High Speed Raster Image Streaming For Digital Presses Using the Hadoop File System

Russell Perry

HP Laboratories
HPL-2009-345

Keyword(s):
Print, VDP, Hadoop, Binary Integer Programming

Abstract:

An application of the distributed Hadoop file system to very high rate variable data printing is described. The raster image processing of a large variable data document is represented as a MapReduce process. The key challenge addressed by this paper is how to stream the resulting raster images off the Hadoop file system to a digital press at multi-gigabit data rates. To achieve this, efficient scheduling of the order in which file blocks are read by the client is beneficial. An approach to scheduling based on binary integer programming is described which generates more efficient schedules compared to a naive approach. The scheduling model allows the exploration of system design choices and helps to identify file block distributions that are problematic to read at high rates. Measured stream rates approaching 4Gb/s were achieved which is close to the required rate for streaming pages containing rich designs to a digital press. This required only a minor extension to the Hadoop client to allow file blocks to be read in parallel from the Hadoop data nodes.

External Posting Date: October 21, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: October 21, 2009 [Fulltext]

High Speed Raster Image Streaming For Digital Presses Using the Hadoop File System[1]

Russell Perry, Member IEEE

***Abstract*—An application of the distributed Hadoop file system to very high rate variable data printing is described. The raster image processing of a large variable data document is represented as a MapReduce process. The key challenge addressed by this paper is how to stream the resulting raster images off the Hadoop file system to a digital press at multi-gigabit data rates. To achieve this, efficient scheduling of the order in which file blocks are read by the client is beneficial. An approach to scheduling based on binary integer programming is described which generates more efficient schedules compared to a naïve approach. The scheduling model allows the exploration of system design choices and helps to identify file block distributions that are problematic to read at high rates. Measured stream rates approaching 4Gb/s were achieved which is close to the required rate for streaming pages containing rich designs to a digital press. This required only a minor extension to the Hadoop client to allow file blocks to be read in parallel from the Hadoop data nodes.**

***Index Terms*—Print, VDP, Hadoop, Binary Integer Programming.**

I. Introduction

Variable data and document customization are increasingly being used in printed marketing and other publications [1][2]. As a result, every page to be printed is potentially unique. A print press must therefore be fed the rasterized image of each page rather than just a single repeated page design. Increasingly powerful digital print presses place growing demands on the IT and software infrastructure that supports the raster image processing and subsequent streaming of the ripped pages to the press. For example, digital web presses can support print throughputs of 122m/min [3]. At such rates, the supporting IT infrastructure costs become significant. Sharing common IT infrastructure between multiple, lower throughput, presses is also desirable to lower costs.

To contain the cost of the press' supporting software and IT infrastructure, an architecture approach is considered here offering a potential low cost of implementation. The approach is based on the Hadoop file system [4] in conjunction with MapReduce [5]. MapReduce is used to implement parallel raster image processing after which the rasterized images, stored on the Hadoop file system, are streamed to the print press. The Hadoop file system splits files into *blocks* and stores the blocks across a cluster of *data nodes*. The file blocks are stored on the file system of a data node's operating system and their location is recorded by a *name node* server. Each file can be replicated a configurable number of times. Examination of this approach is motivated by the fact that the Hadoop file system was developed with deployment on low cost commodity servers in mind. Hadoop was also developed to easily scale-out. As such using Hadoop provides an alternative approach to using a high performance SAN with shared file system to store the rasterized images. Whilst SAN systems are designed to be robust to individual component failures, should the SAN suffer a system-level failure then the printing operation will be interrupted. Certain failures in the Hadoop system can be tolerated with only an incremental loss in performance. The Hadoop File System supports redundancy (through block replication and operation retries) to allow for recovery from individual data node failures. However, whilst retries are acceptable in batch processing applications without tight time constraints, they are not acceptable in the print press domain where continuous real time streaming of raster images is required.

The specific challenge addressed in this paper is how to reliably stream files containing raster images from off a Hadoop file system (HDFS) to a print press at very high data rates. The raster image processing is not as time critical and therefore, is not the main focus of this paper. The task of streaming data to the press is usually performed by a *press client*. The press client's main function is to read data from a file system, buffer pages in memory and write the pages, in order, to the print press. Typical data rates are in the multiple Gbps range and once a stream starts it cannot be interrupted without disrupting the entire print run. For example, the Web Press can require input data rates in excess of 5Gb/s even with compression, assuming complex page layouts. When streaming at high rates, scheduling the transfers of file blocks in parallel from the HDFS to avoid resource contention is beneficial. E.g. reading different blocks from the same disk at the same time will slow data transfers and is ideally best avoided. This problem is a focus of the paper and is described in greater detail in section IV.

This paper's first contribution lies in the development of a model and algorithm for efficiently reading, in parallel, the sequence of file blocks making up a file to maximize transfer rates. The model is represented as a timetabling problem that is amenable to solution using binary integer programming (BIP). Solving the scheduling problem for a range of file block distributions allows various system design trade-offs to be explored and cost effective configurations identified. The BIP algorithm is compared with a simple naïve scheduling algorithm indicating the value of using the former, more

[1] Prepared for International MultiConference of Engineers and Computer Scientists 2010.

sophisticated, approach.

The paper's second contribution is to provide measurements of the file transfer rates when reading multiple file blocks in parallel from a HDFS deployment. For comparison, data rate measurements are provided using both Netperf [14] and Java's non-blocking IO when transferring data using a single stream.

The structure of the rest of the paper is as follows. Section II discusses prior work. Section III then describes the print streaming problem in detail and the application of Hadoop along with a brief overview of the experimental test bed. In section IV, the model of the scheduling problem is specified and two algorithms are developed to solve the scheduling problem. Section V presents results for algorithm performance across a range of file block distributions from both analysis and experiment.

II. Prior Work

In [6], the problem of scheduling a university timetable was modeled as a binary integer programming problem. The problem addressed here is also amenable to such an approach and is described fully in section IV.

BitTorrent provides a file transfer protocol underpinned by a peer-to-peer network [7]. If multiple peers have a copy of a file, then a BitTorrent client can discover which peers are currently hosting the file and begin downloading different parts of the file from each peer in a non-sequential manner. This avoids flash loads occurring on a single server hosting the file. At a conceptual level this approach is similar to using HDFS with multiple file replicas such that a client has multiple locations from which it can read a file block.

Other higher speed file transfer mechanisms have been proposed including GridFTP [10]. Performance improvements over *ftp* and *scp* protocols have been reported by splitting a TCP connection into multiple segments. GridFTP also makes use of parallel TCP connections at the application layer, which increases the TCP recovery rate from errors whilst decreasing the rate at which it slows throughput in response to errors. It also increases the total effective TCP-buffer space available. Alternatively using UDP for file transfers with reliability and error recovery provided by the application layer appropriate to the files to be transferred is also possible.

