Quantifying Temporal and Spatial Localities in Storage Workloads and Transformations by Data Path Components

Cory Fox, Dragan Lojpur, and An-I Andy Wang
Florida State University, Department of Computer Science, 253 Love Building, Tallahassee, FL 32306, USA.
{fox, lojpur, awang}@cs.fsu.edu

Abstract

Temporal and spatial localities are basic concepts in operating systems, and storage systems rely on localities to perform well. Surprisingly, it is difficult to quantify the localities present in workloads and how localities are transformed by storage data path components in metrics that can be compared under diverse settings.

In this paper, we introduce stack- and block-affinity metrics to quantify temporal and spatial localities. We demonstrate that our metrics (1) behave well under extreme and normal loads, (2) can be used to validate synthetic loads at each stage of storage optimization, (3) can capture localities in ways that are resilient to generations of hardware, and (4) correlate meaningfully with performance.

Our experience also unveiled hidden semantics of localities and identified future research directions.

1. Introduction

To increase performance, modern storage consists of many data path components, from the front-end file system cache and disk layout management to the back-end disk controller cache and on-disk caching. Various components generally exploit the temporal and spatial localities in workloads to achieve performance gain. However, how localities of a workload are transformed through individual optimizations is not well understood, resulting in designs that are more reflective of the understanding of the front-end workload than that of the locality characteristics immediately before the component. The problem worsens as the storage data path components proliferate over time.

This research aims to develop metrics that quantify temporal and spatial localities in workloads as well as transformations by various data path components. These metrics can help us understand (1) how front-end workloads (e.g., references sent from a user-level application to the operating system) stress back-end data path components (e.g., disks), (2) how synthesized front-end workloads have different effects within the data path from the original workloads from which they are derived, and (3) how each storage component shapes localities.

Although conceptually simple, quantifying localities in the context of a storage data path is challenging for many reasons: (1) Storage components such as the file system cache can introduce internal system traffic due to prefetching, buffered writes, page replacement policies, metadata accesses, and system events that are sensitive to physical time and memory resources. Therefore, the accesses before one storage component do not always have a one-to-one mapping to the accesses after a storage component. (2) The semantics of locality depend on the granularity of analysis. At a high level, accesses can be analyzed in files and directories (although internal storage components do not operate at these granularities). At a low level, accesses can be analyzed in blocks and sectors. Locality computed based on the distance between adjacent references to files is likely to be poorer than locality based on the distance between references to blocks, since many blocks are referenced sequentially within files. (3) Locality metrics need to be comparable across workloads and system environments. A workload that exhibits a "90% spatial locality" on a 50-GB drive should exhibit meaningful behaviors when applied to a 100-GB drive.

Existing quantifications of locality are largely performed within the context of caching. Studies on temporal and spatial localities also exist independently. However, there are limitations. The popular metric of the cache-hit rate [21] measures the effectiveness of various caching policies, but the metric is not applicable when evaluating data path components such as the disk scheduler. Commonly used stack and block distances [4] can measure how temporal and spatial localities are transformed by caches. However, they are highly sensitive to various system settings, and are difficult to use to compare workloads from different environments. Some studies evaluate the effects of caching algorithms and cache sizes on the reference stream using analytical and

simulation methods. However, these analyses often exclude the effects of traffic internal to a storage system [18]. Researchers have also studied tiered cache management [3, 12, 22], but their focus remained on improving I/O efficiency within a system, not on making measured effects comparable across workloads and environments.

This paper proposes two affinity metrics to evaluate how data path components transform workloads in terms of temporal and spatial localities. Through analysis of workloads with extreme locality characteristics, as well as applying normal workloads on different data path components under different environments, we were able to show that our metrics behave well, are meaningful when comparing workloads from different environments, and reflect performance characteristics. Our exploration further unveiled the intricacy of locality concepts and identified research directions to refine our metrics.

2. Background & related work

This section highlights the existing ways to quantify temporal and spatial localities.

2.1. Spatial locality

For disk-based storage, spatial locality measures the degree to which data objects stored in the physical vicinity of one and other are used together (i.e., within a short timeframe), since accessing nearby objects is faster on disks. Although the mapping of logical disk blocks to physical sectors and the timing behavior of modern disks are not straightforward [2], good spatial locality can be often achieved by accessing logical disk blocks consecutively.

Interestingly, spatial locality is a by-product of physical disk layout policies, which are governed by file systems. Therefore, spatial locality measures how well a workload matches the underlying disk layout. That is, should a workload make references to random disk blocks, and should the disk layout pack those blocks contiguously in the same random order, the spatial locality of this workload is high. However, sequential access of randomly stored disk blocks exhibits poor spatial locality.

Most file systems exploit spatial localities in three ways: (1) sequentially accessed disk blocks are stored contiguously [14], (2) files stored in the same directory are collocated [13], and (3) disk blocks are prefetched, assuming that most accesses are sequential [16, 4]. Given these assumptions, the "spatial locality" of a workload is often measured in ***block distance***.

With knowledge of the underlying disk layout, block distance measures the differences in block numbers between adjacent references. Figure 2.1 shows that the first reference has a block distance initialized to the block number (i.e., 1,000). If block 4,000 is referenced next, its block distance is 3,000. Thus, the spatial locality of a workload is the average block distance for all references. A smaller average indicates a better spatial locality.

