# StepAhead: Rethinking Filesystem Namespace Translations

Debadatta Mishra
Indian Institute of Technology Bombay
deba@cse.iitb.ac.in

Purushottam Kulkarni
Indian Institute of Technology Bombay
puru@cse.iitb.ac.in

Raju Rangaswami
Florida International University
raju@cs.fiu.edu

**Abstract**

A hierarchical namespace is a common abstraction used for data organization within modern file systems. Fast translation of namespace objects to physical locations is necessary to carry out efficient file system operations. For reasons attributed to modularity, security, and to some extent legacy, namespace translations involves iterative translation of intervening directory objects from the root of the namespace. Namespace resolution is typically a multi-step process, potentially involving serialized I/O operations at each step.

In this paper, we propose a *rethink* of the strategy to fetch pathname entries. Our technique, `StepAhead`, proactively utilizes hints about namespace translation lookup failures to enable parallel and just-in-time fetching of necessary path translation data into memory to increase cache hits significantly. With `StepAhead`, we measure an increase in cache hit rates for path translation data across a set of six workloads by as much as 51%, which in turn results in application speed-up of as much as 20%.

## 1. Introduction

File systems provide a hierarchical namespace abstraction consisting of directories and files. The resolution of pathname targets involves fetching file system meta data and data blocks. This resolution is a multi-step process. Each *directory entry* in the path (from root of the namespace tree to the target file system object) is first translated to an `inode` object, access permissions are checked, and the contents of the directory are fetched in order to resolve the next entry in the path; this continues in a *serial* manner for subsequent directory entries until the target's meta data is resolved. Thus, pathname translation potentially requires serialized I/O operations for translating each entry of the target's path.

The costs involved in pathname resolution within file systems have been dominated by meta data related operations as documented in prior works [15, 22, 26, 28]. For example, Lensing et al. reported up to two times improvement in application performance when all meta data was cached in memory.

To minimize I/O operations during pathname resolution, most operating systems employ translation caches [26] (e.g., the Linux `dentry` cache) to store the recently resolved entries. However, such caches have limited effectiveness. First, while the translation cache can reduce the translation overheads for the frequently accessed path entries, cache size limitations restrict such efficacy to extend to all path translations. Further, a translation entry's lifetime in the cache is dependent on presence of the associated file system object in memory (elaborated in §2). To improve effectiveness of the cache and to reduce the probability of I/O required for translation lookups, translation mechanisms also implement storage-local prefetching [5, 7] of spatially co-located translation meta data [16, 19]. Benefits of such prefetching are based on two fundamental assumptions: *(i)* file translation meta data for spatially located files in the namespace tree exhibit spatial locality on storage, and *(ii)* namespace entries that exhibit spatial locality in the namespace tree also exhibit temporal locality of access. The first assumption may fail for a sufficiently aged file system [2, 23] name space and the second is highly workload dependent. We demonstrate the limited effectiveness of existing translation caches and conventional storage-locality based meta data prefetch techniques in §2.

In this paper, we explore an alternate proposition: *Can pathname lookups be made efficient through parallel prefetch of necessary disk data required for successful translation?* To enable parallel prefetch of translation data, the file system can determine the disk blocks required for the complete path translation to fill the cache before the corresponding pathname entries are requested. Doing so, however, is non-trivial. In most Unix-like operating systems, the generic path translation layer (e.g., VFS layer of Linux) is decoupled from the file system implementation. Traditionally, the seri-


*APSys 2016*,, August 04-05, 2016, Hong Kong, Hong Kong..
 ISBN 978-1-4503-4265-0/16/08. . . $15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2967360.2967370

alized path resolution never requires meta data corresponding to more than one path entry at every step. We propose a space efficient file system *knowledge base* which can be used by the path translation layer to fetch the required disk blocks of multiple path entries. We demonstrate that the benefits of using such knowledge is significant; the cache hit ratio of translations is improved by up to 50% hit resulting in workload speed ups of 8-20%.

We present `StepAhead`, a solution that populates the in-memory cache with path translation data *just-in-time*. All the required data block read requests are queued simultaneously to enable *parallel fetching* of meta data. `StepAhead` extends the generic VFS abstractions to enable file system specific *accurate* parallel prefetch implementations and improve the VFS path translation performance (§3). We evaluate different filesystem workloads (§4) to explore the extent of benefits and overheads.