Striping, either by file or by disk has a long history. The original term RAID (redundant array of inexpensive disks) was introduced by Patterson et. al. in 1988 [8]. The goal is to provide greater reliability and/or bandwidth by combining multiple physical disks to provide a single logical, high performance disk. Hadoop is most similar to the RAID-1 configuration albeit the data striping is at the file system rather than disk level. In a RAID-1 configuration, data is mirrored across all available disks, whereas in Hadoop the number of replicas can be configured on a file by file basis. Later, striped network file systems were introduced which generally provide better performance with larger files. An early example was Zebra [9] which striped files according to the client rather than by file. Fault tolerance was also provided by making redundant copies of files, at the expense of increasing complexity needed to maintain synchronization between file copies. In this application, the raster image files are large, but not subject to change so file synchronization is not an issue.

Although in this paper the Hadoop File System is used, the approach is likely to be suitable for other distributed virtual file systems that support file replication.

III. Print Streaming Problem

The MapReduce programming model introduced in [5] and also implemented by Hadoop [4], is tailored to implementing large-scale data intensive processing which can be performed in parallel with the processing results then combined. It provides an abstraction that shields the developer from the intricacies of scheduling program execution across many machines and handling data distribution and management. The "map" and "reduce" are operations where the map operation takes as input a key (e.g. a document reference) and value (the document contents). The key/value pair is essentially a record from an input file. The output of the map is one or more intermediate key values. The reduce operation combines the intermediate values sharing the same key and emits zero or one outputs. The input to the MapReduce system is a configuration object. It specifies the set of files to be processed and other items such as the number of map and reduce operation instances to use and how the records in the input file should be interpreted. As well as the map and reduce operations, many of the internal MapReduce processes can be customized by the user.

A model for parallel raster image processing using MapReduce is now described. Initially, the large variable data PDF source document to be printed (it could contain $>10^5$ pages), is logically divided into a sequence of sub-documents. The sub-document size is chosen such that the time to rasterize it is sufficiently long so as to limit the overhead of the raster image processor initialization time. The input to the MapReduce engine is a text file containing key/value pairs specifying the list of all the sub-documents by their start and end byte positions in the source document. The MapReduce engine splits the list and distributes each part to an available map instance. During the map stage, each sub-document is read from the source document made available from a single URL address. The address is provided as a property to the MapReduce job. Within the map stage, the document is ripped and the resultant raster image is written to a file on the HDFS. Each map operation then returns an intermediate key/value pair using the same key name, but with the value set to the raster image output filename. The filename is chosen to indicate the original sub-document. Using the same key name ensures all map outputs are sent to a single reduce operation.

For the reduce phase, a single reduce instance is used which orders the raster files from each map stage according

to their original position in the source document and emits a single text file hereafter called the *stream list*. The stream list contains the set of files that need to be streamed to the print press. Note that the stream list collectively represents the complete rasterized source document, so henceforth the problem may be considered to be that of streaming a single rasterized source file even though it is decomposed into separate sections.

Using an active open source project is in itself attractive, whilst, in addition, many features of Hadoop (e.g. block replication) are also useful in addressing the problem. However, certain aspects of the problem are not ideally matched with the MapReduce paradigm. For example, the volume of output data produced by a MapReduce process is typically much smaller than the original input data. In the case here, the intermediate raster image files are very large and so the map stage actually yields a greater volume of output data. Another significant difference is that the map nodes read their input data from a common source file, albeit from different positions within that file (the input key/value pairs indicate which part of the source document to read). The source document is thus hosted on a web server accessible to all mapper nodes. Finally, the set of key/value pairs emitted by the map stage are interpreted as file *references* to the raster images which strictly speaking contain the actual intermediate values. This allows large intermediate values to be handled analogous to the way that the iterator type for the reduce operation input parameter, allows long lists of intermediate values to be handled. The reduce stage is used to collate the file references to produce the stream list. In effect, the reduce stage is actually carried out by the press client since it generates the final print stream based on the stream list.

A. System Setup

The test infrastructure on which Hadoop and the press client are deployed is shown in Figure 1. For illustration purposes, the locations of file blocks (replication factor 2) are indicated by the small green numbered rectangles shown above the set of 8 data nodes. Each green rectangle symbolizes a file block and the number represents the block sequence identifier. In general there are *N* blocks stored across *M* data nodes. In our system there are 31 data nodes and one name node (not shown). The Hadoop nodes are HP blade servers (bl2x220c) with 300GB of local disk storage and 16GB of memory. They have 4CPUs. The name node server is identical to the data nodes. The press client is an HP blade server (bl460c) with 64GB of memory and 8CPUs. Each server runs Red Hat Enterprise Linux 5 (v2.6.18).

At the press client a process runs containing multiple readers (*R*) for reading file blocks off the HDFS according to a pre-computed schedule. This is illustrated in the expanded blue bubble. The blocks, once read, are buffered by the press client before they are streamed to the press. The press client has two HP NC512m Dual Port 10GbE network interface cards; one is connected to the print press and the other connects to a switch that connects each data node over a 1Gb/s link. The Hadoop client code was extended to allow the data nodes on which each file block is stored to be determined and to allow file blocks to be read independently, not just sequentially as part of reading the complete file.

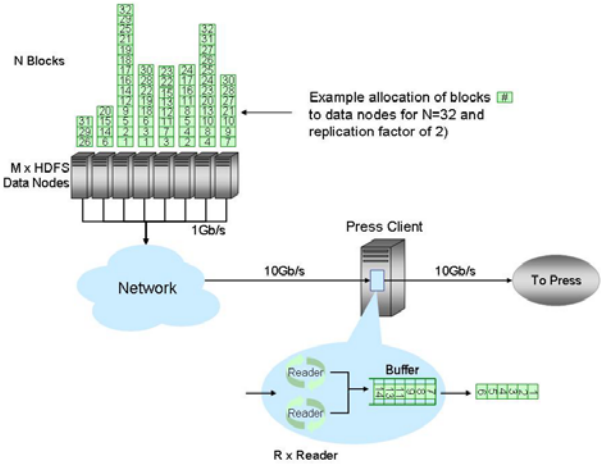

Figure 1: System Architecture for Streaming File Blocks off the HDFS

In production, the infrastructure must support ripping of new documents whilst rasterized images of previous documents are read from the file system. This could reduce block read speeds if a write operation on the same node occurred at the same time. To avoid such contention it would be reasonable to deploy multiple instances of Hadoop on separate groups of servers and to alternate which deployment is used for ripping whilst streaming data from another instance. This would avoid disk writes occurring on nodes at the same time as disk reads during when the time critical file streaming is on-going.