Block distance is sensitive to the number of unique data objects referenced in a workload. Suppose analyses are performed at the granularity of a 4-KB block and a 256-KB block. Two identical reference streams that request 256-KB at a time can yield very different block distance numbers. The average block distance based on 4-KB blocks may be 64 times higher than that computed in 256-KB blocks. The *a priori* knowledge that both workloads are identical allows us to trivially convert the block distance based on one block size into the distance of another. However, such knowledge is not always available. Should published numbers only present block distance in arbitrary block sizes, results obtained would not be easily comparable.

Block distance is also sensitive to access patterns with different block ranges, which are in many cases a by-product of the disk size. Therefore, if workload *A* (perhaps with a large disk) contains the 10 block references 1, 2, 3, 4, 5, 1001, 1002, 1003, 1004, and 1005, the average block distance is 100.5. If workload *B* (perhaps with a small disk) contains the 10 block references 1, 50, 100, 150, 200, 250, 300, 350, 400, and 450, the average block distance is 45. Based on the averages, one can conclude that workload *B* has a better spatial locality, while workload *A* has more references made to adjacent blocks. Therefore, it is difficult to use block distance alone to compare workloads running in different environments.

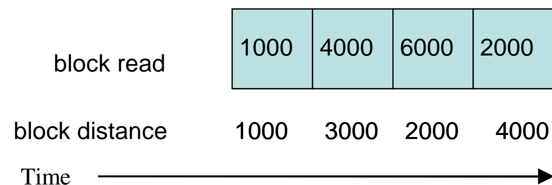

Figure 2.1. Block distance calculation.

2.2. Temporal locality

Temporal locality measures how frequently the same data object is accessed. Temporal locality exhibited in workloads is crucial for many storage optimizations (e.g., caching) and thus is an essential characteristic to quantify.

One common metric to measure temporal locality is ***stack distance*** [17], defined as the number of references to unique data objects before referencing the same object. Since our proposed metrics are built on this concept, we shall detail it further.

Suppose the granularity of a reference is a file, and the algorithm begins with an empty stack. When a file is first

referenced, it is pushed onto the top of the stack. The stack distance for this reference is either infinite or a pre-defined *Max* ($\geq$ maximum number of unique files). Whenever a file is accessed again, the depth ($\geq$ 0) of the file in the stack is the stack distance. The referenced file is then removed from the stack and pushed onto the top of the stack again.

To illustrate, Figure 2.2 begins with an empty stack (*NULL*). File *A* is referenced for the first time, its stack distance becomes *Max*, and the stack now contains one file (*A*). Files *B* and *C* then are referenced consecutively for the first time, and the stack distance for each is again *Max*; *B* and *C* are each pushed onto the top of stack in order of reference, and the stack now contains three files (*C*, *B*, *A*). At this point, if file *B* is referenced, the stack distance of this reference is 1, which is the depth of File *B*. File *B* then is removed from the stack, and pushed onto the top of stack (*B, C, A*). A low average stack distance across all references in a workload indicates a good temporal locality. While stack distance can quantify the temporal locality of a workload, it has limitations. First, similar to block distance, stack distance is sensitive to the granularity of analysis (e.g. file vs. 4-Kbyte blocks vs. 512-byte sectors).

Second, while the lowest possible stack distance is 0, the *Max* value is not bounded. To one extreme, when *Max* >> the total number of unique data objects, the average stack distance approaches (*Max**the number of unique data objects)/the number of references, reflecting little about the ordering of the data references. Although some variants of stack distance computation omit first-time references, it becomes problematic when a significant fraction of the references are first-time (e.g., Web workloads).

Third, since stack distance is sensitive to *Max*, the number of unique data objects, and the total number of references, the resulting average stack distance has no reference point other than 0 and an arbitrary *Max* to indicate whether a workload exhibits good temporal locality. The metric is mostly useful when performing relative comparisons between two workloads under similar settings and applied to similar environments. When a given workload is exercised in different environments, the results represented by this metric are not comparable.

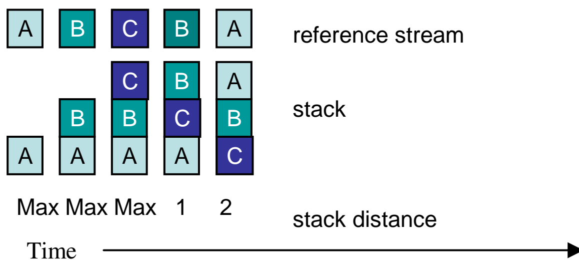

Figure 2.2. Sample stack distance.

2.3. Effectiveness of caching

Locality has been widely studied within the context of caching. However, the fact that we are still seeing major storage innovations based on exploiting locality reflects ample opportunities to advance this area [7, 11, 5, 23]. In particular, few studies address the issue of quantifying localities. A popular metric is the cache-hit ratio, which is computed by dividing the number of references served from the cache by the total number of references, in either files or blocks. Variants of cache-hit ratios are used to compare various caching policies [9, 3].

Cache-hit ratios can reveal information such as the working set size. However, a high cache-hit ratio can be caused by a cache size greater than the working set size, effective caching policy, or good temporal locality within a workload. Therefore, this metric provides confounding information on how a workload is transformed in terms of spatial and temporal localities. Most important, cache-hit ratios cannot be applied to analyze non-cache-related storage data path components (e.g., disk scheduler).

2.4. Effects of cache transformations

The effects of cache transformations have also been studied in distributed systems. For example, the concept of log-based file systems was introduced based on the observation that the client cache absorbs the majority of reads, leaving the write-mostly traffic to the server side [16]. Multi-tiered coordinated caching examines how to remove unwanted interactions between cache layers [12, 22].

