USENIX Association

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

In-Place Rsync: File Synchronization for Mobile and Wireless Devices

David Rasch and Randal Burns
Department of Computer Science
Johns Hopkins University
{rasch,randal}@cs.jhu.edu

Abstract

The open-source rsync utility reduces the time and bandwidth required to update a file across a network. Rsync uses an interactive protocol that detects changes in a file and sends only the changed data [18, 19]. We have modified rsync so that it operates on space constrained devices. Files on the target host are updated in the same storage the current version of the file occupies. Space-constrained devices cannot use traditional rsync because it requires memory or storage for both the old and new version of the file. Examples include synchronizing files on cellular phones and handheld PCs, which have small memories. The in-place rsync algorithm encodes the compressed representation of a file in a graph, which is then topologically sorted to achieve the in-place property. We compare the performance of in-place rsync to rsync and conclude that in-place rsync degrades performance minimally.

1 Introduction

Rsync [18, 19] makes efficient file synchronization a reality. It enables administrators to propagate changes to files or directory trees. To save bandwidth and time, rsync moves a minimum amount of data by identifying common regions between a source and target file. When synchronizing files, rsync sends only the portions of the file that have changed and copies unchanged data from the previous version already on the target. Faster and more efficient methods for synchronizing copies make it easier to manage distributed replicas.

Despite rsync's efficiency, its shortcomings sometimes preclude its use. We address one specific shortcoming. Each time rsync synchronizes a file, it reserves temporary space in which it constructs the new file version. Rsync maintains two copies (one new, one old) on the target for the duration of the transfer. Rsync cannot be used without sufficient temporary space for two copies of a file.

The construction of the new target file in temporary space often renders rsync unusable on mobile devices with limited memory. A popular device by Palm contains only 16MB of memory. For the Palm to keep enough temporary space available might require up to 8MB free (Figure 1). Insufficient space often excludes the Palm from performing traditional rsync and forces a transfer of the entire file. Ironically, handheld systems, compact and convenient machines that can benefit from an efficient propagation of updates, cannot always afford the space overhead of rsync.

Rsync cannot backup or replicate block devices. Although the benefits of compression make rsync well-suited to the task, systems rarely have spare block devices on which to put temporary data.

We have modified rsync so that it performs file synchronization tasks with in-place reconstruction. We call this in-place rsync or *ip-rsync*. Instead of using temporary space, the changes to the target file take place in the space already occupied by the current version. This tool can be used to synchronize devices where space is limited.

In-place reconstruction eliminates the need for additional storage by using the space already occupied by the file [2, 3]. In-place reconstruction seems trivial, but the process must account for hazards that arise when moving a block of data from its original location in the old file to its location in the new file – an operation called a COPY command. Not only does each COPY read a block of the file, but it also overwrites k bytes. Overwritten regions cannot be used in future COPY commands because they no longer contain the original data.

The goals of the ip-rsync algorithm include: (1) prevent the copying of corrupted data, which has been previously written by another COPY operation; and, (2) minimize compression loss. To prevent the copying of corrupted data, ip-rsync identifies COPY commands that write into regions from which other COPY commands read and then performs the read operation (on the original data) before executing the write. It is not always possible to reorder COPY commands to avoid all conflicts. In this case, ip-rsync discards the conflicting COPY operation. The data corresponding to the COPY are sent from the host to the target. Sending the additional data, instead of copying from the file already on the target, reduces compression and increases the time needed to synchronize files. Ip-rsync implements several heuristics for selecting COPY commands to eliminate that min-

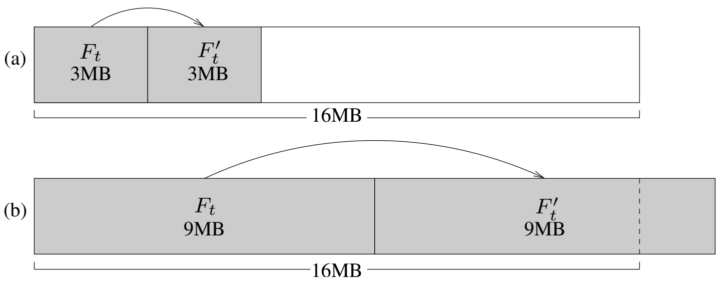

Figure 1: With 16MB of memory: (a) There is enough space for the file F_t (3MB) and a temporary copy F'_t. (b) There is insufficient space for a temporary copy F'_t of F_t (9MB) and rsync cannot be performed.

imize compression loss.

We describe the in-place rsync utility as an extension to rsync. We start with an overview of the rsync algorithm and a discussion of its performance optimizations. We follow with our algorithm for performing rsync in-place and a discussion of the effect of in-place reconstruction on the algorithm's optimizations.