B. Design Options

Since there are multiple replicas of each file block (normally maintained for redundancy), it is possible for the readers to read blocks from different data nodes during the same time period and so avoid a *read collision* i.e. reading from the same disk at the same time. This can allow higher data transfer rates subject to disk, network and processor limitations. Conversely, it might be possible to use lower speed and therefore, lower cost disks in conjunction with a greater number of readers to maintain a given throughout.

There are various system trade-offs that can be made. A low replication factor, reduces the total disk storage occupied by the raster image files. Also it will reduce the time taken to save each raster image to file during the map stage because fewer replicas need to be created. Given the size of rasterized image files, the cost of replication can be considerable. However, with fewer replicas, it can be more difficult to schedule reads to avoid read collisions. To counter this, the press client buffer size and delay streaming the output (*stream delay*) can be increased to smooth out scheduling bottlenecks. The number of data nodes used in the HDFS deployment is a major cost factor. More data nodes allow file blocks to be more widely spread and can help simplify scheduling. It also allows more parallel raster image processing to be performed which is more CPU intensive. In

fact the raster image processing workload may demand more nodes than are required by the scheduling algorithm to achieve a given throughput. In summary, there are several key design parameters that will affect system cost and performance. The scheduling analysis and experimentation can be used to explore the design trade-offs and to determine the most cost effective system configurations to use.

IV. Modeling the Print Streaming Schedule

The problem of reading file blocks off the HDFS is modeled as a timetable scheduling problem. Each reader (1...*R*) is instructed to read specified file blocks from the data nodes and cache them in a local buffer. An output stream writer will read the blocks from the cache and write them to an output stream connected to the press at a rate *R* times the rate that blocks are read individually. Two scheduling algorithms are compared; a simple naive algorithm and a binary integer programming algorithm. To model the scheduling problem several variables are introduced which are summarized in Table 1.

TABLE I
VARIABLES USED TO MODEL THE SCHEDULING PROBLEM

Variable	Description
R	The number of client side readers.
M	The number of data nodes in the HDFS deployment.
N	The number of file blocks to read.
$T_{\max}$	The maximum number of time slots in the schedule.
S_b	The ordered set of indices of blocks still to be scheduled. $S_b(k)$ represents the k^{th} element from the ordered set.
TN	A $M \times T_{max}$ matrix representing the schedule timetable generated by the naïve algorithm
TIP	A $M \times T_{max}$ matrix representing the schedule timetable generated by the binary integer programming algorithm (see Figure 2 for illustration)
b_i	The i^{th} block, an integer value.
b_{itn}	A binary decision variable that represents whether to read (1) the i^{th} block at time t from n or not (0).
b_{in}	Indicates that a replica of the block with index i, is stored on node n (1) or not (0).
$\underline{x}_b$	The vector form of the timetable schedule for the integer programming based algorithm. This is a vector of the binary decision variables b_{itn}. The mapping of the decision variables to a one dimensional vector is done first by block index, then data node number and finally by time slot. Thus the decision variable b_{itn} is stored at location $i+(n-1)N+(t-1)NM$.
$\underline{c}$	The cost vector for the set of decision variables.
$b(t,i)$	The buffer cost function.

It is assumed that blocks can be read in a bounded time period called a *time slot*. There may be some variation in the time taken to read a block, but this is not an issue if a block can be read in the next time slot without colliding with an on-going read then it can proceed immediately. In fact, if each reader is assigned to read blocks from just one node throughout the whole schedule, then there cannot be a read collision irrespective of the overlap between timeslots. This is the preferred implementation approach. For example, with reference to Figure 2, a reader assigned to node 1 would read blocks 1, 3, 6 and 10 whilst a second reader assigned node 2 would read blocks 5, 4, 7 and so on. This is only possible if the complete schedule is computed in advance rather than scheduling read tasks to each reader during streaming.

Each set of blocks read by the readers are scheduled to occur within a single time slot. The scheduling problem amounts to deciding which block should be read from which node during which time slot. Geometrically, the decision variables can be visualized as points in a three dimensional space with axes of time, node index and block index. The binary decision variable b_{itn} denotes whether to read (1) the i^{th} block at time t from node n or not (0). This is represented in the two dimensional plot in Figure 2 with the block index axes projected onto the time-node plane with the block index shown rather than the binary decision variable. E.g. the time slot containing number 5 corresponds to decision variable b_{522}=1, which indicates that block 5 is read during time slot 2 from node 2.

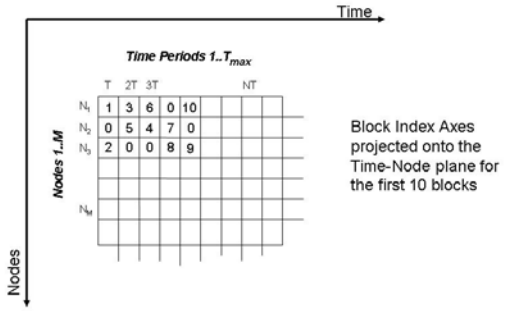

Figure 2: Visualisation of the block read schedule in two dimensions

The schedule is subject to certain constraints that are modeled in the integer programming model. The naïve algorithm incorporates the read collision constraints explicitly within the algorithm and subject to those constraints, aims to read blocks in order as much as possible. In both algorithms, the aim is to read as many blocks in parallel without read collisions, thus maximizing data streaming rates with as least buffering and stream delay as possible.

A. The Naïve Algorithm

The general strategy taken by the naïve algorithm is to read each block in index order, but without colliding with any other reads in the same time slot from the same node. If the next block cannot be read without a read collision, then the reader will be allocated the next block to read and so on. All readers are allocated blocks to read at each time slot if at all possible. Let *numToAssign* be the number of blocks to read in the t^{th} timeslot, then the naïve algorithm can be summarized as follows.

$t = 1$ # start at the first time slot
$S_b = \{b_1 .....b_M\}$ # list of all indices of all blocks to be read
while $S_b \neq \varnothing$ # while there are sill blocks to be scheduled
$S_t = \varnothing$, # set of indices of blocks to be read in time slot t
number of blocks to assign in this time slot

$numToAssign = \min(R, |S_b|)$

skip to next timeslot when there are no possible assignments for one
of more remaining readers.

