

Performance Update

Performance-related product changes since the 2003 Perforce user conference

Michael Shields
Richard E. Baum
March, 2005

Abstract

Perforce takes seriously the “Fast” in its “Fast Software Configuration Management System” tag line. This paper discusses performance features and fixes introduced since the 2003 Perforce User Conference. Also discussed are a few tips and tricks that may help improve the overall performance of your Perforce installation.

1 Introduction

There are a great many factors that effect how well a Perforce installation functions. These include hardware configuration, software configuration, operating system tuning, and software functionality. While these all play a role in performance, for purposes of this discussion, we will concentrate on new functionality within Perforce. Since the 2003 user conference, we have introduced a number of new features to assist you in getting the most out of your installation. While some of these require that you upgrade to a newer release, the relatively minor cost of doing so is rewarded with faster throughput, better performance, and happier users. The topics covered herein will generally be discussed in the order of the release in which they first appeared.

2 2003.2 Performance enhancements

2.1 Narrow sync optimization

The number of revisions considered by a Perforce command can be referred to as the “scope” of the command. The scope of commands can be narrowed by the client view, the protections table, more specific file arguments, and revision specifiers. In general, the more narrow the scope, the faster the command will run.

The revision specifier `@<changelist-number>, #head` is certainly not new. It specifies a range of revisions using the `@<changelist-number>` and `#<revision>` specifiers that have long been part of Perforce. The revision specifier `@<changelist-number>, #head` specifies the intersection of all revisions submitted beginning with changelist `<changelist-number>` and all revisions ending at the head revision. This simplifies to all revisions submitted beginning with changelist `<changelist-number>`.

In the 2003.1 release, using the @<changelist-number>, #head revision specifier could significantly speed the computation of revisions needing to be synced to a client. For one example 2003.1 configuration, the amount of time required to compute that two revisions need to be synced in a client that already has 730,075 revisions is approximately seven and a half seconds without the @<changelist-number>, #head revision specifier, and approximately two and a half seconds with the revision specifier:

```
$ time p4 sync -n
//depot/main/file2#1 - added as /p4clients/r03.1/depot/main/file2
//depot/main/file3#1 - added as /p4clients/r03.1/depot/main/file3

real    0m7.687s

$ time p4 sync -n @1880,#head
//depot/main/file2#1 - added as /p4clients/r03.1/depot/main/file2
//depot/main/file3#1 - added as /p4clients/r03.1/depot/main/file3

real    0m2.466s
```

In the 2003.2 release, the @<changelist-number>, #head revision specifier has been optimized to use the db.revex index, which is ordered descending by changelist number. Commands using this revision specifier now position at the beginning of the db.revex index and scan through changelist <changelist-number>. If the changelist specified is close to the last changelist number in the repository, the performance improvement can be substantial. Using the same dataset and configuration as the 2003.1 configuration above, the same sync completes nearly instantaneously:

```
$ time p4 sync -n @1880,#head
//depot/main/file2#1 - added as /p4clients/r03.2/depot/main/file2
//depot/main/file3#1 - added as /p4clients/r03.2/depot/main/file3

real    0m0.005s
```

2.2 Label optimizations

The usefulness of an @<same-label>, @<same-label> revision specifier may not be immediately obvious. One of the practical uses for this revision specifier is syncing the revisions in a label without disturbing the remaining revisions already on the client. This behavior is far different than simply syncing to a label (that is, using an @<label> revision specifier), which will not only sync the revisions in the label, but will also delete from the client those revisions not in the label.

Specifying the files to be synced qualified by the @<same-label>, @<same-label> revision specifier where the files specified did not exist in the label resulted in less than optimal performance in the 2003.1 release. For one example 2003.1 configuration, the amount of time required to determine that the 7,300 files specified did not exist in a label containing 14,600 revisions when syncing a client that has 730,075 revisions is approximately 85 seconds:

```
$ time p4 sync -n //depot/main/10/...@label1,@label1
real    1m24.648s
```

In the 2003.2 release, the processing of the @<same-label>, @<same-label> revision specifier has been changed to better handle the case where the files specified do not exist in the label. Using the same dataset and configuration as the 2003.1 configuration above, the same sync completes in less than a second:

```
$ time p4 sync -n //depot/main/10/...@label1,@label1
real    0m0.041s
```

2.3 Locking and memory use

The Perforce server uses OS file locks on the db.* tables to synchronize between commands. Synchronization is required to maintain the integrity of the metadata stored in the db.* tables. Several changes were made in the 2003.2 release that affected the locking of the db.* tables. These changes reduce the locks taken and better utilize resources when locks are held. As a result of these changes, overall concurrency is improved, that is, more commands can run concurrently.

In the 2003.1 and earlier releases, read locks were taken on tables such as db.have and db.label for commands such as p4 filelog and p4 files in case an argument used either an @<client> or @<label> revision specifier. While these commands computed the revisions to be processed for each argument, the read locks held on tables such as db.have and db.label blocked commands that needed to write to these tables, such as a p4 sync that needed to update the client's have list, or a p4 labelsync that needed to update the revisions in a label. Of particular note is the p4 sync command, which in the 2003.1 and earlier releases, takes a read lock on the db.label table even when there is no @<label> revision specifier used in the arguments. With the read lock on the db.label table held for the duration of a sync's compute phase (which can be lengthy), labelsync commands needing to update the revisions in a label could not complete until the compute phase of all syncs had completed.

One of the changes in the 2003.2 release eliminates the read locks taken on tables such as db.have and db.label if they are not needed for either an @<client> or @<label> revision specifier used in an argument and the read locks are otherwise not needed for the command. If either an @<client> or @<label> revision specifier is used in an argument, a read lock will be taken only on the table as required to compute the revisions to be processed for that argument. If an @<client> revision specifier is used in an argument, a read lock will be taken on db.have to

compute the revisions to be processed for that argument. If an @<label> revision specifier is used in an argument, a read lock will be taken on db.label to compute the revisions to be processed for that argument. Only when an argument uses both an @<client> and an @<label> in its revision specifier are read locks taken on both the db.have and db.label tables to compute the revisions to be processed for that argument if these locks are otherwise not needed for the command.

