Repository Structure Considerations for Performance

Michael Shields
Performance Lab Manager
Perforce Software

Tim Brazil
Performance Lab Engineer
Perforce Software

March, 2009

Abstract

The structure of a repository can affect performance. Decisions such as the length of paths and products or projects as depots or first-level directories can affect the structure of Perforce metadata stored in btrees within the Perforce db.* files. Different btree structures can result in different Perforce Server performance characteristics.

This paper discusses how the structure of metadata stored in btrees within the db.* files is affected by fundamental decisions of structuring a repository. Also discussed is the effect of these decisions on Perforce Server performance. The different performance characteristics will be illustrated by results of Perforce benchmarks. Some internals of the db.* files will be provided to further the understanding of the effects.

1 Introduction

Perforce performance might be affected by several factors. These factors include the hardware on which Perforce components are deployed, Perforce usage, and the structure of the Perforce repository. Each of these factors might have a significant performance impact.

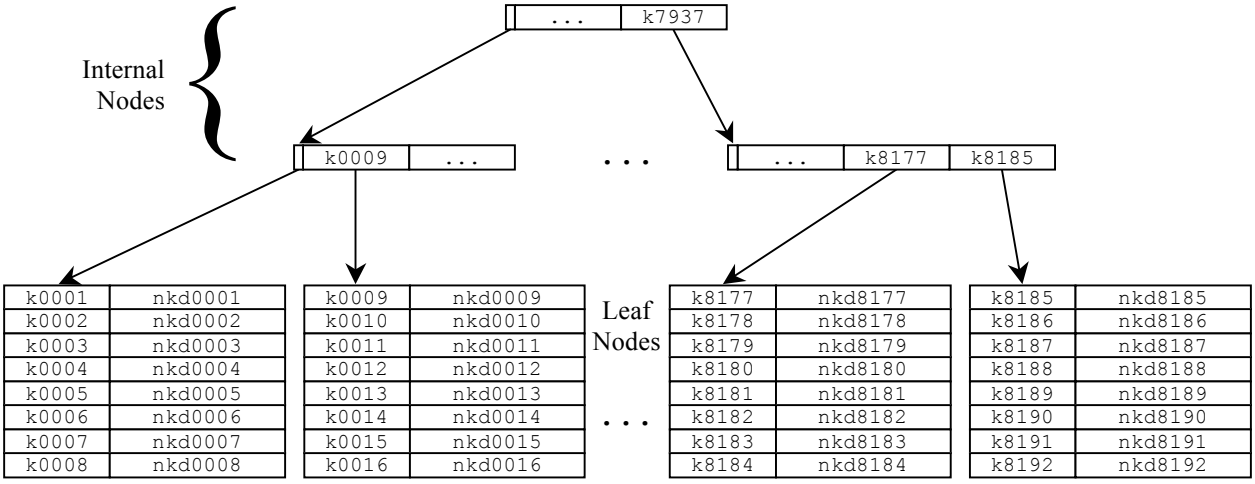
There are many decisions affecting the structure of a Perforce repository. Two important decisions are the length of paths and the placement of products or projects within the hierarchy of the repository. Historically, there has been little information available describing the effects on Perforce btrees and the performance implications of these two decisions.

Some of the descriptions and behavior shown below are undocumented and not supported by Perforce Technical Support. Undocumented behavior is subject to change at any time.

2 Perforce btrees

Perforce metadata in a db.* file is stored in a btree structure. Each db.* file actually contains two btree structures (starting with the 2005.1 release of the Perforce Server). One btree is for the Perforce metadata stored in the db.* file. The second btree in each db.* file is for free page management, which reuses contiguous free pages close to a page that already contains data proximal (in sorted order) to data being inserted. Unless otherwise noted, the btrees discussed in the remainder of this paper are the first of the two in each db.* file, that is, the btree for the Perforce metadata stored in the db.* file.

An example of the Perforce btree general structure is:



Perforce btrees generally consist of internal nodes and leaf nodes. (These might also be referred to as internal pages and leaf pages; in this context, node and page are often used interchangeably). Each entry in an internal node contains a key value and a reference to a node at a lower level, except the first entry in each internal node, which has a zero-length key. The implicit key value of the first entry in each internal node is the implicit or explicit key value of the entry that referred to the internal node. The implicit key value of the first entry in the internal node at the top level (often referred to as the root node) is empty; it is a key value that is less than all other key values. Each entry in a leaf node contains both the key and non-keyed data for a record of Perforce metadata.

Perforce btrees can also have overflow pages, which are used to store a record's non-keyed data that cannot fit on a leaf node. Overflow pages are generally only needed for storing lengthy text, such as a particulary verbose changelist description. Overflow pages can be chained together as necessary to accommodate the non-keyed data of a record. An overflow page or the start of an overflow chain is referenced in an entry of a leaf node. Because of their unique properties, the btree page discussions in the remainder of this paper do not apply to overflow pages.

The default size of each node in a Perforce btree is 8192 bytes. Empty and nearly empty btrees will not have any internal nodes and only a single leaf node. Therefore, the minimum size of a default btree is 8192 bytes. The size of a db.* file containing an empty or nearly empty default btree is 16384 bytes. The additional 8192 bytes is a metapage that is present in all db.* files. The metapage contains information about the two btrees in the db.* file. The second btree, which is for free page management, is not created until data needs to be inserted into a nearly empty btree and the only leaf node is full.

