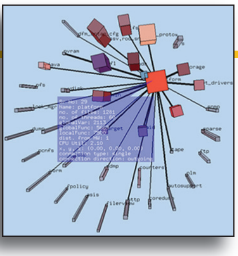


MultiVizArch: Multiple Graphical Layouts for Visualizing Software Architecture

by **Amit Prakash Sawant and Naveen Bali**

Abstract

This article presents an automated technique for visualizing large software architectures using multiple graphical representations, including multi-dimensional scaling, 2-D grid, and spiral layouts. We describe how our software visualization methods were applied to the Network Appliance operating system known as Data ONTAP 7G (ONTAP). We show how each method can be applied to comprehend a specific aspect of ONTAP. This approach can be used by software engineers, architects, and developers to better understand the architecture of their code.

Introduction

An accurate and complete software architecture that represents the entire implementation in code is very difficult, if not impossible, to obtain in the software industry. ONTAP, Network Appliance's storage operating system, is no exception—we found that an accurate and complete software architectural representation does not exist for ONTAP. ONTAP is a very large software system, consisting of approximately 10,000 files and nearly 1 million lines of code). A number of ways to examine and interpret the components of ONTAP exist, but they all oversimplify the system and leave out critical details [1, 5, 6].

In this article we describe multiple graphical methods for visualizing the software architecture of ONTAP. We encapsulate the natural partitions that exist in the system, enabling us to consider individual components and the interactions between them. We utilize existing categorizations of ONTAP as embedded directly in the code and employ visualization techniques to view the identified components. Moreover, since we use raw source code to generate the software architecture, the architecture can be updated frequently to capture changes as they happen. Although our work deals exclusively with ONTAP, the techniques described here can be applied to other large systems including operating systems, middleware, and application software.

Related Work

Knodel et al. [10] propose that software architecture visualizations should be thoroughly assessed by software architects in an industrial environment. They successfully applied the approach of software development, visualization, and validation to their software architecture visualization tool. Auer et al. [2] applied multi-dimensional scaling techniques to visualize high-dimensional software portfolio information gathered from different sources. They proposed a simple method to define raw software metrics data that can be analyzed and processed using multi-dimensional scaling methods. They also conducted a validation study to demonstrate the usability of their proposed approach for software decision support.

GASE (graphical analyzer for software evolution) [7] is another tool to visualize software structural change. This tool uses color to represent new, common, and deleted parts of a software system, thereby enabling the software developer to gain insight about structural changes that might otherwise be difficult to detect. The RECONSTRUCTOR project focuses on interactive visualization of software systems and is concerned with the representation of architecture information in different forms [8].

Data Collection

Kazman and Carrière [9] describe a workbench called Dali, which can extract, manipulate, and interpret software architectural information using raw source code as input. We have borrowed some of Dali's methods in our work.

Software architecture is defined as a set of components that interact with one another via a set of connectors [6]. The connectors in our software architecture are direct procedure calls and accesses to global variables. We chose global variable and global function (collectively known as symbols) references as connectors because ONTAP is writ-