So this change in the 2003.2 release allows commands that need to update tables such as db.have and db.label to complete without waiting for needless read locks to be released. Commands such as p4 filelog and p4 files without an @<client> or @<label> revision specifier in the arguments no longer block commands such as p4 sync and p4 labelsync. And p4 sync without an @<label> revision specifier in the arguments no longer blocks p4 labelsync commands.

A change to better utilize resources when locks are held was also made in the 2003.2 release. This change affects updating a client's have list during a sync. In all releases, clients acknowledge each revision as it is received (though due to buffering within Perforce and the network, it may appear as though revisions received by a client are acknowledged in groups). In 2003.1 and earlier releases, the server updates the client's have list for each acknowledgement that it receives from the client. So each acknowledgement incurs the overhead of taking a write lock on the db.have table, recording that the revision is on the client's have list by reading and modifying a small number of pages in the db.have table's cache, flushing the modified pages from the db.have table's cache, and releasing the write lock. This overhead for each acknowledgement is lock intensive and does not make the best use of the db.have table's cache. In the 2003.2 release, the server instead batches the acknowledgements received from the client. After receiving 100 acknowledgements (or the last acknowledgement), the server takes a write lock on the db.have table, records that the revisions are on the client's have list by reading and modifying a larger number of pages in the db.have table's cache, flushes the modified pages from the db.have table's cache, and releases the write lock. The advantages of the server updating the client's have list in batches is that fewer calls to acquire and release locks on the db.have table are required, and each page in the db.have table's cache can be modified several times before being flushed from the cache.

Additional changes were made in the 2003.2 release that will free resources earlier. These changes include releasing some db.* table locks and memory earlier than was done in prior releases.

2.4 Mapping optimizations

Client views and the protections table are combined to form a map table. As the complexity of the client view and the protections table increases, the combinations forming the map table also increases, perhaps significantly. In the 2003.1 release, the map table was searched linearly to find an applicable map for each revision. For very large map tables, the linear search could require significant CPU resources. The map table is typically searched during a command's compute phase, with read locks held on several important db.* tables. The CPU resources required for the linear search of a very large map table could result in an extended compute phase, and the read

locks held blocked other commands that needed write locks on the same db.* tables. The linear search was replaced with a search tree in the 2003.2 release. The search tree reduces the amount of CPU resources required to find an applicable map for each revision, resulting in a shorter compute phase with read locks held for a shorter duration. And since the read locks are not held as long, commands needing write locks on the same db.* tables acquire the write locks faster and complete.

An admittedly worst-case client view and protections table can be used to demonstrate the improvement due to the implementation of the search tree in the 2003.2 release. The client's view contains 50 entries:

```
//depot/main/00/... //large-client/depot/main/00/...
//depot/main/01/... //large-client/depot/main/01/...
//depot/main/02/... //large-client/depot/main/02/...
...
//depot/main/47/... //large-client/depot/main/47/...
//depot/main/48/... //large-client/depot/main/48/...
//depot/main/49/... //large-client/depot/main/49/...
```

The protections table contains 51 applicable entries:

```
write user excluded-user * //depot/main/...
write user excluded-user * -//depot/main/.../00/...
write user excluded-user * -//depot/main/.../01/...
write user excluded-user * -//depot/main/.../02/...
...
write user excluded-user * -//depot/main/.../47/...
write user excluded-user * -//depot/main/.../48/...
write user excluded-user * -//depot/main/.../49/...
```

In the 2003.1 release, the compute phase for a sync of 182,500 revisions from a server with 730,077 revisions takes approximately 98 seconds due to the linear search of the map table:

```
$ time p4 sync -n | wc
182500 912500 34820000
real    1m37.510s
```

In the 2003.2 release, using the same dataset and configuration as the 2003.1 configuration above, the compute phase for the sync is reduced to approximately 20 seconds due to the implementation of the search tree:

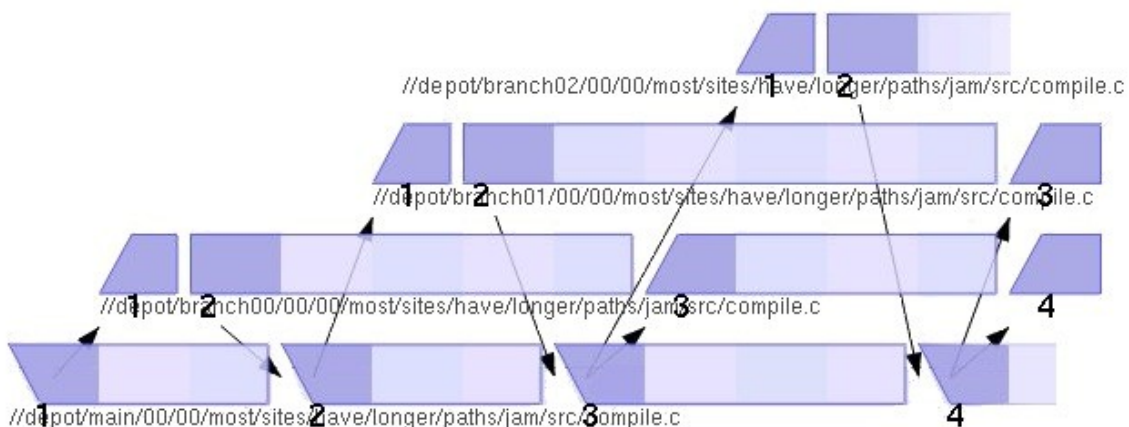
```
$ time p4 sync -n | wc
182500 912500 34820000
real    0m20.258s
```

2.5 Indirect integration faster

Indirect integration searches for relationships between files by following integration history. Using the integration history, it is possible to determine if the files are distantly related through intermediate branches. If the files are found to be related, the server can pick a base revision that may simplify merging. Otherwise, the first revision on the source branch is used as the base, which can make merging difficult.