Our prototype implementation leverages the flexibilities of virtualized systems to provide a minimally intrusive solution using virtual machines (VMs). The hypervisor builds the knowledge base for the guest VM file system through virtual disk image introspection [17, 24]. The VFS extensions are implemented through hypercalls, initiated from within the virtual machine.

## 2. Inefficiency of Pathname Lookups

Most POSIX compliant filesystems represent and expose files and directories within a hierarchical namespace in the form of a tree. This hierarchical structure induces translation dependencies across objects in the hierarchy leading to inefficiencies in pathname lookup operations.

### 2.1 Anatomy of a Pathname Lookup

With UNIX-like operating systems, a *virtual file system* (VFS) layer acts as the interface between the user space API and the actual implementation of the filesystem. Translation of a VFS path to a file system object—file or directory, requires translation of each intermediate entry of the path until the entry containing the target object. Most significantly, the translation of intermediate components of the path is serialized as each path entry has to be validated and the corresponding `inode` or data block location has to be identified. This verification involves retrieving the data associated with intermediate (directory) objects, which in turn may require access to the disk through the filesystem. To minimize the translation overheads due to disk access, most UNIX-like systems cache the frequently accessed intermediate directories as memory objects (e.g., `dentry` objects in Linux) [3, 26] [1].

An illustration of the VFS and filesystem operations corresponding to opening the file `/home/userB/images/2.jpg` (for a filesystem mounted at `/`) is shown in Figure 1. Assuming that the entries for `/`, `home` and `userB` are available in the `dentry` cache, the sequence of serial I/O operations needed for remainder of the path translation are shown. When the `dentry` lookup for `images` is unsuccessful, the VFS subsystem requests the filesystem to create a `dentry` object for `images`. A typical filesystem like EXT4 [16] would require two I/O operations, *(i)* fetch the necessary directory content of the directory `userB` to translate the name (i.e., `images`) to an inode, and *(ii)* fetch the disk inode object for `images`. The name resolution of `2.jpg` cannot start until the name resolution for `images` is complete. Resolution of `2.jpg` also requires two accesses to disk, as shown in part 3 of Figure 1. This *serialized* execution can result in significant overheads with each step requiring possible disk-access for resolution.

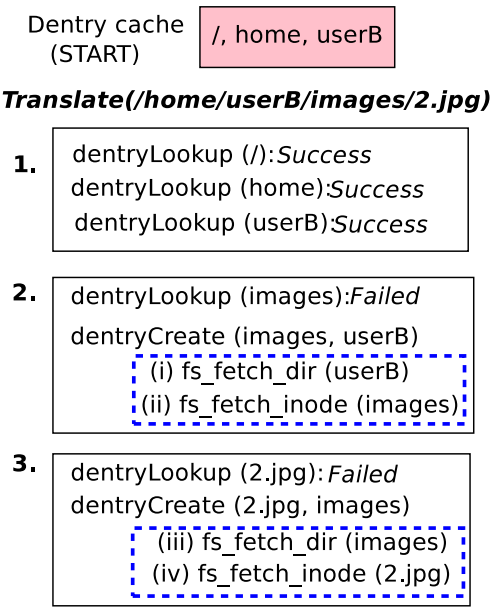


Figure 1: Serial I/O operations required to translate `/home/userB/images/2.jpg` with initial valid dentries for `/, home, userB` in the `dentry` cache. `images`, not found in the `dentry` cache, triggers *four I/O operations in total*, *two* involving directory data and *two* involving inodes.

### 2.2 Effectiveness of Translation Caches

Translation caches [3, 26] serve limited purpose because of the following reasons. *First*, the translation caches can not serve first time lookups in a *cold* cache scenario. *Second*, translation caches compete for memory in the system against other caches [18, 30] (e.g., disk block cache or page cache) and may be evicted due to memory pressure over the application's life time. *Third*, even though the memory footprint of `dentry` cache is small, a `dentry` is tightly coupled with file system objects of *significant size*. The tight coupling is a result of inherent dependencies across multiple independent OS software layers performing pathname resolution, man-

[1] In this paper `dentry` is used to denote an in-memory directory or file translation entry.

| Workload | % of failed dentry-lookups | % of prefetched inodes un-used |
|---|---|---|
| boot | 17 | 22 |
| webserver | 73 | 92 |
| find | 33 | 54 |
| ccode | 64 | 27 |
| fileserver | 60 | 72 |

Table 1: Performance of dentry cache and static storage-local prefetching.

age retrieved pathname objects, and cache pathname objects. For example, in Linux, a dentry-cache entry for a file depends on the parent directory data (content) block and the inode corresponding to the file dentry, cached and managed by the file system. If an inode memory object is evicted out of memory, the translation entries of paths containing the inode will need to be evicted as well to ensure meta data/data consistency of files.

