FINAL: Flexible and Scalable Composition of File System Name Spaces

Michael J. Brim
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
mjbrim@cs.wisc.edu

Barton P. Miller
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
bart@cs.wisc.edu

Vic Zandy
IDA Center for Computing Sciences
17100 Science Drive
Bowie, MD 20715
zandy@cs.wisc.edu

ABSTRACT

Group file operations enable tools and middleware to operate upon a large group of files located across thousands of independent servers in a scalable fashion, a necessary requirement for effective use of today's largest distributed systems. Our initial prototype of group file operations showed scalability benefits for several tools, but also revealed the importance of having a scalable method for defining useful groups. We have developed a language called FINAL for describing name space composition in a flexible and scalable manner. Clients of our TBON-FS distributed file system can use this language to compose a single-system image (SSI) name space that automatically creates useful groups in a scalable fashion. We provide many examples of traditional and SSI name space compositions that can be described using FINAL. We report how TBON-FS can compose a global name space from tens of thousands of independent name spaces in one-quarter second, and show that using the global name space eliminates hundreds of seconds of overhead when working with distributed file groups.

Categories and Subject Descriptors

D.3.2 [**PROGRAMMING LANGUAGES**] Language Classifications - *concurrent, distributed, and parallel languages, specialized application languages*.

D.4.3 [**OPERATING SYSTEMS**] File Systems Management - *access methods, directory structures, distributed file systems*.

H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**] Systems and Software - *distrubuted systems*.

General Terms

Management, Performance, Languages

Keywords

name space, composition, file system, scalability, middleware

1. INTRODUCTION

Tools and middleware face a daunting challenge to operate effectively on the world's largest distributed systems containing tens of thousands of hosts and hundreds of thousands of processors. A large class of problems encountered at this scale result from system designs that force group operations to use serial interactions with operating systems and file systems. As the target group size grows, the resulting group operation latency grows linearly or worse. For example, even today's most powerful supercomputers have system software that forces a parallel debugger to issue a control or inspection request for each process in a parallel application, often requiring interactions with thousands of compute hosts. Our research helps identify and eliminate these iterative barriers to scalable operation, and provides scalable solutions that enable developers to easily create new scalable tools or improve the scalability of existing software.

In previous work [4], we introduced *group file operations*, a solution to the problem of applying the same file operations to a large group of files located across thousands of independent hosts. The keys to the group file operation idiom are explicit identification of file groups using directories as the grouping mechanism, and the ability to name a file group as the target for conventional file system operations such as `read` and `write`. Group file operations provide an interface that eliminates forced iteration, thus enabling scalable implementations. To support scalable group file operations, we developed the TBON-FS distributed file system, which employs a tree-based overlay network (TBON) to provide scalable communication of group file operation requests and distributed aggregation of response data. TBON-FS provides client tools with a single-system image (SSI) name space containing files from thousands of independent file servers. Single-system image name spaces enable applications to access and operate on distributed resources as if they were local, easing the development effort by allowing developers to focus on features rather than distributed access and communication.

Several classes of tools and middleware can benefit from group file operations, including systems for distributed system administration and monitoring, parallel application runtimes, and distributed debuggers. For instance, tools for distributed monitoring and debugging often need to access the synthetic files for process control or inspection as provided by `/proc` across a large set of independent hosts. Using group file operations, these tools can easily control or monitor groups of processes by defining file groups containing the relevant files and performing group read or write operations. Previously [4], we demonstrated the utility and scalability benefits of group file operations by using them to quickly develop several new tools for distributed system management and monitoring, including a parallel version of the standard `top` utility, and integrating them within Ganglia, a distributed monitor for clusters and grids.

Although our initial investigation clearly showed the scalability benefits of group read and write operations, it also revealed a significant piece was missing, the ability to define file groups in a scalable fashion. TBON-FS originally used a simple, static composition strategy for constructing its SSI name space – each file server's name space was placed in an independent directory hierarchy of the global name space. This inflexible structure results in inefficient group definition for groups that contain files from many servers. For each new group, the TBON-FS client must create a directory and populate it with symbolic links to each member file in a non-scalable, iterative manner that can take hundreds of seconds for groups containing tens of thousands of distributed files. To avoid this centralized, iterative group definition, we began investigating scalable approaches for distributed construction of the name space that could be implemented using the TBON.

After considering a few straightforward techniques for addressing the problem of scalable group definition, including parallel path matching using regular expressions, it quickly became clear that no single approach to constructing the TBON-FS name space would meet the group definition requirements for a wide variety of tools and middleware. For instance, consider a strategy for creating groups from the synthetic files provided by `/proc` across a large set of independent hosts. A parallel debugger or job management system may wish to create a file group representing all the processes of a specific parallel application, while a distributed system load monitoring program may want groups consisting of all processes from every host or all processes running the same executable. We believe each TBON-FS client is best-suited to the task of constructing and organizing the global name space, and our goal is to develop a method for specifying global name space composition that is both flexible and scalable. Clients should be able to easily identify the files or directory hierarchies from each server's name space to include in the global name space, and to control how files from independent servers are correlated to achieve a name space tailored for use with group file operations. A key to achieving the latter property is *an efficient and automated method for creating file groups as directories within the composite name space*.

Although name space composition has been studied for over forty years, prior approaches are ill-suited to the construction of global name spaces that comprise tens of thousands of independent service name spaces, due to three factors. First, previous systems are inefficient, requiring either pair-wise composition [13,17,18,19,26,28] or fine-grained manipulation of directory entries [15,24]. Neither approach is scalable for composing thousands of name spaces. Second, systems that have provided a global name space at smaller scales (i.e., hundreds to a few thousand services) use an inflexible structure that isolates service name spaces into separate directory hierarchies, which avoids use of inefficient composition [4,5,8,10,26]. Finally, the semantics adopted by previous systems are fixed for specific compositions, although many valid choices may exist. For example, name space unions may have shallow or deep semantics, and may treat duplicate names using renaming or overlay semantics that hide duplicates.

To address prior deficiencies and our name space composition goals for TBON-FS, we developed a language for describing compositions with three key qualities:

- Scalability - many name spaces can be combined using efficient distributed name space construction, avoiding centralized pair-wise operations.
- Simplicity - name space composition is easily described using a simple tree abstraction for name spaces and a set of tree composition operators with clear semantics.
- Flexibility - many interesting compositions can be specified by combining declarative tree operations with prescriptive programming constructs.

The language provides a semantic foundation that guides our approach for efficient large-scale name space composition within TBON-FS, and can be adopted by previous or future systems requiring flexible name space composition.

Our language is FINAL, for **Fi**le **N**ame space **A**ggregation **L**anguage. FINAL treats name space composition abstractly as operations on rooted trees of names, and provides five tree composition operations: `subtree`, `prune`, `extend`, `graft`, and `merge`. . All composition operations on trees produce new trees. Each tree node is associated with a directory or file provided by some file service. However, path resolution uses only the parent/child edges maintained by the current tree view, rather than the state of underlying file services. Specifications containing FINAL declarations are translated at runtime to produce a name space accessible via a library interface. The Cinquecento language [30] serves as the base syntax for FINAL, with extensions for our tree and file service abstractions and the tree composition operations.