2 Background

Ip-rsync builds on the rsync algorithm for propagating changes between two files located at different sites [18, 19]. Rsync is widely used for backup and restore, as well as file transfer. Rsync finds blocks of data that occur in both the target file and the source file. It saves bandwidth and transfer time when updating the target file by not sending blocks from the source file that exist already in the target file. Rsync operates on a fixed block size in a single pass, or round, over the files. A multi-round version of rsync (*mrsync*) increases compression by taking multiple passes over files, halving the block size in each subsequent round [9]. Multi-round rsync detects common sections of the files at a fine granularity. However, in multi-round rsync, the sender (source) does not send unmatched data until all rounds are complete. In this way, multi-round rsync loses much of the benefit of of the interleaved transfer found in rsync. Rsync transmits unmatched data while searching within the file for matching data. Rsync outperforms mrsync when the similarities and differences between files consist of large sequences. Mrsync performs well when files consist of short matching sequences separated by small differences. Mrsync is more suitable in lower-bandwidth networks in which transfer time dominates. Although our in-place algorithm is suitable for multi-round rsync, we did not implement an ip-mrsync because of mrsync's limited adoption.

The concepts of rsync influence the design of many distributed systems. In particular, the combination of a weak and strong checksum has been used in a low-bandwidth file system [12], migrating virtual computers [15], and a transactional object store [16].

Rsync has much in common with *delta compression*. Both encode changes between files using COPY and ADD commands. Delta compression differs in its semantics because it compares two files that are collocated, rather than two files separated by a network. Algorithms for delta compression are based on hashing [1, 10, 13] or extensions to Lempel-Ziv compression [4, 6, 8]. The problem of compactly representing versions as a small set of changes was introduced by Tichy as the string-to-string correction problem with block move [17].

In-place reconstruction has been previously addressed for delta compression [2, 3]. The problem and solution are similar to ip-rsync, because they both reorder the execution of commands to avoid conflicting COPY commands.

We feel that in-place reconstruction is more widely applicable in rsync than in delta compression. In-place delta compression can transmit data to a resource limited device. However, it precludes a resource limited sender because it requires both versions of a file to generate a delta encoding. Rsync allows versions to be synchronized between two resource-constrained devices and, therefore, may be used in peer-to-peer and serverless applications.

3 Design

Rsync synchronizes two files, bringing an old version of a file on the *target* up to date with a new version of a file on the *source*. Rsync sends as few bytes as possible by detecting common sections of the two versions and using the common data in the old version when building the new version. To detect the common sections, the target generates a weak and strong checksum for blocks in the target file. The target transfers the checksums to the source. On the source, rsync stores the weak checksums in a hash table. The checksums take approximately 100 times less space and bandwidth than the file. The source file is scanned, calculating the weak checksum at each offset. Rsync probes the hash table with each checksum. Upon finding a matching checksum, a strong checksum

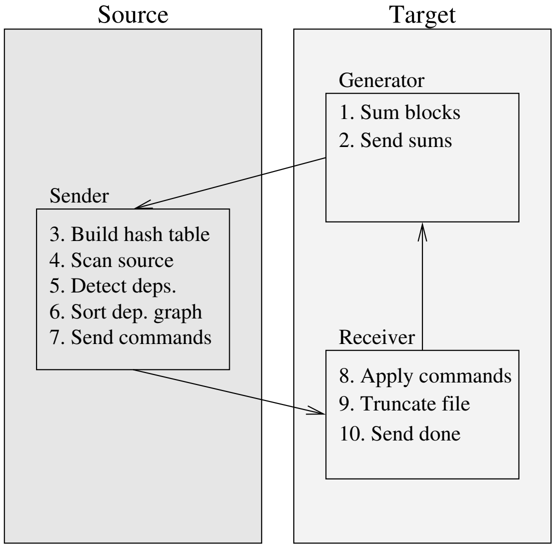

Figure 2: The rsync process triangle with the sequence of steps necessary for an ip-rsync transfer.

is computed and compared. Matching strong checksums indicate matching data with high probability. Rsync encodes matched data as a COPY command. The COPY encodes an action on the target that duplicates data from the reference file in the updated file. The algorithm encodes unmatched data as an ADD command, which includes the data to be added. The target receives encodings from the source. Encodings describe the new file sequentially (from first byte to last) so that the target can reconstruct the new version in a single pass. The use of a weak and strong checksum saves computation by allowing the source to generate and compute a strong checksum only when the weak checksum already matches. Also, the chosen weak checksum saves computation by rolling from offset n to offset $n+1$ without recomputing the checksum based on all k bytes [7]. Rolling checksums observe that strings of length k at offsets n and $n + 1$ differ in only two bytes – the first byte of the string starting at n and the last byte of the string starting at $n+1$. The algorithm computes a rolling checksum for offset $n + 1$ by subtracting the contribution of the first byte of the checksum at offset n and adding the contribution of the last byte of checksum at offset $n + 1$.

3.1 Algorithm

The rsync implementation uses a programming construct referred to as a *process triangle* that defines three processes, one on the source and two on the target (Figure 2). The *generator* process runs on the target and generates the checksums for the target file (F_t) that get sent to the *sender* process ($G \Rightarrow S$). The sender scans the source file (F_s) for the sums it receives and transmits the results to the *receiver* process ($S \Rightarrow R$). The receiver applies the delta commands and notifies the *generator* if a file needs to be resent ($R \Rightarrow G$). The *generator* and *receiver* processes run independently on separate files concurrently and proceed independently.