Serialized name resolution occurs upon misses of intermediate objects in the dentry cache. To understand the significance of dentry cache misses, we executed several workloads and measured the occurrences of name resolution failures. To cover the cold cache situation, we used the system boot and find workloads with an empty dentry cache. The webserver and fileserver workloads are load profiles of the Filebench [1] benchmark. ccode is a multi-threaded file read workload where every thread picks a random file within the Linux source tree to open and read. The experiment was conducted on a Linux machine with 512 MB RAM using the EXT4 filesystem. Table 1 shows the number of name resolution failures (at any level of translation). The total number of path translations requested for boot, webserver, find, ccode and fileserver are 42K, 378K, 12.5K, 174K and 239K, respectively. On average, across these workloads, 50% of the name resolution operations fail. The number of failed lookups is significant for the webserver, fileserver and ccode workloads. The dentry failures of webserver, fileserver and ccode workloads were not only due to the dentry cache miss for the first time access but because of the cache contention between the data and meta data in the steady state.

### 2.3 Effectiveness of Storage-local Prefetch

Some filesystems try to localize the inodes of a directory tree in the disk block space at the time of creation. The existing prefetching mechanisms for file block caches, therefore, apply spatial locality based prefetching to improve sequential reads. Upon an inode read from the disk, the surrounding blocks containing inodes are prefetched to expedite future inode lookups in the same directory tree [16]. This approach can be effective if the data access pattern is sequential or accessed in chunks in a relatively young file system.

The effectiveness of such *static prefetching* for path lookups, however, is limited because it relies on two additional assumptions. First, the built-in assumptions that *(i)* there is significant temporal and spatial locality of creation time of files and directories within a directory tree, and *(ii)* second, that false positives due to the application usage pattern, do not result in significant wasted memory space and do not induce I/O congestion. Table 1 demonstrates that the extent of memory wasted in prefetching inodes that would not be eventually used is between 10 and 80% across the five workloads. Even more significantly, since inode grouping is preferentially sibling-oriented, static prefetching may not directly expedite future inode lookups of inodes at lower levels of the namespace hierarchy, thereby compromising its effectiveness for path lookups. Furthermore, *directory content access* in the process of the path resolution does not benefit from spatial locality based prefetching because *(i)* it is difficult and restrictive to organize directory data in consecutive storage blocks, and *(ii)* the exact access path is not known at the time of creation of the directory tree.

Optimizations to address the limitations of dentry cache and storage local prefetching should, *(i)* improve the path translation performance while preserving the benefits of dentry cache, *(ii)* perform *targeted* (pre)fetch of disk blocks before they are required by the path translation process. To meet the above objectives, StepAhead performs *parallel* fetch of disk blocks [27] in a *targeted* manner in the event of dentry lookup failures.

## 3. The StepAhead Approach

StepAhead is designed to perform targeted prefetching of only the disk blocks necessary for invoked namespace translation operations. It is designed for zero cache pollution and *efficient cache population* via parallel prefetching of filesystem meta data necessary for namespace translations. In this section, we discuss aspects of StepAhead's design and challenges to realize them.

### 3.1 Design and Architecture

Figure 2 presents the architecture of StepAhead. At its core, StepAhead enables the parallel fetch of disk blocks required for translating the residual path fragment of a requested namespace object when the *first* namespace object lookup failure is encountered. While we use Linux/UNIX specific terms to make our discussion of the design more concrete, StepAhead's design principles apply more broadly.