The indirect integration feature was introduced in the 2002.2 release. The algorithm used in the initial implementation considered more integration history than necessary when searching for distant relationships. This could have resulted in excessive reads from the db.integed table during an indirect integration's compute phase. The excessive reads could have extended the indirect integration's compute phase, and the read locks held could have blocked other commands. In the 2003.2 release, the indirect integration algorithm was reworked, which can result in less integration history considered. Fewer reads from the db.integed table can result, reducing the indirect integration's compute phase and blocking other commands less often.

The following integration history can be used to demonstrate the improvement due to the algorithm in the 2003.2 release:



In the 2003.1 release, the compute phase for an indirect integration from `branch02` to `branch00` of 7,300 files with the same integration history as shown above takes approximately 26 seconds to determine that no integrations are necessary due to the algorithm used in the initial implementation:

```
$ time p4 integ -I -n //depot/branch02/00/... //depot/branch00/00/...
//depot/branch02/00/... - all revision(s) already integrated.

real    0m25.587s
```

In the 2003.2 release, using the same dataset and configuration as the 2003.1 configuration above, the compute phase for the indirect integration is reduced to approximately three seconds

due to the rework of the algorithm:

```
$ time p4 integ -I -n //depot/branch02/00/... //depot/branch00/00/...  
//depot/branch02/00/... - all revision(s) already integrated.  
  
real    0m2.706s
```

2.6 MD5 Checksums computed and stored at submit

Starting with release 2003.2, Perforce client programs will compute MD5 digest values for file revisions as they are transmitted a 2003.2 or later the server. This eliminates the server's having to calculate this value after the transfer, and ensures that the value is in the depot from the time of file submission. There should be no negative performance impact from this, as the values are calculated as data is chunked from the client to the server, with no extra read of the data required. Files submitted by pre-2003.2 clients will cause the server to calculate the value after the file revisions are committed.

Prior releases required a run of `p4 verify -u` to cause these values to be calculated and stored within the depot. If not stored, MD5 hash values used by Perforce functions had to be calculated on-the-fly whenever calculations required. Functions that can be slowed by such on-the-fly calculations are `p4 diff`, `p4 diff2`, and `p4 revert -a`.

This feature also means that it is no longer necessary to periodically run a `p4 verify -u` to cause the server to store the MD5 hash values. Instead, a single pass after upgrading to 2003.2 will suffice.

3 2004.2 Performance enhancements

3.1 Enhanced triggers

In addition to the pre-submit triggers available in earlier releases, the 2004.2 release of Perforce adds both mid-submit and post-submit triggers as well as specification triggers that can be fired whenever a user generates or modifies a Perforce specification form. It is now also possible, with the mid-submit triggers, to access file the contents of files being submitted to the depot. There are a total of five new trigger types in this release.

Mid-submit triggers run after the file transfer from the client to the server is complete, but before the changelist is further processed by the server. They offer a set of commands that allow access to file revisions on the server prior to submission. These can be used to test an operation for certain conditions and to prevent submission of the specification if desired. The `@=` operator can be used with the `p4 diff2`, `p4 files`, `p4 fstat`, and `p4 print` commands to access file revision contents. This kind of trigger can also be used to stop a submit from completing.

Post-submit triggers are executed after the commit of a submit transaction and can not stop it from being submitted. They can be used to perform actions after a submit has taken place. One obvious use for these is as a replacement for the Perforce review daemon. Rather than having a

daemon running automatically every few minutes, a trigger could easily be set up to fulfill this function and run only when a submission to the depot actually takes place.

Specification triggers are fired when a user generates or modifies a Perforce specification form. They can run before or after form generation, and also after submission. They allow automation of forms, including the ability to both pre-populate form fields for users as well as to validate data prior to submission.

The triggers available in the 2004.2 release are as follows:

Trigger type	Trigger name	When it runs / notes
Pre-submit	Submit	After changelist creation but before file transfer. Has no access to file contents.
Mid-submit	Content	After changelist creation and after file transfer. Has access to file contents
Post-submit	Commit	After changelist creation, file transfer, and changelist commit.
Pre-spec	Out	Before form is presented to user. May modify form
Mid-spec	In	Before submitted form is parsed/validated by the server. May modify form
Post-spec	Save	After parse, but before storage in depot. May not modify form

Overall, the additional trigger types provide more configuration choices. Scripts set to run on a regular basis to check on the status of a branch, the number of users, etc. can now be set to run only when certain operations take place. This obviates the use of cron jobs for such functions as running the Perforce review daemon to send email to users when a changelist has been submitted. In addition, it is now possible to perform tasks that were impossible before. These include the pre-population of forms with custom data, and validation of forms prior to submission to the depot.

3.2 Uncompression offloaded to clients

Files that are stored as compressed binaries are now left in their compressed state when being transmitted to a client. They are then uncompressed on the client. Prior to the 2004.2 release, these files were uncompressed on the server and the uncompressed data stream was sent over the network to the client.

By default, Perforce client specifications do not have compression turned on. This results in the server uncompressing files that are stored in the versioned file tree prior to sending them over the network. The result is more data being sent over the network and a longer transmit time. This change removes the uncompression work from the server, and causes less data to be transmitted from the server to the client.

While the performance improvement resulting from these changes is not dramatic, it certainly can be seen. With a 2003.2 server the time to sync a sample 75mb file is as follows:

```
% time ./p4 sync //depot/big_file.zip#1
//depot/big_file.zip#1 - added as /p4/big_file.zip

real 0:12.02
```

In this example, the file was then synced to #none and the server and depot were immediately upgraded to 2004.2. The same operation then took slightly less time:

```
% time ./p4 sync //depot/big_file.zip#1
//depot/big_file.zip#1 - added as /p4/big_file.zip

real 0:11.68
```

The savings from this change are not immense, but they are there and they do not require anything other than an upgrade to a 2004.2 or later server.

3.3 Overlay mappings now supported

Overlay mappings are now supported in client views. Without overlay mappings, Perforce matches server files to the last line of the client view that matches the file. Overlay mappings allow one branch of files from the depot to overlay another branch that is already mapped on the client. These mappings maintain a one-to-one relationship between the client files and the server files and cause no confusion or conflict for the server. If an operation on a file does not match the overlay mapping, the regular mapping is used.

