application variables. All accesses must go through the read*() and write*() methods of the DataInput and DataOutput Java interfaces, which only allow reading/writing a single primitive data value at a time (e.g. .readLong(), .writeDouble(), etc.) The only bulk operations provided are for byte arrays, which are used to implement the stream interface. However, the type safety of Java prevents you from doing anything useful with these byte arrays other than passing them into the constructor for a stream or a String, where you can parse them out one value at a time. Needless to say, this implies significant software overheads associated with large amounts of I/O, as the number of method calls is linear in the number of primitive input/output values, rather than in the number of discrete data chunks the application wants to read/write[2]. As a result, a common programming construct for file I/O in Java involves tight little loops that transfer file data to or from a Java array where it is worked on – this is especially true for Titanium programs and in general, any scientific application. This is a significant oversight on the part of the Java designers, and one important finding of this research is that limiting I/O in this manner creates enormous I/O bottlenecks in applications that do large amounts of bulk I/O.

4.0 AsyncFile Interface

The solution proposed and implemented by this research project is the new, Ti.io.AsyncFile class. The goals of the interface, in order of importance, are performance, power, and simplicity. Performance issues are addressed by providing support for asynchronous file I/O with overlapped computation, and by supplying array bulk I/O primitives that remove the limitations imposed by the existing Java/Titanium I/O libraries. Power, that is to say expressiveness, was achieved by studying the existing interfaces for asynchronous I/O and formulating a design that takes the best features of each and places them in a Java framework. Specifically, the synchronization primitives were designed to be very flexible and expressive. Lastly, simplicity was achieved by adhering to the Java "flavor" wherever possible and minimizing the number of methods in the class and the number method calls required to perform an I/O operation, thereby making the interface easy to learn and use.

Figure 1 presents the current AsyncFile class that was designed and implemented in the prototype. The interface is somewhat similar to the RandomAccessFile class, in that the user is given the ability to seek to random positions in the file, although the default behavior is to service requests sequentially. The AsyncFile class is the abstraction for a non-blocking, buffered file with support for bulk I/O (there is no explicit buffering in the current implementation but it's provided within the OS). The methods for file-level control are self-explanatory and identical to those provided by RandomAccessFile.

The remainder of Figure 1 (the second page) documents the I/O initiation and synchronization methods.
The I/O initiation methods are read() and write(), which take as arguments a single-dimensional Java array of primitive types, a starting offset into that array, and an element count. When the method is called, it performs bound-checking and EOF-checking, then initiates an asynchronous I/O operation and returns an AsyncFileRequest object, which serves as the application's handle to the non-blocking operation that was initiated. The AsyncFileRequest class is also documented in Figure 1.

Once the asynchronous request has been initiated, control is returned to the application, which can then proceed to perform arbitrary computations while the non-blocking I/O completes in the background. The application can also initiate other non-blocking file operations to run concurrently. When the application wishes to synchronize with the ongoing operations, it calls one of the "Done" methods in the AsyncFile class or AsyncFileRequest class. The Done methods each take as a parameter a time-out interval in milliseconds that ranges anywhere from zero (poll) to infinity (a specially provided constant that means block indefinitely). The semantics of the Done methods are to check the status of the AsyncFileRequests the application is querying, raise any I/O exceptions that may have occurred, and return a boolean indicating whether the Done condition was satisfied. For example, the AsyncFile method AllDone(int timetowait) waits for all the pending operations on this AsyncFile object to complete or for the timeout to expire (whichever happens first), then returns true or false to indicate which happened. The static AllDone(int, AsyncFileRequest[]) method is similar, except it waits for all the requests in the provided list to complete or for the timeout to expire. The static AnyDone(int, AsyncFileRequest[]) method returns when at least one of the listed requests is complete (or the timeout expires). The AsyncFileRequest.Done(int) method is used for checking just a single request, and is semantically identical to calling AsyncFile.AnyDone and passing just this

[2] To make matters worse, the composition of streams often implies a chain of method call invocations for each primitive input/output call. Furthermore, the implementation of the I/O primitives for multi-byte values often involves multiple calls to the single-byte primitives (e.g. DataInputStream.readLong() is currently implemented as 8 calls to the DataInput.read() method that inputs a single byte).

request. The AsyncFileRequest.cancel() method is used to cancel a particular request, a handy capability for some applications (and incidentally, one that is not provided by most implementations of non-blocking file I/O - the Solaris library is one of the few that does provide it).