The Virtual File System (VFS) interface provides abstract methods, implemented by actual file systems, to perform low level file and directory operations. StepAhead proposes extensions to the VFS interfaces to provide dentry cache look up *failure hints* to the file system. With StepAhead, the file system layer uses the VFS layer hint to fetch *necessary* disk blocks corresponding to the file system objects (inodes and

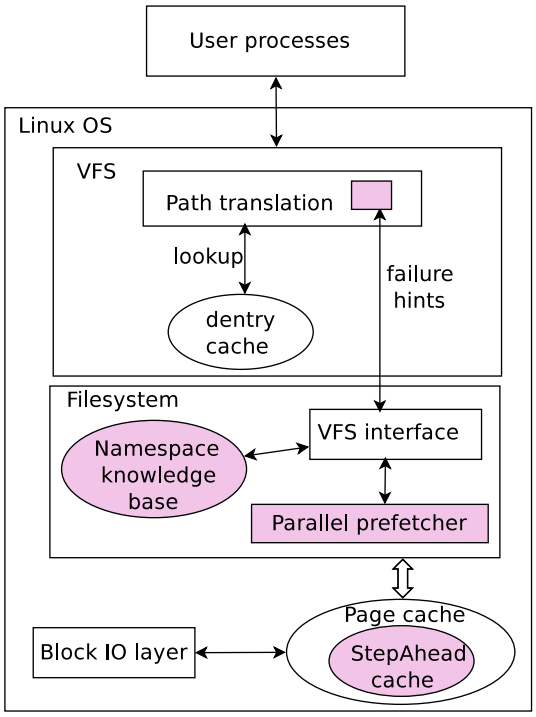


Figure 2: File system parallel prefetching with VFS layer hints on `dentry` cache lookup failure. Shaded components represent `StepAhead` additions/modifications.

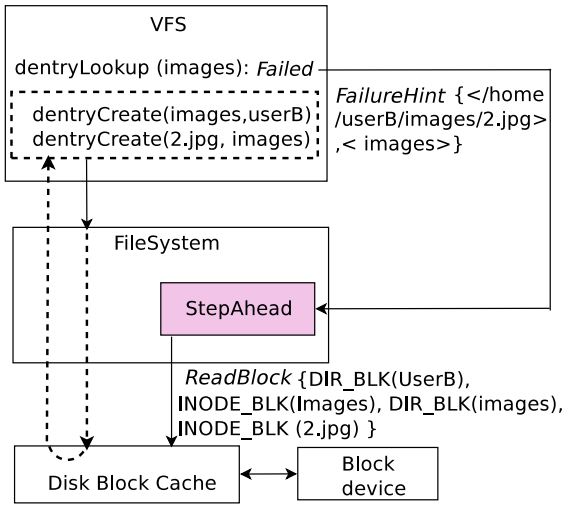


Figure 3: `StepAhead` optimizations in translation of `/home/userB/images/2.jpg` when `images` not found in `dentry` cache. INODE_BLK and DIR_BLK represent the disk blocks containing the inode data and the directory content, respectively.

`directory data blocks`) required for successful translation of the residual path.

`StepAhead` builds and utilizes a namespace knowledge base (as shown in Figure 2) to prefetch the necessary disk blocks required for namespace translation. The namespace knowledge base identifies the mapping between a namespace object and the corresponding disk blocks as maintained by the file system. The knowledge base can be created either by a walk of the name space tree during the file system initialization or can be progressively built by observing the name space object translation operations throughout the life time of the file system. We use the former approach in a non-intrusive prototype implementation explained in (§3.2). The name space knowledge base tree maintains block address information pertinent to each name space entry (files and directories). More specifically, block addresses of the following types of entries are stored in the knowledge base: *(i)* file system storage primitives like super blocks and `inodes`, *(ii)* file or directory translation information (e.g., indirect blocks), and *(iii)* directory data.

The parallel prefetcher issues all the required block I/O requests through the page cache layer to the block I/O layer. Upon successful prefetch, file system objects for the residual path requested by the VFS step-wise translation can be served from the page cache. Consider the example path translation presented in Figure 1 for path `/home/userB/images/2.jpg` where the first `dentry` lookup failure occurs for `images`. The corresponding `StepAhead` processing in such a scenario is shown in Figure 3. The VFS layer provides a failure hint with the complete path and the lookup entry that failed, e.g., in the example above, the hint is the tuple {<`/home/userB/images/2.jpg`>, <`images`>}.