The number of entries in each node of a Perforce btree varies inversely with the actual length of each entry's key. As the lengths of the keys increase, the number of entries decrease. The node fan-out, which is the average number of nodes referenced by the internal nodes, decreases as the number of entries in the internal nodes decrease. As the node fan-out decreases, the number of internal nodes increase, and the number of levels in the btree might increase. The number of levels in a Perforce btree is the number of levels for only the internal nodes. An excessive number of levels can be problematic in that additional work is needed to traverse from the root node down to a leaf node.

The number of entries in each leaf node is further dependent upon the actual length of each entry's non-keyed data. As the lengths of either the keys or non-keyed data increase, the number of entries in the leaf nodes decrease. And as the number of entries in the leaf nodes decrease, the number of leaf nodes must increase to store the same amount of metadata records in the leaf nodes. Additional internal nodes will be needed to reference the increased number of leaf nodes. The additional internal nodes might propagate up the btree, possibly requiring additional levels in the btree.

There are several Perforce utilities that can be used to examine the structure of the btrees in the db.*files. The *p4d -xv -vdb=2* command can be used to determine a number of the btree characteristics discussed thus far. Since the *p4d -xv* command scans both the internal and leaf nodes, one of the statistics reported is the number of records, which is labeled as “items”. An example of the output from a *p4d -xv -vdb=2* command is:

```
$ p4d -r $P4ROOT -xv -vdb=2
...
GetDb db.rev mode 1
Validating db.rev
tree stats:     leafs: 1865619  internal: 23234 free: 0 levels: 4
                items: 72625000 overflow chains: 0      overflow pages: 0
                missing pages: 0        leaf page free space: 2%
                leaf offset sum: 11412075       wrinkle factor: 6.12
                main checksum: 1829071194 alt checksum 0
Unlocking db.rev.
...
```

A faster command that can be used to determine most of the same btree characteristics is the *p4 admin dbstat* command. This command is faster because only the internal nodes are scanned. The *p4 admin dbstat* command will also report several other btree characteristics, such as the node fan-out. If the *-h* flag is specified with the *p4 admin dbstat* command, a histogram showing distances between leaf pages is also reported. The distances between leaf pages can be useful for understanding disk seek behavior resulting from scans of a subset of leaf nodes. An example of the output from a *p4 admin dbstat -h* command is:

```
$ p4 admin dbstat -h db.rev
db.rev
        internal+leaf 23234+1865619
        page size 8k end page 1888854
        generation 4 levels 4 fanout 81
        ordered leaves: 97%
        Checksum 1829071194

         .... : -1000       283
        -1000 :  -100         0
         -100 :   -10     22944
          -10 :    -1         0
                    1   1819446
            1 :    10         1
           10 :   100     22663
          100 :  1000         0
         1000 :  ....       281
```

3 Path Length

Paths are used throughout the Perforce metadata. The length of the paths can affect Perforce performance. It is recognized that some characteristics of paths might be a function of other tools. For example, the Java™ development environment can influence some characteristics of paths. Perforce development continues to consider enhancements to the Perforce product that will minimize the effects of path length on performance. Alternatively, a Perforce workaround that will manage path length yet maintain interoperability with other tools will be described in "Path Length Recommendations" (section 3.4).

3.1 Effects on Perforce btrees

Paths are part of the key in the btree of many Perforce db.* files. Depending upon the db.* file, the paths might be in depot-syntax (//<depot-name>/<depot-path>, often referred to as depot paths) or client-syntax (//<client-name>/<client-path>, often referred to as client paths). As of the 2008.2 Perforce Server, the following db.* files have a btree with keys containing paths:

db.* file	paths in key
db.archmap	lbrFile, depotFile
db.have	clientFile
db.integed	toFile, fromFile
db.label	depotFile
db.locks	depotFile
db.resolve	toFile, fromFile
db.rev	depotFile
db.revcx	depotFile
db.revdx	depotFile
db.revhx	depotFile
db.revpx	depotFile
db.revsx	depotFile
db.working	clientFile

As the lengths of the above paths increase, the actual length of the keys in the btrees increase. This affects the Perforce btrees as described in section two. As the lengths of the keys increase, the number of entries in the internal nodes decrease. The fan-out of the internal nodes decreases, increasing the number of internal nodes, which might increase the number of levels in the btrees. And since the keys are also part of the metadata records in the leaf nodes, increasing the lengths of the above paths decreases the number of entries in the leaf nodes. The number of leaf nodes must then increase, which increases the number of internal nodes referencing the leaf nodes. This also might increase the number of levels in the btrees as the additional internal nodes propagate up the btrees.

Paths are also part of the non-keyed data in several Perforce db.* files. Depending upon the db.* file, the paths might be in depot-syntax or client-syntax. As of the 2008.2 Perforce Server, the following db.* files have paths as part of their non-keyed data:

db.* file	paths in non-keyed data
db.change	root
db.protect	depotPath
db.review	depotPath
db.trigger	depotPath
db.view	viewPath, depotPath

Since non-keyed data only appears as part of the metadata records in the leaf nodes, increasing the lengths of the above paths affects the btrees by decreasing the number of entries in the leaf nodes. The subsequent effects, which include an increased number of leaf and internal nodes and perhaps an increased number of levels, have already been discussed.