$skip = 0$

while $((numToAssign > 0) \wedge (skip \equiv 0))$

$k = 1,\ assigned = 0$

iterate through remaining blocks in order

while $((assigned \equiv 0) \wedge (k < |S_b|))$

$i = S_b(k),\ n = 1$

while $((assigned \equiv 0) \wedge (n < M))$

if $((b_{in} \equiv 1) \wedge (b_i \notin S_t) \wedge (TN_{nt} \equiv 0))$

$S_t = S_t \cup \{b_i\},\ TN = i$

$assigned = 1$

end

$n = n + 1$

end

$k = k + 1$

end

$skip = 1 - assigned$

$numToAssign = numToAssign - assigned$

end

$t = t + 1$ # advance to next time slot

$S_b = S_b \ \Delta \ S_t$ # Δ is the set difference operator

end

B. The Binary Integer Programming Algorithm

Integer programming formulations require that a cost function is minimized subject to some set of constraints. The cost function, *C*, in this case is

$$C = \sum_{t=1}^{T_{\max}} \sum_{n=1}^{M} \sum_{i=1}^{N} [t + b(t,i) + s(t,i)] \times b_{itn} \qquad (1)$$

where

$$\begin{aligned} &b(t,i) = 0 \quad where\ \lceil i/R \rceil - t < 0\ \ see\ read-by\ constraint \\ &b(t,i) = \lceil i/R \rceil - t\ where\ 0 \le \lceil i/R \rceil - t \le W; \\ &b(t,i) = (\lceil i/R \rceil - t)^2\ where \lceil i/R \rceil - t > W; \\ &s(t,i) = ((b_i - t)/N)^2 \end{aligned},$$

and *W* is an integer number of time slots (in the reported experiments a value of 2 was used). The equations (1) can be converted into vector form $C = \underline{c}.\underline{x}_b$ where the decision variables b_{itn} are mapped into the vector $\underline{x}_b$ firstly by block index, then node index and finally time index. The values of $\underline{c}$ are pre-computed and are constant coefficients as required for integer programming. The terms from (1) involving the ceiling function are used to map the block index *i*, to the time slot *t*, since *R* blocks are to be read per time slot; blocks 1…*R* should be read during time slot 1 and so on. The cost function encourages blocks to be read as soon as possible, by weighting the decision variables by their associated time slot. At the same time, a cost *b*(*t*, *i*) is applied for buffering blocks far in advance of when they will be written to the output stream. Specifically, when a block is read more than *W* time slots before it is required, the cost function coefficients in (1) are generated from a quadratic rather than linear function. The term $s(t,i)$ is a small bias to ensure that block *i* will be read before block *j* when *i* < *j*, with all other costs equal.

The set of constraints are now considered in turn. The read-once constraints

$$\sum_{t=1}^{T_{\max}} \sum_{n=1}^{M} b_{itn} = 1,\ \forall i, \qquad (2)$$

ensure that each block is read only once. The read collision constraints are defined by

$$\sum_{i=1}^{N} b_{itn} \le 1, \quad \forall t,n \quad where \quad b_{in} \ne 0 \qquad (3)$$

which ensures that only one block can be read from a node during a timeslot. The completion time constraints are given by

$$b_{itn} = 0 \quad for \quad t > T_{\max} \qquad (4)$$

To support this constraint, the number of time slots is limited to $T_{\max}$, which should be set to a little larger than N/R to allow for a feasible solution to be obtained even with problematic block distributions that may prevent *R* blocks being read in one or more time slots.

The read-by constraints are required to ensure that each block has been read before it will be written to the output stream. This can be expressed in the form

$$\sum_{t=T_{\min}}^{T_{\max}} \sum_{n=1}^{M} b_{itn} = 0 \quad \forall i\ where\ b_{in} = 1 \qquad (5)$$

where $T_{\min} = \lfloor i/(R+\delta) \rfloor + D_{\min}$ and $D_{\min}$ is a fixed delay allowed before the block must be read and δ is a small arbitrary fraction to ensure the floor function evaluates to the required integer value. A hard buffer constraint can be enforced as

$$\begin{aligned} &0 \le b_{itn} \le 1, \quad \forall t - m \le i \le t \\ &b_{itn} = 0 \quad \forall i < t - m \quad and \quad i > t \end{aligned} \qquad (6)$$

to ensure that a block is only read within some fixed window prior to it being streamed. Note the cost function incorporates a soft buffer constraint by raising the cost of reading a block far in advance of it being streamed.

The number of readers is constrained according to

$$\sum_{n=1}^{M} \sum_{i=1}^{N} b_{itn} \le R \quad \forall t \qquad (7)$$

Finally the number and distribution of blocks across the Hadoop data nodes is a function of the number of replicas specified in the Hadoop configuration and the random assignment of blocks to data nodes. The distribution of the file's blocks represents the final set of constraints limiting which blocks can be read from each node. In these cases the decision variable is set to 0 if the block is not readable from the associated node.

Some of the constraints described above are implicitly handled in the naïve algorithm. The other constraints could be built in to the algorithm, at the cost of increased complexity.

The size of the integer program grows with increasing numbers of nodes and file blocks making solution more difficult. However, the scheduling problem can be readily sub-divided into a sequence of smaller sub-problems with the end state of one sub-problem imposing additional constraints on the next sub-problem. If during the last time slot in the first sub-problem there are one or more unallocated readers, then that time slot will become the first time slot of the next sub-problem. In such a case the constraints (2) and (7) are adjusted for the first time slot according to the remaining number of available readers and free nodes. In this way several sub-problems can be stitched together to provide a tractable means of solution for very large files containing many blocks.

C. Constraints for Fault Tolerance

A key feature provided by Hadoop is automatic replication of file blocks to provide fault tolerance. In the current application, this redundancy is used to allow non-sequential blocks belonging to the same file to be read in parallel. Each replica provides an extra degree of freedom providing flexibility in the schedule. However, the redundancy is still valuable to recover from a data node failure part way through printing a document. In this case, it is possible to amend the approach described above to allow block reads to be rescheduled in the event of a failed read.

The general approach to provide data node fault tolerance is to insert contingency time slots in the schedule to allow time to read a block from another data node. Contingency can also be provided by increasing the number of block replicas, but this would slow down the time taken to store the rasterized images. Using contingency time slots requires that after every T_c time slots, a spare time slot is reserved, to provide an opportunity to re-read blocks, scheduled to be read from a failed data node, from the other nodes. This requires additional client side buffering, and requires adjustment to the constraints above. Firstly, the read once constraint is relaxed to allow blocks to be read multiple times. Secondly, additional constraints must be satisfied such that the schedule will allow all of the blocks scheduled to be read from any failed data node in the preceding T_c -1 time slots to be re-read from other the remaining data nodes. Thus the collision constraint at the contingency time slots should be relaxed to allow up to T_c -1 blocks to be read from each remaining active node, but noting that the non zero entries represent *alternative* blocks to be read depending on which node fails.