Not until recently has the size of caches become sufficiently large for standalone machines [20], and their transformations on temporal and spatial localities have thus become an area of research interest. [24] examined the effects of L1 and L2 caches on memory references. Our study extends this study to the entire storage data path.

Temporal and spatial localities in Web reference streams have been analyzed using stack distance and the number of unique sequences, respectively [1]. Although our studies share similarities in methodology, we focus on the transformations at various data path locations.

Researchers have advocated a more thorough analysis of real-world workloads before creating accurate synthetic workloads [19, 15]. Hsu et al. [9] introduced a way of viewing reference streams. By plotting a referenced address modulo 32MB against the access number, they demonstrated differences between real-world workloads and synthetically generated ones.

2.5. Aggregate statistics

Various high-level statistics are used to characterize the localities of a workload [1, 15]. For example, a workload can be analyzed for the average number of bytes referenced per unit of time, which can be decomposed into bytes from unique block locations, or ***unique bytes*** [6]. The ratio of unique bytes to total bytes can be used to quantify temporal locality, in terms of how often bytes are repeatedly referenced.

One concern is that very different reference patterns can yield similar aggregate statistics, which is particularly pronounced in synthetic workloads that mimic real-life workloads via matching aggregate statistics [9]. For example, synthetic workloads often match well with aggregate statistics before the file system cache, but their after-cache behavior can deviate from the after-cache behavior of the real-life workload significantly, as demonstrated in this paper.

3. Affinity metrics

We propose two metrics to measure temporal and spatial localities of workloads—stack affinity and block affinity respectively.

$$stack_affinity = \frac{1}{\log_{10}(10 + stack_dist)} \quad (1)$$

$$block_affinity = \frac{1}{\log_{10}(10 + block_dist)} \quad (2)$$

Although our metrics seem simple and are built on existing stack and block distances, our metrics ease comparing different workloads in different environments. Conceptually, locality is inversely proportional to the orders of magnitude changes in stack and block distances. The rationale reflects the exponential speed of hardware evolution and how certain performance metrics (e.g., cache hit rate) improve linearly as the system resources increase exponentially (e.g., memory size) [15]. We will first demonstrate the inherent characteristics of these metrics, and use them to measure how storage data path components transform localities.

Resiliency to different maximum values: One drawback of distance metrics is the high sensitivity to the maximum value due to first-time references and the size of the disk. To reduce such sensitivity, we first move the distance metrics to the denominator. So, large distance values due to various causes push locality metrics toward a common minimum 0, which means poor locality. Additionally, we take a logarithmic weighting of the distance, to achieve two effects. (1) Since hardware improvements in terms of performance, disk/cache capacity, and cost are exponential, the logarithmic function dampens the computed affinity differences due to nearby generations of hardware. For example, referencing a block 200 GB away on a disk degrades spatial locality just as significantly as referencing a block 500 GB away. (2) First-time references skew the affinity numbers only in a limited way, such that the resulting affinity values still largely depend on the ordering of references.

Boundary conditions: Another drawback of distance metrics is the difficulty in interpreting locality when the maximum value is not bounded and specific to environments. With affinity metrics, we can describe localities between 0% (poorest) and 100% (highest). The addition of 10 to the denominator makes the minimum value of denominator 1 when either the stack distance or block distance is 0, which represents 100% in both metrics.

References that lead to good locality behavior are more exponentially weighted based on the observed relationships between performance metrics and available system resources [15]. Recall Section 2.1, workloads with good localities may exhibit worse original distance values than those of workloads with poor localities. Consider the same example from Section 2.1: with the block-affinity metric, the reference stream on blocks 1, 2, 3, 4, 5, 1001, 1002, 1003, 1004, and 1005 yields a 90% spatial locality, while the reference stream on blocks 1, 50, 100, 150, 200, 250, 300, 350, 400, and 450 on a small disk yields a 61% spatial locality. These numbers are more reflective of how adjacent disk blocks are referenced as opposed to the differences in block ranges.

Granularity of analyses: Although our simple alterations of the stack and block distance metrics overcome many existing limitations, affinity can yield very different numbers for different granularity of analyses. Above the operating system, logged references are directed to files and directories, although the storage data path operates in blocks. Our current solution is to convert the analysis granularity to blocks, which is the highest common denominator between the two. (Note that we do not preclude the possibility of analyzing reference affinities at the level of physical data locations on the disk).

This conversion requires locating file blocks on the disk. However, in many cases, there is no one-to-one mapping of the referenced blocks, which poses challenges when applying our metrics to evaluate data path components such as file system caching. First, the file system cache generates internal references to the storage systems; thus, reference blocks after cache may have no corresponding reference before cache. One example is the prefetching of consecutive blocks into the cache in anticipation of sequential access patterns. Another is the committing of modified memory content to the disk when the available memory is running low. Second, references to cached content may not have corresponding after-cache references. For example, reads (and sometimes writes) to cached data

and resolved cached file path components will not yield after-cache references. Thus, the total number of unique data requests can be different across individual data path components, which is captured by our affinity metrics.

4. Evaluation

Our experiments included (1) stressing affinity metrics under workloads with extreme temporal and spatial localities, (2) observing affinity metrics under normal trace replays, (3) applying affinities to compare characteristics of a trace-replayed workload and a synthetic workload based on the trace, (4) studying the sensitivity of affinity metrics across nearby generations of hardware, (5) correlating affinity and performance metrics, and (6) testing affinity metrics under workloads from more environments.

To see the locality transformations by the entire storage data path, we gathered the affinity values at the data path front end before going through the file system cache, and at the back end before requests are forwarded to disk. To see the effects of individual optimizations, we could selectively disable optimizations. To illustrate the effects of disk scheduling, we could bypass file system caching. Additionally, we can minimize the effects of write-back policies by using a read-mostly workload.