For example, look at this client view with mappings that do not represent a one-to-one relationship between client and server files:

```
//depot/joe/... //clientname/...
//depot/jan/... //clientname/...
```

These mappings overwrite each other and conflict. No files from `//depot/joe/...` will ever be synced to the workspace and this first line is effectively ignored.

If overlay mappings were used instead, files from both the `joe` branch and the `jan` branch could be synced to the client. An example of overlay mappings that could solve this problem is as follows:

```
//depot/joe/... //clientname/...
+//depot/jan/... //clientname/...
```

Files in `//depot/jan/...` are mapped, and take precedence over files with the same name in `//depot/joe/...`

With overlay mappings, if the files `//depot/jan/my_file.c` and `//depot/joe/my_file.c` both exist in the depot only `//depot/jan/my_file.c` will be synced to the client.

The ability to overlay mappings allows clearer client specification views. Prior to this, conflicting sets of mappings that were not one-to-one would at best cause syncs to silently fail to bring the proper set of files to the client. At worst they would cause a `Client map too twisted` error message to be displayed. Simplified mappings are easier to read for humans and they mean less data and faster processing for the Perforce server.

One additional note, overlay mappings are only functional in client specification views. They are prohibited in branch views and they have no effect in label or other views.

3.4 Indirect integration faster yet

Indirect integration is now the default integration method as of the 2004.2 release. Changes were made to the indirect integration algorithm so that the base revision selected for the merge does not have to be on the source branch. By allowing the base revision to be on any branch, the base selected should be a common ancestor close to both revisions being merged. The selection of a better base revision should result in a more intuitive merge.

The performance of indirect integration improved in 2004.2, even though the functionality of selecting a better base revision for the merge was added. Continuing with the same dataset and configuration used in section 2.5, we can compare the performance of the same indirect integration in the 2003.2 and 2004.2 releases. In the 2003.2 release, the compute phase for an indirect integration from `branch02` to `branch00` of 7,300 files with the same integration history as shown in section 2.5 takes approximately 2.7 seconds to determine that no integrations are necessary:

```
$ time p4 integ -I -n //depot/branch02/00/... //depot/branch00/00/...
//depot/branch02/00/... - all revision(s) already integrated.

real    0m2.706s
```

In the 2004.2 release, using the same dataset and configuration as the 2003.2 configuration above, the compute phase for the indirect integration is reduced to approximately 2.4 seconds:

```
$ time p4 integ -n //depot/branch02/00/... //depot/branch00/00/...
//depot/branch02/00/... - all revision(s) already integrated.

real    0m2.386s
```

4 2005.1 Performance enhancements

4.1 MapState redux

A client's have list contains all revisions that are currently on the client, expressed in both client-syntax (`//<client-name>/<path>/<file>`) and depot-syntax (`//depot/<path>/<file>`). The client's have list is keyed on the filename expressed in client-syntax. Oftentimes, commands are

provided with arguments in depot-syntax. If a command with a depot-syntax argument needs to retrieve revisions on the client's have list using depot-syntax, without some type of optimization, the client's entire have list must be scanned. For large clients, scanning the client's entire have list can have a significant impact not only for the user executing the command, but also for the users that are blocked by the read locks held while the client's entire have list is scanned.

MapState was introduced in the 2002.1 release as an optimization that reduces the need to scan a client's entire have list. Prior to the 2005.1 release, a client's MapState flag set (that is, a value of one) indicated that the client's have list was in alignment with the client's view. A client's have list was in alignment with the client's view when for all revisions on the client's have list, the filename expressed in depot-syntax could be translated using the client's view to the filename expressed in client-syntax on the client's have list. For example, if a client's have list and view were:

```
$ p4 have
//depot/main/file1#1 - /p4clients/r04.2/extra-directory/main/file1
//depot/main/file2#1 - /p4clients/r04.2/main/file2

$ p4 client -o
...
Root: /p4clients/r04.2
...
View: //depot/... //mapstate-client/...
```

The above client's have list is not in alignment with the client's view since `//depot/main/file1` does not translate using the client's view to `/p4clients/r04.2/extra-directory/main/file1` (which is the local-syntax equivalent of the client-syntax `//mapstate-client/extra-directory/main/file1`). `//depot/main/file1` translates using the client's view to `/p4clients/r04.2/main/file1` (`//mapstate-client/main/file1`), which is not what is on the client's have list.

Syncing the client to the revisions on its have list usually moved the revisions that were not in alignment with the client's view:

```
$ p4 sync \#have
//depot/main/file1#1
- deleted as /p4clients/r04.2/extra-directory/main/file1
//depot/main/file1#1
- added as /p4clients/r04.2/main/file1
```

The client's have list was then usually aligned with the client's view:

```
$ p4 have
//depot/main/file1#1 - /p4clients/r04.2/main/file1
//depot/main/file2#1 - /p4clients/r04.2/main/file2
```

In the 2004.2 and earlier releases, a client's MapState flag was set if and only if for all revisions on the client's have list, the filename expressed in depot-syntax could be translated using the

client's view to the filename expressed in client-syntax on the client's have list. With a client's MapState flag set, commands that needed to retrieve revisions on the client's have list using depot-syntax could simply translate the filename expressed in depot-syntax using the client's view, and use the resulting filename expressed in client-syntax to probe directly into the client's have list, which is keyed on the filename expressed in client-syntax. Most scans of a client's entire have list were eliminated if the client's MapState flag was set.

Prior to the 2005.1 release, keeping a client's MapState flag set could be a challenge. Changing a client's view would often require syncing the client, which could be forgotten or cumbersome for users at large sites. If a revision was open for edit and not in alignment with the client's view, the client's MapState flag could not be set until the revision was either submitted or reverted and then synced to the correctly aligned location. Or if list access was denied for a revision on a client's have list and not in alignment with the client's view, the client's MapState flag could not be set since without list access, the revision could not be synced to the correctly aligned location.