Figure 1: AsyncFile Interface (rev 1.4)

```
package Ti.io;

// new exceptions (all other exceptions are from java.io.* or java.lang.*)
public class AsyncIOCanceledException extends IOException {...};
public class FileNotOpenException extends IOException {...};
```

```
public class AsyncFile {
  // this class is the abstraction for a non-blocking file with support for bulk-I/O

  // -------  AsyncFile methods for file-level control ---------

  public AsyncFile(String filename, String mode) throws IOException, IllegalArgumentException;
  public AsyncFile(File filename, String mode) throws IOException, IllegalArgumentException;
    // open the named file    If mode is "rw", then the file may be both read and written.
    // If mode is "r", then the file may be read but may not be written
    // (any write method calls for this object will throw an IOException).
    // If mode is not "r" or "rw", then this constructor throws an IllegalArgumentException.
    // If the filename is not found and mode is “rw” the file is created
    // If the filename is not found and mode is “r” an IOException is thrown

  public final String filename;
    // the filename, for future reference

  public local long length() throws IOException;
    // returns the current file size (may be changed by subsequent writes)
    // if there are writes in-flight which are “growing” the file, the result is undefined

  public long getFilePointer() throws FileNotOpenException;
    // returns the position where the next read or write will begin
    // measured in bytes from the beginning of the file (which is position 0)
    // if close() has been called, will throw FileNotOpenException

  public local void seek(long offset) throws IOException;
    // set the file byte offset where the next read or write will begin
    // IOException is thrown if pos is less than zero or greater than the length of the file
    // if close() has been called, will throw FileNotOpenException

  public local void close() throws FileNotOpenException;
    // queues a close request
    // the file will not actually be closed until outstanding operations complete
    // in order to force an immediate close, call close() then cancel()
    // any operations initiated after a close() will throw FileNotOpenException

  public local void cancel() throws IOException;
    // cancel any outstanding operations for this file (see same function in AsyncFileRequest)
```

```
  // ------- AsyncFile methods for request dispatch & synchronization ---------

  public local AsyncFileRequest read(Object local primjavaarray, long arrayoffset, long count)
    throws IOException, NullPointerException, IndexOutOfBoundsException;
  public local AsyncFileRequest write(Object local primjavaarray, long arrayoffset, long count)
    throws IOException, NullPointerException, IndexOutOfBoundsException;
    // initiate a non-blocking file operation at the current file position
    // to read/write into/from the array elements arrayoffset..(arrayoffset+count-1)
    // primjavaarray must be a 1d-java array of primitive type (int[], long[], byte[], etc.)
    //  or a 1d-java array of immutables whose non-static member variables are all primitive
    // successful calls return an AsyncFileRequest which acts as a handle for this request
    // and immediately update the file position, advancing it to just past the end of this op
    // attempts to read past EOF will throw an EOFException
    // writes past the EOF will grow the file by the necessary amount
    // if close() has been called, will throw FileNotOpenException
    // may throw other exceptions if failure condition can be immediately determined
    // user promises not to read or write the primjavaarray buffer until operation is complete
    //  (until one of the Done functions returns true for this AsyncFileRequest)
    // user MUST use one of the Done functions at some time in the future to ensure all
    //  operations complete successfully (e.g. before the end of the program and before losing
    //  the last reference to AsyncFile and AsyncFileRequest) or requests may be canceled
    // see notes on consistency for details about concurrent accesses to a single file

  public local boolean AllDone(int timetowait) throws IOException;
  public static final int TIME_INF;
    // block until all outstanding operations initiated on this file complete, or
    // until <timetowait> milliseconds have expired, whichever comes first
    // returns true if there are no outstanding operations,
    //  or false if the timer expired
    // if timetowait is zero, the call will not block,
    //   (it will simply poll the status of outstanding requests and return)
    // if timetowait is TIME_INF, the block timer will never expire
    //   (it will block until there are no outstanding requests)
    // will throw an exception if any of the outstanding requests are in an exceptional state

  public static boolean AnyDone(int timetowait, AsyncFileRequest local [] local requestlist)
     throws IOException;
  public static boolean AllDone(int timetowait, AsyncFileRequest local [] local requestlist)
     throws IOException;
    // same as above, except limited to only those outstanding operations in <requestlist>
    // note these operations need not be from the same AsyncFile object
  }
```

```
public class AsyncFileRequest {
    // this class is the abstraction for a single non-blocking operation in progress

  public final AsyncFile local parent; // the AsyncFile which initiated this AsyncFileRequest

  public local boolean Done(int timetowait) throws IOException;
  public local boolean Done() throws IOException;
    // same semantics as the AsyncFile "Done" methods,
    // except that check only applies to this particular operation
    // if timetowait is omitted it defaults to AsyncFile.TIME_INF (block)

  public local void cancel() throws IOException;
    // cancel this operation if it's still outstanding
    // will have no effect if the operation has already completed
    // further calls to the "Done" functions on this request throw AsyncIOCanceledException

  public local void clearException();
    // clears the exceptional state for this AsyncFileRequest so that future calls to
    // the Done methods won't throw exceptions on account of this request
  }
```

Any of the Done() methods may throw an I/O exception if one of the requests being queried has ended in an exceptional condition. Figure 2 illustrates the state diagram for AsyncFileRequest objects. AsyncFileRequest objects begin their life in the "In Flight" state when they are returned from the read() and write() methods. The common behavior is for the I/O to complete successfully and for the AsyncFileRequest to move into the "Completed Successfully" state where it remains until it is destroyed. However, if an I/O error occurs, the AsyncFileRequest moves into the "Completed with I/O Error" state – future calls to the Done() method which include this AsyncFileRequest will cause an exception to be thrown. In most cases, scientific applications are programmed with fail-abort semantics – that is, I/O exceptions are generally so rare in debugged code that the programmer doesn't care what happens if one occurs. This default case is handled nicely by the Java exception mechanism – in the absence of catch code, the exception thrown propagates to the top of the program and halts execution with an error. However, support is also provided for applications that wish to catch I/O exceptions and possibly perform some form of recovery - for example, checkpointing their state before shutting down to minimize the cost of lost work. In this case, the programmer may want to call the AsyncFileRequest.clearException() method, which moves the exceptional request into a cleared state, so further calls to Done methods which include this request won't re-throw the exception. This is really only important when using the AllDone(int) method, which checks the state of all requests ever initiated on a particular file object – an application which uses only the static AllDone(int, AsyncFileRequest[]) and AnyDone(int, AsyncFileRequest[]) methods which specifically list the requests to be checked need never call clearException().

Finally, the AsyncFileRequest.cancel() method cancels a request if it's still in-progress - freeing I/O and OS resources, and moving it into the cancelled state (which is identical to the Exception state, except the Exception thrown will always be AsyncIOCanceledException). This method may be useful for applications that are reading from slow or highly contended mirrored disks (where a response from either is sufficient, so a request can be issued to both disks and the "loser" can be canceled) or applications which wish to issue speculative prefetches and cancel on misprediction (probably rare).