Rsync transmits ADD and COPY commands in the order in which they were detected during a sequential scan of the source file. The target applies the commands in the order received. No destination offsets need to be specified for COPY and ADD commands. The algorithm calculates the destination offset from the previous destination offset and the length of the previous command.

With ip-rsync, the commands undergo a reordering step to facilitate corruption-free, in-place reconstruction. As such, the transmission of the ADD and COPY commands in a non-sequential order requires the explicit specification of a destination offset with each command. The destination offset allows commands to be applied in any order at the target. The extra data in the ip-rsync codeword adds a small overhead to the bandwidth requirements.

Ip-rsync takes the following actions to synchronize a file (the bold text indicate where ip-rsync and rsync differ):

1. *Generator:* Generate weak and strong checksums for each block in the reference target file.
2. *Generator:* Send checksums to the *sender*.
3. *Sender:* Build a hash table from the checksums received.
4. *Sender:* Scan the source file for matches. **Buffer all COPY and ADD commands.**
5. *Sender:* **Construct a dependency graph among COPY commands.**
6. *Sender:* **Topologically sort the dependency graph, breaking cycles as they are detected.**
7. *Sender:* **Send sorted COPY commands, followed by ADD commands, to the *receiver*.**
8. *Receiver:* Apply commands when received. For each command, **seek to the given offset**, and either copy data or insert the included data.
9. *Receiver:* Truncate the file if it decreased in size.
10. *Receiver:* Alert the *generator* of the file's completion.

In detecting and resolving dependencies, ip-rsync sacrifices some concurrency. The sender conducts an analysis of all COPY commands prior to transmitting any commands to the receiver. This causes ip-rsync to wait until all COPY commands are found and analyzed before transmitting data. Traditional rsync overlaps detecting and transmitting matches by sending data to the

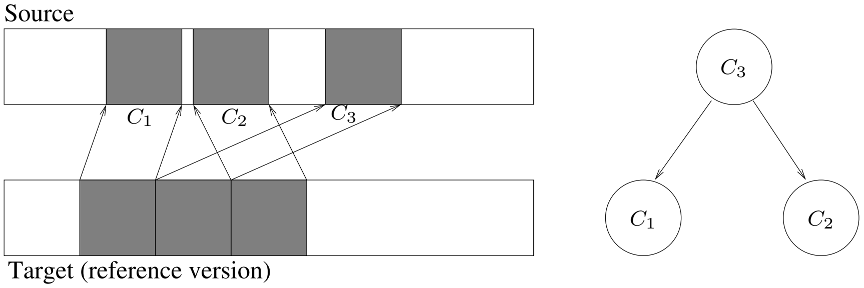

Figure 3: An example of a file synchronization and the associated dependency graph. The blocks on the target will move from their original location (bottom) to match their location on the source (top). To paraphrase the graph, C_3 must be completed before C_1 or C_2 since the destination of these blocks will corrupt the source for C_3; *i.e.*, perform C_3 first so that C_1 or C_2 do not overwrite its source data.

receiver immediately. The sender detects dependencies among all COPY commands and waits for the discovery of all commands before reordering.

3.2 A Simple Example

We now digress to a simple, yet flawed algorithm, which one might propose as an alternative to the in-place rsync algorithm. The following algorithm is a trivial modification to rsync. Its flaw results in significant compression loss. This example motivates the need for the complexity (buffering and dependency detection) of our solution. The simple algorithm runs rsync and employs the following heuristic: if any COPY command refers to a block location which precedes the current write offset, then that block has been overwritten with new data and can no longer be used for COPY commands. The lost data is resent over the network link as an ADD command. The advantage of this algorithm is the similarity to the original algorithm and its ability to overlap I/O with the computation of checksums.

This simple algorithm is sensitive to changes which insert data into the source file. Such changes cause the simple algorithm to generate many ADD commands, resulting in large compression losses. Consider synchronizing a pair of files that are identical except that the source file has some data (as few as 1 or 2 bytes) inserted at the beginning. These few bytes prevent any data from being copied.

Reordering commands is necessary in spite of the rarity of cycles in dependency graphs. To evaluate the value of ordering, we look at compression in the naive algorithm. Run on our data set, the simple algorithm uses ADD commands to replace 52% of the COPY commands. Each ADD sends k extra bytes over the network link. This naive algorithm cuts compression in half when compared with traditional rsync. Our final algorithm's complexity proves necessary to avoid a drastic loss in compression.

3.3 Encoding Dependencies

Our final implementation of ip-rsync uses a graph-based algorithm that detects and resolves dependencies, preventing corruption of the target file. This comes at the cost of a delay to generate and process the conflicts, losing some of the benefit of overlapping network transfer with checksum generation.