3.2 Effects on Perforce Server

Lengthy paths affect the Perforce Server in at least three ways. First, additional CPU cycles will be required when comparing longer paths. Second, the increased number of leaf and internal nodes will require additional disk space and I/O. Third, additional memory will be required for Perforce Server child processes or threads, and an effective OS filesystem cache.

The Perforce Server compares strings efficiently, usually requiring only reasonable CPU cycles. Some of the string comparisons start at the beginning of the strings and compare characters until an unequal character is found. Paths are often compared using this algorithm. For example, this string comparison algorithm is used when traversing a btree from the root node to a leaf node when the btree's key contains a path; additional levels in a btree with a key containing a path might result in additional string comparisons. This string comparison algorithm is also used when checking if a scan of a subset of leaf nodes should continue in a btree with a key containing a path.

Additional CPU cycles will be required during string comparisons in at least two different ways as the length of paths increase. First, if the leading part of the paths are lengthy, as is the case in //TheVerboseProject/MAIN-will-always-build/..., the string comparison algorithm mentioned above will always compare the leading part of the paths before ever possibly finding an unequal character. Even a few extra characters in the leading part of the paths can result in a measurable number of additional CPU cycles required by the many string comparisons performed in a typical Perforce installation. Second, as the overall length of the paths increase, additional CPU cycles are required to ensure that two paths are identical. Identical paths can only be assured by comparing each and every character of the two strings; equal lengths alone does not ensure that the paths are identical.

Additional internal and leaf nodes obviously require additional disk space. Perhaps not quite so obvious, additional disk I/O will be required when traversing a btree from the root node to a leaf node through any additional levels resulting from longer paths. Additional disk I/O might also be required for reading and writing the same number of metadata records on an increased number of leaf nodes as the path lengths increase. Probably not at all obvious is that as a result of fewer metadata records on each leaf node due to the longer paths, scans of a subset of leaf nodes might require additional disk I/O for internal nodes. Scans of multiple leaf nodes rely on the internal nodes to determine the next leaf node in the scan; the Perforce Server does not use forward and backward pointers in the leaf nodes, and is therefore not susceptible to corruption of these pointers as might be encountered in other btree systems.

Paths are often part of data stored during each compute phase[1] of a Perforce command in data structures internal to a Perforce Server child process or thread. As the lengths of the paths increase, some of the internal data structures increase in size. To accommodate the increased sizes of these data structures, additional virtual memory is used by each child process or thread. Many Perforce commands execute concurrently in a typical Perforce Server; each command executes in the context of its own child process or thread. The multiplicative effect of the number of child processes or threads and the additional amount of virtual memory for each requires additional physical memory in the machine on which the Perforce Server executes. The additional physical memory is required so that excessive paging of the child processes or threads does not occur and ample physical memory remains available for the operating system's filesystem cache and other resources critical to Perforce Server performance.

Some data structures internal to a Perforce Server child process or thread are limited in size. One such data structure is a cache for each db.* file used by the child process or thread. Pages in each of these caches can be used by the child process or thread without the need of an I/O request, thus enhancing performance. But since the caches are limited in size (though some tuning is available as of the 2008.2 release of the Perforce Server), only a limited number of pages are in each cache at any point in time. If the number of entries in each btree node has been decreased as a result of using longer paths, then the cached pages have less data that is immediately available to the child process or thread. This can result in additional I/O requests to process the same amount of data.

If needed data is not immediately available in a db.* file's cache, the child process or thread will execute an I/O request. For best performance, it is advantageous if the I/O request can be satisfied from the operating system's filesystem cache rather than waiting for a physical disk I/O to complete. Depending upon the operating system, the size of the filesystem cache is usually limited to some portion of the machine's physical memory that is not otherwise used. If the number of entries in each btree node has been decreased as a result of using longer paths, then the pages in the filesystem cache have less data that can be processed without additional physical disk I/O operations. In most environments, I/O requests that can only be satisfied by physical disk I/O operations will take much longer to complete, degrading Perforce Server performance.

The additional overhead attributable to longer paths discussed in the preceding paragraphs can affect concurrency in the Perforce Server. Some of this overhead occurs during each compute phase of a Perforce command. Since locks are held on some of the db.* files[2] during a compute phase, any additional time required to complete the compute phase will result in some of the db.* files locked for a longer period of time. A locked db.* file might block other Perforce commands until the lock is released; locks held during a compute phase will be released by the time the compute phase completes. The concurrency consequences have been somewhat mitigated by the lock "de-wedge" logic[3] introduced in the 2008.1 release of the Perforce Server.

---

[1] Each command generally has one or more compute phases; one compute phase per argument. Each compute phase is generally followed by an execution phase before the next compute phase for the command begins.

[2] For a more thorough discussion of metadata locking, see section two of *Perforce Performance*, presented at the 2007 Perforce User Conference.

[3] See changelist 135973 in the 2008.1 Perforce Server release notes.