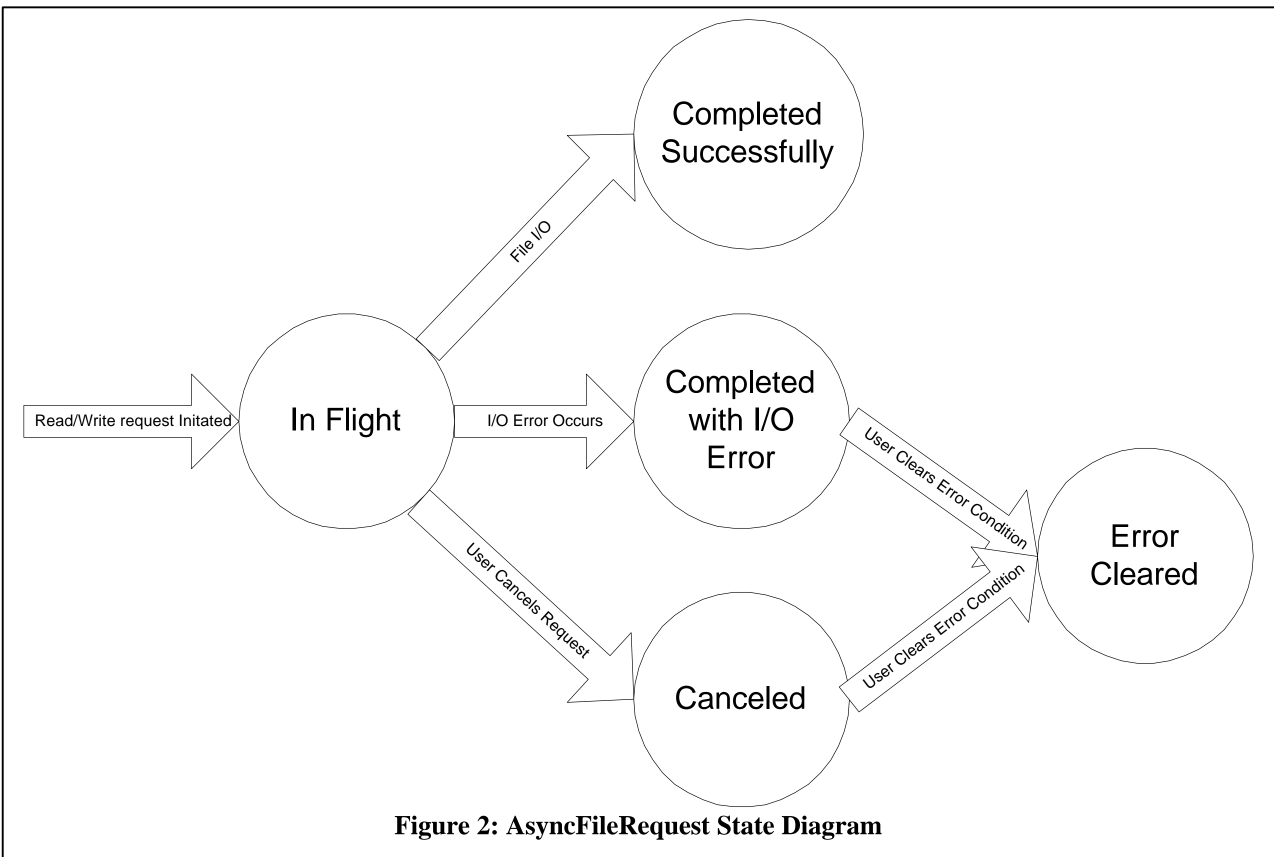


Figure 2: AsyncFileRequest State Diagram

5.0 Design Results

The AsyncFile interface seems to meet the design goals of performance, power and simplicity. Performance is provided by enabling overlapped computation and providing direct support for bulk I/O (empirical results will be presented in the next section). In terms of power, initial usability results indicate the interface is very expressive and a natural match for the types of synchronization which applications wish to perform. Specifically, the interface

makes it very easy to open a file, initiate a slew of requests, then work on each of them as they complete, allowing the OS disk scheduler to choose the optimal evaluation order. The random access feature allows applications to jump around to arbitrary parts of a file, and is especially useful for applications where different processors write their results to different areas of a file chosen apriori (for example, in parallel radix sort). The interface also very naturally accommodates double-buffering algorithms, which use a buffer leap-frogging approach, performing I/O on one buffer while computing on the other, then switching them. The exception semantics fit cleanly into the Java exception model and default to intelligent behavior. Finally, the implementation fully supports 64-bit file offsets for I/O on massive files.

To test usability and performance, I wrote a simple program that simulates the disk and CPU activity of the initial sorting pass in an external merge sort. During the first pass of external merge sort, the algorithm repeatedly reads a memory-sized chunk of a file, performs an in-memory sort, and writes the sorted chunk out to disk. Subsequent passes merge these sorted fragments into larger sorted fragments until only a single sorted fragment remains. The regular pattern of computation and I/O on large buffers makes this application a prime candidate for optimization using the new library.

The optimized version of the application uses double-buffering on the input and output, and is somewhat slick in that it manages to do this with only three buffers – this is a useful optimization because the fewer buffers required, the larger they can be while still fitting in physical memory, leading to larger sorted fragments and fewer merge passes. This optimization is enabled by the fact that multiple I/O operations can be outstanding simultaneously. The relevant piece of the source code is presented in Appendix A (I've omitted the performance instrumentation code and the body of the sort function, which is an O(n log n) operation) . Although this program has a rather complicated pattern of file I/O, I designed and wrote the entire I/O skeleton and got it working in less than 30 minutes. This seems to be a major victory in terms of simplicity.

As far as expressiveness is concerned, the 3-buffer, double-buffered code that uses AsyncFile occupies about 50 lines of non-comment code, whereas the analogous single-buffered blocking code written for one of the existing Java libraries occupies about 20 lines. One less obvious feature of the AsyncFile interface is it allows one to perform blocking I/O as a special case when no useful overlapped computation can be performed (for example, while reading the first buffer or writing the last buffer during double-buffering, or for reading file header information) and still get the performance benefits of bulk I/O. One example is the first call to the read() method in Appendix A:

```
inf.read(buf1, 0, buffersz).Done(); // blocking read(buf1) – read first chunk
```

A simple version of the sorting program written using only blocking calls to AsyncFile (single-buffered, no overlapped computation) occupies only about 10 lines of code because it eliminates value-reading loops, file stream composition, and an EOF exception handler from the code utilizing the Java libraries.

Finally, the AsyncFile library will perform regular Active Message polling while the application is blocked inside a Done method, allowing it to continue servicing network requests and prevent blocking other processors as occurs with the Java I/O libraries (the prototype doesn't yet do this, but it soon will).

6.0 Performance Benchmarking

The double-buffered sort presented in Appendix A which uses the AsyncFile library was reimplemented as a straightforward, single-buffered, blocking algorithm using various configurations of the existing Java/Titanium I/O libraries in order to evaluate the relative performance of the prototype library implementation. All versions were instrumented to measure various aspects of performance using the language-provided timing facilities (System.currentTimeMillis()), and run in various configurations. The I/O library configurations considered are listed in Table 1. The test suite was run on the various hardware configurations listed in Table 2. The tests were run using input files that ranged in size from 1 MB to 16 MB, with buffer sizes ranging from 32 KB to 512 KB. I collected a large amount of data, but in the interests of space will show only a few of the more interesting results.