In ip-rsync, the generator (steps 1 and 2) mirrors the corresponding actions of traditional rsync. The process which builds the hash table in step 3 also remains unmodified. However, during the scan in step 4, ip-rsync's sender buffers ADD and COPY commands in memory, instead of sending them immediately. Buffering all commands allows ip-rsync to detect dependencies and reorder commands prior to sending data.

Buffering COPY and ADD commands consumes much less space than traditional rsync requires. For a COPY command, ip-rsync needs space to store the source block and the target offset only. ADD commands require an extra field for the length of the data. The algorithm does not store the raw data for ADD commands in memory. Rather, ip-rsync stores the meta-data in memory and reads data directly from the source file when sending an ADD command.

The algorithm constructs a directed graph in which edges represent ordering constraints among commands (Figure 3). A topological sort of the graph determines an execution order for processing COPY commands on the target. When a total topological ordering proves impossible because of cycles in the graph, ip-rsync converts nodes that copy data in the file to commands that add the data explicitly. This results in lost compression. We later describe several heuristics for breaking cycles.

The source puts buffered nodes into a structure that points to the COPY command and contains fields necessary for dependency detection, topological sorting, transmission to the receiver, and deallocation. The fields of the graph node are a pointer to the COPY command that the node represents, a "visited" field used to encode

Algorithm 3.1: DFS($graph$)

```
procedure VISIT($node$)
  if not $node.VISITED$
    then { $node.VISITED \leftarrow$ true
           for each $edge \leftarrow$ EDGES($node$)
             do VISIT(TARGET($edge$))

main
  for each $node \leftarrow$ NODES($graph$)
    do VISIT($node$)
```

Figure 4: Pseudo-code for depth-first search. Initially nodes are in the *UNVISITED* state.

Algorithm 3.2: MODIFIED-DFS($graph$)

```
procedure VISIT($node$)
  if $node.STACK$
    then { $node.DELETED \leftarrow$ true
           DELETE($node$)
  if not $node.VISITED$
    then { $node.STACK \leftarrow$ true
           for each $edge \leftarrow$ EDGES($node$)
             do VISIT(TARGET($edge$))
           $node.VISITED \leftarrow$ true

main
  for each $node \leftarrow$ NODES($graph$)
    do VISIT($node$)
```

Figure 5: Modified depth-first search.

states during topological sort, a reference counter for deallocation, a pointer to the next node in topological order (initially NULL), and finally a pointer to the first in a linked list of outgoing edges.

The sender also buffers ADD commands. ADD commands are self-describing and, therefore, require no data from the old version. As a result, ADD commands need no reordering.

After all commands have been buffered, the COPY graph is passed to a dependency-detection function (step 5). This function detects all dependencies with an $O(n \lg n)$ algorithm, where n is equal to the number of bytes in the larger of the source file or the target file [2]. Each generated edge has two fields: the target node, and a pointer to the next outgoing edge from the source. With the full graph constructed, the sender topologically sorts the nodes using a modified depth first search (DFS) algorithm.

The algorithm used to topologically sort the graph in step 6 modifies DFS to make it operate on graphs that contain cycles. We present pseudo-code for depth-first search (Figure 4) and the modified depth-first search used in our algorithm (Figure 5). DFS (for acyclic graphs) is a "stack" algorithm, pushing nodes onto a stack as they are visited. In DFS, nodes take on two states – *UNVISITED* and *VISITED*. A node remains *UNVISITED* until the algorithm pops it off the *STACK* during its traversal. The algorithm marks the completed node *VISITED*. Modified-DFS for topological sort uses an additional *STACK* state to record the order in which the algorithm visits each node. The *STACK* state helps detect and break cycles. Modified-DFS also adds a *DELETED* state to encode nodes that have been deleted. Just as in DFS, each node begins *UNVISITED*. Analogous to DFS, the algorithm calls *Visit* on each node. *Visiting* a node places it on the stack and marks it in the *STACK* state. Subsequently, the algorithm *Visits* any neighbors of the node that are not *VISITED* or *DELETED*. When done with the neighbors, the algorithm removes the node from the stack, marks the node *VISITED* and places it on the front of an output list. If the *Visit* procedure finds a neighbor already marked *STACK*, a cycle exists in the graph. The algorithm marks this node *DELETED*, which breaks the cycle. A compensating ADD command is created in place of the *DELETED* COPY. The topological sort algorithm is $O(V + E)$ as it examines each vertex at least once and traverses every edge. We will examine alternative metrics and procedures for resolving cycles in the next section.

Ip-rsync sends the COPY commands to the target in topologically sorted order. Upon sending a COPY command, it deallocates the corresponding node along with its remaining edges. After sending all COPY commands, ip-rsync sends ADD commands according to the ADD list, including the ADD commands that correspond to deleted COPY commands (step 7). If necessary, the target truncates the file to the new size and the synchronization is complete.

3.4 Cycle Breaking

Any cycles found in the dependency graph represent a set of COPY commands that mutually depend on each others' completion. The only ways to find a valid ordering of the COPY commands in a cycle are to remove an edge of the cycle or remove a node (COPY command) entirely.