3.3 Benchmarks Varying Path Length

Performance degradation as a function of path length can be seen in a number of Perforce commands. The benchmarks shown below use a publicly available script that measures the performance of critical portions of two important Perforce commands. The datasets used in these benchmarks can made available if independent verification or further experimentation is desired.

3.3.1 Methodology

The Performance Lab tested the p4d server to determine the effect that path length might have on performance. The branchsubmit benchmark -- a single-threaded test that obtains metrics related to the commit phase during branch and submit operations -- was well suited for this purpose.

We ran the branchsubmit benchmark against each of five datasets. Since operating systems have a tendency to perform file system caching, it is unknown whether stored data may have an effect on an initial test run. For this reason, the branchsubmit benchmark was run twice to take advantage of this caching effect, resulting in a more consistent test behavior.

The hardware used for these benchmarks was:

p4d server	Make/Model	Dell 2950 PowerEdge III Server
	Processor(s)	(2) Quad Core Intel(R) Xeon(R) CPU X5450 @3.00GHz
	Memory	16 GB
	Local Disk	(4) 146.8 GB 15k SAS
	OS	SUSE Linux Enterprise Server 10 (SP1)
	Kernel	2.6.16.46-0.12-smp
	Release	P4D/LINUX26X86_64/2008.2/190934 (2009/03/05)
db storage	Make/Model	Violin 1010 DRAM Solid State Disk
	Capacity	413GB formatted XFS

3.3.2 Datasets

Five datasets were created for this test. The overall content of each depot was the same, although the directory names varied depending on the required test path length. For example, for the 064 dataset, the path length plus the average filename length equaled 64 bytes.

Each of the datasets contained a single depot, which included a mainline and thirty-five release branches. The mainline and each release branch contained 100 subdirectories. Additionally, each of those subdirectories contained 100 subdirectories. The paths for each dataset were lengthened by adding a series of numbers to the directory names. In the case of the longest paths (datasets 112 and 128), letters were also used.

Due to the variance in path lengths, the resulting depots ranged from ~34GB to ~57GB in size. The increase in size can be attributed to the effect the increasing path length had on db.rev* and other tables. Each depot contained 25.2M files, ~72.6M revisions, and ~49M integration records, with default db.protect protections but without db.have table entries.

The following formula for increasing path length was used.

064 *//depot/.../most/sites/have/longer/...*
080 *//depot/.../most[0...3]/sites[0...3]/have[0...3]/longer[0...3]/...*
096 *//depot/.../most[0...7]/sites[0...7]/have[0...7]/longer[0...7]/...*
112 *//depot/.../most[0...9][A...B]/sites[0...9][A...B]/have[0...9][A...B]/longer[0...9][A...B]/...*
128 *//depot/.../most[0...9][A...F]/sites[0...9][A...F]/have[0...9][A...F]/longer[0...9][A...F]/...*

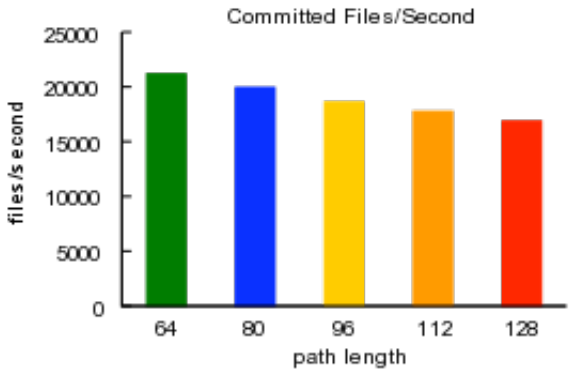
3.3.3 Results

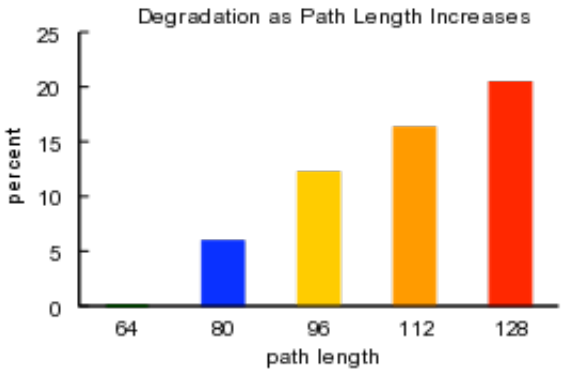
Path length does indeed have an effect on performance. As path length increased, the compute phase and commit duration took longer to complete. In addition, the commit rate performance decreased. Commit rate -- the number of files per second (f/s) that are committed during a submit transaction -- is a good indicator of submit performance.

The table below lists results from each test run.

Depot	Compute Phase	Commit Duration	Commit Rate
064	4210ms	3305ms	21180 f/s
080	4311ms	3513ms	19925 f/s
096	5117ms	3764ms	18597 f/s
112	5112ms	3947ms	17734 f/s
128	5289ms	4152ms	16859 f/s

The following charts illustrate the commit rate performance presented from the table above. While the chart on the left depicts what appears to be a gradual "Files Per Second" degradation as "Path Length" increased, the chart on the right identifies that actual percent degradation. Using the 64 byte path length as the baseline (0 degradation), the 128 byte path incurs ~20% performance degradation.