Figure 3 shows the normalized performance (total run time for each block size / avg. total run time of all block sizes) of the algorithms changes relatively little with varying block sizes, which justifies the presentation of a single block size for all comparisons. The normalized performance of async gets slightly worse for larger block sizes, because the impact of the O(n log n) computation increases with larger block sizes – the other configurations are so dominated by poor I/O performance that the effect of the slightly increased computation cost barely affects the normalized total run time.

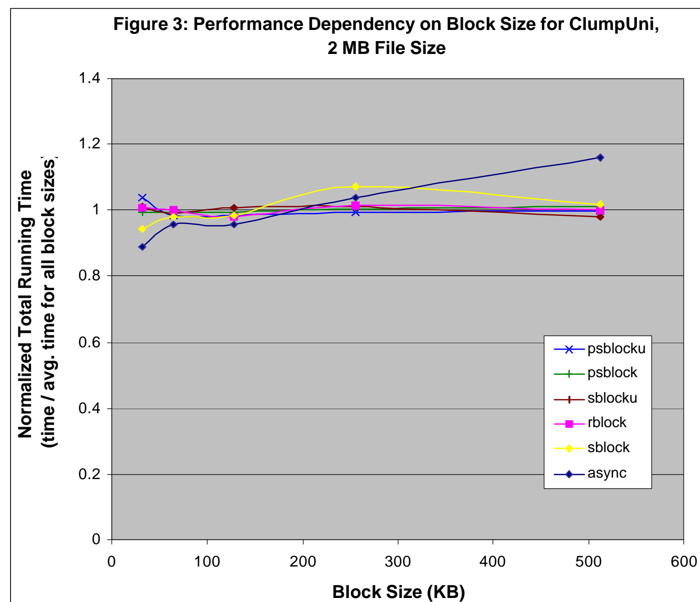
name	Description
async	The AsyncFile library designed in this project, using the asynchronous, double-buffering algorithm illustrated in Appendix A. Primitives were the bulk I/O primitives: AsyncFile.read() and AsyncFile.write()
ablock	The AsyncFile library designed in this project, using a blocking, single-buffered algorithm. Primitives were the bulk I/O primitives: AsyncFile.read() and AsyncFile.write()
rblock	The unbuffered RandomAccessFile interface, using a blocking, single-buffered algorithm Primitives were: RandomAccessFile. readLong() and RandomAccessFile.writeLong()
sblock	A buffered sequential access stream, using a blocking, single-buffered algorithm The streams used were: DataInputStream(BufferedInputStream(FileInputStream())) and DataOutputStream(BufferedOutputStream (FileOutputStream())) Primitives were: DataInputStream. readLong() and DataOutputStream.writeLong()
sblocku	An unbuffered sequential access stream, using a blocking, single-buffered algorithm The streams used were: DataInputStream(FileInputStream()) and DataOutputStream(FileOutputStream()) Primitives were: DataInputStream. readLong() and DataOutputStream.writeLong()
psblock	A buffered sequential access textual stream, using a blocking, single-buffered algorithm The streams used were: DataInputStream(BufferedInputStream(FileInputStream())) and PrintStream(BufferedOutputStream (FileOutputStream())) Primitives were: Long.parseLong(DataInputStream.readLine()) and DataOutputStream.println()
psblocku	An unbuffered sequential access textual stream, using a blocking, single-buffered algorithm The streams used were: DataInputStream(FileInputStream()) and PrintStream(FileOutputStream()) Primitives were: Long.parseLong(DataInputStream.readLine()) and DataOutputStream.println()

All tests were compiled using titaniumc-0.643 and run on Solaris 2.6.
The BufferedInputStream object uses the default 2KB buffer, and PrintStream object uses the default lazy flushing.

Table 1: Library Configurations Considered

Name	Description
ClumpUni	8-way SMP with 2GB memory – uniproc backend, local file system
ClumpPthread	8-way SMP with 2GB memory – pthread backend, local file system
NOWUni	Single processor SUNW,Ultra-1 (a NOW node) – uniproc backend, local file system
NOWPthread	Single processor SUNW,Ultra-1 (a NOW node) – pthread backend, local file system
NOWSplitCNFS	NOW Parallel job with a single node - Split-C backend, NFS file system

Table 2: Hardware Configurations



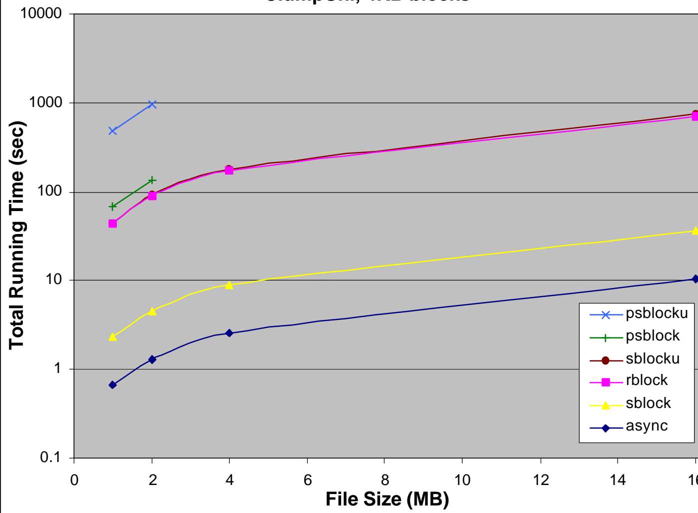


Figure 4: Performance Comparison for Varying File Sizes, ClumpUni, 4KB blocks

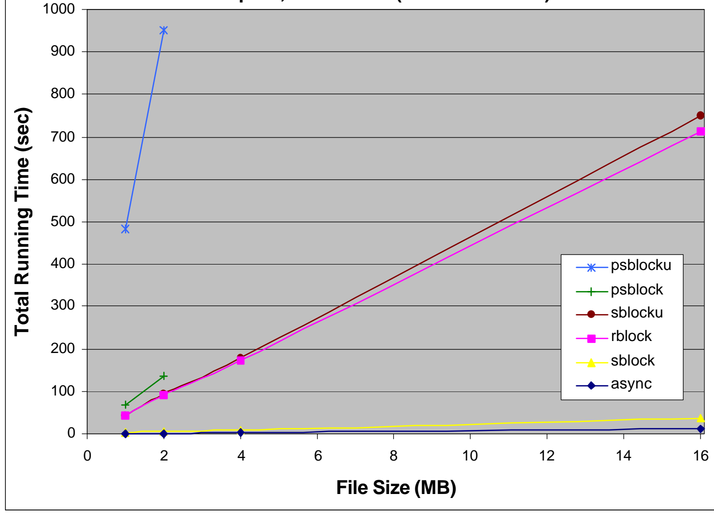


Figure 5: Performance Comparison for Varying File Sizes, ClumpUni, 4KB blocks (linear time scale)