	admin	counters	dense	filerview	ftp	http	kernel	nfs	nvram	platform	storage	tape	target	vdisk	....	# Files	# Threads	# Global Var	# Global Func	# Local Func	HW dist	CPU Util
admin	11998	40	20	0	10	50	580	38	0	472	155	17	125	1	....	491	500	812	2187	2670	4	0
counters	50	195	0	0	0	0	3	0	0	0	0	0	0	0	....	407	1	12	121	94	4	0
dense	10	1	340	0	0	0	2	0	0	0	0	0	0	0	....	26	4	48	92	88	4	0
filerview	0	0	0	0	0	0	0	0	0	0	0	0	0	0	....	289	0	0	0	0	4	0
ftp	1	0	0	0	1140	6	4	0	0	0	1	0	0	0	....	70	8	102	438	44	3	0
http	9	1	0	0	0	1245	12	0	0	22	0	0	0	0	....	92	8	45	615	445	3	0
kernel	6619	439	276	0	548	899	10556	3365	342	22478	9943	555	4954	2056	....	781	64	1404	2105	1400	1	505.79
nfs	39	48	0	0	1	1	12	5358	0	0	81	0	0	0	....	152	18	352	1154	376	3	78.71
nlm	0	0	0	0	0	0	1	14	0	0	0	0	0	0	....	23	3	31	201	62	3	0
nvram	18	6	0	0	0	0	139	0	981	716	28	0	17	7	....	30	3	135	259	176	1	0.03
platform	114	24	7	0	1	0	887	4	280	31820	1232	27	269	12	....	1281	60	2113	5669	7500	1	51.3
storage	116	45	0	0	0	0	65	0	0	1950	17698	122	64	3	....	422	66	1397	3654	3010	1	24.32
tape	0	0	0	0	0	0	87	0	0	92	48	803	0	0	....	23	5	159	182	264	1	0
target	148	33	0	0	0	0	18	0	0	437	86	0	7264	254	....	251	12	440	1489	1344	1	0
vdisk	217	2	0	0	0	0	6	0	0	28	0	0	110	1631	....	86	2	150	661	342	2	0

Table 1: Subset of global symbol cross-reference table with column and row platform highlighted.

ten mostly in the C programming language. However, our approach can easily be adapted to other languages, such as Java.

In order to identify the components within ONTAP, we used BURT, an in-house bug-reporting tool widely used at Network Appliance. All bugs are manually assigned a type and a subtype. For example, bug types can be *hardware*, *software*, or *firmware*, and subtypes within *software* can be *storage*, *platform*, or *kernel*. Since every line of code in ONTAP has a BURT type associated with it, and since every type has a subtype, it follows that the entire code base can be partitioned into BURT's software subtypes. After eliminating obsolete and redundant software subtypes in BURT and merging overlapping ones, we were left with a set of 38 software components. This categorization process can be automated using tools such as **Bugzilla** or accomplished manually with input from software architects and developers.

We used a commercially available tool called Understand for C++ [15] to extract symbols defined within each file in the source tree and obtained cross-references for these symbols. We then used homegrown Perl scripts to extract the symbols and symbol cross-references in HTML format and created a new mapping of symbols to software components. The global symbol cross-reference table contains the number of references between software components (the higher the value, the more connected the pair of components) along with seven other attributes: number of files, number of threads, number of global variables, number of global functions, number of local functions, distance from hardware layer, and CPU utilization. All the attributes are in relation to the corresponding software component.

The CPU utilization measurements were statically gathered from profiles which told us how much CPU is consumed by each function while running a set of performance benchmarks. Since we already had a mapping of functions to software components, we created another set of homegrown Perl scripts to map the CPU utilization numbers to software components. A zero value for CPU utilization means that code from corresponding software components was never called in the benchmark test. From the above description it is obvious that we have used tools to automate the process of data collection, which makes the whole exercise repeatable. Table 1 shows a subset of a global symbol cross-reference table with the software component *platform* highlighted.

MultiVizArch Design

We design our visualizations by first constructing an object to represent a single software component. Next, we position the objects appropriately to produce a static visualization of the software architecture. While designing visualizations, we take into consideration the properties of the data (dimensionality and number of elements) and the visual features (visual-feature salience and visual interference) used to represent the data elements.

Glyph Representation and Data-Feature Mappings

We use **glyphs** that vary their spatial position (x position, y position), color (hue, luminance), and texture (height, size, orientation, line thickness, transparency) properties to represent the attribute values of the software component. The most important attributes should be mapped to the most salient features, and secondary data should never be visualized in a way that would lead to visual interference.

Placement Algorithm

Glyphs representing the attribute values embedded in a dataset must be placed appropriately in order to create an environment that enables easy visual comprehension of complex information. We decided to use three graph layout methods: multi-dimensional scaling (MDS) [3], a traditional 2-D grid, and a spiral technique to visually represent the relationship among the set of software components as explained below. Sawant et al. have used 2-D grid and spiral visualization techniques to represent storage server performance data [12].