We demonstrate FINAL's expressive power by using it to describe many interestingly diverse compositions. Examples include mount and union mount, private name spaces for convenience and security, single-system image name spaces for management and monitoring, and name spaces for organizing heterogeneous resources in cloud computing environments. We also report how TBON-FS applies FINAL in a distributed manner, *using trees to compose trees*, to provide scalable construction and use of a global name space.

The rest of the paper is organized as follows. Section 2. introduces our core abstractions and operations for composing name spaces as trees. Section 3. presents the FINAL language and demonstrates its flexibility through several specification examples. Section 4. evaluates the use of FINAL within TBON-FS for scalable global name space construction, focusing on the issues of efficiency and scalability. Related work is summarized in Section 5..

2. NAME SPACE COMPOSITION

The two main abstractions used within FINAL are name spaces and file services. We first describe these abstractions, and then introduce our name space composition operations that are based on these abstractions. This section concludes with a few examples of how the composition operations can be used to describe various mount semantics provided by current operating systems.

2.1 FINAL Abstractions

We assume file system name spaces are organized as a traditional tree of directory entries, where internal vertices are directories and files are leaves. A path is a sequence of name elements that represent a traversal of the directory tree starting at its root and ending at the tree vertex corresponding to the named file system entry. Thus, name spaces map paths to file system entries. Figure 2-1 presents an example name space.

A *file service* is our abstraction for local or remote file systems. File services provide access to a *physical name space*, which con-

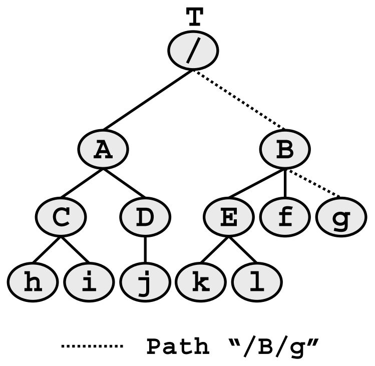

Figure 2-1. Name spaces and Paths
A *name space* is a rooted tree of names. A *path* is a tree traversal starting at the root node. In all figures, uppercase names are used for directories, and lowercase names are used for files. For the tree **T**, the path **"/B/g"** yields file "g".

tains entries for the file system(s) accessed through the service. File services also provide a set of common file operations that can be used on entries in their physical name space (e.g., `open`, `read`, `write`, and `close` for files, and `list` for directories). The most common example of a file service is the file system interface provided by the operating system on a local host, which typically presents a name space including several local file systems. Remote file services use file access protocols such as 9P [12], NFS [22], SMB [23], or Chirp [25] to provide operations on files and directories in remote physical name spaces.

For ease in describing name space composition, we abstract name spaces as trees and name space composition as tree alterations and combinations. We call these trees *name space views* to distinguish them from the physical name spaces of file services. All views used as inputs to composition operations are immutable, and operations produce new immutable views as results. The immutability of views provides clear semantics for composition, as there are no side-effects to the input views, and views can be freely used as inputs to many operations. Since file services are used as the building blocks for name space composition, services conventionally export a view that resembles their physical name space. If a file service removes file system entries after providing a view that contained them, lookups in the composite name space will fail.

Unlike the name spaces of conventional operating systems where each file system maintains the structure for its portion of the name space, FINAL's name space views cannot rely on the underlying file services to maintain the tree structure. Instead, views use a strategy similar to attachments in the Cedar file system [8] and skeleton directories in the Jade file system [19], where each entry in the view is associated with a path on some (possibly remote) file service.

2.2 FINAL Composition Operations

FINAL treats name space composition as abstract operations on immutable trees. The composition algebra consists of the five tree operations presented in Table 2-1.

The first three operations, `subtree`, `prune`, and `extend`, support common manipulations of single trees. These operations have two operands: a tree `T` and a path `P`. `subtree(T,P)` yields

Table 2-1. Final Composition Operations

`subtree``( Tree, Path ) => Tree` Returns a copy of sub-tree at the specified path in the input tree.
`prune``( Tree, Path ) => Tree` Returns a copy of the tree with sub-tree at path removed.
`extend``( Tree, Path ) => Tree` Returns a copy of the tree with the specified path prepended.
`graft``(` $\texttt{Tree}_1$, $\texttt{Tree}_2$, `Path ) => Tree` Returns a copy of Tree_1 with Tree_2 inserted at the specified path.
`merge``(` $\{\texttt{Tree}_k\}$, `conflict_fn ) => Tree` Returns a new tree that contains all unique paths and entries returned from applying the conflict function to all shared paths.

```
rename_merge( shared_path, conflicts )
{
  results = [ : ]; // initialize empty hash table
  if( all_directories(conflicts) ) {
     // return single, common directory
     results[shared_path] = conflicts[0];
     return results;
  }
  else if( all_files(conflicts) ) {
     // return all files, renamed to not conflict
     for( int i=0; i < length(conflicts); i++ ) {
        // new path is shared path plus version
        p = shared_path + "." + i;
        results[ p ] = conflicts[i];
     }
     return results;
  }
  // bad input structure
  return nil;
}
```

Figure 2-2. Example `merge` Conflict Resolution Function
Pseudocode that renames all files having the same path by appending a version number. Conflicting directories are merged to a single directory.

a view of the sub-tree whose root is found by traversing `P` in `T`. If the target sub-tree is a single file, `subtree` returns a tree consisting of a root directory whose only child is the file. `prune(T,P)` complements `subtree`, and results in a view of `T` with the subtree rooted at `P` removed. `extend(T,P)` yields a copy of the input tree with `P` prepended to its root. Figure 2-3 applies these three path operations to the example name space of Figure 2-1.

The fourth operation, `graft`, inserts one tree into another. As shown in Figure 2-3, $\texttt{graft(T}_1\texttt{,T}_2\texttt{,P)}$ produces the tree resulting from inserting $\texttt{T}_2$ into $\texttt{T}_1$ at `P`. Any path elements of `P` not present in $\texttt{T}_1$ are created in the result tree.

Together, the first four tree compositions can describe arbitrarily complex tree compositions. Unfortunately, such flexibility comes with a cost when one wants to perform deep composition of two or more trees, such as is needed for name space overlays. Overlays are name space compositions that include unique paths within all input trees, and where precedence is given to a top-layer name space when resolving shared path conflicts. A deep overlay can be described by traversing the trees in level order, identifying