The MapState definition and implementation has changed in the 2005.1 release. Alignment between a client's have list and the client's view is no longer a consideration. Instead, the server internally maintains for each client an additional set of mappings that when appended to the mappings in the client's view (the internally maintained mappings for the client appended to the mappings in the client's view can be collectively referred to as the haveMap), translate filenames expressed in depot-syntax to the filenames expressed in client-syntax on the client's have list. The haveMap yields the same results as the earlier MapState implementation in that the resulting filenames expressed in client-syntax can be used by commands working with depot-syntax to retrieve revisions by probing directly into the client's have list, eliminating most scans of a client's entire have list.

The MapState flag set in a client now indicates that the internally maintained mappings for the client have been verified and reconstructed as necessary following a modification of the client's specification. The modification of a client's specification clears the client's MapState flag. The flag is then set immediately after the next scan of the client's entire have list, during which the internally maintained mappings for the client are verified and reconstructed as necessary. The next scan of the client's entire have list following a modification of the client's specification could occur executing a simple command such as `p4 have`, or executing a command working with depot-syntax that must scan the client's entire have list to retrieve revisions since the internally maintained mappings for the client have not yet been verified and reconstructed as necessary. With the MapState flag set, the haveMap can be used by commands working with depot-syntax to translate filenames expressed in depot-syntax to the filenames expressed in client-syntax on the client's have list, which can then be used to retrieve revisions by probing directly into the client's have list.

The internally maintained mappings for the client are a minimal set of mappings that as part of the haveMap are only used for translating filenames expressed in depot-syntax to the filenames expressed in client-syntax on the client's have list. The internally maintained mappings are not used to determine which files can be synced to the client. The files that can be synced are specified by the mappings in the client's view and restricted by the protections table; this functionality remains unchanged in the 2005.1 release. The internally maintained mappings can

only be viewed in a checkpoint or journal.

As an example of the internally maintained mappings for a client, if the client's view and have list are:

```
$ p4 client -o
...
Root:  /p4clients/r05.1
...
View:
      //depot/... //mapstate-client/...

$ p4 have
//depot/main/file1#1 - /p4clients/r05.1/main/file1
//depot/main/file2#1 - /p4clients/r05.1/main/file2
//depot/main/file3#1 - /p4clients/r05.1/main/file3
```

The haveMap for the above example consists of the one mapping in the client view and no internally maintained mappings for the client. The entire haveMap is:

```
//depot/... //mapstate-client/...
```

This haveMap with the single entry can translate filenames expressed in depot-syntax to the filenames expressed in client-syntax for all revisions on the client's have list.

Changing the client's view and syncing //depot/main/file1, resulting in the client's have list:

```
$ p4 client -o
...
Root:  /p4clients/r05.1
...
View:
      //depot/... //mapstate-client/extra-directory/...

$ p4 sync //depot/main/file1
//depot/main/file1#1
  - deleted as /p4clients/r05.1/main/file1
//depot/main/file1#1
  - added as /p4clients/r05.1/extra-directory/main/file1

$ p4 have
//depot/main/file1#1 - /p4clients/r05.1/extra-directory/main/file1
//depot/main/file2#1 - /p4clients/r05.1/main/file2
//depot/main/file3#1 - /p4clients/r05.1/main/file3
```

The client's haveMap now consists of two mappings, the first of which is the one mapping in the client view and the second is an internally maintained mapping:

```
//depot/... //mapstate-client/extra-directory/...
//depot/... //mapstate-client/...
```

The first mapping in the haveMap translates `//depot/main/file1` to `/p4clients/r05.1/extra-directory/main/file1` (which is the local-syntax equivalent of the client-syntax `//mapstate-client/extra-directory/main/file1`). The second mapping in the haveMap translates both `//depot/main/file2` and `//depot/main/file3` to `/p4clients/r05.1/main/file2` (`//mapstate-client/main/file2`) and `/p4clients/r05.1/main/file3` (`//mapstate-client/main/file3`), respectively.

4.2 obliterate faster

Obliterating revisions is time-consuming and potentially resource-consuming for most sites in releases prior to the 2005.1 release. Significant time was required since large tables such as `db.have`, `db.label`, and `db.integed` were scanned to find records to obliterate. These tables had to be scanned since they were not keyed to efficiently find the records to obliterate. The revisions to obliterate are processed using filenames expressed in depot syntax, but `db.have`'s leading key is the filename expressed in client-syntax, `db.label`'s leading key is the label name, and `db.integed` is keyed such that only one-half of the records to obliterate (either a forward or reverse integration record) can be efficiently found. The tables are keyed to favor production SCM activities, not the obliterate command.

Significant disk space resources could be consumed by an obliterate in releases prior to the 2005.1 release. If a revision being obliterated was the source of a lazy copy for many revisions, the obliterate would materialize the revision once for each revision that was the target of a lazy copy from the revision being obliterated. For example, if revision five of `file1` on `branch1` was branched to `file1` on `branch2`, `branch3`, and `branch4`, obliterating `file1` on `branch1` would add revision five's content to the archive files for `file1` on `branch2`, `branch3`, and `branch4`. If `file1` was a text file on `branch2`, `branch3`, and `branch4`, the content added to the archive files could be minimal differences between `file1`'s revision five on `branch1` and `file1`'s revision two on each of `branch2`, `branch3`, and `branch4`. But if `file1`'s revision five on `branch1` was a large binary file, obliterating `file1` on `branch1` could copy revision five's content to revision one on each of `branch2`, `branch3`, and `branch4`.

Obliterating revisions in the 2005.1 release can be far less time-consuming and removes the potential for consuming disk space resources. The algorithm used by the obliterate command was changed so that it is able to derive the key values necessary to probe directly into the large tables, saving significant time as compared to scanning the tables. The algorithm now uses each client's MapState flag (discussed in section 4.1) to verify that it can probe directly into the client's have list to efficiently find the records to obliterate. If a client's MapState flag is not set, then the client's entire have list must be scanned to find the records to obliterate. Since scanning a client's entire have list will slow the obliterate, it will be helpful to set the MapState flag in all clients prior to starting the obliterate command. The MapState flag can be set in all clients by executing:

```
$ p4 clients \  
| sed 's/^Client \([^ ]*\).*$/p4 -c \1 fstat \\//depot//file/' \  
| /bin/sh > /dev/null 2>&1
```

Disk space resources are no longer consumed by obliterating the source of a lazy copy for many

revisions in the 2005.1 release. This intelligent handling of lazy copies is now possible due to the introduction of the new db.archive table. The db.archive table tracks the revisions that use a revision in the archive files as the source for their lazy copies. (The size of the db.archive table is dependent upon the site's usage of lazy copies; an estimate is 30% of the size of the db.rev table.) Now when a revision is obliterated from the archive files, the revisions using the obliterated revision as the source for their lazy copies can be efficiently found, one of which will be made the new source of the lazy copies and the remaining revisions redirected to the new source of the lazy copies.