3.4 Path Length Recommendations

A wise man once wrote "Long, descriptive names may help the first time reader but hinder him every time thereafter."[4] While the original context for this quote was a bit different (and granted, it's a bit of a stretch for inclusion here), it holds true in this context as well. The "help" resulting from long, descriptive names in paths is that a new user more readily understands the structure of a repository. The "hinder" is that the resulting increased path length always affects Perforce performance for that user and all other users of the repository.

Existing repositories with lengthy paths need not be abandoned, nor is checkpoint surgery necessary. New projects using shorter paths can be started in the existing repository. As more activity shifts to the new projects, the longer paths are used less routinely and performance will begin to improve. Most of the causes of performance degradation associated with longer paths only affect Perforce performance if the longer paths are actually used. For example, pages are cached only if they have actually been used; if longer paths have been recently used, the cached pages will contain less data than if shorter paths have been recently used. But if the use of longer paths increased the number of levels in a btree, additional work will still be needed to traverse from the root node down to a leaf node, regardless of the paths used.

If possible, any lengthy leading part of paths should be shortened. The example mentioned above, //TheVerboseProject/MAIN-will-always-build/..., might easily be shortened to //tvp/MAIN/..., saving 32 character comparisons every time two paths must be compared by looking for the first unequal character. By shortening the leading part of paths, the overall path length will be decreased, reducing the performance degradation attributable to longer paths.

Client view mappings can be used as a workaround to manage path length yet maintain interoperability with other tools. Default view mappings can be automatically provided and enforced by Perforce Server triggers on the client form[5]. The view mappings can be constructed so that depot paths within a Perforce repository map to a location in the client workspace where other tools expect the files. For example, a client view mapping of:

```
//epiph/MAIN/jsrc/...          //my-client/com/our-company/epiphany-project/...
```

saves 27 characters in each depot path while placing the files in the client workspace at a location compatible with a Java™ development environment.

The length of depot paths can be effectively managed using client view mappings. But client view mappings will not affect the length of client paths. Lengthy client paths only contribute a small amount to the performance degradation attributable to long paths. Since lengthy depot paths is the more significant contribution, managing the length of depot paths using client view mappings can be an effective workaround to the performance degradation attributable to long paths.

---

[4] Christopher Seiwald, *Seven Pillars of Pretty Code*

[5] Additional information on Perforce Server triggers is available in Chapter 6 Scripting Perforce: Triggers and Daemons of the Perforce System Administrator's Guide.

4 Products or Projects as Depots or First-Level Directories

The hierarchical structure of a Perforce repository can affect performance. The placement of products or projects within the hierarchy is one decision that affects the structure. Making each product or project its own depot can result in one set of performance characteristics. Alternatively, a single depot containing all products and projects can result in a different set of performance characteristics.

4.1 Effects on Perforce btrees

Making each product or project its own depot will usually shorten the length of paths. The decrease in path length results from removing what was the leading depot from the path and shifting the product or project into its place. The shortened paths affect the Perforce btrees as one would expect based upon the above discussions. The btrees will generally have more entries in each node, resulting in fewer internal and leaf nodes, and perhaps fewer levels, than if a single depot contained all products and projects.

For each new depot created, a record is added to db.depot and db.domain. This can increase the number of internal and leaf nodes, and perhaps increase the number of levels in the btree in each of db.depot and db.domain. The btree in db.depot is usually trivial; adding one record for each product or project probably does not result in a btree with more than one internal node in db.depot. But db.domain can have a more interesting btree since db.domain also has records for client, label, and branch specifications. A typical Perforce site has many more client workspaces than product or projects, usually by multiple orders of magnitude. Adding one record to the btree in db.domain for each product or project amongst all of the other specifications does not result in a sufficiently more interesting btree.

4.2 Effects on Perforce Server

The location of products or projects within the hierarchy affects the Perforce Server beyond the performance implications attributable to path length. The effects of path length detailed above included considerations for string comparisons, the amount of disk I/O needed, and caching effectiveness. Decreasing the path length by making each product or project its own depot will affect Perforce Server performance as one would expect based upon the above discussions.

There is another performance implication of making each product or project its own depot. As the number of depots grows excessively large, the Perforce Server will use a significant number of CPU cycles for most commands as the mapping code works through the depot map. The depot map is a data structure internal to each Perforce Server child process or thread. Though the mapping code is fairly complex, there might be optimizations possible that reduces the overhead associated with working through an excessively large depot map. Perforce development continues to consider enhancements to the Perforce product that will minimize the effects of an excessively large number of depots.

There is also a performance implication related to a single depot containing all products and projects. As the number of directories at the first-level within a depot grows excessively large, the number of I/O requests made by the Perforce Server increases for commands such as *p4 dirs //<depot-name>/** and *p4 fstat //<depot-name>/**. The number of I/O requests increases linearly with the number of first-level directories. But each I/O operation could be expensive, depending upon factors such as the speed of the I/O subsystem and the amount of physical memory used as filesystem cache. Since the *p4 dirs //<depot-name>/** and *p4 fstat //<depot-name>/** commands can be frequently executed by GUI applications such as the Perforce Visual Client, it is important to ensure that these commands are not excessively burdened by I/O operations.