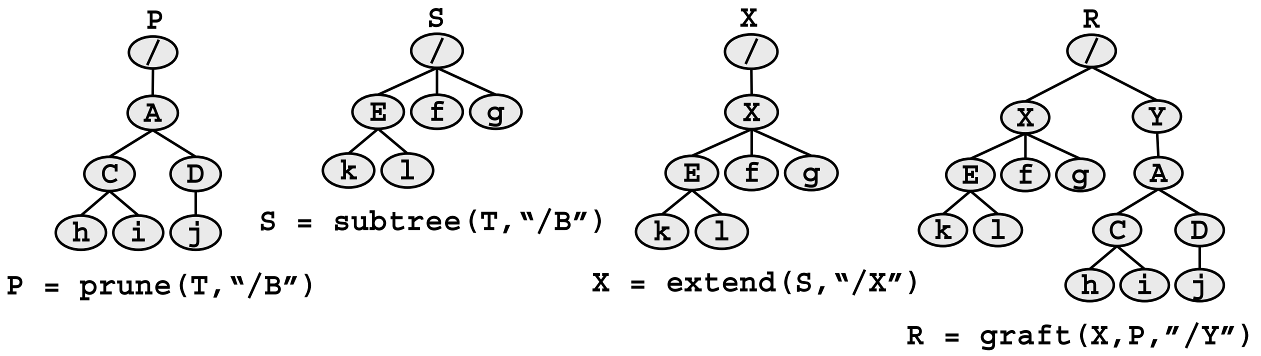

Figure 2-3. Path Composition Operations

The two left trees represent the results of applying the **`prune`** and **`subtree`** operations, respectively, to the name space of Figure 2-1 with the path "**`/B`**". The third tree shows the result of extending the **S** sub-tree with the path "**`/X`**". The rightmost tree shows a **`graft`** of tree **`P`** into tree **`X`** at path "**`/Y`**".

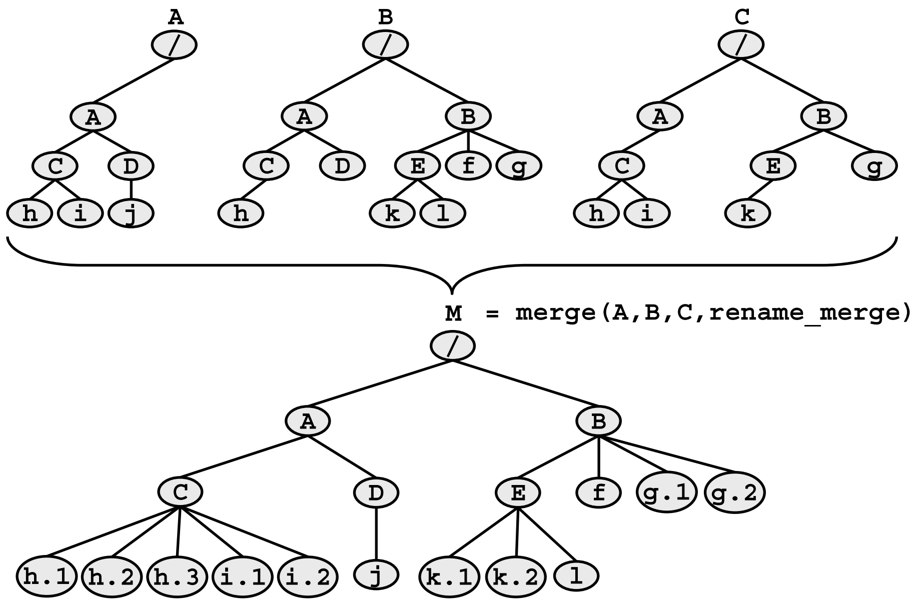

Figure 2-4. `merge` Tree Composition Operation

The upper three trees, **`A`**, **`B`**, and **`C`**, are merged using the conflict function of Figure 2-2, resulting in the lower tree **`M`**. As specified by the conflict function, common directories are merged into a single directory in **`M`**, and common files are renamed by appending a version number.

shared and unique paths. Sub-trees at unique paths can be grafted into the result tree, while vertices common to both trees result in a vertex copied from the top-layer service. A specification describing these overlay semantics may be extremely verbose for large trees, and the efficiency of composing the trees would be limited to serial evaluation of the composition operations. Similarly, as `graft` operates on two input trees, composition of a large set of trees would need to iterate and compose in a pair-wise fashion. One of our goals for name space composition is to eliminate such complex specifications and serialized pair-wise composition. Instead, we favor operations that provide the required semantics while still providing efficient and scalable composition of many trees.

Our last composition operation, `merge`, is designed to achieve this goal by providing a general-purpose, operation that captures the behavior of a large class of deep compositions, while still supporting customizable composition semantics. `merge({`T_k`},conflict_fn)` provides deep composition of a set of trees, where the trees are combined at all levels from their roots to leaves. The tree produced by `merge` contains all unique paths that occur in one input tree but not the others, as well as the results of applying a customizable conflict resolution function to paths that are shared among the input trees. Unique and shared paths are determined using a level-order traversal across all input trees. At each level starting with the roots, the conflict resolution function is called for each path that is shared among two or more trees. The function is passed the shared path and a set of conflicting vertices, and produces an output set of *(vertex, path)* pairs. Each *vertex* in the output set is added to the result tree at its paired *path*. By letting users define custom conflict resolution functions, FINAL provides flexibility to perform fine-grained manipulation, similar to how directory filters are used within the Virtual System Model [15], while still describing merge compositions at a high-level.

Pseudocode for an example function that uses a simple file renaming strategy is shown in Figure 2-2; this function returns a single directory for shared directory paths, or a set of renamed files for shared file paths. An example `merge` operation using this conflict function is shown in Figure 2-4. In Section 4., we use a minor variation of the conflict resolution function of Figure 2-2 that creates a new directory containing renamed files to merge thousands of name spaces without hiding any files from the constituent trees. Due to the common use of overlay semantics [17,18,28], we provide a pre-defined conflict resolution function called `overlay`. This function outputs the first vertex in the conflicting input set.

Although overlays are well-defined for pair-wise composition, their semantics become less clear for composition of large sets of trees as supported by `merge`. One approach to clarifying the semantics would be to assume an ordering over all trees in the merge input set, and then use that ordering to determine precedence for conflicts occurring between two or more trees. However, such an ordering constraint poses difficulties when performing the `merge` in a scalable distributed fashion. As described in Section 4., we use a tree-based overlay network (TBON) to implement scalable merges of large tree sets. Computation within a TBON generally requires commutativity and associativity of computation inputs, and support for hierarchical execution where results from computation at children vertices will be used as inputs to their parent's computation [21]. Requirements for commutativity and associativity result from both the hierarchical computation and asynchrony within the overlay network, as results from children can arrive in any order and will be passed to the parent's conflict function in the order received. In general, we believe there are a class of conflict functions that assume pair-wise behavior, and may not make sense for large scale, distributed merges. As we demonstrate later Section 4., this class of functions is still useful for specifying how individual name spaces are constructed before combining those name spaces in a distributed fashion.

Finally, though `merge` captures the essential behavior of many types of deep name space composition, it cannot emulate shallow merges, such as the unions provided by Plan 9 in which only first level names below the root are overlayed [18]. Still, shallow composition can be reproduced using iterative application of `graft` and `subtree`.

2.3 Traditional Composition Examples

FINAL can be used to specify a wide range of compositions in a simple, declarative manner. Here we focus on giving example compositions that describe the various forms of mounts provided by current operating systems, including standard, union, and bind mounts. Section 3. provides additional examples that fully use the flexibility of FINAL.

A traditional UNIX mount operation that attaches a file service to the current name space at a specified path, the *mount point*, is easily described using `prune` and `graft`:

```
T = graft( prune(O,P), N, P )
```

where `O` is a view of the original name space before the mount, `P` is the mount point, `N` is the new tree to be mounted, and `T` is the resulting tree.