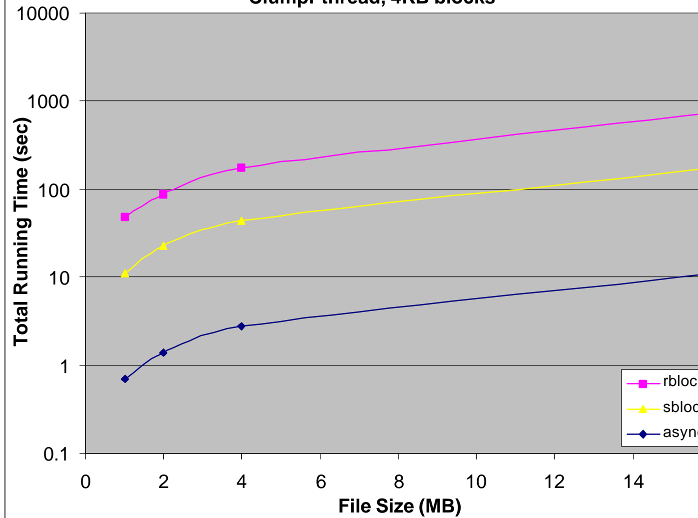


Figure 6: Performance Comparison for Varying File Sizes, ClumpPthread, 4KB blocks

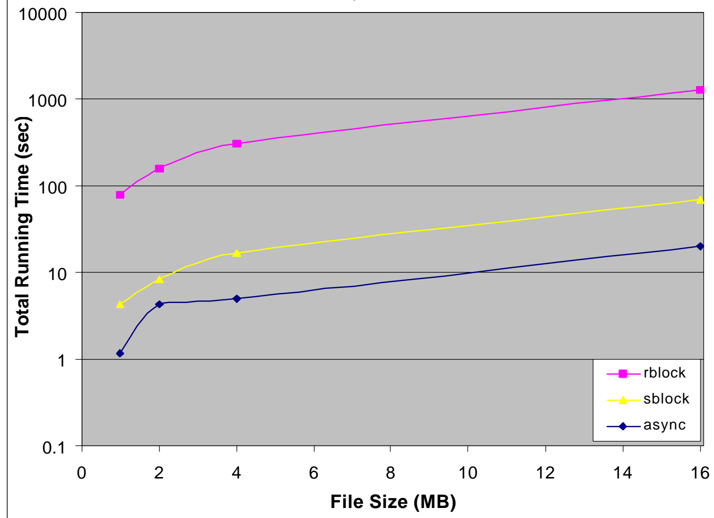


Figure 7: Performance Comparison for Varying File Sizes, NOWUni, 4KB blocks

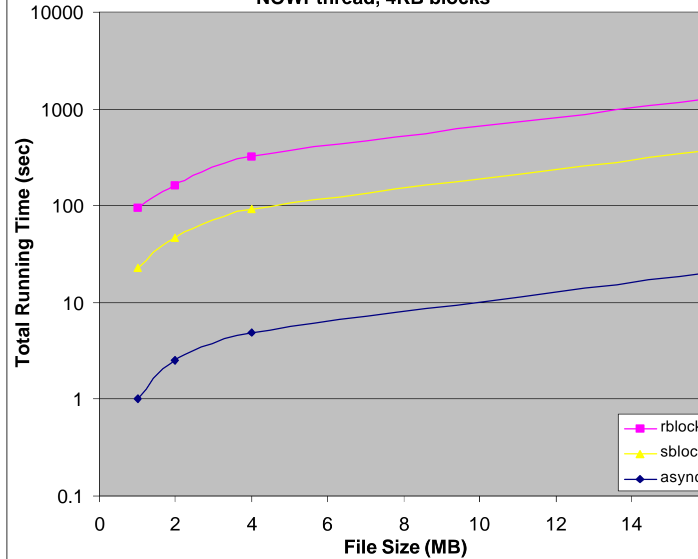


Figure 8: Performance Comparison for Varying File Sizes, NOWPthread, 4KB blocks

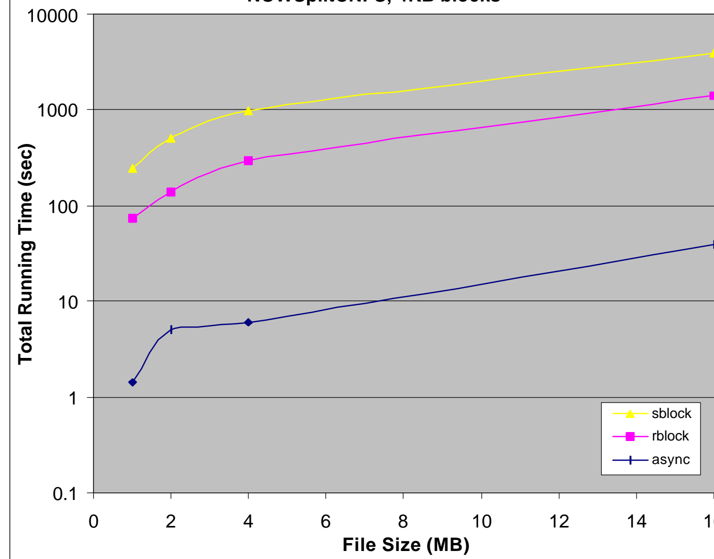


Figure 9: Performance Comparison for Varying File Sizes, NOWSplitCNFS, 4KB blocks