Web workloads: We gathered HTTP access logs from two Web servers; one from the Department of Computer Science at Florida State University (FSU) between 11/14/2004 and 12/7/2004 and the other from the Laboratory of Advanced Systems Research at UCLA, between 5/8/2005 and 6/7/2005. We selected the week with the most bytes referenced. Table 1 summarizes the chosen workloads.

For each log, we also obtained the file system snapshot, which consists of all files, directories, and links as well as their i-node creation, modification, and access timestamps. Before replaying our traces, we recreated each file system in the order of their creation dates.

The Web workloads were replayed on two machines, one acting as a server and the other as a client (Table 2). The server hosted an Apache 2.2.2 Web server while HTTP requests were generated via a multi-threaded replay program running on the client machine. Each thread corresponded to a unique IP address.

To accelerate the evaluation process, we sped up the trace replay by a constant factor derived using the following method. (1) We replayed the trace with a zero-time delay between references to measure the maximum speed-up factor. (2) We divided this factor by two, and approximated it with the nearest power of two. With this method, we sped up both traces by a factor of 128.

The front-end reference stream data were captured on the server side. To extract file and directory block numbers, we used `debugfs` provided by ext2. In addition, we had to account for the implicit traffic generated during path resolutions when applying our affinity metrics for analysis. For example, a reference to `/dirA/file1` involves a reference to `/` and `/dirA` before referencing `/dirA/file1`.

Table 1. Workload characteristics.

	FSU	UCLA	Desktop	OS-class
Bytes referenced	4.3 GB	19 GB	50 GB	2.0 GB
Unique bytes referenced	133 MB	668 MB	11 GB	1.5 GB
Number of requests	150K	841K	13M	532K
Mean interarrival time	4.03 secs	3.08 secs	3.16 msec	27.9 msec

Table 2. Experimental hardware configurations.

	Server	Client
Processor	2.8GHz Pentium 4, 1024-KB cache	2.4GHz Intel Xeon 512-KB cache
RAM	512-MB Netlist DDR PC3200	2-GB Micron DDR PC2100
Disks	2 160-GB 7200-RPM Seagate Barracuda 7200.7	40-GB 7200-RPM Maxtor 6E040l0
Network	Intel 82547Gi Gigabit Ethernet Controller	Intel 82545EM Gigabit Ethernet Controller
Operating system	Linux 2.6.5	Linux 2.6.16.16
File system	Ext2 0.5b	Ext2 0.5b

We monitored `bios`, which are the Linux internal representations of block requests. We timestamped a `bio` in `generic_make_request()` in `ll_rw_blk.c` before it was submitted to the I/O scheduler. When the `bio` returned and called its finalization code, we logged the start time, end time, and block number in the memory and dumped them at the end of the replay. We set aside preallocated memory for logging, specified as a kernel parameter in `grub.conf`, to ensure the same memory size setting as the original Web servers.

Software development workloads: We gathered traces from machines used for operating system research and undergraduate course projects at FSU. The former desktop trace was taken from 8/20/07 to 8/22/07, and contained 32K processes. The latter OS-class trace was gathered from 3/8/07 to 3/14/07, and contained 33K processes. Unlike the read-mostly Web traces, these desktop traces consist of both read and (up to 24%) write activities.

We used Forensix [8] to gather front-end traces, which required a different playback system. The file system recreation and block mapping steps are identical to the Web workload ones. However, replaying was performed only on the server, with one process created for each process in the trace. We sped up both traces by 32 times.

For data gathering, the front-end references were logged as the system replays. Each process kept its own list of

files referenced and bytes accessed. On replay completion, these file references, including path resolutions, were mapped to the block level. The back-end reference stream data were recorded by reusing the Web logging framework.

We used the front-end reference stream to calculate the blocks referenced during the trace. For the backend, we used the blocks reported by our modified kernel.

Metrics: In addition to affinity metrics, we also measured the bandwidth and latency. The backend was measured right before the block I/O was sent to the storage device and as soon as the block I/O returned. The latency was measured on a block-by-block basis.

For the front end, the latency and bandwidth could be measured based on either the requested data only (excluding metadata) or all blocks related to a data request (including metadata). To align the front-end measurement with the back, we chose the latter. The front-end bandwidth and latency for the Web workloads were measured on the client, and include network effects, while the software-development measurements were performed on the server. The performance numbers reflect the end-user experience.

5. Extreme localities

To understand the dynamic range of our metrics when being transformed by the entire storage data path, we synthesized workloads to exercise all combinations of high and low temporal and spatial localities. These workloads are based on the FSU trace, so the affinity values can be compared with regular FSU replays. Although these loads are read-mostly, they serve as a starting point to understand the rich locality behaviors. As a further simplification, these replays were single-threaded. The synthesized load has the total number of references equal to that of the FSU trace. The request timing is based on an exponential distribution, with the mean set to the average interarrival time of the FSU trace. Figure 5.1 summarizes the median affinity values for various locality settings. We used the median since the lack of back-end traffic sometimes leads to 0 affinity values.

High temporal and spatial localities: To achieve high spatial and high temporal localities, we created two 1-MB files in / and read those files alternatively and repeatedly.

Figure 5.1 shows that the front-end stack and block affinities are about 0.77 and 0.73, respectively. The front-end stack affinity was higher than expected, given that each file sequentially accessed 256 4-KB blocks should result in affinity ~0.4. It turned out that repeated references to ext2 i-nodes and indirect index blocks improved the temporal affinity.