4.3 Benchmarks Varying Depots and First-Level Directories

The benchmarks in the previous section showed the performance implications related to path length. Though path length does decrease when each product or project is in its own depot, the benchmarks shown below eliminate the effect of changes in path length by ensuring that the lengths of the paths do not change from one dataset to the next. With changes in path length eliminated, the effect of incrementally varying between a single depot with an excessively large number of first-level directories and an excessively large number of depots with fewer first-level directories can be more readily observed.

The benchmarks shown below use a publicly available package that measures the performance of repeated Perforce commands. The datasets used in these benchmarks can made available if independent verification or further experimentation is desired.

4.3.1 Methodology

The Perforce Performance Lab set out to determine whether there was a performance advantage in the way a product directory is organized within the depot. Which is better: to place a product directory structure under its own depot or to place all products under a parent //depot? For example, which of the following configurations have the greatest performance benefits.

//productA		//depot/productA
//productB	or	//depot/productB
//productC		//depot/productC

It is very challenging to predict exactly how individual users navigate through a repository. Therefore, we based our testing on the "application" approach. Applications have a more consistent behavior, and can be modeled in a test scenario, as we did with the browse benchmark.

The browse benchmark was modeled using the P4V (Perforce Visual Client) application and similarly to P4V, initially extracts the names of all depots. The browse benchmark then navigates down random directories within a depot performing *dirs* and *fstats.* It completes the series of commands with a final *filelog* command at the end of the directory tree. With the use of a specified seed value within the test, randomness can be reproduced for each test run.

The browse benchmark was designed as a load test and generates numerous test users relentlessly voyaging down random paths. It does not necessarily replicate human user behavior. Executing the browse benchmark under various configurations can easily identify trends corresponding to environment changes. For our purpose, the environment changes relate to variations in the depot directory structure.

The browse benchmark was executed against each of fifteen datasets in separate test runs. Two computers were used: a host machine running the p4d server, and a client machine enlisted to launch a varying number of children. Each browse child performed sixteen browses. A "browse" is a series of p4 client commands including *depots*, *dirs*, *fstat*, and *filelog*. The *depots* command collected a list of available depots. The *dirs* and *fstat* navigated down a random directory path. At the end of the "browse" the *filelog* command was executed.

The hardware and operating system environment used to test Varying Depots and First-Level Directories was the same used during the Varying Path Length benchmark.

4.3.2 Datasets

To accurately measure performance differences, the Performance Lab required a depot design that accommodated varying structures while maintaining the overall size and qualities of the depot. To accomplish this, the depot and first-level directory structures were generated using a binary model naming convention.

//depot[binaryValue]/[binaryValue]top/[main | release]/...

In the table below, a single depot named "//depot" has 16384 directories with a naming convention of 00000000000000top…11111111111111top. Each subdirectory below the <binaryValue>top level contains a mainline and eighteen release branches. The path below each of the release branches was ./most/sites/have/longer/paths/jam/src where the ./src directory contained seventy ASCII source files.

As the number of depots increased for other datasets, the number of first-level directories decreased. The overall content of each resulting dataset remained the same.

	Depot Naming	Directory Naming	
1	depot	00000000000000top…11111111111111top	16384
2	depot0…depot1	0000000000000top…1111111111111top	8192
4	depot00…depot11	000000000000top...111111111111top	4096
8	depot000…depot111	00000000000top...11111111111top	2048
16	depot0000…depot1111	0000000000top...1111111111top	1024
32	depot00000…depot11111	000000000top...111111111top	512
64	depot000000...depot111111	00000000top...11111111top	256
128	depot0000000...depot1111111	0000000top...1111111top	128
256	depot00000000...depot11111111	000000top...111111top	64
512	depot000000000...depot111111111	00000top...11111top	32
1024	depot0000000000...depot1111111111	0000top...1111top	16
2048	depot00000000000...depot11111111111	000top...111top	8
4096	depot000000000000...depot111111111111	00top...11top	4
8192	depot0000000000000...depot1111111111111	0top...1top	2

The following example shows the *dirs* within the single depot dataset. The directory structure is comprised of a mainline and eighteen release branches (nineteen directories total).

```
$ p4 dirs //depot/00000000000000top/*
//depot/00000000000000top/main
//depot/00000000000000top/r01.0.0
//depot/00000000000000top/r02.0.0
//depot/00000000000000top/r03.0.0
…
//depot/00000000000000top/r18.0.0
```

The following directory path examples illustrate that although the //depot/first-level directories are re-structured, the length of the overall path remains the same.