The cycle breaking method described in the previous section deletes an entire node to resolve a cyclic dependency. To compensate, the sender sends an ADD command with the data that should have been copied by the original command. This deletion costs k bytes of compression. The sender marks the node as deleted and the process continues. In-place delta compression [2] uses another metric that finds and deletes the smallest COPY command in a cycle in order to minimize compression loss. Ip-rsync need not distinguish between nodes based on size. Rsync has a fixed block size; all COPY commands have the same length.

We implement an alternative method for breaking cycles. This "trimming" method removes an edge of the cycle, rather than a node. An edge occurs when the read region of one COPY overlaps the write region of another COPY. The edge is eliminated by shrinking one COPY command, so that it no longer overlaps with the other COPY. An ADD command is generated that covers the trimmed region. We call this trimming a node, because it reduces the read and write regions described by the node. Trimming preserves some of the benefit of existing COPY commands when breaking cycles. When ip-rsync with trimming detects a cycle, it scans through all nodes in that cycle. The scan examines the overlap between each pair of nodes. It trims the dependency with the least overlap. The goal of this policy is to minimize compression loss. After trimming a node, ip-rsync checks existing edges pointing to the trimmed node to ensure that dependencies have not changed. It is possible that the trimmed node no longer conflicts (overlaps) with other nodes in its edge list.

To preserve the format and encoding of COPY commands, the algorithm does not change a COPY command when trimming the corresponding graph node. Instead, the algorithm allows the COPY command to write corrupt data and repairs the corrupt bytes with an ADD command. This preserves the fixed-size block used by rsync. During the transmission phase, no changes are made to the protocol and the target copies the full block for a trimmed node, which includes corrupted bytes in the overlapping region. The ADD command generated when trimming the node repairs the corrupted data.

4 Performance

Our experimental results compare ip-rsync to traditional rsync and evaluate different policies for breaking cycles found in ip-rsync graphs. Results indicate that ip-rsync degrades performance in bandwidth constrained environments, which we expect since it always transmits more data over the network. However, when factors other than bandwidth limit performance, our tests show that in-place rsync outperforms traditional rsync. Although ip-rsync requires extra computation, it reduces disk writes and file-system block allocations.

Our experimental data set provides an example of files used on handhelds that need to be updated over wireless networks. The data set consists of 1523 pairs of versioned files obtained from `http://www.handhelds.org/`. The files include a variety of kernel binaries and compiled programs that are intended to be downloaded to handheld computers. The files in our dataset target the same audience and devices that ip-rsync benefits the most. To collect data, we downloaded the software archive and ran scripts that search the archive for multiple versions of the same files. The original and processed data are available from the Hopkins Storage Systems lab at `http://hssl.cs.jhu.edu/ipdata/`.

In our experiments, we synchronize each pair of versions with rsync and in-place rsync. For in-place rsync, we compare two different cycle breaking methods: "delete node" that deletes the final node found in a cycle and "trim node" that trims the least possible number of bytes in each detected cycle.

Ip-rsync incurs some compression loss from encoding overhead. In-place rsync adds four bytes to each twelve byte codeword in order to encode offsets in the target file. These four bytes have a very different effect on compression and on bandwidth overhead. We illustrate this point with a simple worst-case example in which all commands are COPY commands. This results in negligible compression loss: 4 bytes degrade compression by 0.55% in the default 700 byte block. However, the 4 bytes increase the bandwidth by 33% – 16 bytes per COPY codeword as opposed to 12 bytes. The overall bandwidth overhead of in-place rsync in our experiments is less than 5%, much less than the 33% worst case bound. Data transferred in ADD commands dominates bandwidth, which mitigates overhead from codewords.

Overall, ip-rsync achieves compression almost identical to rsync. In addition to encoding overheads, ip-rsync loses compression when eliminating cyclic dependencies. With the delete-node policy, the compression lost by ip-rsync compared to rsync was 0.543%. The trim-node policy cost 0.545% in compression. The difference is negligible. The overall increase in transmitted data averages 0.544% of the size of the original file.

In-place rsync pays for its decreased parallelism and compression with longer transfer times at low bandwidths. For low bandwidths, the in-place algorithm spends 10% more time in the scanning and command transmission steps combined than the original rsync algorithm requires to complete the parallel hash search and command transmission phase. The extra time spent in these steps is fundamental to algorithms that reorder commands and is therefore unavoidable.

The performance of ip-rsync scales almost identically to that of rsync (Figure 6). Only at the smallest bandwidths can the effect of transferring the offset with each command be seen. Otherwise, ip-rsync has negligible latency and bandwidth overhead.