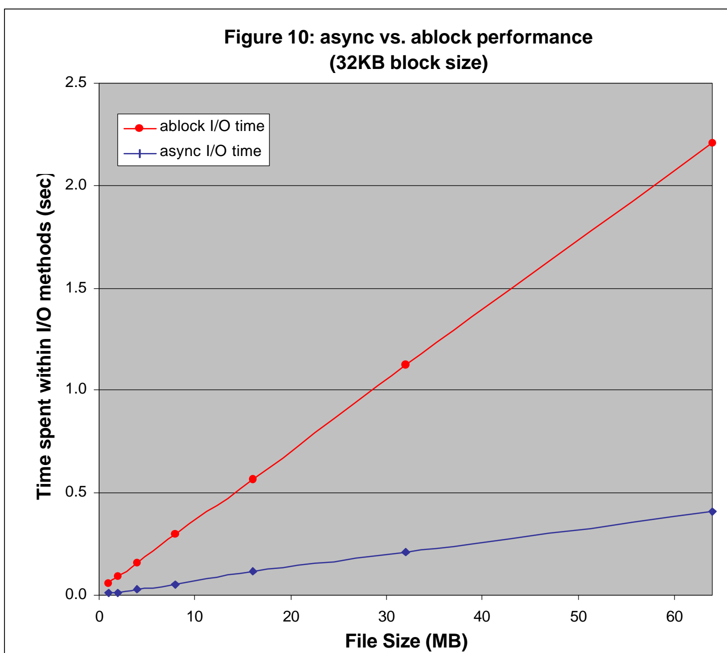
Figures 4-9 show total running time in seconds for the entire first pass of the sorting algorithm, ranging over the file sizes on the various configurations. Note the time scale in figures 4 and 6-9 is logarithmic to accentuate the performance characteristics of the faster configurations while accommodating the slower ones – figure 5 shows the same data as figure 4 on a linear scale for comparison.

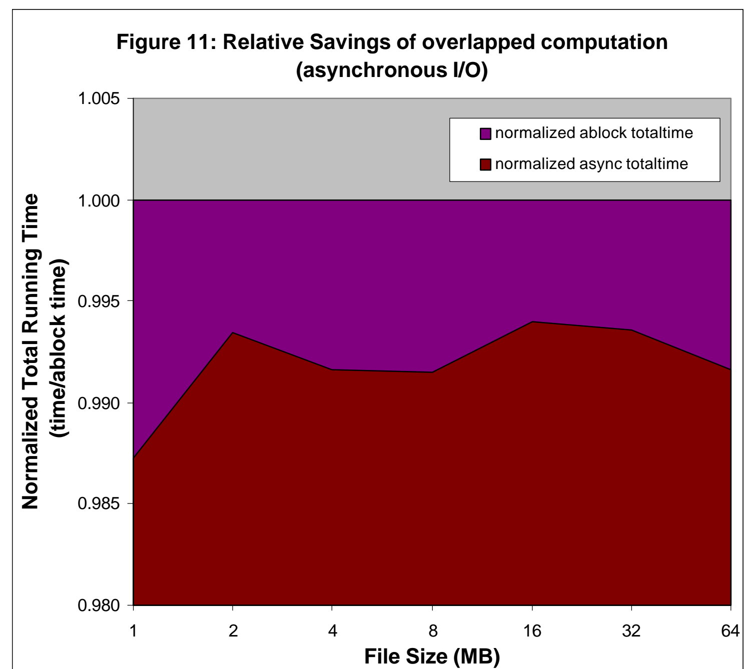
One immediate observation is that the code using the AsyncFile library is always much faster, for all platforms and file sizes, beating the other configurations by amounts ranging from 10x to 100x. The primary explanation seems to be the support for bulk I/O – this effect is so overwhelming that the performance gain of overlapped I/O with computation is almost negligible in comparison. In order to really observe the performance gains of overlapping I/O and computation for this application, it's necessary to go to much higher file sizes (it was not feasible to wait for the other configurations to complete on files larger than 16MB).

As far as the blocking algorithms stack up, Figure 4 shows that the performance of rblock and sblocku are essentially identically bad. This is not surprising because the operations performed by their implementations are actually very similar. Furthermore, this graph shows that using the textual output primitives (psblock and psblocku) give VERY poor performance, and should be avoided at all costs (these are only shown on figure 4 – they took so long to run that it was necessary to limit those tests to under 2MB on the fastest hardware configuration available). sblock seems to be the best configuration that the existing Java/Titanium libraries provide, but async seems to always offer better performance for this application (generally about 5x to 10x faster).

Another observation is that NFS (used for the tests shown in figure 9) leads to very poor performance in the existing libraries, as compared to local disk file I/O – which in itself is not all that surprising, but the fact that it does so much worse is somewhat stunning. The running times for the AsyncFile configuration were affected, but only mildly – the bulk I/O results in far fewer calls to the read() and write() system functions, presumably leading to less NFS communication. It's not clear why the sblock configuration did worse than rblock on NFS – this bears more investigation.

As previously mentioned, the effects of the overlapped computation in the async configuration was overshadowed by the performance gain of bulk I/O in this application for the smaller file sizes discussed above – results are not shown for the ablock configuration (single-buffered, blocking sort using the AsyncFile bulk I/O primitives) but they were virtually identical to the numbers for async. To further investigate the difference, the async and ablock configurations were run with larger input files on the ClumpUni configuration – the performance difference was not dramatic, but it was noticeable for larger files. Figure 10 illustrates the time spent by each configuration within just the I/O methods (doesn't include sorting time, etc.). The I/O time for async is non-zero because the Solaris non-blocking primitives actually block for a short time while initiating a new request. This overhead is believed to be approximately constant – it appears linear in the graph because the block size is fixed (thus the total number of requests issued increases with file size). Figure 11 summarizes the relative savings of overlapping computation with I/O in this application – the purple area represents the percentage of the total time saved using the double-buffering technique enabled by the non-blocking capabilities of AsyncFile (note the y-axis begins at 98%).





7.0 Limitations and Future Work

There are several limitations in the current prototype implementation that have yet to be addressed. Perhaps the most important remaining task is porting the library to Titanium platforms other than Solaris – the library should be easily portable to any OS providing the minimal non-blocking I/O primitives described in section 2, but no real attempt has been made to do so thus far. Other future work includes implementing a subclass of AsyncFile that performs data validation on the data input and output routines – for example, to raise exceptions when attempting to perform I/O on floating point NaN's or illegal Unicode character bit patterns.