The sequence of `dentry` lookup (and create) operations in the VFS layer remain unaltered while the `StepAhead` extension of the file system issues block IO read requests for—directory data block(s) for `userB`, block containing the `inode` for `images`, directory data block(s) for `images` and the block containing the inode for `2.jpg`. All these IO requests are serviced in a parallel and asynchronous manner w.r.t. the `dentry` operations during the path lookup. Therefore, in an ideal scenario, I/O requests after the `userB` directory data block read can be served from the page cache (as shown by a dotted round trip in Figure 3) resulting in efficient VFS path translation.

Effectiveness of the prefetch operation in improving namespace translation cache hit rates depends on the accuracy of the namespace knowledge base. One challenge involves keeping the namespace knowledge base up-to-date in presence of namespace updates. However, it is important to note that the correctness of the namespace knowledge base is *desirable but not mandatory*; it improves translation performance when accurate but does not affect the correctness of file system operations when the knowledge base in not accurate. In the worst case, the parallel prefetch operations based on out-of-date namespace knowledge base information may be inaccurate. In such cases, namespace translation would

simply fetch the correct disk blocks corresponding to file system objects when they are not found in the StepAhead cache, without compromising correctness.

### 3.2 Implementation Details

We implemented a prototype of StepAhead for virtualized systems using the Linux-KVM virtualization solution. Flexibilities enabled due to virtualization are useful to implement StepAhead in virtualized systems to realize a quick initial prototype. Our implementation avoids any intrusive changes in the file system (of the virtual machine) and implements the StepAhead functionalities for the virtual machine at the hypervisor using *virtual machine image introspection* and the host machine's (filesystem) page cache.

In a virtualized platform, the virtual machine disk devices are backed by virtual machine image files in the hypervisor. From the hypervisor, the virtual machine image files (or logical disk partition) can be accessed and analyzed [17, 24, 25] using simple user level file IO libraries and system calls. Virtual disk image introspection [17, 25] is used to extract name space knowledge from the virtual machine disk image. Given the knowledge of the format for storing image files and the filesystem used by guest OS to manage files, the hypervisor can build the namespace knowledge base for the guest OS's filesystem. We have implemented name space knowledge extraction for the EXT4 file system [16] from the virtual machine disk image files. The knowledge base is created and periodically refreshed for the virtual machine under test.

The VFS layer of the guest operating system is modified to invoke a *hypercall* on a dentry lookup failure to realize the *failure hint* notification (§3.1). The hypercall handler of the KVM hypervisor implements the parallel fetch through namespace knowledge lookup to populate the hypervisor page cache. In steady state, VFS dentry lookup failures are passed to the KVM hypervisor, the namespace knowledge base is consulted for the block mappings and disk blocks are fetched in parallel. A dedicated Linux host process receives information about lookup failures via the hypercall handler inside the kernel and implements StepAhead's parallel prefetching. The fetched disk blocks are stored in the host machine's page cache and are used to serve the guest OS's serialized VFS translation requests. This is possible because the disk accesses from within the virtual machine traverse via the host page cache which is used as the StepAhead cache (Figure 2) in our implementation.

## 4. Evaluation

We evaluated StepAhead using a machine equipped with an Intel Xeon E5507 processor (8 cores) and 8 GB of physical memory running Ubuntu 12.04 Linux server with kernel version 3.10. We used the KVM hypervisor to create a virtual machine with 512MB RAM, 1 VCPU and a virtual disk of size 24 GB with two EXT4 partitions. We used well known file system utilities—find, du, stat, touch, grep and reading of directories (referred to as readdir), as workloads operating on different segments of the virtual machine namespace (Table 2). The host machine's page cache (operating as a second chance cache underneath the guest) cached the disk blocks fetched by StepAhead.

| Workload | # of files | # of directories |
|---|---|---|
| find | 365000 | 31751 |
| du,stat and touch | 260042 | 17507 |
| readdir | 0 | 25788 |
| grep | 2250 | 95 |

Table 2: Workloads and the operation domains.

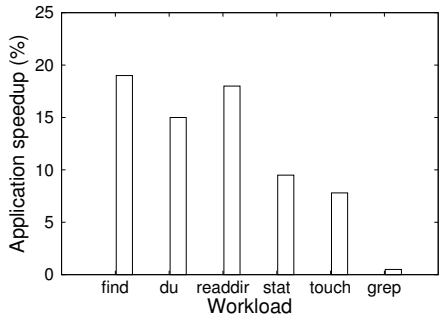


Figure 4: Application speedup with StepAhead.