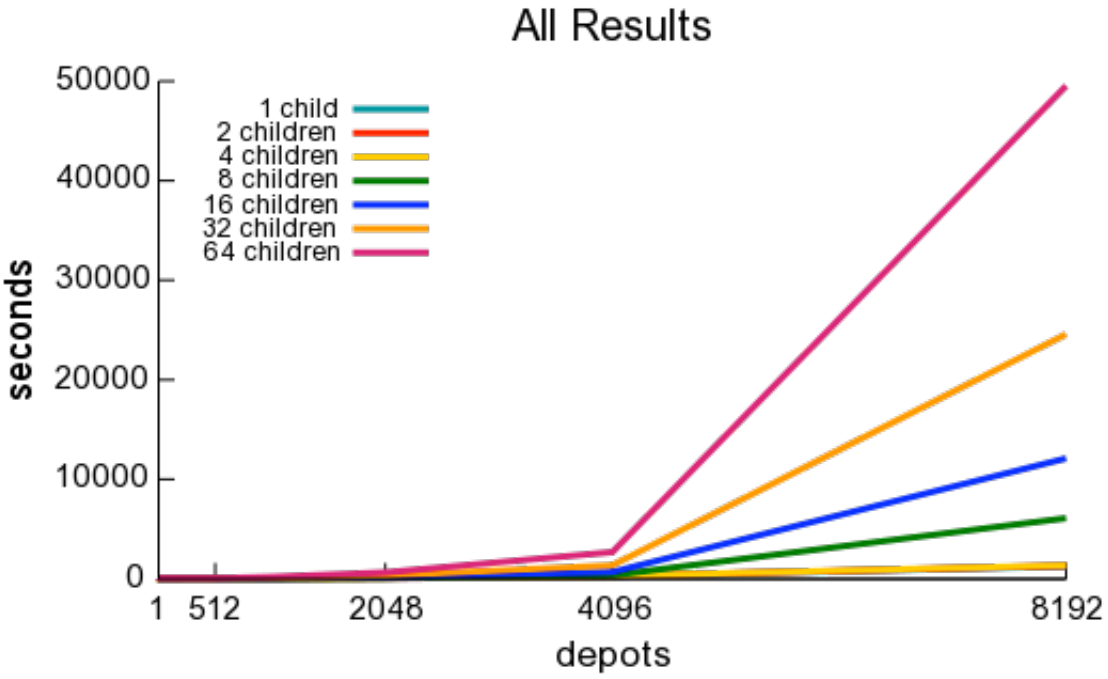
```
//depot0/1111111111111top/main/most/sites/…/jam     # 2 depots
//depot0000000/1111111top/main/most/sites/…/jam     # 128 depots
//depot0000000000000/1top/main/most/sites/…/jam     # 8192 depots
```

The resulting size of each dataset was ~42GB. Each dataset was comprised of ~21.8M files, ~63.9M revisions, ~41.3M integration records, with default db.protect protections but without db.have table entries.

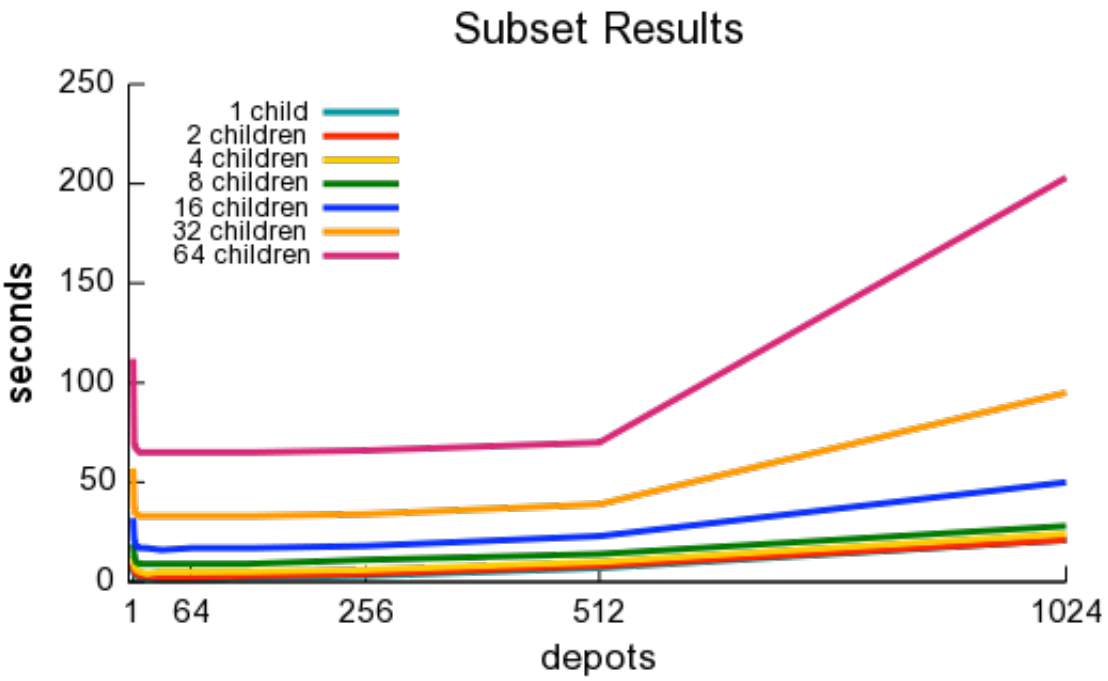
4.3.3 Results

A repository structure that has a large number of depots (//productx) can have an effect on performance. Additionally, it is not optimal to use a single depot containing an excessively large number of first-level directories and intensive browsing is a normal activity of the user community.

The graphs below illustrate the performance variations as the number of depots increased.



The **All Results** chart presents the total statistics from all test runs (1 depot to 8192 depots) with significant degradation apparent at ~4000 depots.



The **Subset Results** chart presents only statistical information for 1 depot to 1024 depots. At this granularity is it evident that the actual degradation begins after 500 depots. It is important to understand that 500 depots is not the hard and fast breaking point. In a production environment several factors need to be considered and each factor may weight the results in either direction.