A bind mount is a variation of the standard mount that uses a sub-tree of the existing name space rather than a new file system as the new name space to be mounted. Thus, bind mounts make a portion of the current name space available at more than one path simultaneously, and can be described by:

```
T = graft(O, subtree(O,P1)), P2)
```

where P_1 is the original path to the bound sub-tree and P_2 is the new path.

A union mount that lays the mounted name space over (or under) the current name space at the mount point, instead of replacing its contents, can be described by:

```
T = graft(O, merge({N,subtree(O,P)},
                   overlay), P)
```

which lays the name space ***over*** the existing sub-tree, or by:

```
T = graft(O, merge({subtree(O,P),N},
                   overlay), P)
```

which lays the name space ***under*** the existing sub-tree.

These simple mount examples show the ease in describing common name space compositions using our tree composition operations. We now proceed to discuss the flexibility afforded for specifying interesting compositions that results from combining the declarative semantics of the tree algebra with the prescriptive programming language capabilities.

3. THE FINAL LANGUAGE

FINAL's abstractions and tree composition operations provide the foundation for describing composition of many name spaces in a declarative manner. Further flexibility in composition specification can be achieved by extending FINAL with prescriptive capabilities such as iteration and conditional program flow constructs. We first describe the motivation for adding our abstractions and operations to an existing language, then discuss our choice to extend Cinquecento [30]. We introduce the key language features of Cinquecento that enhance the flexibility of FINAL for describing interesting compositions. The section concludes with simple examples of complete FINAL specifications.

3.1 Prescriptive Extensions for FINAL

While the declarative, functional nature of FINAL's composition operations is appealing due to its simplicity, there exist interesting composition strategies that are hard to describe using only these operations. To provide additional flexibility in writing practical specifications, we believe that imperative specification constructs are necessary. For example, when composing related sub-trees of name spaces from distributed services, the organization of the services' physical name spaces may differ such that the sub-trees are located at different paths. To support such heterogeneous name space composition using a declarative specification, the user would need to know a priori the specific paths to the sub-tree on each service. In these cases, it would be convenient to allow the specification to employ run-time queries to discover name space contents or environment context. Another instance where prescriptive capabilities are useful is when the composition is selective based on file attributes, and the specification could query name space entries to find those that satisfy the desired criteria.

Rather than design a completely new language to add prescriptive functionality, we have decided to embed the FINAL abstractions and operations within an existing programming language that provides the desired features. Cinquecento is a dynamically typed functional language that supports C expression syntax and data operators. Cinquecento programs are sequences of expressions that are dynamically evaluated in order. Originally designed to enable mixed-domain debugging, the language provides many useful functional capabilities such as lambda expressions as well as built-in types for high-level data structures including lists, vectors, and dictionary hashes. Cinquecento's C-based syntax should feel familiar to systems programmers, and its functional characteristics allow FINAL's composition operations to be used in a natural manner.

Although we consider Cinquecento a natural fit as the basis for FINAL, it is important to note that the flexibility that results from combining FINAL abstractions and composition operations with prescriptive language capabilities is not dependent on the choice of Cinquecento. Similar benefits could be realized using other dynamically evaluated languages such as Perl, Python, or Java.

3.2 FINAL Language Constructs

Here, we briefly introduce the useful constructs Cinquecento provides for writing FINAL specifications. A full tour of the Cinquecento language is available online [6].

Cinquecento expression syntax mirrors that of C, with the exception being that functions and variables are dynamically typed. FINAL specifications can use standard C control statements, including `if`, `do`, `while`, `for`, `switch`, and `goto` to provide prescriptive iteration and conditional program flow. The code snippet below demonstrates how closely Cinquecento resembles C, with the only prominent difference being the definition of two list data structures.

```
evens = []; /* an empty list */
odds = [];
for( i = 0; i < 10; i++ ) {
   if( i % 2 ) append(odds, i);
   else append(evens, i);
}
```

Cinquecento also provides functional language constructs, including lambda expressions and `apply`, `foreach`, and `map` operations that apply lambdas to collections of variables or containers. Combined with our extensions to Cinquecento for FINAL's abstractions and composition operations, these imperative and functional constructs provide the basis for writing flexible FINAL specifications.

3.3 FINAL Abstractions and Composition Operations

We have embedded two new data types in Cinquecento, `nstree` and `filesvc`, and functions that operate on these types to support FINAL's tree and file service abstractions.

The `nstree` data type represents a vertex in a name space tree. Each `nstree` corresponds to a name space entry on some file service, and thus contains a path and `filesvc` reference. To maintain the name space organization, each `nstree` has references to its parent and children `nstree` vertices. Initial views for a service are created using the `mknstree` function, which takes a `filesvc` operand and returns the root `nstree` for a tree that has an identical structure to the physical name space of the service. Figure 3-1 presents interfaces for creating and using the `nstree` data type.

Construct a tree from a file service.
`mknstree( svc ) → nstree`

Construct an empty tree.
`nulltree() → nstree`

Walk path in tree to target tree. Returns nil for bad path.
`treewalk( tree, path ) → nstree | nil`

Retrieve names for children of tree.
`treelist( tree ) → string[]`

Get vertex name, file service path, or file service for tree.
`treename( tree ) → string`
`treepath( tree ) → path`
`treesvc( tree ) → filesvc`

Figure 3-1. `nstree` Interfaces

Construct an instance of the named file service; "..." represents an optional set of service specific operands.
`mkfilesvc( name [, ...] ) → filesvc`

Define new file service with name and file operations.
`svcdefine( name, file_ops ) → int`

Figure 3-2. `filesvc` Interfaces

The `filesvc` type represents an instance of a defined file service. The `mkfilesvc` function creates a new instance of the named file service. `mkfilesvc` supports optional parameters that can be used to customize the service instance, such as passing host and mount point information for a remote file service like NFS or 9P. For convenience, a pre-defined service named `local` provides access to the local name space. Additional file services can be defined with the `svcdefine` function, which associates a set of service operations with the given name. `mkfilesvc` simply returns the service instance obtained by passing the optional parameters to the `init` operation of the named service. Figure 3-2 presents the interfaces for `filesvc`.

3.4 Example FINAL Specifications

In every FINAL specification, the variable `root` is used to denote the `nstree` that should be used as the resulting name space. The value bound to this variable at evaluation time should be of type `nstree`, and will be used to instantiate the name space that is used by applications. A specification to access the sub-tree of the local name space located at path "`/usr/bin`" would be:

```
L = mknstree( mkfilesvc("local") );
root = subtree( L, "/usr/bin" );
```

As expected, these simple specifications look similar to the composition examples of Section 2..

The following example specification demonstrates the use of procedural constructs to deal with heterogeneous context. This example checks for the presence of three common paths for temporary storage within the local name space and chooses the first available, and defaults to the user's home directory if none of the paths were found:

```
L = mknstree( mkfilesvc("local") );
tmp = subtree( L, "/tmp" );
if( tmp == nil ) {
   tmp = subtree( L, "/temp" );
   if( tmp == nil ) {
       home = getenv("HOME");
       tmp = subtree( L, home );
   }
}
root = tmp;
```

This example also demonstrates Cinquecento's support for accessing the environment, which is useful for parameterizing specifications that handle heterogeneous context.