### 4.1 Effectiveness of StepAhead

Workload speedup with a cold cache is shown in Figure 4 and is compared against a baseline system with same configurations for the guest and hypervisor page caches without StepAhead optimizations. Relative to the baseline system, StepAhead improved application performance by 19%, 18% and 15% for find, readdir and du, respectively. For a data intensive application like (grep), StepAhead did not result in any noticeable application improvements due to the minimal presence of namespace translation operations relative to file data fetch operations. Namespace translation cache hit ratio improvements with StepAhead over the baseline are shown in Figure 5. We verified that application speed up improvements for find, du and readdir were on account of StepAhead's effectiveness in improving namespace translation cache hit rates by 51%, 50% and 43%, respectively. These improvements are attributed to VFS layer hints which in turn resulted in fetching the translation data in one go. For example, for readdir-workload, $\sim$21000 blocks containing inodes and $\sim$36000 directory data blocks were fetched using parallel prefetching technique of StepAhead.

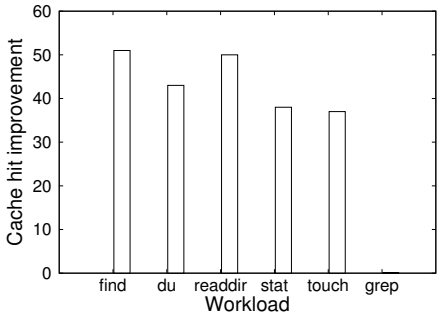


Figure 5: Cache hit improvement with StepAhead.

| Disk image size (GB) | #of files and directories | dentry cache size (MB) | StepAhead knowledge base size (MB) |
|---|---|---|---|
| 8 | 80K | 152 | 5 |
| 16 | 200K | 259 | 10 |
| 24 | 275K | 412 | 14 |
| 40 | 470K | 662 | 23 |

Table 3: Memory requirements to store all entries in dentry cache vs. size of the StepAhead knowledge base.

### 4.2 Overhead Comparisons

The overheads of StepAhead involve two resources, the memory required to store the namespace knowledge base and the CPU cycles required for path matching in order to identify the blocks to prefetch. In Table 3, memory requirements of StepAhead's knowledge base for different disk image sizes and varying number of files and directory are shown. The memory requirement for the namespace knowledge base is directly proportional to the number of files in the disk partition. On the other hand, to maintain entries in dentry cache for in-memory translation of every file and directory, the memory requirement is significantly more ($\sim$25x) compared to the StepAhead knowledge base size. For workloads used in this experiment (§4.1), the CPU overhead of the StepAhead process was negligible (5% CPU utilization in the worst case). We expect the CPU overheads to further reduce for a native system implementation of StepAhead.

## 5. Related work

Several research efforts have proposed hashing based techniques [14, 15, 26] to reduce meta data lookup latencies. Full path hashing based resolution techniques present challenging issues like access control and prefix checks requiring intrusive changes to VFS functionalities. Our approach is transparent to the VFS layer functionalities (e.g., access control mechanisms) and is complimentary to translation cache optimizations. Meta data storage and management in specialized structures (e.g., key-value stores) for large scale distributed systems [3, 4, 10, 29, 33] operate at a higher level of abstraction (on top of the local file systems). StepAhead is orthogonal to these solutions.

File system organization improvements like multi-level directory partitioning [30], coalition of meta data object for—files in a directory [11] and semantically related files [29, 31, 32], have been proposed to improve the efficiency of meta data operations. While in principle these are sound optimizations, ubiquitous file systems are slow in adopting the proposed changes because of the inherent complexities and lack of generality. Extent of StepAhead benefits will depend on the meta data placement strategy and therefore will be an interesting evaluation of StepAhead.

Preferential treatment of meta data using differentiated caching strategies [18, 19, 30] and usage of faster persistent storage devices (like SSD devices [6]) can improve access speeds of file system meta data. Effects of these optimizations are dependent on the file system size and individual memory load scenarios. StepAhead is complementary to these efforts. Our approach is similar to optimistic path resolution [8] and meta data prefetch [25] at NFS clients to address the overheads of multiple network round trips. StepAhead's basic proposition borrows from these ideas and proposes a generalized solution for local file systems.