However, the front-end block affinity was not as high as the expected value, which would be close to 1 for all sequential accesses. This is because accessing a file also involves referencing directories in the file path. Although we created our files in / to minimize directory lookups, for the purpose of accounting, each front-end file reference still involves looking up /, which is not stored near the files, thus, driving down the front-end block affinity significantly.

The back-end affinities show values above the 0.95 range. This value is slightly misleading since frequent directory traversal causes directories to be cached, leaving mostly timestamp updates to the disk.

High temporal locality and low spatial locality: For this workload, we created a reference stream that reads the two files with the lowest and highest logical block numbers.

Figure 5.1 shows that the front-end stack affinity was 0.72; block affinity, 0.22. The back-end stack affinity increased to 0.94, while the block affinity increased to 0.56.

Low temporal locality and high spatial locality: This case is achieved by reading files in the order of increasing block numbers, while references within a file remain sequential. Figure 5.1 shows that the front-end stack affinity was 0.41; block affinity, 0.47. The back-end stack affinity was 0.01 and block affinity, 0.94.

Low temporal and spatial localities: One way to generate a workload with poor localities is to shuffle the reference ordering of files randomly in the FSU trace.

Figure 5.1 shows that the front-end stack and block affinities were 0.62 and 0.22, respectively. The front-end stack affinity is relatively high, suggesting that temporal locality is inherent in the file systems' hierarchical naming structure. Also, the random reference stream does not generate the worst-case locality, because the previous scenario shows worse front-end stack affinity numbers. The back-end stack- and block-affinity values were 0.00 and 0.17, respectively.

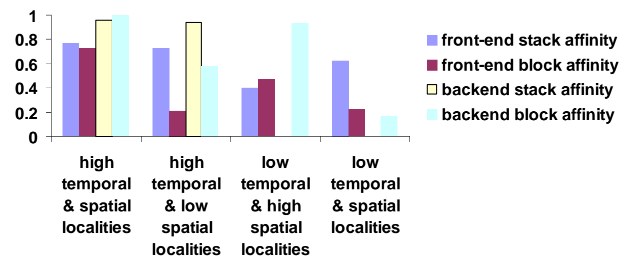

Figure 5.1. Affinities for combinations of temporal and spatial localities, at the front end (before file system cache) and backend (before disk) of a storage data path.

Overall: The dynamic range of affinity can capture both high and low values for temporal and spatial localities. Front-end stack affinity values tend to reflect the

directory structure captured by the trace, while the back-end affinity values can span the entire dynamic range.

6. Web workloads

Understanding how affinity metrics behave under extreme localities enables us to better interpret the numbers under the FSU and UCLA Web workloads. Figure 6.1 shows affinity values over time. The affinity values are averaged (within each hour) because our notion of locality is inversely correlated with the average orders of magnitude changes in stack and block distances. The front-end affinity values are in the mid-range, with high back-end block affinity and low back-end stack affinity. In reference to Figure 5.1, the Web workload displays the case of low temporal and high spatial localities. The back-end stack affinity increases over time as compulsory misses taper, but its growth appears to be asymptotic. We confirmed that the compulsory misses within Web traces are more uniformly scattered throughout the trace.

Figure 6.2 shows affinities for the UCLA trace. The front-end stack affinity was 0.74, which is higher than the FSU case. The variance of affinity typically reflects the number of references. In this case, the UCLA front end has eight times more references per interval than the FSU case. Also, front-end affinity numbers typically have a lower variance compared to the backend because the number of references in the backend tends to be orders of magnitude lower than the front end due to caching. The front-end block affinity was only 0.28, which is lower than the FSU case, also with a lower variance.

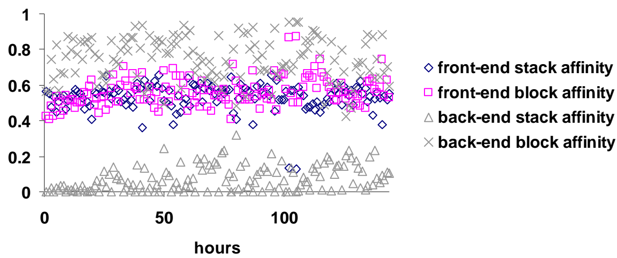

Figure 6.1. Affinities for the FSU trace.

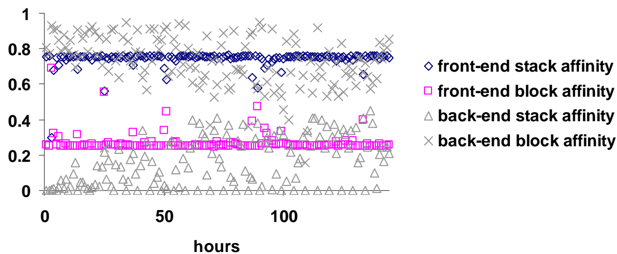

Figure 6.2. Affinities for the UCLA trace.

7. Trace vs. synthetic workloads

Affinity metrics enable us to verify the fidelity of synthetic workloads against trace replays, beyond the front-end aggregate statistics. Figure 7.1 shows affinities over time for the low temporal and spatial locality case, based on random shuffling of references in the FSU trace used in Section 6. This synthesis technique also preserves many front-end aggregate statistics (e.g., file size distribution).

Since the frequency of referencing popular files is preserved, the synthesized load can preserve front-end stack affinity. However, random shuffling of references degrades front-end block affinity significantly. The uniform distribution of references significantly increases the number of references for many time intervals, resulting in lower variance of affinities throughout. This uniformity also fails to capture how real world workloads can change significantly from hour to hour.