Another limitation of the current prototype implementation is that it only supports file I/O on single-dimensional Java arrays of primitive types and Java arrays of immutable classes containing only primitive types. The reason is these are the only arrays in Titanium whose elements are guaranteed to be stored in contiguous memory locations. Multidimensional Java arrays are stored as a hierarchy of references to single-dimensional arrays (which could possibly differ in size). In any case, a programmer could certainly perform I/O on the constituent single-dimensional fragments of a multi-dimensional array with the caveat that the application may have to store some additional application-dependent meta-information in order to recover the shape of a multi-dimensional array read in this fashion. It's not clear what it means to perform I/O on non-primitive (i.e. reference) types, although the object serialization approach pioneered in Java 1.1 is probably a good start (Titanium does not include this because it's not a Java 1.0 feature). Similarly, there's no support for I/O on Titanium arrays, which may also be multidimensional and even strided, and are similarly not guaranteed to be stored contiguously in memory (even for the single dimensional case, because Titanium supports generating single-dimensional array "slices" from a multi-dimensional array, thereby creating an array stored non-contiguously). This is a more serious issue, and definitely merits further study – for the time being, Titanium developers wishing to perform I/O on Titanium arrays are advised to allocate and use a Java array as an intermediate copy and use the bulk I/O operations in the AsyncFile class – this would probably still be significantly faster than passing the values to the Java blocking I/O classes one value at a time. This functionality could easily be implemented as a subclass of AsyncFile and probably will be in the near future.

There are several open issues dealing with the lack of strong consistency and locking semantics for files opened using an AsyncFile. Specifically:

- The AsyncFile library provides no file locking capabilities
- The AsyncFile library provides no guarantees about the durability of writes (i.e. they may be buffered for awhile by the OS and no flush() method is currently provided)
- Whenever a write is in-flight, new read requests that partially or completely overlap that same area of the file will return undefined results in the area of the array corresponding to the overlap. New overlapping write requests will leave the file area within the overlap in an undefined state
- The library provides no support for concurrent accesses to a single file from different machines, especially for files on an NFS file system – attempting this may give rise to very highly system-dependent effects. Specifically, reads and writes are likely to be buffered in the OS, so writes made from another machine may not be visible immediately (even after the write request has completed) and may observed non-atomically on the reading machine (might see half a write, even after the write request has completed).

The first two seem manageable and will probably be handled by a future version of AsyncFile. The third really seems like more of an application-level problem, but it may be desirable to add some additional checking to detect overlap conditions and raise exceptions (again, certainly feasible). Finally, the last issue is a very sticky one that results from the lack of consistency guarantees in many network file systems, and is not likely to be solved in the near future. Incidentally, the Java libraries don't address any of these issues.

A final issue involves access from multiple processors in a parallel program. Currently, the only usage model supported is one where each processor wishing to perform I/O creates its own AsyncFile object and initiates I/O on local data buffers. The reasons are three-fold. First, the GLUnix operating system which coordinates parallel Titanium jobs on distributed memory systems doesn't currently support the sharing of file descriptors across processors, so if one processor in a distributed system could access the methods in an AsyncFile object created on a remote processor, requests it initiated would always fail – this limitation is shared by the existing I/O libraries and seems unlikely to be fixed anytime soon. The second problem is that the asynchronous I/O primitives in the base Solaris operating system know nothing about the remote memory pointers used to implement the global memory space in distributed systems, so it's not possible to initiate a non-blocking file I/O request on a buffer residing on a remote processor. That is, one processor couldn't initiate an asynchronous I/O operation using a buffer residing in another processor's memory space. It's conceivable to solve this problem by allocating a local buffer and performing an extra copy from global to local memory before a write and from local to global memory after a read, thereby achieving the same effect at the cost of a network bulk data transfer. An esoteric, but possibly more efficient solution to this problem would be to write an Active Message request handler that initiates non-blocking I/O directly

on the remote processor. The final limitation is the current prototype implementation is not reentrant, so some synchronization would need to be added in order to ensure correctness of concurrent accesses to a single AsyncFile object from multiple processors if we ever decide to support this. The Titanium "local" annotations on the AsyncFile methods currently ensure that no code that could possibly violate these access conditions will compile. However, the compiler local variable analysis is conservative, so these annotations make using the interface slightly more tedious - in the future they may be replaced by an (efficient) runtime check which ensures the necessary locality properties for the current platform and throws an exception if they are not met.

Some other areas for future work include providing some form of implicit data distribution– this is an important facet of parallel I/O optimization that is currently left up to the explicit control of the application. Second, there is no support for implicitly asynchronous file operations – one could imagine a compiler analysis which converts blocking file operations to non-blocking ones (perhaps moving them earlier as well), and automatically inserts Done() operations before the first use of a buffer after a non-blocking read. However, since there are currently no blocking bulk I/O primitives in the language, it's questionable how effective such an optimization could be. Furthermore, it seems that programs designed without thought given to such optimizations would probably not exhibit a large degree of CPU/disk parallelism.

8.0 Conclusions

The new AsyncFile class effectively adds support to Titanium for asynchronous file I/O operations, enabling the application to mask I/O latency with overlapped computation, using algorithms such as double-buffering. The library also provides support for bulk (array) I/O operations, removing much of the overhead associated with the Java I/O libraries which require the programmer to read arrays a single primitive element at a time. Experimental results show the performance of the new library is a huge win over the existing solutions in Titanium and Java. There is considerable room for future work, but it seems clear that this runtime library satisfies its goals of performance, power and simplicity, and is a valuable addition to the Titanium programmer's toolkit.

9.0 Acknowledgements

I would like to thank the entire Titanium team, especially Kathy Yelick, Ben Liblit, and Andy Begel for all their invaluable help.

10.0 References

[ACH95] Acharya, Uysal, et al. “Tuning the Performance of I/O Intensive Parallel Applications”, Rice Center for Research on Parallel Computation, 1995.

[AIK98] Aiken, Alexander and Gay, David. “Memory Management with Explicit Regions”, Programming Language Design and Implementation, Montreal, Canada, June 1998.

[AND95] Anderson, Culler, and Patterson. “A Case for Networks of Workstations: NOW”, IEEE Micro, Feb, 1995.