The contingency periods, do not have to be used if there are no data node failures. Thus in the absence of failures, the contingency period can be skipped, but the aggregate data transfer rate will be reduced according to the frequency of contingency periods.

D. Scheduling for Multiple Streams

As noted in the introduction, sharing the same IT infrastructure across multiple print presses is attractive. To support this, the scheduling approaches described above can be adapted to allow multiple streams for different presses to be generated in parallel. There are various approaches that can be taken such as dividing resources evenly between streams and running independent algorithms, or adapting the algorithms to share the available resources between multiple competing streams. E.g. blocks read from a node by one stream will prohibit the other streams from reading from the same node in that same time slot. It is relatively straightforward to adapt of some of the constraints (e.g. read-by constraints) to support parallel streams.

E. Smart Placement

It is possible to invert the scheduling problem described above by ensuring that blocks are written to nodes in such a way as to permit efficient streaming of the data off the HDFS. This will require modifying the block distribution algorithm used by the file system to allocate blocks to data nodes in the first instance. This may conflict with fault tolerance concerns such as replicating blocks on nodes hosted on different equipment racks with independent power supplies. Optimally writing blocks on nodes to allow efficiently reading those blocks at a later time will require solution of a similar optimization problem to that described.

V. Results

Sets of test cases were generated by randomly generating different block distributions across data nodes for varying numbers of nodes, replication factor, file sizes and readers. Then the two scheduling algorithms are applied to each test case to generate a schedule. From each schedule, the *stream delay* and last time slot that the final block was read are determined under the condition that blocks are written to the stream at a continuous rate of R blocks per time slot. This ensures that the cache utilization will remain relatively stable. The stream delay is defined as the number of time slots elapsed before the output stream can be written. The prior condition requires that the stream delay is sometimes increased to ensure continuity of the data stream once started. The frequency with which the stream delay is increased is reflected in the average values shown later. The average size of the buffer, measured in blocks, is calculated and normalized by the number of readers. Normalization is applied to allow comparison for varying R, because as R increases the number of blocks that need to be buffered is necessarily greater, but relative to the read rate the buffer size may be smaller. All the results are averaged over 500 test cases.

The experiments were performed using MATLAB™ and the binary integer programming solver provided in the Optimization Toolkit [12].

A. *Scheduling Algorithm Results*

In Figure 3, the performance obtained using the BIP algorithm is shown for varying numbers of replicas and readers. In the key, the first digit is the number of readers, the second digit is the number of replicas and the last digit is the number of nodes. 16 data blocks had to be scheduled in each case. The same performance data is shown in Figure 4 for the naïve algorithm. The best results are closest to the origin i.e. no buffering and no stream delay required. The dash lines represent contours of circles of radius 3 through 6 centered on the origin to help identify points at equi-distance from the origin. Points which represent tests using the same number of readers are grouped inside the numbered ellipsoids for clarity.

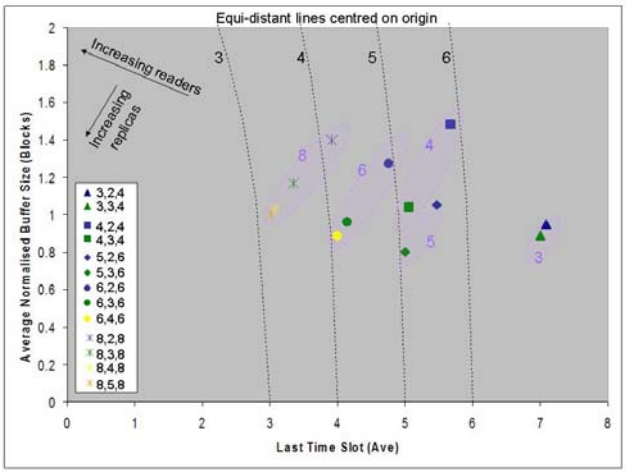

Figure 3: Performance of the binary integer programming scheduling algorithm with varying numbers of replicas and readers

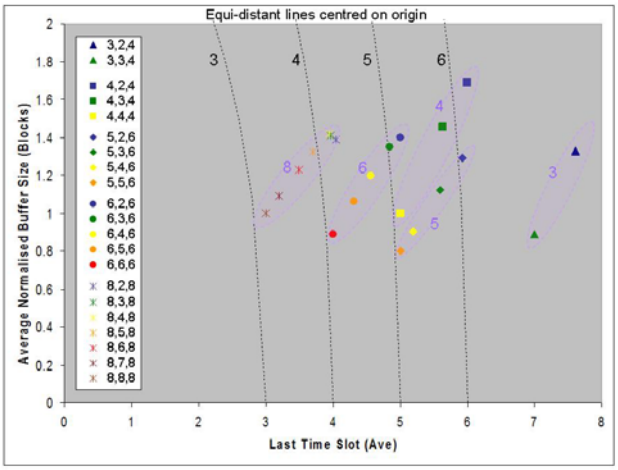

Figure 4: Performance of the naive scheduling algorithm with varying numbers of replicas and readers

The BIP algorithm generally provides better results unless the file was nearly or, fully replicated on every data node. In general for a given number of readers, the naïve algorithm requires more replicas to support the same data rate (hence the greater number of data points shown for the naïve algorithm). This is a benefit of using BIP since reducing the number of replicas reduces both storage capacity and the time taken to save each raster image file.

Note the performance using 5 readers is not markedly better than using 4 readers for this particular experiment simply because with 16 blocks to be scheduled the last block will always need to be read in the 4th time slot. Figures, 3 and 4 also show the average normalized buffer sizes at less than 1. Strictly speaking the steady-state average normalized buffer size should approach 1. This is because with a minimum required stream delay of 1 and *R* readers there will always be at least *R* blocks buffered at the end of each time slot. After normalization by *R*, this is equivalent to a normalized buffer size of 1. However, at the end of the schedule, the number of blocks remaining in the buffer will decrease as the last few are read by the output stream writer. If, for example, in the penultimate time slot, only the last block remains in the cache, this will reduce the average buffer size. This is most apparent in the case of 5 readers. In the ideal schedule 5 blocks are read and written each time period. However, when reading 16 blocks, there is always one remaining block to be read in the 4th time slot. This gives an average buffer size of 4 blocks and therefore a normalized buffer size of 0.8. Over a longer schedule the impact of the artifact would be less pronounced. The impact of this is to shift the plotted points downwards.