The back-end stack affinity values diverge over time as the number of back-end references decreases. Unlike the original trace, toward the end of the trace, we saw more repeated references to the popular blocks for timestamp updates, and fewer compulsory misses (Figure 7.2).

The back-end and front-end block affinities shared similar initial values, reflecting initial compulsory misses. The back-end block affinity then declined asymptotically to 0.20 as most directory and metadata blocks are cached 60 hours into the trace.

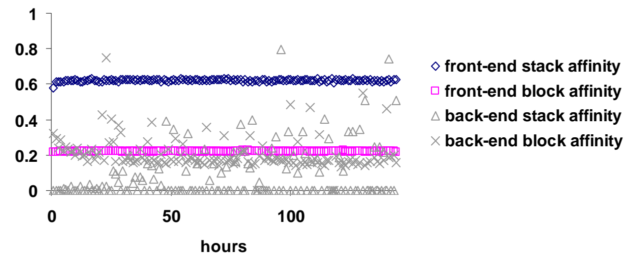

Figure 7.1. Affinities for the synthetically generated FSU trace.

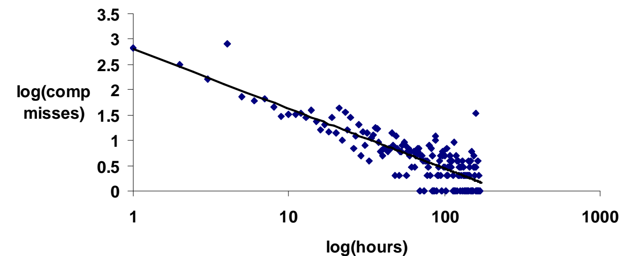

Figure 7.2. Compulsory misses over time show a log-log-linear relationship in a Web trace with randomly shuffled references.

In addition, the back-end affinity numbers reveal that the front end and the backend of a system reach steady states at different times. Studies conducted without this awareness can yield misleading results and system designs.

8. Portability of affinities

To show the portability of affinity metrics across neighboring generations of hardware, we recreated the file system snapshots according to the file creation dates, on hard drives with 40 GB and 160 GB. With a larger drive, we verified that the data blocks were stored over a larger range of block locations. Although the range of reference data block increased by 4 times, Figure 8.1 shows affinity characteristics that are very similar to those in Figure 6.1, suggesting our logarithmic transformations in the metrics enable us to characterize traces in a way that is more resilient to the exponential rate of hardware evolutions.

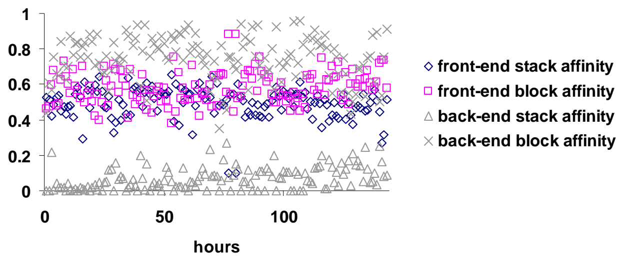

Figure 8.1: Affinities for the FSU trace with 4 times the disk size.

9. Affinity vs. performance

To demonstrate the relationship between our affinity metrics and performance, we measured the locality transformation by the default anticipatory disk scheduler [10] in Linux 2.6.5. We replayed the FSU traced blocks with multiple threads on the server, with the `O_DIRECT` flag to bypass file system caching. The anticipatory scheduler attempts to reduce 'deceptive idleness' by waiting for additional requests from a process before switching to requests from another process. Without this style of scheduling the localities inherent in a program would be broken up by request switching.

The baseline comparison is the noop scheduler, which sends disk requests in a FIFO order. To ensure sufficient requests for reordering, we replayed the FSU traces with zero-think-time delays. To provide a fuller context, we compared these results with the normal FSU replay (Section 6) with half of the zero-think-time speed-up factor, and with both the file system cache and the anticipatory scheduler running.

Figures 9.1, 9.2, and 9.3 show that the noop scheduler had the same front-end and back-end affinity values and performance numbers, since the scheduler performs no transformations on locality. The anticipatory scheduler improved the stack affinity by 10% and block affinity by 54%. Interestingly, although the back-end bandwidth increased by 111%, the front-end bandwidth improved by only 18%. In terms of latency, the improvement is by two orders of magnitude, reflecting a significant reduction in disk seeks due to switching requests among processes.

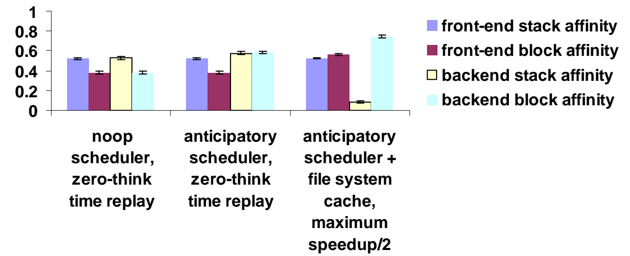

Figure 9.1: Affinities for FSU trace replay under different configurations.

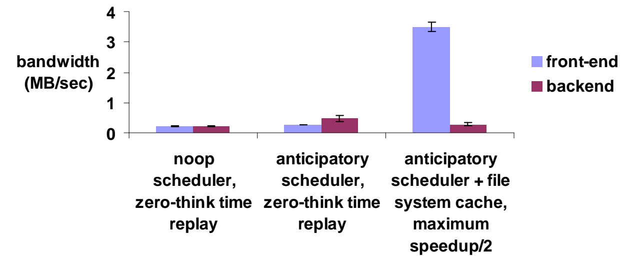

Figure 9.2: Bandwidth for FSU trace replay under different configurations.