Multi-Dimensional Scaling Layout

The multi-dimensional scaling (MDS) technique [16] provides a visual representation of the pattern of proximities among a set of objects. The relationship between input proximities and distances among points on the map can be (1) positive: the smaller the input proximity, the closer the distance between points (e.g., visualization of cities); or (2) negative: the smaller the input proximity, the farther the distance between points (e.g., visualization of software architecture).

MDS finds a set of vectors in two-dimensional space such that the matrix of Euclidean distances among them correspond as closely as possible to some function of the input matrix according to a criterion function called *stress*. Consider a matrix D containing pairwise distances among a set points. The simplified algorithm is as follows:

1. Assign points to arbitrary coordinates in two-dimensional space.
2. Compute Euclidean distances among all pairs of points, to form the matrix D'.
3. Compare D' with the input matrix D by evaluating the stress function. The smaller the value, the greater the correspondence between the two.
4. Adjust coordinates of each point in the direction that best reduces the stress value (a fraction of $\|D-D'\|$).
5. Repeat steps 2 through 4 until stress does not get any lower.

2-D Grid Layout

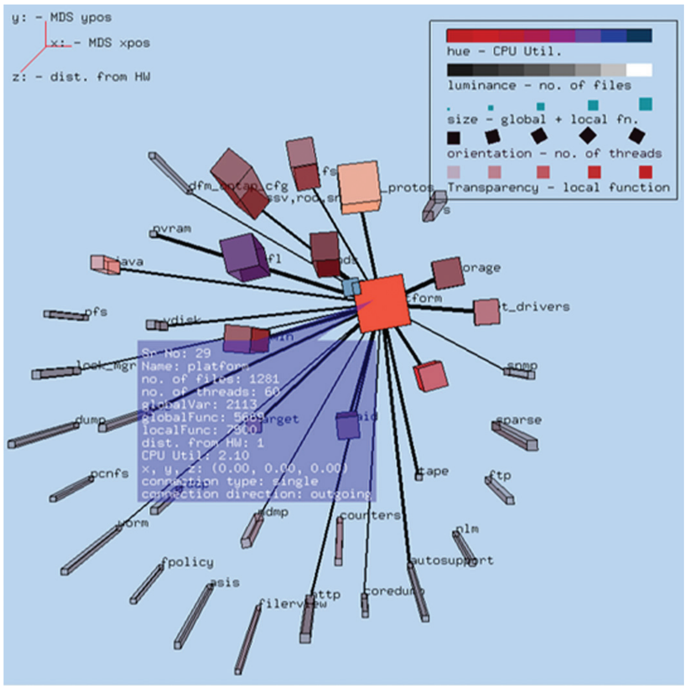
A 2-D grid layout is very intuitive and a commonly used placement algorithm in visualization systems. A two-dimensional ordering is imposed on the data elements through user-selected scalar attributes.

Spiral Layout

A one-dimensional ordering is imposed on the data elements through a user-selected scalar attribute. We map this ordering to a 2-D spatial position for each element by using a 2-D space-filling spiral [4].

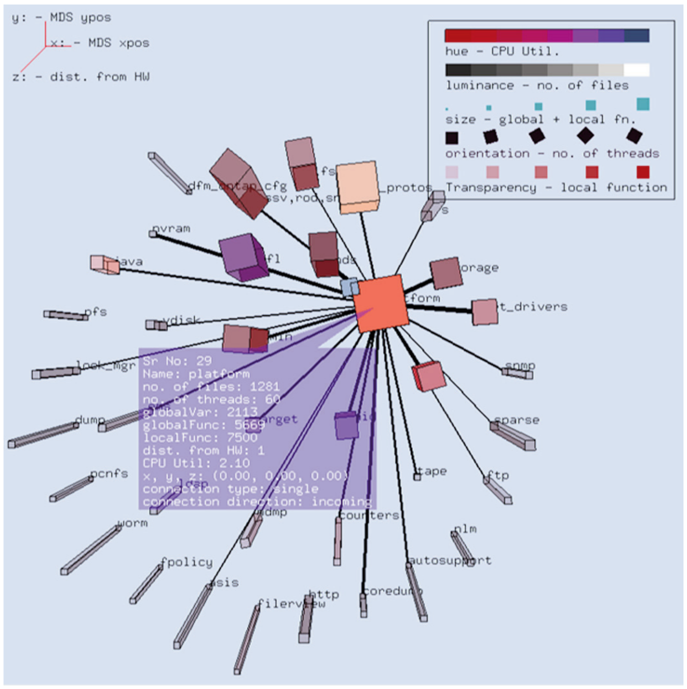
Multiple Layouts of the Software Architecture