It is observed that just increasing the number of readers ultimately provides diminishing returns, assuming a fixed number of replicas i.e. the scheduling algorithms are unable to realize the maximum theoretical increase in stream rates. This is because of the increasing likelihood of read conflicts when using many readers. This issue can be particularly acute in certain cases e.g. where the distribution of a block's replicas are similar for many sequential blocks such that the read collision constraints limit scheduling options. There are many other less obvious block distributions that are difficult to schedule efficiently. By determining a schedule in advance of streaming can allow for these problematic distributions to be identified and for remedial steps to be taken (see later).

Average statistics do not fully characterize the relative performance of the two algorithms. When specifying the system configuration, the possible peak stream delays and maximum buffer sizes need to be determined. However, the average values do indicate the relative frequency of longer stream delays and consequently the need for larger buffer sizes.

In Figure 5 the performance of a 4-node cluster using 4 readers and either 2 or 3 replicas for a varying number of file blocks are compared. A smaller number of blocks were chosen for this experiment to allow schedules to be generated for a larger numbers of file blocks.

As the number of blocks is increased, the out-performance of the BIP algorithm is reduced. This is thought to be due to an averaging effect whereby large groups of blocks tend to contain sub groups which cannot be scheduled efficiently irrespective of the scheduling algorithm. Also when the ratio

of number of replicas to number of nodes is lower (in this case 2 replicas and 4 nodes), the schedules generated using the BIP algorithm are only slightly more efficient. This is because the read collision constraint more frequently limits scheduling options when there are few block replicas as noted previously. This is examined in more detail next.

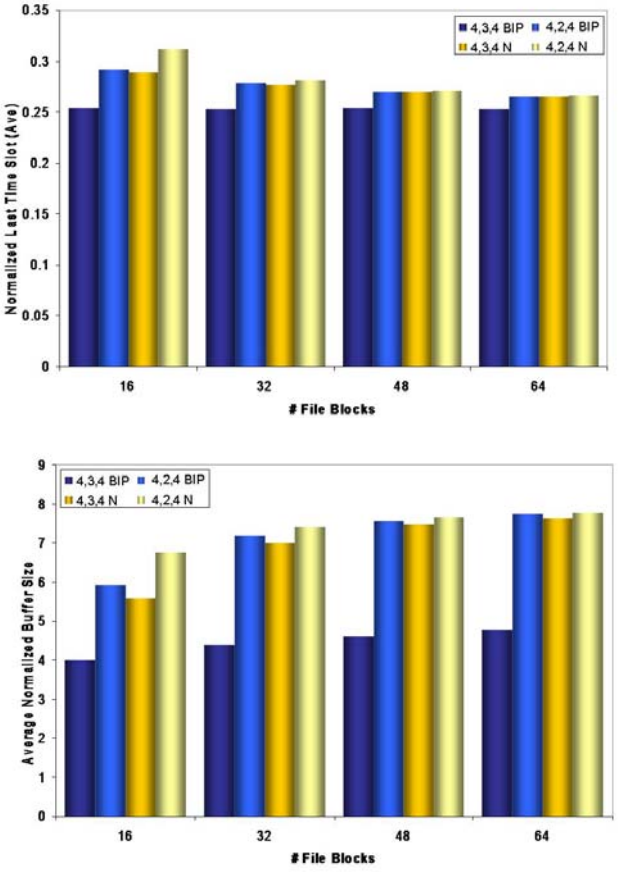

Figure 5: Performance comparison of the BIP and naïve scheduling algorithms for a 4 node Hadoop cluster using 4 readers and varying numbers of replicas and blocks. Normalized Last Time Slot (top) and Average Normalized Buffer Size (bottom)

In the next experiment, minimum stream delays were obtained from schedules derived using 5 readers and 2 replicas for a varying number of nodes. The minimum stream delay is the delay (in time slots) required before the files blocks can be written to the output stream without interruption assuming a sustained output stream rate of 5 (=*R*) times the average transfer rate from the Hadoop data nodes. Note that a stream delay of 1 is the minimum; a delay of 1 time slot is required to allow the blocks read during the first timeslot to be cached before writing them to the output stream.

It is very clear that the BIP algorithm results in schedules with many fewer stream delays of 2 compared to the naïve algorithm. The naïve algorithm also resulted in stream delays of 3 time slots when using just 5 nodes. A longer stream delay will require additional cache capacity to be provided. For both algorithms, increasing the number of nodes reduces the number of stream delays greater than 1. This general pattern is observed across a range of system configurations.

It is possible to reduce the occurrence of long stream delays by pre-reading a few selected blocks which are problematic to schedule owing to poorly distributed blocks (e.g. as noted above a sequence of blocks that are distributed across broadly similar nodes is problematic to schedule). Examining a schedule to identify problematic blocks and pre-caching them can avoid long stream delays at the expense of slightly more buffering. With fewer cases of long stream delays, this additional step is required less often. It's worth noting that it is only by determining a schedule up-front that any problematic block distributions can be identified.

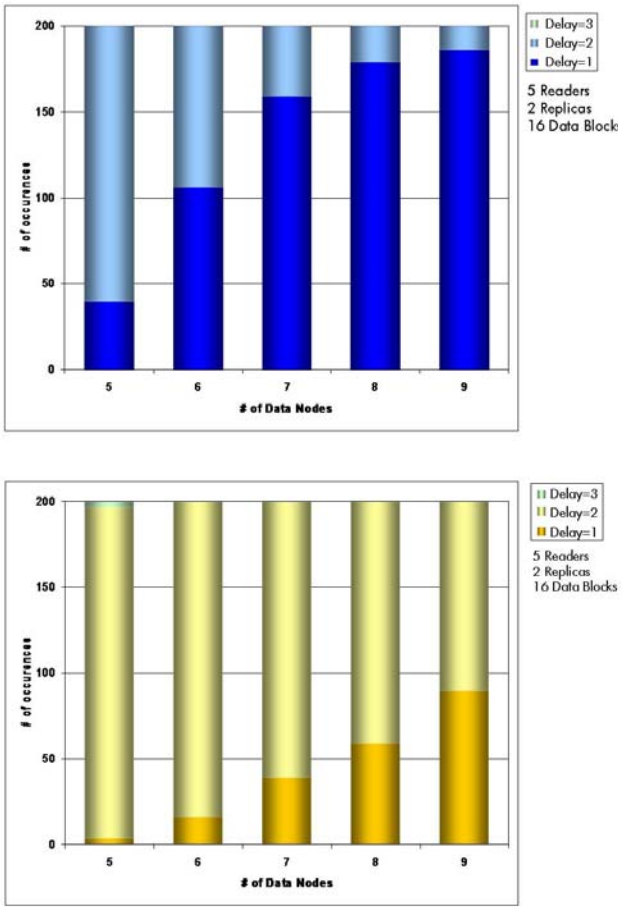

Figure 6: Distribution of stream delays for schedules generated by the BIP (top) and naïve (bottom) algorithms for a 5 node Hadoop cluster with 5 readers, 2 replicas and varying numbers of nodes

The number of Hadoop nodes required in a deployment may be dictated by the RIP speed requirements where performance is CPU bound rather than IO bound. If this is the case, a benefit of using the extra nodes for raster image processing is the potential for reducing the buffer requirements at the press client. Conversely, if the number of nodes utilized during the RIP stage is relatively low then it may be necessary to consider higher degrees of file replication.