4.3 Delete triggers

The 2005.1 release adds a new trigger type to Perforce. A new specification trigger of type, `delete`, runs prior to deletion of any specification. It can be used to limit when a specification is deleted, or to log information about the deletion. This brings the total number of Perforce triggers available to seven, as follows:

Trigger type	Trigger name	When it runs / notes
Pre-submit	Submit	After changelist creation but before file transfer. Has no access to file contents.
Mid-submit	Content	After changelist creation and after file transfer. Has access to file contents
Post-submit	Commit	After changelist creation, file transfer, and changelist commit.
Pre-spec	Out	Before form is presented to user. May modify form
Mid-spec	In	Before submitted form is parsed/validated by the server. May modify form
Post-spec	Save	After parse, but before storage in depot. May not modify form
Delete-spec	Delete	Prior to deletion of a specification

One of the most promising uses for this is to prevent unauthorized users from deleting job specifications from the depot.

4.4 File size stored

In order to display the sizes of depot file revisions to users within Perforce clients, the server needs obtain the file size. Since Perforce stores files on in the versioned file tree in as compact a manner as possible (by default, RCS reverse deltas for text files and compressed images for binary files) the file size information is not readily available from the server filesystem. Prior to the 2005.1 release, the server needed to fully render each file revision and count the number of bytes that it contained in order to calculate and pass the size value to a client. This proved to be an expensive way to generate a small amount of data, particularly from an I/O and throughput standpoint.

Starting with the 2005.1 release, Perforce calculates and stores the file size for a revision at the same time that the MD5 hash value for that revision is computed and stored. It is updated at the

same times, too. Thus, it can be easily reported to clients without any server-side calculations. With older server releases, the server calculates the file size on-the-fly whenever an `fstat -O1` (old syntax: `p4 fstat -l`) command is issued.

With older clients and a 2005.1 server, the value is calculated on the server at submit time, but it is stored within the server's metadata. This is true for any client version when `text+k` typed files are used. RCS keywords calculations take place on the server and need to be factored into the file size and MD5 hash value, so the values can not be calculated on the client under these circumstances.

In order to populate upgraded depots with file size information for existing revisions, the `p4 verify -u` command has been updated. It will compute the necessary values for selected file revisions, as long as the MD5 value, the file size, or both, need to be calculated. Similarly, the `p4 verify -v` command has been updated to refresh both the MD5 hash value and the file size for any files specified. This is done regardless of what is already stored in those values for the specified file revisions.

This new functionality makes it much less expensive to display a file's size in a Perforce client or to make it available to a script. The server used to have to fully render the desired revision of the file in question and count the number of bytes it contained in order to compute this. Since the value remains static, a single calculation at storage time saves the server a considerable amount of work when it is being used with client software that constantly calls for file sizes to be displayed. One Perforce client program that makes extensive use of this is the 2005.1 release of P4V.

4.5 verify changes

4.5.1 verify -m

Sites upgrading to the 2005.1 release will likely want to have file size values calculated for each file revision. This will allow Perforce clients like the 2005.1 release of P4V to make use of this value without putting a load on the server for each file revision being reported on. At sites with large repositories, running a long `p4 verify` job, can be time consuming and can cause the server to use a large amount of memory.

For release 2005.1, an extra flag has been added to the `p4 verify` command, `-m <value>`. This is designed to be used in conjunction with the `-u` flag to limit the number of operations performed by a single `verify` command. For example, `p4 verify -m 1000 -u //depot/...` tells Perforce to add MD5 hash and file size values to up to 1000 file revisions and then exit. If there are more than the specified number of file revisions to be updated, the `p4 verify -u` command will need to be run multiple times. The command should be repeated until there are no more values to be updated.

When running in update mode with the `-q`, or quiet, flag, `p4 verify -u` reports nothing if it has not run out of revision records that require updating:

```
% p4 verify -qum 1000 //...
%
```

When there are no revision records to update with file size or MD5 information, the command reports this:

```
% p4 verify -qum 1000 //...
//... - file(s) already have digests.
%
```

When running without the `-q` flag, `verify` displays one line of output for each file revision processed, as it always has:

```
% p4 verify -um 10 //...
//depot/fee#1 - branch change 9 (text) 0FA210DD41DA8F35EA4A45
//depot/fee#1 - branch change 9 (text) 0FA210DD41DA8F35EA4A45
//depot/fie#1 - branch change 9 (text) CDE564481B883E2D6B98B0
//depot/fie#1 - branch change 9 (text) CDE564481B883E2D6B98B0
//depot/file1#1 - branch change 9 (text) FF44EB7681559452B56696
//depot/file2#1 - branch change 9 (text) 8F105FDB2429800859D47B
//depot/file3#1 - branch change 9 (text) C884BF7D9C154E2A607D05
//depot/file4#1 - branch change 9 (text) 70CC6D7CA8978BE85285BF
//depot/foe#1 - branch change 9 (text) A0D566CE4C86EA41096F45
//depot/foe#1 - branch change 9 (text) A0D566CE4C86EA41096F45
//depot/foo#1 - add change 1 (xbinary) 103D8391745376FF42849F
//depot/foo#2 - edit change 2 (xbinary) 3BDDA4F1D515FBC542F808
//depot/foo#3 - edit change 3 (xbinary) CBE97DA82352CEAA732099
%
```

It is important to note that the `-m` flag does not limit a `p4 verify` operation to the specified number of operations per file. It operates on a global basis and counts the total number of operations performed. It exits either when there is no more work to be performed or when the defined maximum number of operations for the command has been reached.