Figures 2 through 7 show visualizations of the global symbol cross-reference table and CPU utilization for storage performance benchmarks. The data-feature mappings are as shown in the legend. The lines in Figure 1, Figure 3, Figure 5 and Figure 6 represent the number of *outgoing* symbol references whereas the lines in Figure 2 and Figure 4 represent the number of *incoming* symbol references. The CPU utilization attribute is calculated for the **SPC1** benchmark in Figure 1, Figure 2, Figure 3, Figure 4 and Figure 5 and for the **SFS** benchmark in Figure 6. Previously, we have used a tool called DiffArchViz [11] to visualize dynamic software architecture by examining the correspondence between multiple runtime profiles for a few storage server performance benchmarks, such as SFS 3.02 [14] and SPC-13 [13].



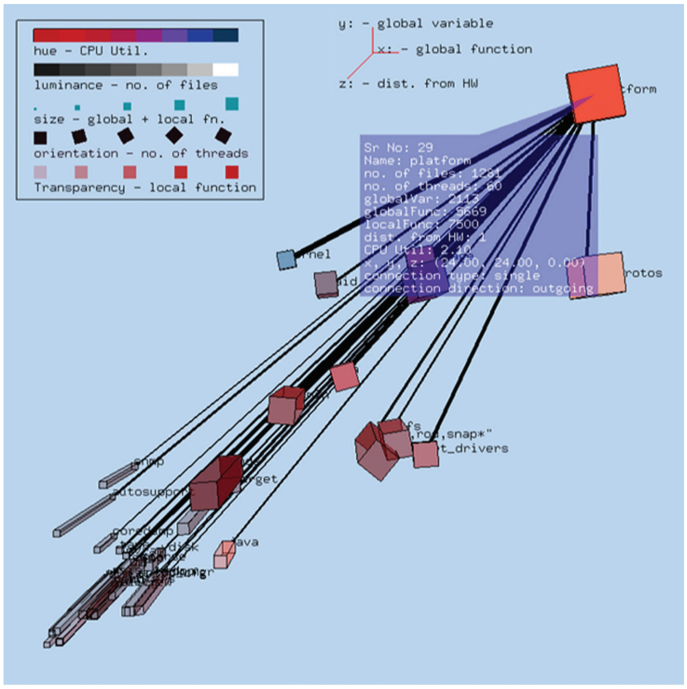
```
Sr No: 29
Name: platform
no. of files: 1281
no. of threads: 60
globalVar: 2113
globalFunc: 5669
localFunc: 7500
dist. from HW: 1
CPU Util: 2.10
x, y, z: (0.00, 0.00, 0.00)
connection type: single
connection direction: outgoing
```

Figure 1: Visualization of global symbol cross-reference table and CPU utilization: MDS outgoing connections from the module platform.