B. Measured Results

To test the rate at which file blocks can be transferred from the HDFS, several measurements of data transfer rates were carried out using the arrangement shown as 'Test Path 3a/b' in Figure 7. The version of Hadoop used in these tests is 0.18. The application running on the press client is expanded in the blue rectangle. In this test, multiple parallel connections to *R* different data nodes are established and data

blocks are read from each data node in parallel by the *readers* executing on the press client. To provide an upper bound on transfer rates, the same data block is read repeatedly from each data node (practically rates would be limited by the underlying disk performance). The readers write the files blocks to files on tmpfs [15], a RAM-based filesystem with 16GB of memory allocated, which serves as the block buffer. When a file block has been successfully written to a data file in tmpfs, a second zero-length 'notification' file is also created which indicates that the associated data file, is available to be read by the output stream writer. This provides a simple, but effective means of notification. The output stream writer polls for each notification file in order and if available then proceeds to read the associated data file. The output stream writer thus reads the blocks in order from tmpfs and writes them to an output stream. In this case the data is written to either a socket connected to a server running on Test Node 2, or to /dev/null, the latter acting as a control case measuring the transfer rate in the absence of additional outbound network traffic being generated as in path 3b.

Prior to carrying out Test 3, the networking configuration was tested and optimized using Netperf 2.4.5 [14]. Netperf was used to measure the transfer rates between the press client and a second identical server in place of the press (Test-Node-2) and the date rates between an HDFS data node (Test-Node-1) and the press client. The latest version of the NetXen Driver and firmware (v4.0.406-4, released Jun 2009) for the HP NC512m NIC was used. In the experiments described here, earlier driver versions resulted in slower transfer speeds.

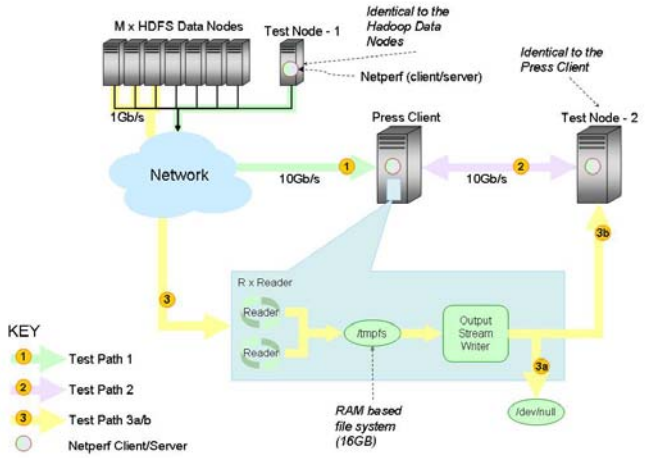

Figure 7: Data Transfer Rate Measurements for three different network paths

To achieve the highest rates possible several modifications were made to the TCP/IP configuration on each node. Useful suggestions are available from [16]. The most impactful change was to increase the MTU (minimum transmission unit) from 1500 to 8000 blocks (i.e. Jumbo Frames). The TCP max buffer size for send and receive was increased to 16777216. TCP Segmentation Offload was enabled. The interface queue length was set to 65535. The TCP auto-tuning buffer limits (net.ipv4.tcp_rmem and net.ipv4.tcp_wmem) were set to (4096 87380 16777216).

The measurements taken when transferring data along paths "Test Path 1" and "Test Path 2" as illustrated in Figure 7 are shown in Table II. Bandwidth and CPU utilization measurements were taken using both Netperf and a Java application. Java was used for measuring transfer rates as this would be the likely development language for a future implementation. The Netperf script tcp_stream_script was used to measure the data transfer rate across a single stream between the press client and test nodes 1 and 2. The highest rates were obtained using send and receive buffer sizes of 512K and a message size of 32-64KB. The CPU utilization (CPU), measured at the sender and receiver are denoted as CPU TX and CPU RX respectively. The peak rate for Test Path 1 is close to 1Gb/s which is as expected. For the Test Path 2, the transfer rate averaged at ~6.2Gb/s which is comparable with the measurements presented in [13]. Table II, also shows results using the Java based application using Non-Blocking IO. The test writes data across a single stream between a client and server instance. Using this test, data rates less than, but comparable to Netperf were obtained.

TABLE II
NETWORK PERFORMANCE RESULTS

Test	BW (Gb/s)	CPU TX	CPU RX
Test-Path 1, Netperf	0.98	2%	3.3%
Test-Path 2, Netperf	6.2	12.5%	8.2%
Test-Path 1, Java	0.94	N/A	N/A
Test-Path 2, Java	5.6	N/A	N/A
Test-Path 2, Java (2 Streams)	8.3	N/A	N/A

The tests were also repeated using multiple instances of the java application executing in parallel. An instance of the java client was run on each data node, with the server instances running on the same press client. In this case (not shown in the Table), the data rate per client connection was maintained at 940Mb/s for up to 8 clients giving an aggregate transfer rate of approximately 7.5Gb/s. Using Test Node 2, two clients were run on the test node and two server instances were run on the press client. The measured aggregate data transfer rate was ~8.3Gb/s.

For the tests relating to Test Path 3a/b, each reader was run as a separate java processes. The modified Hadoop client code was used by the readers to establish connections to the HDFS and to read the blocks. Note the connections to the Hadoop data nodes do not use non-blocking IO as this was not supported by Hadoop at the time of writing. The output stream writer begins to stream blocks from tmpfs as soon as each reader has completed reading a first block. A larger delay may cause the measured to be artificially inflated because it allows the output stream writer to read blocks from tmpfs faster than the readers are actually writing them, at least for a period of time.

The measured transfer rate along paths 3a and 3b for varying numbers of readers are shown in Figure 8. The typical data rate for a single reader, reading a file block off HDFS was approximately 700Mb/s. This is the rate along Test Path 1 and is significantly less than the rates obtained using Netperf and the test java application (see Table II).