[BEG98] Begel, Andrew. “Titanium Threads”, CS267 Final Project, 1998.

[CHO93] Choudhary, Rosario and Bordawekar. “Improved Parallel I/O via a Two-phase Run-time Access Strategy”, IPPS Parallel I/O Workshop, 1993.

[CUL97] Culler, Arpaci-Dusseau, Arpaci-Dusseau, Chun, Lumetta, Mainwaring, Martin, Yoshikawa, and Wong. “Parallel Computing on the Berkeley NOW”, 9th Joint Symposium on Parallel Processing, 1997.

[EIC92] Eicken, Culler, Goldstein, and Schauser. “Active Messages: a Mechanism for Integrated Communication and Computation”, 19th International Symposium on Computer Architecture, 1992.

[GAY98] Gay, David and Aiken, Alex. “Barrier Inference”, Principles of Programming Languages, San Diego, California, January 1998.

[GJS96] Gosling, Joy and Steele. “The Java Language Specification”, June, 1996.

[HIL99] Hilfinger, Paul. “Titanium Language Working Sketch, rev 0.22”, September, 1999.

[JAC91] Jacobson, David M. and Wilkes, John "Disk scheduling algorithms based on rotational position", HP Laboratories Technical Report, 1991.

[KEN94] Kennedy, Koelbel, and Paleczny. “Scalable I/O for Out-of-Core Structures”, Rice Center for Research on Parallel Computation, 1994.

[MAI95] Mainwaring, Alan and Culler, David. “Active Messages Applications Programming Interface and Communication Subsystem Organization”. UC Berkeley Technical Report, 1995.

[MPI97] Message Passing Interface (MPI) Standard 2.0 Specification, Chapter 9, 1997. http://www-unix.mcs.anl.gov/mpi/

[MS99] Microsoft Windows 32-bit API, Microsoft Developer Network, 1999.

[NOW99] NOW Project web page. http://now.cs.berkeley.edu/

[ROS93] Rosario, Juan and Choudhary. “High Performance I/O for Parallel Computers: Problems and Prospects”, IEEE Computer, July 1993.

[RUE91] Ruemmler and Wilkes. “Disk Shuffling”. Hewlett-Packard Software and Systems Laboratories Technical Report, 1991.

[RUE94] Ruemmler and Wilkes. “An introduction to disk drive modeling”. Hewlett-Packard Software and Systems Laboratories Technical Report, 1994.

[SHI96] Shriver, Merchant and Wilkes. “An analytic behavior model for disk drives with readahead caches and request reordering”, 1996.

[SOL99] Solaris 2.6 Reference Manual Answerbook. http://docs.sun.com/

[TIT99] Titanium Project web page http://www.cs.berkeley.edu/Research/Projects/titanium

[WOR94] Worthington, Bruce, Ganger, Gregory and Patt, Yale "Scheduling for Modern Disk Drives and Non-Random Workloads", University of Michigan, 1994.

[YEL98] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. “Titanium: A High-Performance Java Dialect”, ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford, California, February 1998.

Appendix A: Code Fragment from dbsort.ti
(first pass of external merge sort with double buffered input & output)

```
// open files
AsyncFile local inf = new AsyncFile(infilename,"r");
AsyncFile local outf = new AsyncFile(outfilename,"rw");

// get file length and calculate the number of chunks (assumed to be >= 3)
long filesz = inf.length();
numchunks = (new java.lang.Long(filesz)).intValue()/(buffersz*8);

long [] local buf1 = new long[buffersz]; // Allocate buffers - only 3 required!
long [] local buf2 = new long[buffersz];
long [] local buf3 = new long[buffersz];

inf.read(buf1, 0, buffersz).Done(); // blocking read(buf1) - read first chunk

int chunksleft = numchunks-1;

AsyncFileRequest local buf1Req = null;  // buffer pointers
AsyncFileRequest local buf2Req = null;  // (null ptrs are ignored by the Done methods)
AsyncFileRequest local buf3Req = null;

while (chunksleft > 1) { // the "steady-state" double-buffering loop

  buf2Req = inf.read(buf2, 0, buffersz); // non-blocking read(buf2)
  sort(buf1);
  buf1Req = outf.write(buf1, 0, buffersz); // non-blocking write(buf1)

  // wait( buf2, buf3 )
  AsyncFileRequest local [] local tmplist1 = { buf2Req, buf3Req };
  AsyncFile.AllDone(AsyncFile.TIME_INF, tmplist1);

  buf3Req = inf.read(buf3, 0, buffersz); // non-blocking read(buf3)
  sort(buf2);
  buf2Req = outf.write(buf2, 0, buffersz); // non-blocking write(buf2)

  // wait( buf1, buf3 )
  AsyncFileRequest local [] local tmplist2 = { buf1Req, buf3Req };
  AsyncFile.AllDone(AsyncFile.TIME_INF, tmplist2);

  // do some swapping
  long [] local oldbuf1 = buf1;
  long [] local oldbuf2 = buf2;
  long [] local oldbuf3 = buf3;
  AsyncFileRequest local oldbuf1Req = buf1Req;
  AsyncFileRequest local oldbuf2Req = buf2Req;
  AsyncFileRequest local oldbuf3Req = buf3Req;

  buf1 = oldbuf3;
  buf1Req = oldbuf3Req;
  buf2 = oldbuf1;
  buf2Req = oldbuf1Req;
  buf3 = oldbuf2;
  buf3Req = oldbuf2Req;

  chunksleft -= 2;
  }

if (chunksleft == 1) {
  buf2Req = inf.read(buf2, 0, buffersz); // non-blocking read(buf2)
  sort(buf1);
  buf1Req = outf.write(buf1, 0, buffersz); // non-blocking write(buf1)

  buf2Req.Done(); // wait(buf2)
  sort(buf2);
  buf2Req = outf.write(buf2, 0, buffersz); // non-blocking write(buf2)

  // wait( buf1, buf2 )
  AsyncFileRequest local [] local tmplist2 = { buf1Req, buf2Req };
  AsyncFile.AllDone(AsyncFile.TIME_INF, tmplist2);
  }

else { // chunksleft == 0
  sort(buf1);
  outf.write(buf1, 0, buffersz).Done(); // blocking write(buf1)
  }
```