Our next specification shows the use of name space inspection and a lambda function to generate a name space containing all files in the "`/var/log`" directory of the local name space that have sizes larger than 4KB:

```
L = mknstree( mkfilesvc("local") );
log = subtree( L, "/var/log" );
entries = treelist( log );
result = nulltree();
check_entry = @lambda(x){
   ent = treewalk( log, x );
   if( ent != nil ) {
      attr = stat( treesvc(ent),
                   treefile(ent) );
      if( isfile(attr) &&
          attrsize(attr) >= 4096 )
          result = graft( result, ent,
                    "/" + treename(ent) );
   }
}
foreach( check_entry, entries );
root = result;
```

The above example introduces two functions for name space navigation using `nstree`'s: `treewalk` and `treelist`. Each function has a `nstree` parameter that indicates the node from which navigation begins. `treewalk` takes a relative path operand, walks the path to the target `nstree`, and returns the target or the Cinquecento special value `nil`. `treelist` returns a list of `string` values containing the names of any children of the current `nstree`. In the example, `treelist` is used to get the entries of the "`/var/log`" directory, and `treewalk` is used to visit each entry.

The previous example also shows how a specification can query attribute information for the file referenced by a `nstree` using the `stat` method, which takes a `filesvc` and path and returns a `fileattr`. A `fileattr` object contains information similar to a `struct stat` as used by the POSIX `stat` operation. In the example, the attributes of each directory entry are queried to determine if the entry is a file and if the file's size is greater than 4KB.

Private name spaces, which originated in the Plan 9 operating system, provide processes the ability to construct a custom view of the default system name space. Common uses for this customization are for user convenience or system security. For convenience, a process may select some subset of the local name space that is necessary for execution. The reduced name space often makes access to target files more efficient, as the files are placed closer to the root of the name space. One common example of private name spaces is when a user wishes to substitute their own executables or libraries for the system installed versions. The following FINAL specification shows how to accomplish such a task.

```
loc = mknstree( mkfilesvc("local") );
home = subtree( loc, getenv("HOME") );
mybin = subtree( home, "/install/bin" );
bin = extend( mybin, "/usr/bin" );
rest = prune( loc, "/usr/bin" );
root = merge( [ rest, bin ], overlay );
```

In the security realm, private name spaces can be used to prevent unauthorized access to files through isolation or omission. An operating system may isolate processes by giving each process its own private name space, thus prohibiting other processes from viewing any newly created files. System administrators may desire to exclude sensitive portions of the system name space from the view of regular user processes. The next specification supports the latter by applying a lambda function to `prune` excluded paths.

```
loc = mknstree( mkfilesvc("local") );
rest = loc;
excludes = [ "/etc", "/root", "/sbin",
             "/usr/sbin", "/var/log" ];
excl_fn = @lambda(x) {
   rest = prune( rest, x );
}
foreach( excl_fn, excludes );
root = rest;
```

These simple example specifications show the flexibility achieved by incorporating prescriptive constructs in FINAL. The following section introduces the integration of FINAL within our TBON-FS distributed file system for scalable single-system image name space composition.

4. SCALABLE NAME SPACE COMPOSITION IN TBON-FS

TBON-FS is a distributed file system we designed to provide scalable group operations on large sets of files located across tens of thousands of distributed servers. A TBON-FS client views a global name space composed from the independent name spaces of servers. We briefly review the TBON-FS architecture and our group file operation idiom. We then discuss the problem for file group creation inherent in the original TBON-FS name space, and our goals for addressing this deficiency using name space composition. We describe our changes to TBON-FS to allow it to scalably construct the global name space, and give example SSI name space specifications that automatically create file groups. The section concludes with an evaluation of the modified TBON-FS system, including performance measured when composing tens of thousands of name spaces.

4.1 TBON-FS Architecture and Name Space

TBON-FS has an architecture as shown in Figure 4-1. A tree-based overlay network, namely MRNet [21], is used to multicast client application file system requests to user-level proxy servers

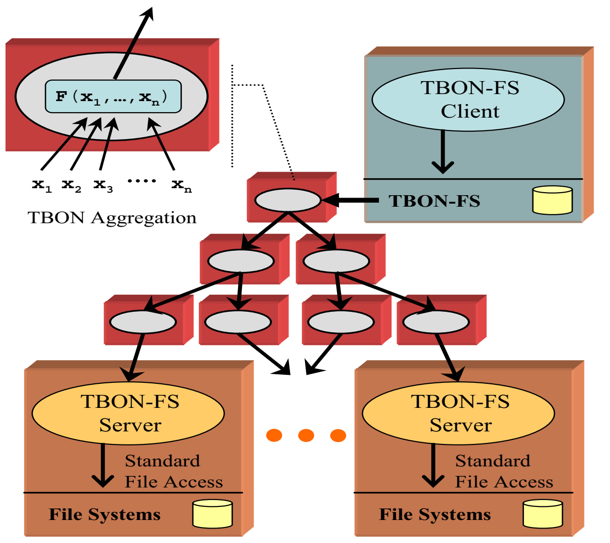

Figure 4-1. TBON-FS Architecture

TBON-FS client file operations are translated into requests that are multicast to user-level file servers using MRNet. TBON-FS servers provide a simple proxy for accessing the server host's local file system. Results from file operations at the servers are aggregated within the TBON before being delivered to the client.

on each distributed host. Servers run in the context of the user who owns the client application, and have the same file system access privileges as that user. File operation responses are aggregated with the TBON and delivered to the client. The set of server hosts and the overlay tree topology are specified by the client as file system specific options when TBON-FS is first mounted.

Although TBON-FS supports operations on individual files, it is designed to provide scalable group file operations [4]. The group file operation idiom allows clients to explicitly define groups using a directory as the group abstraction, and to open groups by passing the group directory as the first operand of `gopen`, a group version of `open`. The group file descriptor returned by `gopen` allows normal file operation such as `read` and `write` to be used on file groups.

TBON-FS originally used a simple composition strategy for constructing its global name space – each server's name space was placed in an independent directory hierarchy. This structure results in excess overhead for defining groups that contain files spanning many servers, as a new directory must be created for the group, and the member files must be either copied or symbolically linked inside the group directory. Thus, group definition proceeds in a non-scalable, iterative fashion. This behavior directly contradicts the goals of the group file operation idiom for eliminating explicit iteration. By integrating FINAL within TBON-FS, we wish to let clients specify how the aggregate name space is constructed, to enable efficient creation of file groups by automatically grouping related files within directories in the constructed name space, thus avoiding the iterative group definition.

4.2 TBON-FS + FINAL

We have extended TBON-FS to let clients provide a FINAL specification that controls how the global name space is constructed. Scalability in name space composition for large numbers of servers requires distributed execution to avoid non-scalable, centralized evaluation at the client. Further, to enable specifications to use server-local context when generating the name space, those specifications must be evaluated on the servers. Our integration of FINAL in TBON-FS takes advantage of the TBON to parallelize name space construction at each server and merge the server trees. Prior work has shown TBONs provide scalable tree merging for parallel analysis of application call graphs [1,11].