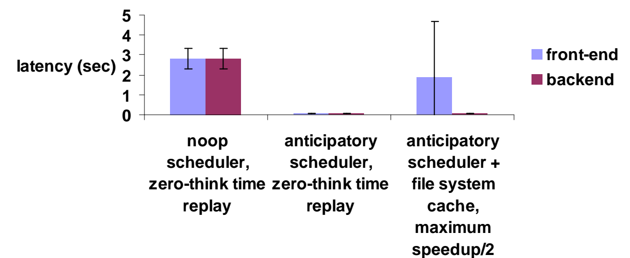

Figure 9.3: Latency for FSU trace replay under different configurations.

For the third case, the FSU trace replay speed was halved, effectively reducing the number of concurrent request streams and the probability of sequential requests being fragmented due to request multiplexing. As a result, front-end spatial affinity was 50% better than that of the noop scheduler. Surprisingly, front-end stack affinity is less sensitive to the ordering of requests compared to block affinity. As long as the number of unique blocks referenced within a time frame is within a similar order of magnitude, stack affinity would not change much. On the

other hand, minor reordering of requests to distant block locations can change block affinity significantly.

The enabled file system caching absorbed 45% of stack affinity from the front, and sequential prefetches improved spatial affinity in the backend by another 31%.

In terms of performance, caching improved bandwidth by 16 times. The front-end latency of the original FSU replay is not directly comparable due to the inclusion of the network component. The significant latency variance can be attributed to either the network or the low back-end stack affinity. The lack of opportunities for the anticipatory scheduler to reorder requests due to slowed replay allowed only 34% latency improvement over the noop case.

Overall, back-end block affinity correlates with improved bandwidth. While back-end stack affinity does not seem to contribute to bandwidth, poor back-end stack affinity seems to introduce high variance to latency. Intriguingly, front-end and back-end affinities are poorly correlated.

10. Read & write workloads

Web traces show how affinity metrics interact with read-mostly workloads. The next step is to understand the behaviors of affinities with the presence of writes.

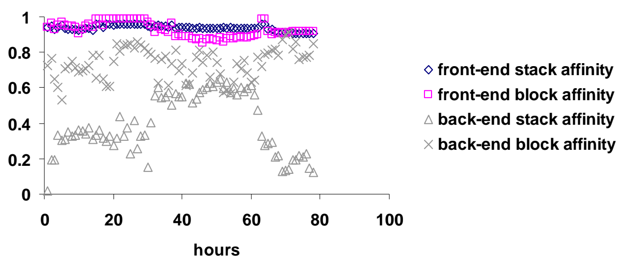

Figure 10.1. Affinities for the desktop trace.

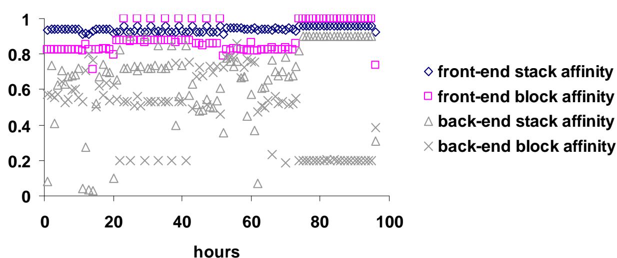

Figure 10.2. Affinities for the OS-class trace.

Figures 10.1 and 10.2 show the behavior of affinity metrics under software development workloads. In both cases, the front-end affinities were high, which have not been observed in the read-mostly workloads. One explanation is that Web accesses mostly read files in their entirety. In the software development environment, writes are often made to the same block (e.g., compilation), resulting in high block and stack affinities.

Interestingly, the back-end stack affinity was relatively high in both cases compared to the Web traces, reflecting synchronous write-through activities (e.g., updating directory blocks). The correlation coefficients between front-end and back-end stack affinities for the OS-class trace and the desktop trace were 0.20 and 0.86 respectively.

11. Lessons learned

Although temporal and spatial localities are basic concepts in the OS arena, our attempt to quantify localities illustrates that we do not know locality as well as we think.

First, cache-hit rates, aggregate statistics, stack distance, and stack affinity actually do not capture the notion of time. Therefore, two consecutive references to the same object, spaced one month apart, can still be considered to exhibit good temporal locality. Interestingly, this property was not obvious until we replayed traces at different speeds.

Second, ironically, spatial locality is more sensitive to replay speeds because of the interruption of sequential transfers due to switching among concurrent reference streams. While our initial observation is not conclusive, future investigation will help us better understand the intricate effects caused by trace accelerations.

Third, spatial locality defines how well the reference stream matches the back-end disk layout. Therefore, when we quantify the spatial locality in a workload, we assume disk layout optimizations shared by common file systems.

Fourth, in many cases locality transformations are relevant only with a sufficient volume of I/O requests. Otherwise, optimizations such as disk scheduling cannot effectively shape the reference localities.

Fifth, we originally thought temporal and spatial localities can capture workload characteristics and data path optimizations well, but we have just begun to grasp the rich behaviors of workloads and data path interactions. Our experience suggests that the element of time is captured by neither locality metric, which warrants future investigations. Also, optimization concepts such as data alignment would not fit well with our metrics, unless we incorporate the notion of distance with the intricate timing of storage devices.