4.5.2 verify and MaxScanRows / MaxResults

Starting with release 2005.1, the MaxScanRows and MaxResults parameters of `p4 group` specifications are observed by `p4 verify`. This should help to prevent undesired performance problems associated with extremely large `p4 verify` operations. An example of one of these parameters in use can be seen here:

```
%p4 verify //depot/...
Too many rows scanned (over 10000); see 'p4 help maxscanrows'.
%
%p4 verify //depot/foo/...
//depot/foo/fee.txt#1 - add change 24 (text) D2FC7D7E7788C2BA58E2EA
//depot/foo/fie.txt#2 - edit change 30 (text) 639654F0C1198C3390ECE3
//depot/foo/fie.txt#1 - add change 24 (text) FC415EA76229A5719B2C25
//depot/foo/foe.txt#1 - add change 24 (text) 9C7F6DAC504F73C8854FC2
%
```

A `p4 verify` attempt of the entire depot was rejected by the server, while a subsequent, more limited request, was accepted and processed.

Sites that are already using these parameters for other purposes should check their `p4 verify` operations to ensure that they are completing properly now that `p4 verify` is observing the limitations that they impose.

4.6 Filetype modifiers

Filetypes modifiers can now be specified without a base filetype. This lets Perforce automatically select the base filetype and combine that with the specified type modifiers. With previous releases, type modifiers could only be specified along with base filetypes.

This new functionality works wherever filetypes are specified within Perforce. This includes the `p4 add`, `p4 edit`, `p4 reopen`, and `p4 typemap` commands. An example of this in use can be seen with this set of typemap entries:

```
TypeMap:
+m //...
+lS //depot/bin/...
```

In the case of the above typemap table, any files in `//depot/bin/...` will have the `+ls` filetype modifiers added instead of the `+m` modifier that is specified on the first line. So, when adding two files the results are as follows:

```
% p4 add bar bin/my_bin_file
//depot/bar#1 - opened for add
//depot/bin/my_bin_file#1 - opened for add

% p4 opened
//depot/bar#1 - add default change (text+m)
//depot/bin/my_bin_file#1 - add default change (binary+ls)
```

As with other Perforce mappings, typemap entries follow the standard behavior of later mappings overriding earlier ones. This can be seen in the example above, where the second file does not inherit the `+m` attribute.

When using filetype modifiers by themselves in commands, the specified modifiers are added to the existing filetype, as is shown here:

```
% p4 reopen -t +l bar
//depot/bar#1 - reopened; type text+lm
```

The ability to use filetype modifiers by themselves allows much more freedom, particularly with the `p4 typemap` command. When selecting files in a certain depot path to all be stored using, for example, the `+s` modifier for a single revision of storage, each file extension used to have to be specified on its own typemap line, like this:

```
TypeMap:
    binary+S //depot/1rev/...pdf
    binary+S //depot/1rev/...doc
    binary+S //depot/1rev/...exe
    binary+S //depot/1rev/...jpg
    binary+S //depot/1rev/...xls
    text+S   //depot/1rev/...log
    text+S   //depot/1rev/...txt
```

Such cumbersome mappings would also fail to apply the desired filetype to files with unspecified file extensions. Now, in order to make all the files in `//depot/1rev/...` have the `+s` modifier, a single typemap line is needed:

```
TypeMap:
    +S //depot/1rev/...
```

This results in less work for the Perforce server and less maintenance for the Perforce administrator.

4.7 Btree passive reorganization

The Btree structures that comprise the Perforce db.* files start out well ordered at the time of a checkpoint restore. With use, however, new disk blocks are allocated and table pages with deleted data are marked as free and then reused. As the table grows, the allocation of new pages makes the table get out of sequential on-the-disk order. The problem has less to do with the relative size of the database files and more to do with the number of records that are added and removed from the Btree structure. After a while, what would otherwise be a sequential read of the Btree can result in the system to having to skip around the disk to put the pieces together. This means increased I/O time and degraded performance.

The 2005.1 release adds a passive database file reorganization function to the Perforce server. As updates are made to the database tables, the server reorganizes portions of them into sequential groups of disk blocks. This results in better performance, as the disk heads can move more quickly from one piece of the table to the next if the pieces are sequential.

Modern disk drives and operating systems look for sequential reads and attempt to pre-cache data that they believe will be requested by programs based on what was already asked for. Thus, performance of all Perforce operations should improve because the server has to move the disk heads around less to find sequential data within the database tables, and caching algorithms can better guess what the server will be looking for. These changes, however, may result in slightly larger database tables due to the allocation of additional contiguous disk pages. In testing, taking sample checkpoints took about 15% less time with the new functionality, while restores took approximately the same amount of time.

The amount of skipping around required can be seen in the “wrinkle factor” for each table displayed by the Perforce server. This value is simply the sum of the number of pages skipped in each leaf to leaf database page seek divided by the number of leaf pages. A value of one is ideal, but this is almost never seen in practice. The wrinkle factor for each table can be seen in the output of the `p4d -vdb=2 -xv` command. Partial output from this command, for one table, can be seen here:

```
Validating db.have
tree stats:      leafs: 433      internal: 7      free: 6756      levels: 2
                  items: 22378    overflow chains: 0    overflow pages: 0
                  missing pages: 0    leaf page free space: 2%
                  leaf offset sum: 94523  wrinkle factor: 218.73
Unlocking db.have.
```

With a high wrinkle factor, the table takes longer to read. Prior to release 2005.1, the way to fix this problem was to take a checkpoint and to restore from it. After taking a checkpoint and restoring from it, the wrinkle factor for this table drops, as shown here:

```
Validating db.have
tree stats:      leafs: 428      internal: 7      free: 0 levels: 2
                  items: 22370    overflow chains: 0    overflow pages: 0
                  missing pages: 0    leaf page free space: 1%
                  leaf offset sum: 985    wrinkle factor: 2.30
Unlocking db.have.
```

4.8 dirs faster

In the 2004.2 and earlier releases, the `p4 dirs` command (without the `-D` flag) could encounter performance problems. The performance problems resulted from scanning revisions while attempting to find a file not deleted at the head revision. If a significant number of files deleted at their head revision were encountered, the scanning that resulted would extend the execution of the `p4 dirs` command. The impact could be significant not only for the user executing the command, but also for the users that were blocked by the read locks held while scanning the revisions.