The charts above were generated from data obtained from the browse benchmark. The p4d log file snippets below reflect the same findings at a more granular level. The log snippets were sampled from three separate test configurations:

1 depot / 1 child
8 depots / 1 child
4096 depots / 1 child

Each of the log snippets were taken from the initial browse transaction from a single child configuration as the test began. Neither high machine loads nor resource constraints were factors in these test runs.

The 1 depot / 1 child test: The initial *dirs* command at the first-level directory finished in .189 seconds and followed with an *fstat* completion at .162 seconds. The following *dirs* and f*stats* complete in less than the minimum p4d log time increment of .000 seconds.

```
2009/04/02 14:03:09 pid 2376 … 'user-dirs -C //depot/*'
2009/04/02 14:03:09 pid 2376 completed .189s 36+140us 0+0io 0+0net 0k 0pf

2009/04/02 14:03:09 pid 2376 … 'user-fstat -P -C -Olh //depot/*'
2009/04/02 14:03:09 pid 2376 completed .162s 20+140us 0+0io 0+0net 0k 1pf

2009/04/02 14:03:09 pid 2376 … 'user-dirs -C //depot/00000001110010top/*'
2009/04/02 14:03:09 pid 2376 completed .000s 0+0us 0+0io 0+0net 0k 0pf

2009/04/02 14:03:09 pid 2376 … 'user-fstat -P -C -Olh //depot/00000001110010top/*'
2009/04/02 14:03:09 pid 2376 completed .000s 0+0us 0+0io 0+0net 0k 0pf
```

The 8 depots / 1 child test: The server log indicates the fastest times in this grouping. The initial *dir* command took only .024 seconds. The initial *fstat* finished on .021 seconds.

```
2009/04/02 09:49:13 pid 12752 … 'user-dirs -C //depot101/*'
2009/04/02 09:49:13 pid 12752 completed .024s 12+12us 0+0io 0+0net 0k 1pf

2009/04/02 09:49:13 pid 12752 … 'user-fstat -P -C -Olh //depot101/*'
2009/04/02 09:49:13 pid 12752 completed .021s 8+12us 0+0io 0+0net 0k 0pf

2009/04/02 09:49:13 pid 12752 … 'user-dirs -C //depot101/00000001110top/*'
2009/04/02 09:49:13 pid 12752 completed .000s 4+0us 0+0io 0+0net 0k 0pf

2009/04/02 09:49:13 pid 12752 … 'user-fstat -P -C -Olh //depot101/00000001110top/*'
2009/04/02 09:49:13 pid 12752 completed .000s 0+0us 0+0io 0+0net 0k 0pf
```

The 4096 depots / 1 child test: Significantly slower results. The initial *dirs* command completes in 1.12 seconds. The subsequent *fstat* is over ½ second. The following *dirs* command at the second-level directory finished as the same initial *dirs* time of 1.12 second.

```
2009/04/02 10:06:29 pid 19663 … 'user-dirs -C //depot101100111000/*'
2009/04/02 10:06:31 pid 19663 completed 1.12s 1108+12us 0+0io 0+0net 0k 1pf

2009/04/02 10:06:31 pid 19663 …'user-fstat -P -C -Olh //depot101100111000/*'
2009/04/02 10:06:31 pid 19663 completed .585s 584+0us 0+0io 0+0net 0k 0pf

2009/04/02 10:06:31 pid 19663 … 'user-dirs -C //depot101100111000/00top/*'
2009/04/02 10:06:32 pid 19663 completed 1.12s 1124+0us 0+0io 0+0net 0k 0pf

2009/04/02 10:06:32 pid 19663 … 'user-fstat -P -C -Olh //depot101100111000/00top/*'
2009/04/02 10:06:33 pid 19663 completed .585s 581+4us 0+0io 0+0net 0k 0pf
```

4.4 Depots versus First-Level Directories Recommendations

While the answer for a specific site is, of course, dependent upon many things (e.g. speed of the I/O system, protections table complexity, etc.), the general answer is that for a reasonable number of products or projects, there will be little performance degradation when using a depot for each product or project. That is, for a reasonable number of depots, there will be little performance degradation as the number of depots increase while the total amount of metadata remains generally constant.

5 Summary

Of the many decisions affecting the structure of a Perforce repository, two important decisions are the length of paths and the placement of products or projects within the hierarchy of the repository.

Increasing the path length affects the metadata btree in some of the Perforce db.* files by increasing the number of internal and leaf nodes, and perhaps increasing the number of levels. The benchmarks above show that doubling the path length from an average of 64 to 128 characters can result in a 20% performance degradation of some Perforce Server operations in some environments.

Making each product or project its own depot can decrease the path length, which will probably decrease the number of internal and leaf nodes, and perhaps decrease the number of levels, in the metadata btree in some of the Perforce db.* files. As the number of depots increases, there will be inconsequential increases in the number of internal and leaf nodes, and perhaps the number of levels, in the metadata btree of db.depot and db.domain. The benchmarks above show that there can be performance improvements as a result of making each product or project its own depot. But the benchmarks also show that making each product or project its own depot can result in performance degradation for an excessively large number of depots. The performance degradation is not a factor until there are more than 500 depots in some environments.