Conceptually, mount requests at the client are treated as a `graft` of a `tbonfs` service into the client's current name space at the requested mount point. A `tbonfs` service represents all TBON-FS servers, and provides a view that merges the individual name spaces of each server. Individual server name spaces are constructed according to a FINAL specification file that is named in the file system options passed to `mount`. Server specifications should be designed to achieve a name space organization that is suitable for merging. The service uses the TBON to multicast the specification to all servers. Since the specification is evaluated at the server hosts, it can rely on server-local context to handle heterogeneity in a manner that promotes easy merging. The conflict function used by the `tbonfs` service to merge the server name spaces is also supplied using file system options, in the form of two names: a shared library name, and the name of a function in that library. Since TBON-FS uses MRNet as its TBON, the conflict function is implemented as an MRNet filter function [21]. A unique feature of TBON-FS is that the client can call `mount` many times, each time passing a unique specification, as a means for generating multiple concurrent global views of the servers.

For efficiency, the merged name space provided by the `tbonfs` service is not fully computed at the time of `mount`. Rather, we use a lazy update strategy that applies the conflict function as necessary to the results returned by each server when specific paths are accessed. When resolving paths (e.g., during `open` or `stat`) or listing directories, the client traverses its local portion of the global name space, which consists of only the mount points for each `tbonfs` service. The service corresponding to the path's prefix is found, the prefix is removed, and the remaining path is passed to the appropriate service operation. The `tbonfs` service handles the operation request by using the TBON to perform a distributed name space lookup on each server and merge the results using its associated conflict function.

4.3 SSI Name Space Examples

Single-system image (SSI) name spaces provide file system clients a view of distributed resources that permits applications to use the resources as if they were local. Such name spaces simplify software development and reduce the effort required for system administration. Here, we present three examples of how clients can specify useful SSI name spaces.

The following examples assume the client specifies that the `tbonfs` service should use a `tbonfs_merge` conflict function that automatically creates file groups as directories in the global name space from common file paths in the independent server name spaces. Similar to the `rename_merge` function presented in Figure 2-2, `tbonfs_merge` produces a single directory for each common directory path. For common file paths, however, the function produces a new directory at the common path. The directory is populated with the conflicting files, after renaming the files using a version number.

```
// overlay files at root of name space
grps = [];
loc = mknstree( mkfilesvc("local") );
grps.append( subtree(loc,"/proc/meminfo") );
grps.append( subtree(loc,"/etc/passwd") );
grps.append( subtree(loc,"/var/log/syslog")
);
root = merge( grps, overlay );
```

(a) Server FINAL Specification

```
/tbonfs/
       /meminfo/
              /[0-99999] # servers 0 to 99999
       /passwd/
              /[0-99999]
       /syslog/
              /[0-99999]
```

(b) Client Name Space

Figure 4-2. Automatic File Groups
(a) Server FINAL specification that selects target files.
(b) Resulting client name space with files in group directories.

```
// define pid remap function
rank = getenv("TBONFS_SERVER_RANK");
@define remap_pid(pid) {
   newpid = (rank * 100000) + atoi(pid);
   return string(newpid);
}
// retrieve pids and remap
loc = mknstree( mkfilesvc("local") );
proc = subtree( loc, "/proc" );
entries = treelist(proc);
pids = [];
for( i=0; i < length(entries); i++ ) {
   name = entries[i];
   if( name[0] > '0' && name[0] <= '9' ) {
      newname = remap_pid( name );
      remap = extend( subtree(proc,name),
                      "/" + newname );
      pids.append( remap );
   }
}
// merge pids
root = merge( pids, overlay );
```

(a) Server FINAL Specification

```
/proc/
     /1/...          # server 0, pid 1
     /...
     /100001/...     # server 1, pid 1
     /...
     /9999900001/... # server 99999, pid 1
     /...
     /9999932768/... # server 99999, pid 32768
```

(b) Client Name Space

Figure 4-3. Global Process Space
(a) Server FINAL specification that remaps process ids.
(b) Resulting client name space that includes all processes.

Our first example shows a custom name space composition for the most common use of TBON-FS, where a client wants to perform group file operations. The target group often corresponds to a file having the same path on all servers. Figure 4-2 shows an example server specification that merges the sub-trees corresponding to three file paths, and the resulting client name space if the mount point is "`/tbonfs`". By using `tbonfs_merge`, group directories for each of the three paths are automatically created and populated.

In prior systems, the complexity in providing a single-system image for large systems often leads to a restriction of type of resources included, such as processes [9]. The specification shown in Figure 4-3 shows a server specification for constructing a global process space from many independent server `/proc` file systems, and the resulting client name space. A numeric renaming scheme is used to maintain a name space structure that is usable by process monitoring utilities like `ps` and `top`, which avoids conflicts between processes with the same process id on different hosts. Although the name space of Figure 4-3 is compatible with existing utilities, the use of this name space would be extremely inefficient, as they would need to iterate over hundreds of thousands of process directories. In [4], we demonstrated `ptop`, a parallel version of `top` that uses group file operations to provide the same interactivity and features when monitoring hundreds of thousands of distributed processes.

Traditionally, SSI name spaces have been used in the area of parallel computing. With the advent of cloud computing, SSI name spaces could be used to help manage the large distributed computing resources provided by cloud providers such as Amazon, Goo-

```
// get local resources
loc = mknstree( mkfilesvc("local") );
bin = subtree( loc, "/usr/bin" );
bin = extend( bin, "/bin" );
if( strstr(arch, "64") == nil )
   lib = subtree( loc, "/usr/lib" );
else
   lib = subtree( loc, "/usr/lib64" );
lib = extend( lib, "/lib" );
mytree = merge( [ bin, lib ], overlay );
// place local resources according to context
os = getenv("OSTYPE");
arch = getenv("MACHTYPE");
osarch = "/os/" + os + "/" + arch;
ostree = graft( ostree, mytree, osarch );
root = ostree;
```

(a) Server FINAL Specification

```
/cloud/
      /os/
         /Linux/
               /x86/
                   /{bin,lib}
               /x86_64/
                      /{bin,lib}
         /Solaris/
                 /...
         /...
```

(b) Client Name Space

Figure 4-4. Cloud Resource Organization
(a) Server FINAL specification that prepends OS and architecture to local executable and library locations.
(b) Resulting client name space that groups servers by context.

gle, and Microsoft. Figure 4-4 shows an example specification that uses server local context to organize various systems by the type of service provided (e.g., the operating system and machine architecture). Cloud administrators could use the resulting name space to simplify common tasks such as software installations and updates.

4.4 Scalability Evaluation

Our hypothesis is that we can use FINAL within TBON-FS to improve the efficiency of group file operations by composing a global name space where file groups are automatically defined, and avoiding the serial cost of group definition. To evaluate this hypothesis, we measured and compared the costs of custom name space composition versus group definition in the original TBON-FS name space.