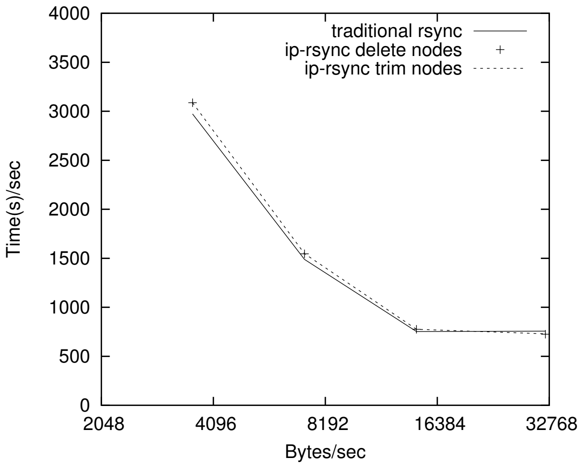

Figure 6: The graph shows transfer times of the entire dataset at different bandwidths using rsync and ip-rsync with two cycle breaking heuristics. Ip-rsync performs more slowly for bandwidth-limited transfers. Rsync incurs extra overhead in the *ext2* file system when allocating space in the temporary file and, as a result, ip-rsync outperforms rsync at higher bandwidths. The cycle breaking strategies make no noticeable difference in transfer time.

Although ip-rsync loses parallelism within a single file, it preserves parallelism across many files. Ip-rsync (and rsync) overlap the transmission of one file with scanning in a subsequent file. Our results show that overlapped execution is the dominant form of parallelism. In one instance of a bandwidth-limited transfer, the entire ip-rsync synchronization took 300 seconds and rsync required 290 seconds. The ip-rsync algorithm transmitted an extra 312,304 bytes, which accounts for the increase in end-to-end time. The standard deviation of the compression loss per file (the average compression loss of is 0.544%) was only 0.8%.

The overhead that arises from decreased parallelism is negligible. Comparing rsync and ip-rsync on a single file would indicate a greater performance loss. However, parallelism lost within a single file is recovered when overlapping multiple files.

While developing ip-rsync, we expected that rsync would outperform ip-rsync in all cases. We believed this because the changes to the algorithm include no improvements in bandwidth, memory usage, or processing. Also, rsync allows for more parallelism between the source and target.

When I/O limits performance (as opposed to bandwidth), ip-rsync outperforms rsync by 2-3% (see Figure 6 at bandwidths greater than 20,000). We account for the decrease in speed by considering the areas where ip-rsync reduces overhead. Ip-rsync performs fewer file-system block allocations using FFS-like file systems [11] that update data in-place. Our tests used the *ext2* file system. However, a copy-on-write file system [5, 14] would negate these benefits, as every write allocates a new file system block. In our further discussion, we refer to our trials on *ext2*. By eliminating the need for temporary space, ip-rsync allocates space only as required to increase the file size. In contrast, rsync allocates all blocks in the temporary file, then deallocates the original blocks. The time required to allocate space for a temporary copy for a traditional rsync balances out the decreased parallelism and compression. Furthermore, ip-rsync requires fewer disk writes when files are changed slightly. Ip-rsync can ignore COPY commands which have the same source and destination. Traditional rsync must copy all blocks regardless of whether they remain at the same offset in the file.

Buffering ADD and COPY commands for in-place rsync requires extra memory, but utilizes far less space than that required by rsync. The amount of extra memory scales with file size, but is much smaller than that of a temporary copy. On average, the extra memory needed is only 3.1% of file size, with a standard deviation of 2.9%. The method for cycle breaking by trimming nodes required 3.2% of the file size in data memory, while the delete node algorithm required only 3.0%. The buffering of in-place rsync can be problematic in resource-limited environments. Although we attempt to minimize the space required, it is possible that the algorithm can exceed available memory. We plan to implement the windowing techniques described in section 6.1 to address this problem.

5 Implementation

Adding in-place reconstruction to rsync changes the operation and the usage of the rsync utility in minor ways. In-place reconstruction affects interfaces, error handling, and the information and statistics that rsync generates.

Because ip-rsync updates a single copy of the data on the target, it changes the operational semantics of the tool, particularly when errors occur. Modifications to the file at the target cannot be isolated from a process reading data concurrently. New failure and recovery scenarios occur; incomplete synchronizations leave a partially updated file on the target that cannot be recovered to either the old or the new version.

5.1 Error Handling

Ip-rsync loses the atomic update property of rsync. Rsync creates a temporary file that contains the updated version. When the update completes, rsync calls `rename()` which unlinks and atomically replaces the existing version of the file. Processes with an open handle to the old file continue to read the old data until they reopen the file. New processes open the new version of the file. When operating on a single version, ip-rsync makes many intermediate changes to a file. The old version of the data is irrecoverably modified. Even if ip-rsync completes successfully, inconsistent views of data may occur during its operation; applications reading the file concurrently with an update by ip-rsync may read from both the old and new version.

To avoid these inconsistent views, ip-rsync opens files exclusively (rsync opens files in shared mode by default). If another process/application has the file open, ip-rsync fails, leaving the original file intact. If another process attempts to open the file during an ip-rsync session, the process either fails to open the file or blocks awaiting ip-rsync's completion. The outcome depends on the arguments to `open()` and operating system semantics.