Prefetching techniques exploring spatial locality of access [5, 9], multiple file stream correlations [7, 12], application directed prefetching [12, 27] and parallel prefetching [27] are interesting optimizations with respect to improving file read performance. StepAhead is capable of going beyond the meta data related optimizations to incorporate some of the above functionalities.

Virtual machine introspection has been used for robust VM image management [17] and monitoring [21], to study and exploit sharing opportunities [13, 24] and locating meta data [25] in a VM image. Privacy sensitive requirements for virtual machine image introspection have also been proposed [20]. StepAhead's usage of VM image introspection is limited to verification of the proposed file system optimizations in a file system non-intrusive manner.

## 6. Discussion

**Namespace knowledge management:** File systems can create and manage the knowledge space by—explicit name space tree walk, or by iterative knowledge gathering over a period of time. Namespace updating can be tricky to manage but can be addressed through periodic knowledge rebuilding or targeted updates to the changed namespace sub trees. Since the file system manages all meta-data updates, a targeted in-band update of the knowledge base also seems feasible. For a virtualized system implementation (as is the case for this implementation), monitoring writes to meta data blocks (e.g., directory content) from the host to selectively update the knowledge base in-band can be implemented.

**Overhead considerations:** StepAhead is designed as an *out-of-band* optimization to ensure correctness. Incorrect path name lookups in the VFS are detected *only when* the first incorrect entry in the path is encountered. StepAhead complements the VFS operations by fetching the meta-data for path components only till the first incorrect entry in the path. Access permissions related issues could reduce the utility of parallel fetching, which we believe is usage dependent and needs to be characterized. StepAhead is cognizant of disk bottlenecks and disable the prefetch optimizations in the event of disk congestion (determined by checking the device state information).

**Extensions for just-in-time data fetching:** The VFS layer hints can also be used to fetch *starting* data blocks along with the meta data blocks, of the file pointed to by a path. A VFS dentry path lookup failure provides an indication regarding the absence of data blocks of the file pointed to by the path in the page cache. This intuition can be used to guide the prefetching related extensions within StepAhead, i.e., prefetch data blocks as well as meta data blocks. Further, StepAhead can be used to estimate access patterns and their correlations across files and design prefetching techniques beyond sequential single file prefetching [7].

## 7. Conclusions

StepAhead is a new approach to speed up namespace translations within modern file systems via file system introspection. It is based on the observation that the serialized I/O involved in pathname translations can be eliminated and replaced with targeted, parallel, just-in-time fetching of all meta-data required for successful translation. StepAhead builds and utilizes its namespace knowledge base to identify and prefetch in parallel the disk blocks required for namespace translation at the first instance of a lookup failure during a translation operation. While the initial results with StepAhead are promising, much work remains to be done in fully exploring and evaluating this direction for applicability in production systems.

## Acknowledgements

We would like to thank our shepherd, Yunxin Liu, and the anonymous referees for their helpful comments. This work is supported in part by NSF awards CNS-1448747 and CNS-1563883, and an Intel ISRA award.

## References

[1] Filebench. www.filebench.sourceforge.net/wiki/index.php/Main_Page.

[2] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *Transaction on Storage 3*, 3 (2007).

[3] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (2010), pp. 1–8.

[4] BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND XUE, L. Efficient metadata management in large distributed storage systems. In *Proceedings of IEEE conference on Mass Storage Systems and Technologies (MSST)* (2003), pp. 290–298.

[5] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. *SIGMETRICS Performance Evaluation Review 23*, 1 (1995), 188–197.

[6] CLOUDBYTE. Moving metadata to flash memory. http://www.cloudbyte.com/wp-content/uploads/2015/05/Metadata-acceleration-CloudByte.pdf.

[7] DING, X., JIANG, S., CHEN, F., DAVIS, K., AND ZHANG, X. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *Proceedings of the USENIX Annual Technical Conference* (2007), pp. 20:1–20:14.

[8] DUCHAMP, D. Optimistic lookup of whole nfs paths in a single operation. In *Proceedings of the USENIX Summer Technical Conference (UTSC)* (1994).

[9] FENGGUANG WU, H. X., AND LI, J. Linux readahead: less tricks for more. In *Proceedings of Ottawa Linux Symposium* (2007).

[10] FU, S., HE, L., HUANG, C., LIAO, X., AND LI, K. Performance optimization for managing massive numbers of small files in distributed file systems. *IEEE Transactions on Parallel and Distributed Systems 26*, 12 (2015), 3433–3448.