All experiments used Jaguar, a Cray XT5 supercomputer located at Oak Ridge National Laboratory. Jaguar consists of over 18,000 compute nodes, each with two six-core Opteron processors and 16GB of memory, connected with Cray's SeaStar 2 network. Our experiments used overlay tree topologies with up to 46,656 leaves (servers). Servers were run on separate compute nodes from those hosting the internal tree processes, and twelve servers were run on each compute node to virtually increase the number of available hosts. Reported results represent averages of three measurements.

Our first experiment measured the time to construct the global name space at the time of `mount`. We measured the total latency observed at the client, and the per-server time to evaluate the FINAL specification and construct the server's local name space. Figure 4-5 shows the `mount` timing results for four specifications: "orig" is the original TBON-FS name space, while "simple", "gproc", and "cloud" correspond to the specifications of Figures 4-2, 4-3, and 4-4, respectively. The top graph shows the average time in microseconds to construct each server's local name space, and the bottom graph shows the total latency for global name space construction as observed at the client. Across all scales, servers require little time to construct their individual name space, regardless of the input specification. At the client, we see excellent performance, as we are able to compose the global name space for over 45,000 servers in approximately 250 milliseconds.

The second experiment measured the performance of defining and opening file groups. In the original TBON-FS name space, group definition is a costly operation that requires a `stat` and `link` operation for each member file. Group definition thus has linear behavior as shown in Figure 4-6, and can take thousands of seconds for very large groups. Using a custom TBON-FS global name space that automatically creates group directories completely eliminates the substantial cost of defining large file groups.

Once a group directory has been created, TBON-FS clients can use our new `gopen` system call to obtain a group file descriptor that can be used for group file operations such as `read` and `write`. To show that name space composition does not degrade the performance of `gopen`, we measured its latency for both the original TBON-FS name space that isolates servers into separate directory hiearchies, and the custom simple name space that automatically creates group directories. Figure 4-7 shows that `gopen` on groups in the simple name space is almost always faster than using the original name space, an unexpected improvement that we attribute to reduced path resolution time at the servers. The spike in latency for the simple name space near 20,000 servers warrants further investigation, but could be caused by shared network interference from other jobs on the XT according to personal discussions with our partners at ORNL. We also measured the latency of group `read` operations in the original and simple name spaces to confirm that name space composition has no effect on the scalable performance observed previously [4]. As expected, there were no measurable differences to group file operations on established file groups.

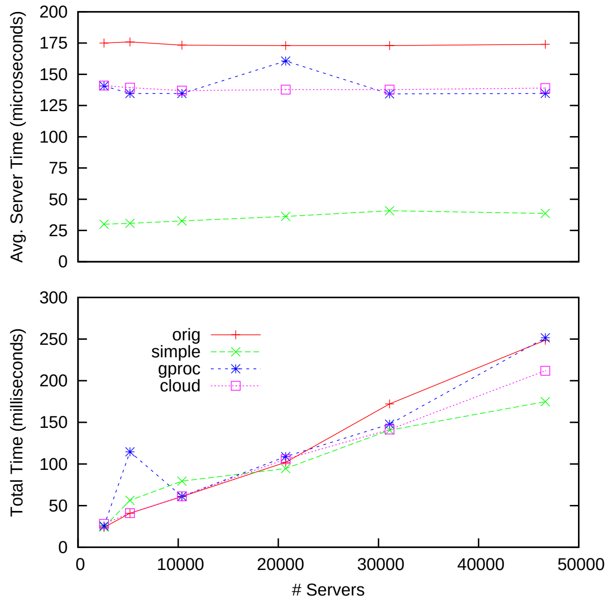

Figure 4-5. `mount` Time: Name space construction
Compares the time required to construct the name space at individual servers and the client using various FINAL specifications.

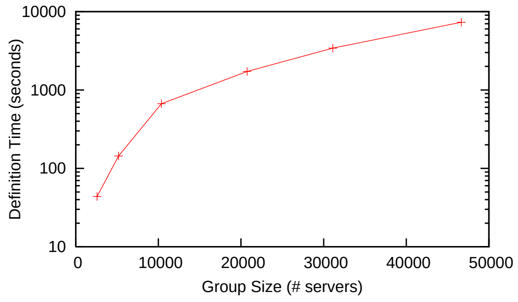

Figure 4-6. Group Definition Time
Time to define the group in the original TBON-FS name space.

5. RELATED WORK

Name space composition has a long history of related work. We focus on three areas of comparison: systems providing either static or flexible name space composition, single-image system name space approaches, and languages or systems that describe custom name spaces.

There are many previous systems that provide either static or flexible name space composition. All UNIX and Linux operating systems provide simple file system mounts that graft a new file

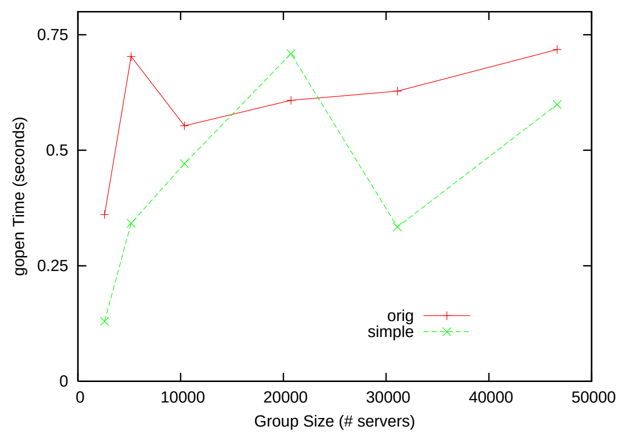

Figure 4-7. gopen Time

Time required to `gopen` a file group in the original TBON-FS name space versus the simple name space that automatically defines the group

system into the local name space. Union mounts that overlay a new file system on the local name space are provided by BSD [17], Plan 9 [18], UnionFS [28], and several user-level file systems built on top of the FUSE framework [24]. All these systems provide fixed name space composition semantics.

Flexible systems for custom name space composition include the Cedar [8], Jade [19], Prospero [15], and Chirp [26] distributed file systems. Cedar has attachments that let users construct custom name spaces containing interesting subsets of the files available in the distributed system. Jade supports arbitrary mounting of logical or physical name spaces using a logical root name space based on skeleton directories. Users can programmatically generate skeleton directories to define the name space. The Virtual System Model underlying Prospero allows users to construct a private view of a global name space as a directed graph. Chirp is a user-level distributed file system for use in Grid or wide-area environments. Chirp provides three distinct name spaces: a global name space with a directory per server, a private name space created by specifying a list of mounts, and shared name spaces that can be created programmatically. Generally, the flexibility provided by these systems comes at the cost of fine-grained name spaced construction, which is not scalable for composing thousands of name spaces. FINAL allows users to describe scalable compositions of large sets of name spaces, while still providing flexibility for prescriptive manipulation when necessary.

Single-system image (SSI) projects have the goal of providing a global name space that includes all the resources of the distributed system. BProc [9] and Mosix [3] limit the global name space to a unified process space, while systems such as LOCUS [27], Kerrighed [14], and OpenSSI [16] provide a distributed file system in addition to the unified process space. Unfortunately, none of these systems permit custom name space composition. Sysman provides a global name space that unifies monitoring and control of a IBM Power clusters, and allows for defining new services that are grafted into the name space [2]. Sysman restricts where new services are inserted, and doesn't provide any mechanism for altering its default name space organization. By integrating FINAL into TBON-FS, we give clients the flexibility to construct custom SSI name spaces.