The algorithm used in the 2004.2 and earlier releases (without the `-D` flag) started by positioning to the first revision matching the argument provided. Revisions were then scanned looking for the first file in a directory not deleted at the head revision (and still matching the argument provided). When a file in a directory not deleted at the head revision was found, the scanning terminated and the file's path truncated to the granularity matching the argument provided was returned. The algorithm then positioned to the first revision following the path just returned, and the process repeated until revisions matching the argument provided were exhausted.

As an example, using the revisions (note that revisions are stored in `db.rev`, which is keyed on ascending depot filename and descending revision number):

```
//depot/main/dir1/file1#2 - delete
//depot/main/dir1/file1#1 - add
//depot/main/dir1/file2#2 - delete
//depot/main/dir1/file2#1 - add
//depot/main/dir2/subdir1/file1#2 - edit
//depot/main/dir2/subdir1/file1#1 - add
//depot/main/dir2/subdir1/file2#2 - edit
//depot/main/dir2/subdir1/file2#1 - add
//depot/main/dir3/file1#2 - edit
//depot/main/dir3/file1#1 - add
//depot/main/dir3/file2#2 - edit
//depot/main/dir3/file2#1 - add
```

The `p4 dirs //depot/main/*` command in the 2004.2 and earlier releases positioned to the first `//depot/main/...` revision. The algorithm then scanned revisions looking for the first file in a directory not deleted at the head revision. This scanned the revisions:

```
//depot/main/dir1/file1#2 - delete
//depot/main/dir1/file1#1 - add
//depot/main/dir1/file2#2 - delete
//depot/main/dir1/file2#1 - add
//depot/main/dir2/subdir1/file1#2 - edit
```

Scanning terminated and the path `//depot/main/dir2` was returned. The algorithm then positioned to the first revision following `//depot/main/dir2` and scanned revisions looking for the first file in a directory not deleted at the head revision. This scanned only the revision:

```
//depot/main/dir3/file1#2 - edit
```

Scanning terminated and the path `//depot/main/dir3` was returned. The algorithm then positioned to the first revision following `//depot/main/dir3` and finding no more revisions, terminated.

When the `-D` flag (include directories with only files deleted at their head revision) was used in the 2004.2 and earlier releases, no additional scanning beyond the scans to find the first file in each directory was required. The scans to find the first file in each directory returned the head revision of the first file in the directory; if the revision was a “delete” was irrelevant since all directories were to be returned, including those with only files deleted at their head revision. Using the `-D` flag could be used as a workaround to the performance problems of the `p4 dirs` command encountered in the 2004.2 and earlier releases, if it was tolerable to include in the output the directories with only files deleted at their head revision.

In the 2005.1 release, the `p4 dirs` command (without the `-D` flag) uses the new `db.rev` index. The `db.rev` index tracks the head revision of files not deleted at their head revision. Since the `db.rev` index contains only the head revision of files not deleted at their head revision, no additional scanning beyond the scans to find the first file in each directory are required. The scans to find the first file in each directory return the head revision of the first file not deleted at its head revision in the directory.

The size of the `db.rev` index is dependent upon the site's number of files not deleted at their head revision. An estimate is 20% of the size of the `db.rev` table, but this estimate is dependent upon the site's percentage of files not deleted at their head revision and the average number of revisions per file. The `db.rev` index is approximately 20% of the size of the `db.rev` table at a production site where approximately 70% of the files are not deleted at their head revision, and there are an average of 3.2 revisions per file. The estimate should be increased as the percentage of files not deleted at their head revision increases. But the estimate should be decreased as the average number of revisions per file increases.

The algorithm used for the `p4 dirs` command (without the `-D` flag) in the 2005.1 release is similar to the algorithm used for the `p4 dirs -D` command in the 2004.2 and earlier releases, with the exception that the `db.rev` index is now used. The algorithm used for the `p4 dirs -D` command in the 2005.1 release is similar to the algorithm used in the 2004.2 and earlier releases; `db.rev` continues to be used for the `p4 dirs -D` command in the 2005.1 release.

The performance improvement of the `p4 dirs` command (without the `-D` flag) in the 2005.1 release can be significant. For one example configuration where 73,000 files are deleted at their head revision in a server with 751,978 files, the `p4 dirs` command executes in approximately 1.2 seconds in the 2004.2 release:

```
$ time p4 dirs //depot/main/* | wc
    90      90     1440
real    0m1.166s
```

Using the same dataset and configuration as the 2004.2 configuration above, the same `p4 dirs`

command completes nearly instantaneously in the 2005.1 release:

```
$ time p4 dirs //depot/main/* | wc
      90      90     1440
real    0m0.045s
```

4.9 Opened files for client

It is often useful to see the files open on a specific client. In the 2004.2 and earlier releases, the only reliable way to use the `p4 opened` command to see the files open on a specific client was to search the results of the `p4 opened -a` command for the desired client. Using the `p4 -c <client> opened` command is not reliable since the client may be locked for usage by only the owner of the client, who could use a password. Using the `p4 opened -a` command to see the open files on a specific client uses additional resources within the server when collecting the open files for all clients, transferring from the server to the client the open files for all clients, and filtering at the client the open files for the desired client. Using the `p4 -c <client> fstat -Ro //...` command (which is not subject to the locked client restriction) is probably more efficient than the `p4 opened -a` command, but the `p4 -c <client> fstat -Ro //...` command accesses additional tables, such as `db.have`, `db.resolve`, and `db.rev`.

In the 2005.1 release, the `-C <client>` argument (following the “opened” keyword) is now available for the `p4 opened` command. The `p4 opened -C <client>` command reduces the overhead of returning open files for all clients, yet accesses only the necessary tables. The `p4 opened -C <client>` command is the most efficient way to see the files open on a specific client.

5 Summary

There have been a number of performance-related features introduced since the last Perforce User Conference. We hope that these features are useful to you and that they help to improve your server’s performance. Please, try them out and as always, if you have questions, contact support@perforce.com.