Finally, the community is well aware of the difficulty of building per-trace-format scripts and replay mechanisms, handling corrupted data entries, analyzing data sets with a large number of attributes and possible transformations, and extracting meaningful trends with limited help from automation. Still, developing empirical metrics based on workloads from diverse environments will remain a difficult task for the foreseeable future.

12. Conclusion

In this paper, we have proposed and demonstrated the use of stack and block affinities to quantify temporal and spatial localities. Through extreme workloads, two Web server workloads, and two software development workloads, affinity metrics behave well, and provide meaningful comparisons across diverse workloads and environments. Moreover, affinity values can be correlated to performance and, thus can reveal how data path components, such as the IO scheduler, contribute to the overall performance gain.

We have illustrated how affinity metrics can be used to evaluate the fidelity of workload generators beyond the front-end aggregate statistics. We also discovered the richness of semantics behind localities and research directions to better characterize storage workloads and data path transformations.

Acknowledgments

We thank Kevin Eustice for providing the UCLA trace and Ashvin Goel for providing the Forensix tool. We also thank Peter Reiher, Geoff Kuenning, and Mark Stanovich for reviewing early drafts of this paper. This research was sponsored by the FSU FYAP Award and the FSU Planning Grant. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of FSU or the U.S. government.

References

[1] Almeida V, Bestavros A, Crovella M, deOliveira A. Characterizing Reference Locality in the WWW. Technical Report. UMI Order Number: 1996-011., Boston University, 1996.

[2] Anderson D. You Don't Know Jack about Disks. *Queue*. 1(4), 2003.

[3] Chen Z, Zhang Y, Zhou Y, Scott H, Schiefer B. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. *Proc. of the 2005 ACM SIGMETRICS*, June 2005.

[4] Cherkasova L, Ciardo G, Characterizing Temporal Locality and Its Impact on Web Server Performance, *Proc. of ICCCN'2000*, October 2000.

[5] Ding X, Jiang S, Chen F, Davis K, Zhang X. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. *Proc. of the 2007 USENIX Annual Technical Conf.*, June 2007

[6] Ferrari D. On the Foundations of Artificial Workload Design. *Proc. of the 1984 ACM SIGMETRICS*, 1984.

[7] Gill B, and Modha D. WOW: Wise Ordering for Writes—Combining Spatial and Temporal Locality in Non-volatile Caches. *Proc. of the 4th USENIX Conf. on File and Storage Technologies*, 2005.

[8] Goel A, Feng WC, Maier D, Feng WC, Walpole J. Forensix: A Robust, High-Performance Reconstruction System. *Proc. of the 2nd International Workshop on Security in Distributed Computing Systems*, June 2005.

[9] Hsu WW, Smith AJ, Young HC. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks—An Analysis at the Logical Level. *ACM Trans. on Database Systems.* 26(1), pp. 96-143, March 2001.

[10] Iyer S, Druschel P, Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. *Proc. of the 18th ACM SOSP*, October 2001

[11] Jiang S, Ding X, Chen F, Tan E, Zhang X. DULO: An Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Localities. *Proc. of the 4th USENIX Conf. on File and Storage Technologies*, Dec 2005.

[12] Li X, Aboulnaga A, Salem K, Sachedina A, Gao S. Second-tier cache management using write hints. *Proc. of the 4th Conf. on USENIX Conf. on File and Storage Technologies*, September 2005.

[13] McKusick MK, Joy WN, Leffler SJ, Fabry RS, A Fast File System for UNIX, *ACM Trans. on Computer Systems* 2(3), pp. 181-197, August 1984.

[14] Ritchie D, Thompson K, The UNIX Time-Sharing System, *Communications of ACM* 7(7), July 1974

[15] Roselli D, Lorch J, Anderson T. A Comparison of File System Workloads. *Proc. of the 2000 USENIX Annual Technical Conf.*, June 2000.

[16] Rosenblum M, Ousterhout J, The LFS Storage Manager *Proc. of the 1990 Summer USENIX*, June 1990.

[17] Spirn JR. Program Locality and Dynamic Memory Management. Ph.D Dissertation, Dept. of Elec. Eng., Princeton Univ., 1973.

[18] Vanichpun S, Makowski AM, The Output of a Cache under the Independent Reference Model – Where did the Locality of Reference Go?, *Proc. of the 2004 SIGMETRICS*, June 2004.

[19] Wang AIA, Kuenning G, Reiher P, Popek G. The Effects of Memory-rich Environments on File System Microbenchmarks, *Proc. of the 2003 International Symposium on Performance Evaluation and Computer Telecommunication Systems*, July 2003.

[20] Wang AIA, Kuenning G, Reiher P, Popek G. The *Conquest* File System: Better Performance through a Disk/Persistent-RAM Hybrid Design. *ACM Trans. on. Storage* 2(3), pp. 309-348, 2006.

[21] Williams S, Abrams M, Standbridge CR, Abdulla G, Fox EA, Removal Policies in Network Caches for World Wide Web Documents, *Proc. of the ACM SIGCOMM*, August 1996.

[22] Wong TM, Wilkes J. My Cache or Yours? Making Storage More Exclusive. *Proc. of the General Track: 2002 USENIX Annual Technical Conf.*, June 2002.

[23] Yadgar G, Factor M, and Schuster A. Karma: Know-it-all Replacement for a Multilevel Cache. *Proc. of the 5th USENIX Conf. on File and Storage Technologies,* February 2007.

[24] Zhou Y, Chen Z, Li K Second-Level Buffer Cache Management. *IEEE Trans. on. Parallel and Distributed. Systems.* 15(6), pp. 505-519, June 2004.