FINAL has its roots in the FLAC language we developed in the past for providing mobile clients with an unchanging name space [31]. FINAL's design addresses some of the deficiencies in describing interesting name space compositions in the purely declarative FLAC, while still providing its intuitive name space composition based on tree operations.

Both FUSE [24] and FIST [29] simplify the development of new, custom file systems. FUSE provides interfaces that closely resemble the file system layer of the operating system, and requires custom file systems to describe their name space programmatically. FIST provides a custom language that mixes C and yacc syntax to describe interactions of stackable file systems. Rules are used for describing the actions taken during file system operations, and filters allow for manipulating data or the name space passed between layers. Because both FUSE and FIST require low-level descriptions based on file system interfaces, neither system allows for easy description of name space composition.

Semantic file systems [7] provide dynamic views of name spaces using virtual directories generated by queries of file attributes. These dynamic views essentially provide transparent name space composition, although limited by the type of queries that can be made and the attributes supported. FINAL provides both attributed-based composition, as well as more flexible whole name space composition.

6. CONCLUSION

We have designed FINAL to enable clear, flexible, and scalable description of name space composition using an intuitive tree abstraction and operations that provide common tree manipulations. By integrating the use of FINAL specifications with our TBON-FS distributed file system, we have shown how to build custom single-system image name spaces. Our experiments demonstrate that our approach is scalable for composing global name spaces from tens of thousands of independent server name spaces. Moving forward, we plan to investigate any limitations of our tree operations, focusing on our `merge` semantics, in the context of building custom SSI name spaces.

7. REFERENCES

[1] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz, "Stack Trace Analysis for Large Scale Applications", *International Parallel and Distributed Processing Symposium (IPDPS '07)*, Long Beach, California, March 2007.

[2] Mohammad Banikazemi, David Daly, and Bulent Abali, "Sysman: A Virtual File System for Managing Clusters", *22nd Large Installation System Administration Conf. (LISA '08)*, pp. 167-174, San Diego, CA, November 2008.

[3] Amnon Barak and Oren La'adan, "The MOSIX multicomputer operating system for high performance cluster computing", *Future Generation Computer Systems* **13**, 4-5, March 1998, pp. 361-372.

[4] Michael J. Brim and Barton P. Miller, "Group File Operations for Scalable Tools and Middleware", *16th Intl. Conf. on High-Performance Computing (HiPC 2009)*, Cochin, India, December 2009.

[5] D.R. Brownbridge, L.F. Marshall, and B. Randell, “The Newcastle Connection or UNIXes of the World Unite!”, *Software-Practice and Experience* **12**, 1982, pp. 1147-1162.

[6] “Cinquecento Manual”, http://cqctworld.org/src/l1/doc/man.html.

[7] David K. Gifford, Pierre Jouvelot Mark A. Sheldon, and James W. O'Toole, “Semantic file systems”, *SIGOPS Oper. Syst. Rev.* **25**, 5, October 1991, pp. 16-25.

[8] David K. Gifford, Roger M. Needham, and Michael D. Schroeder, “The Cedar File System”, *Communications of the ACM* **31**, 3, March 1988, pp. 288-298.

[9] Erik Hendriks, “BProc: The Beowulf distributed process space”, *Intl. Conf. on Supercomputing*, pp. 129-136, New York, NY, June 2002.

[10] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, “Scale and performance in a distributed file system”, *ACM Trans. on Computer Systems* **6**, 1, 1988, pp. 51-81.

[11] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores", *Supercomputing 2008 (SC2008)*, Austin, TX, November 2008.

[12] Lucent Technologies, “Introduction to the Plan 9 File Protocol”, 2010, http://plan9.bell-labs.com/magic/man2html/5/0intro.

[13] Ronald G. Minnich, “9.2u: A User-mode Private Name Space system for Unix”, DARPA Technical Report, September 1998.

[14] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, R. Badrinath, and Louis Rilling, “Kerrighed: A Single System Image Cluster Operating System for High Performance Computing”, *9th International Euro-Par Conference (Euro-Par 2003)*, Klagenfurt, Austria, August 2003. Appears as Lecture Notes in Computer Science **2790**, Harald Kosch et al (Eds.), Springer, Berlin/Heidelberg, Germany, January 2003, pp. 1291-1294.

[15] B. Clifford Neuman, “The Prospero File System: A Global File System Based on the Virtual System Model”, *Computing Systems* **5**, 1992, pp. 407-432.

[16] “OpenSSI (Single System Image) Clusters for Linux”, http://www.openssi.org/, August 2006.

[17] Jan-Simon Pendry and Marshall Kirk McKusick, “Union mounts in 4.4BSD-lite”, *USENIX Technical Conference*, New Orleans, LA, 1995.

[18] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, “The Use of Name Spaces in Plan 9”, *ACM SIGOPS Oper. Sys. Review* **27**, 2, April 1993, pp. 72-76.

[19] Herman C. Rao and Larry L. Peterson, “Accessing Files in an Internet: The Jade File System”, *IEEE Trans. on Software Engineering* **19**, 6, 1993, pp. 613-624.

[20] Dennis M. Ritchie and Ken Thompson, “The UNIX time-sharing system”, *Communications of the ACM* **26**, 1, 1983, pp. 84-89.

[21] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller, “MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools”, *Supercomputing 2003 (SC’03)*, Phoenix, Arizona, November 2003.

[22] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, “Design and Implementation of the Sun Network File System”, *USENIX Technical Conference*, pp. 119-130, Portland, OR, June 1985.

[23] Richard Sharpe, “Just what is SMB?”, October 2002, http://samba.anu.edu.au/cifs/docs/what-is-smb.html.

[24] Miklos Szeredi, 'FUSE: Filesystem in user space”, http://fuse.sourceforge.net.

[25] Douglas Thain, “Chirp Protocol Version 2”, June 2008, http://www.cse.nd.edu/~ccl/software/manuals/chirp_protocol.html.

[26] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes, “Chirp: a practical global filesystem for cluster and Grid computing”, *Journal of Grid Computing* **7**, 1, March 2009, pp. 51-72.

[27] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, “The LOCUS distributed operating system”. *SIGOPS Oper. Syst. Rev.* **17**, 5, December 1983, pp. 49-70.

[28] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair, “Versatility and Unix Semantics in Namespace Unification”, *ACM Trans. on Storage* **2**, 1, February 2006, pp. 74-105.

[29] Erez Zadok and Jason Nieh, “FIST: A Language for Stackable File Systems”, *USENIX Annual Technical Conference*, June 2000.

[30] Vic Zandy and Dan Ridge, “First-class C Contexts in Cinquecento”, IDA CCS Technical Report, April 2008. http://cqctworld.org/docs/cqct.pdf.

[31] Victor C. Zandy, "Application Mobility", Ph.D. dissertation, University of Wisconsin-Madison, 2004.