If a failure occurs during synchronization, ip-rsync may leave the target file in an inconsistent state. This occurs when the network, receiver process or sender process fails after the receiver has written data. If the receiver process fails, no recovery action can be taken. Rsync leaves a temporary file in the file system and ip-rsync leaves an incompletely synchronized file. The inconsistent file left by ip-rsync does not present a problem upon restart; the source contains the new version of the data which is synchronized in a new ip-rsync session against the inconsistent data. If the sender or network fails, the receiver process continues to run and may take recovery action. Rsync merely removes the temporary file, preserving the state of the file prior to synchronization. Ip-rsync cannot recover the original state.

We balance several factors when deciding how ip-rsync should handle inconsistent files. Options include: (1) deleting the file at the receiver and (2) leaving the inconsistent file in the file system. The first approach prevents the application from reading inconsistent data, but discards a file that contains data that makes a subsequent rsync complete quickly. The second approach has the opposite properties, allowing inconsistent data to be read, but preserving the file for faster synchronization. We realize both benefits by having ip-rsync rename the corrupt file, creating a hidden recovery file that contains the inconsistent data. The original file is effectively deleted so that applications cannot access inconsistent data under the old file name. However, the recovery file is available to be used in a subsequent rsync session. When ip-rsync uses the recovery file, it updates the recovery file in-place and then renames the hidden file to the original file name. Renaming the recovery file avoids producing two copies of the file, which might exceed to storage capacity of the target. The recovery file is not implemented in our current release.

5.2 Usage

Ip-rsync is available at the Hopkins Storage Systems Lab Web site and can be downloaded and compiled in a manner identical to rsync (`http://hssl.cs.jhu.edu/iprsync/`). Our modifications introduce no additional dependencies to the build process of ip-rsync.

Making use of ip-rsync should be pose no problems for anyone familiar with the use of rsync. Ip-rsync accepts all command-line arguments of rsync with a few additions. To enable in-place reconstruction you must specify `-i` on the command line. This directs rsync to reconstruct the file in-place rather than using temporary space. The `--stats` option now displays statistics relevant to ip-rsync. The statistics include memory overhead, bandwidth overhead, and compression loss due to in-place reconstruction.

To synchronize a file using in-place reconstruction across the network, a user invokes rsync with the source file and the destination file: `rsync -i source.txt host.example.com:/path/to/dest.txt`. A few messages will appear noting the progress of the synchronization and indicating its successful completion.

Ip-rsync's in-place reconstruction is not compatible with previous versions of rsync. Thus, both hosts involved in the transfer must support in-place reconstruction. Ip-rsync maintains backward compatibility and will synchronize with a peer running rsync.

6 Future Work

Ip-rsync requires more detailed experiments in order to quantify tradeoffs between compression and parallelism and to identify further opportunities for optimization. Rsync itself is highly optimized for parallel execution within a single file and between multiple files [19]. Rsync is widely used because it works so well. We intend to follow this example. Although we have identified unfinished work items (in windowing and error recovery), the most important future work will feed experimental results back into the design of ip-rsync.

Even though we designed ip-rsync for mobile and wireless devices, our experiments reveal that ip-rsync works well in higher-bandwidth networks as well. They show that for bandwidths greater than 20,000 KB/sec, ip-rsync actually outperforms rsync. This result implies that ip-rsync would perform well for synchronizing block devices. In-place reconstruction is necessary because block devices are large (much larger than memory) and there is not spare capacity to write a temporary copy of a whole block device. Experiments need to be conducted to validate these claims.

Reduced concurrency within a single file degrades performance in ip-rsync. In our current experiments, much of this loss is regained through the semi-parallel transfer of multiple files; sending the data of a file while scanning for matching data in another file. However, rsync is commonly used to synchronize single files. More detailed experiments that examine performance when transferring single files will isolate performance losses from reduced parallelism. and indicate ip-rsync's suitability for single file transfer. Such experiments are essential to understand the benefit of reduced I/O in ip-rsync. When combined with semi-parallel execution, the gains of reducing I/O can make ip-rsync outperform rsync. Experiments on single files would allow us to measure the relative value of reduced I/O when compared with semi-parallel transfer.

6.1 Windowing

Ip-rsync runs out of space when the size of the buffered commands exceed the available space. Although buffered commands are much smaller than the original data, the algorithm must gracefully handle cases in which the buffered commands exhaust the available storage. This problem arises frequently when synchronizing block storage devices that may be orders of magnitude larger than a system's memory.

There is no trivial way to address buffer overflow by pruning the buffered commands and the graph they induce, nor by completing some commands early. Ip-rsync must retain all commands until the encoding is complete. Otherwise, if the algorithm were to discard a command, then dependencies between the discarded and subsequent commands remain undetected. Undetected dependencies cause an incomplete ordering and corrupt data.

We have designed, but not implemented, an *overlapped windowing* technique to address buffer overflow that organizes a file into multiple regions each of which are processed independently. It is an on-line process in which variable sized independent windows are created as ip-rsync approaches its memory limit. The read regions of the windows may overlap, but the write regions are disjoint.