This is partly due to the more complex code path of the reader application, but indicates a possible point of future optimization. Along path 3a, there is a slightly less than linear increase in the aggregate output rate as the number of readers is increased, reaching approximately 4Gb/s (500Mb/s per reader). Using more than 8 readers (the number of CPUs on the press client), the data rate only increases marginally.

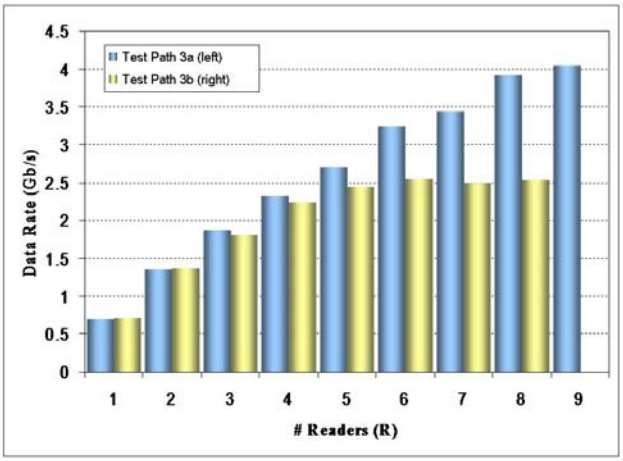

Figure 8: Data transfer rate measurements over Test-Paths 3a and 3b when reading file blocks from Hadoop with varying numbers of readers. The blocks are cached before writing over Test-Paths 3a and 3b.

When transferring data over path 3b, the rate of increase with respect to *R* follows a similar slope to 3a, but after *R*=4, the data rate no longer increases. This limit is because the same network interface was used for both reading blocks from DHFS and writing the file blocks (the second 10Gb/s networking interface was not available for testing). In effect the network interface is supporting roughly 5Gb/s transfer rates in total. The results for 3a indicate strongly the scope for higher data rates if the second network interface was used for writing data to the press. The data rates presented in Figure 8 represent an upper bound on achievable data rates in a working system since the same block on each data node is repeatedly read in the tests.

A few additional measurements were taken to estimate the transfer rates if using a second network interface. In a first test, a large number of blocks were first pre-cached in the tmpfs buffer. The average transfer rate obtained when reading files from tmpfs and writing them directly to a server along path 3b was measured. This test ensured that the stream being written along 3b was no longer competing for network bandwidth with the readers reading blocks from the Hadoop file system. The measured rate was ~3.7Gb/s which is more representative of the data rate that could be achieved in the previous experiment if a second network interface was available. A second test, similar to the first, repeatedly wrote the contents of an in-memory buffer over path 3b. An average rate of 4.4Gb/s was measured indicating that reading the cached file blocks from tmpfs significantly reduced transfer rates. In both cases, the rate was less than the 5.6Gb/s reported in Table II, because the test used the more complex output stream writer code and so is not directly comparable.

VI. Conclusion

This paper describes an application of the Hadoop file system [4] to high speed digital printing. Hadoop provides a large scale distributed fault tolerant file system made up of a set of data nodes, running on low cost servers with direct attached storage. Fault tolerance is provided by storing multiple replicas of each file block across different data nodes. The redundancy provided by Hadoop is used in the present application to enable high speed raster file streaming over a 10Gb/s LAN to a press. The paper shows how to apply a binary integer programming algorithm to make use of Hadoop's file block replication to derive efficient schedules for reading file blocks in parallel from the Hadoop file system. Based on the schedule, a set of readers running in parallel on a press client, are directed to each read file blocks from a specific data node and write them to a local in-memory cache. The blocks can then be written from the cache to the print press over a single higher rate stream. Transfer rates up to 4Gb/s were measured when using 8 parallel file block readers with only a few minor extensions to the Hadoop client code. The scheduling results suggest that when using 8 readers, at least 3 and preferably 4 replicas can be beneficially used to limit the buffer size required at the press client.

Acknowledgment

I would like to acknowledge Eduardo Ceballos and Jon Brewster of HP for introducing the problem and motivating the use of Hadoop. I would also like to thank Stephen Pearson, Niall Saunders and Stuart Martin for their help setting up the infrastructure and Tony Wiley for supporting the work.

References

[1] L. Chudzinski, A. Peck, A. Hale, C. Sherburne, E. Padula, "Customized Communications: Advanced VDP", Infotrends 2008,.
[2] F. R. Meneguzzi, L. L. Meirelles, F. T. M. Mano, J. B. de Souza Oliveira, A. C. Benso da Silva, "Strategies for Document Optimization in Digital Publishing," in Proceedings of the 2004 ACM symposium on Document engineering, ACM, 2004, pp. 163-170.
[3] HP Ink jet Web Press http://www.hp.com/go/inkjetwebpress
[4] Apache Hadoop http://hadoop.apache.org/
[5] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI'04: Sixth Symposium on Operating System Design and Implementation. 2004.
[6] S. Daskalaki, T. Birbas, E. Housos, "An integer programming formulation for a case study in university timetabling," in European Journal of Operational Research, vol. 153, Elsevier B.V., 2003, pp 117-135.
[7] BitTorrent http://www.bittorrent.com/
[8] D. A. Patterson, G. Gibson, R. H. Katz , "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in Proceedings of the 1988 ACM SIGMOD international conference on Management of data, 1988, pp. 109-116.
[9] J. H. Hartman, J. K. Ousterhout, "Zebra: A Striped Network File System," in Proceedings of the USENIX Workshop on File Systems, in ACM Transactions on Computer Systems, May 1992.
[10] P. Rizk, C. Kiddle, R. Simmonds "A GridFTP Overlay Network Service," in Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, 2006, pp. 41-48.
[11] http://h20000.www2.hp.com/bizsupport/TechSupport/Document.jsp?lang=en&cc=us&taskId=120&prodSeriesId=3432831&prodTypeId=329290&objectID=c01073636
[12] MATLAB http://www.mathworks.com/products/matlab/
[13] S. Pope, D. Riddoch, "10Gb/s Ethernet performance and retrospective," in SIGCOMM Computer Communication Review , Volume 37 Issue 2, March 2007.

[14] R. Jones, “Netperf Homepage” http://www.netperf.org/netperf/ .
[15] D. Robbins, “Common threads: Advanced filesystem implementor's guide, Part 3” http://www.ibm.com/developerworks/library/l-fs3.html
[16] Lawrence Berkely National Laboratory “TCP Tuning Guide” http://fasterdata.es.net/TCP-tuning/linux.html.