[11] GANGER, G. R., AND KAASHOEK, M. F. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the USENIX Annual Technical Conference* (1997).

[12] HE, J., BENT, J., TORRES, A., GRIDER, G., GIBSON, G., MALTZAHN, C., AND SUN, X.-H. Io acceleration with pattern detection. In *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing* (2013), pp. 25–36.

[13] JAYARAM, K. R., PENG, C., ZHANG, Z., KIM, M., CHEN, H., AND LEI, H. An empirical analysis of similarity in virtual machine images. In *Proceedings of the Middleware Industry Track Workshop* (2011), pp. 6:1–6:6.

[14] LENSING, P., MEISTER, D., AND BRINKMANN, A. hashfs: Applying hashing to optimize file systems for small file reads. In *IEEE workshop on Storage Network Architecture and Parallel I/Os (SNAPI)* (2010), pp. 33–42.

[15] LENSING, P. H., CORTES, T., AND BRINKMANN, A. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of Systems and Storage Conference (SYSTOR)* (2013), pp. 5:1–5:11.

[16] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., VIVIER, L., AND S, B. S. A. The new ext4 filesystem: current status and future plans. In *Proceedings of Ottawa Linux Symposium* (2007).

[17] REIMER, D., THOMAS, A., AMMONS, G., MUMMERT, T., ALPERN, B., AND BALA, V. Opening black boxes: Using semantic information to combat virtual machine image sprawl.

In *Proceedings of the Conference on Virtual Execution Environments (VEE)* (2008), pp. 111–120.

[18] REN, K., AND GIBSON, G. Tablefs: Embedding a nosql database inside the local file system. In *APMRC Digest* (2012), pp. 1–6.

[19] REN, K., AND GIBSON, G. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference* (2013), pp. 145–156.

[20] RICHTER, W., AMMONS, G., HARKES, J., GOODE, A., BILA, N., DE LARA, E., BALA, V., AND SATYANARAYANAN, M. Privacy-sensitive vm retrospection. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud)* (2011).

[21] RICHTER, W., ISCI, C., GILBERT, B., HARKES, J., BALA, V., AND SATYANARAYANAN, M. Agentless cloud-wide streaming of guest file system updates. In *Proceedings of the nternational Conference on Cloud Engineering (IC2E)* (2014), pp. 7–16.

[22] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference* (2000), pp. 1–15.

[23] SMITH, K. A., AND SELTZER, M. I. File system aging—increasing the relevance of file system benchmarks. *SIGMETRICS Performance Evaluation Review 25*, 1 (1997), 203–213.

[24] SUNEJA, S., ISCI, C., DE LARA, E., AND BALA, V. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the Conference on Virtual Execution Environments (VEE)* (2015), pp. 133–146.

[25] TARASOV, V., JAIN, D., HILDEBRAND, D., TEWARI, R., KUENNING, G., AND ZADOK, E. Improving i/o performance using virtual disk introspection. In *Proceedings of the USENIX Conference on HotStorage* (2013), pp. 1–5.

[26] TSAI, C.-C., ZHAN, Y., REDDY, J., JIAO, Y., ZHANG, T., AND PORTER, D. E. How to get more value from your file system directory cache. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2015), pp. 441–456.

[27] VANDEBOGART, S., FROST, C., AND KOHLER, E. Reducing seek overhead with application-directed prefetching. In *Proceedings of the USENIX Annual Technical Conference* (2009).

[28] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of EuroSys* (2014), pp. 14:1–14:14.

[29] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 307–320.

[30] XING, J., XIONG, J., SUN, N., AND MA, J. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), pp. 26:1–26:11.

[31] ZHANG, S., CATANESE, H., AND WANG, A.-I. A. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *Proceedings of the Usenix Conference on File and Storage Technologies (FAST)* (2016), pp. 15–22.

[32] ZHANG, Z., AND GHOSE, K. hfs: A hybrid file system prototype for improving small file and metadata performance. *SIGOPS Operating Systems Review 41*, 3 (2007), 175–187.

[33] ZHU, Y., JIANG, H., WANG, J., AND XIAN, F. Hba: Distributed metadata management for large cluster-based storage systems. *IEEE Transactions on Parallel Distributed Systems 19*, 6 (2008), 750–763.