During synchronization, the ip-rsync sender encodes data in an active window, which it uses until it exhausts memory. The active window has a start offset and goes to the highest byte in the file. When ip-rsync does not reach the memory limit, synchronization completes in a single window. When the memory limit is reached, the sender stops encoding data and completes processing on the buffered commands, *i.e.*, commands are topologically sorted and sent to the receiver where they are applied. The buffered commands are discarded and processing begins in the next window, starting at the first un-encoded byte.

Ip-rsync defines independent windows based on read/write dependency information. The algorithm encodes commands that read data in the active window only.

Windowing can be expressed equivalently as rewriting the dependency graph used by ip-rsync. When the graph becomes too large, the buffered graph is rewritten as a single node that writes data to the region starting at offset 0 and ending at the highest encoded offset. Furthermore, the algorithm cannot reorder this rewritten node with respect to the subsequent commands, because the command that the node represents has already been completed.

The implementation of windowing offers an opportunity to further parallelize ip-rsync. Our current windowing design (described above) minimizes compression loss by transferring as much of the file as possible in a single window; frequently this means the whole file when memory is not exhausted. We have identified two alternative windowing designs that increase parallelism in exchange for compression. One design partitions the files into fixed-size windows before scanning. Then, ip-rsync operates on each window independently, treating each window as a separate pair of files and overlapping transfer between windows. This approach does degrade compression, because blocks that match in different windows cannot be encoded. Another design for large files implements two active windows that are sized dynamically. One at the front of the file and one at the end of the file. This allows two processes to execute concurrently on the same file. When the algorithm exhausts memory, one window (or both windows) are completed to reclaim space. The algorithm completes when the windows collide in the middle of the file. An experimental comparison of different windowing policies will help us understand compression versus parallelism tradeoffs in large files.

Overlapped windowing is somewhat similar to the windowing technique used in delta compression [8], in which files are partitioned into non-overlapping regions. Delta compression defines codeword formats for windows, but does not specify how they are constructed or evaluated. Overlapping windowing differs from the rolling window techniques commonly used in data compression. The "Lempel-Ziv" family of algorithms uses a fixed-sized buffer, discarding the oldest strings, in order to bound the amount of space used by the string library. Rolling windows create a compression versus memory tradeoff — there are no correctness implications and the file is not "partitioned".

7 Conclusions

We have described the design, implementation, and performance of in-place rsync for synchronizing files among mobile and wireless devices. Ip-rsync is a modification to the open-source rsync utility so that files may be updated in-place: in the memory or storage that the current version occupies. The algorithms of ip-rsync use a graphical representation of the operations in an rsync encoding to detect when in-place updates would corrupt data, and then topologically sorts these operations to avoid such conflicts. When compared with rsync, ip-rsync loses 0.5% compression from encoding overheads and breaking cyclic dependencies. Ip-rsync increases transfer time in bandwidth-constrained environments, but can increase performance in I/O constrained environments by avoiding the creation of a temporary file.

Although in-place reconstruction can degrade both compression and transfer time, it makes file synchronization available in space-constrained environments where rsync alone does not function. The benefits of file synchronization can be brought to mobile and wireless devices in exchange for minor performance losses.

References

[1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. *Journal of the ACM*, 49(3), 2002.

[2] R. Burns and D. D. E. Long. In-place reconstruction of delta compressed files. In *Proceedings of the Symposium on Principles of Distributed Computing*, 1998.

[3] R. Burns, L. Stockmeyer, and D. D. E. Long. In-place reconstruction of version differences. *IEEE Transactions on Knowledge and Data Engineering (to appear)*, 2003.

[4] M. C. Chan and T. Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of the IEEE INFOCOM Conference*, 1999.

[5] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter Conference*, 1994.

[6] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Proceedings of the 6th Workshop on Software Configuration Management*, March 1996.

[7] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[8] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[9] J. Langford. Multiround Rsync. Technical Report Available at www.cs.cmu.edu/∼jcl/research/mrsync/-mrsync.ps, Dept. of Computer Science, Carnegie-Mellon University, 2001.

[10] J. MacDonald. Versioned file archiving, compression, and distribution. Technical Report Available at http://-www.cs.berkeley.edu/∼jmacd/, UC Berkeley, 2000.

[11] M. K. McKusick, W. N. Joy, J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3), 1984.

[12] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the Symposium on Operating Systems Principles*, 2001.

[13] C. Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1991.

[14] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[15] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[16] Z. H. Stephen, S. M. Blackburn, L. Kirby, and J. Zigman. Platypus: Design and implementation of a flexible high performance object store. In *Proceedings of the 9th International Workshop on Persistent Object Systems*, 2000.

[17] W. F. Tichy. The string-to-string correction problem with block move. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[18] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[19] A. Tridgell and P. Mackeras. The Rsync algorithm. Technical Report Available at http://samba.anu.edu.au-/rsync/tech_report/tech_report.html, Australian National University, 1998.