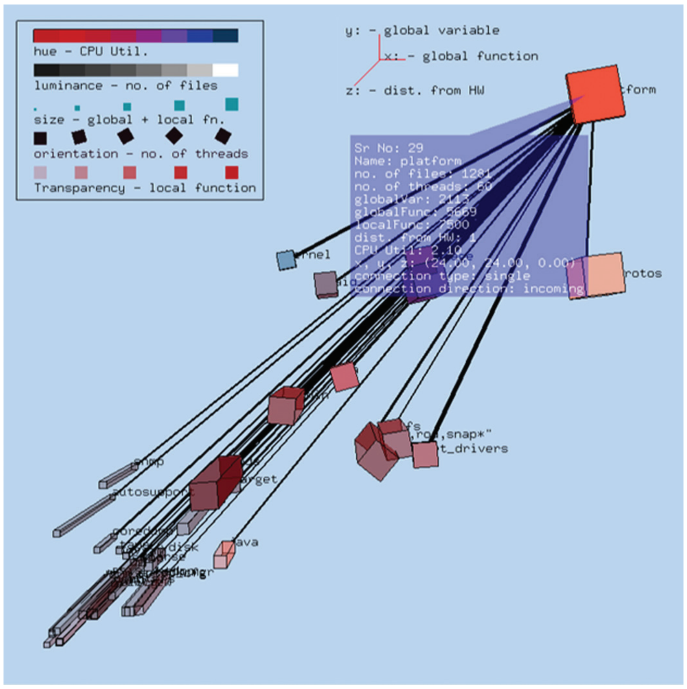
```
Sr No: 29
Name: platform
no. of files: 1281
no. of threads: 60
globalVar: 2113
globalFunc: 5669
localFunc: 7500
dist. from HW: 1
CPU Util: 2.10
x, y, z: (0.00, 0.00, 0.00)
connection type: single
connection direction: incoming
```

Figure 2: Visualization of global symbol cross-reference table and CPU utilization: MDS incoming connections from the module platform.



```
Sr No: 29
Name: platform
no. of files: 1281
no. of threads: 60
globalVar: 2113
globalFunc: 5669
localFunc: 7500
dist. from HW: 1
CPU Util: 2.10
x, y, z: (24.00, 24.00, 0.00)
connection type: single
connection direction: outgoing
```

Figure 3: Visualization of global symbol cross-reference table and CPU utilization: 2-D grid outgoing connections from the module platform.

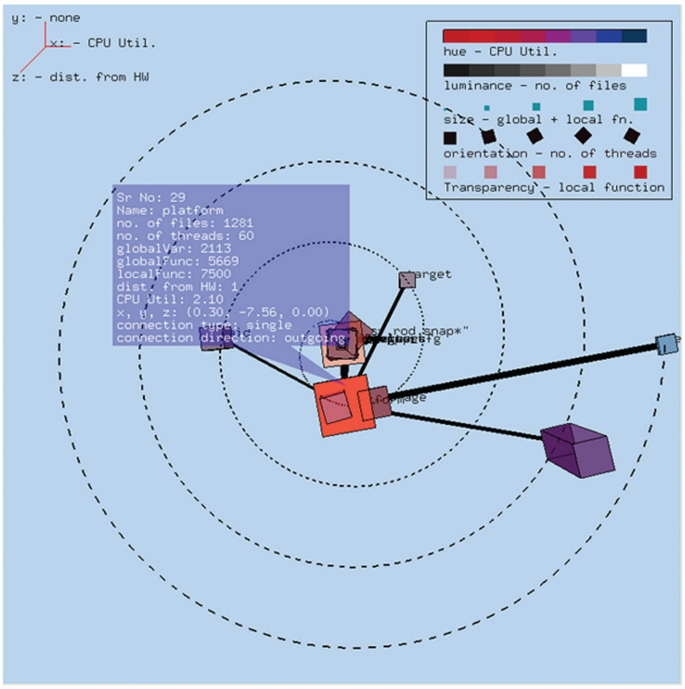


```
Sr No: 29
Name: platform
no. of files: 1281
no. of threads: 60
globalVar: 2113
globalFunc: 5669
localFunc: 7500
dist. from HW: 1
CPU Util: 2.10
x, y, z: (24.00, 24.00, 0.00)
connection type: single
connection direction: incoming
```

Figure 4: Visualization of global symbol cross-reference table and CPU utilization: 2-D grid incoming connections from the module platform.

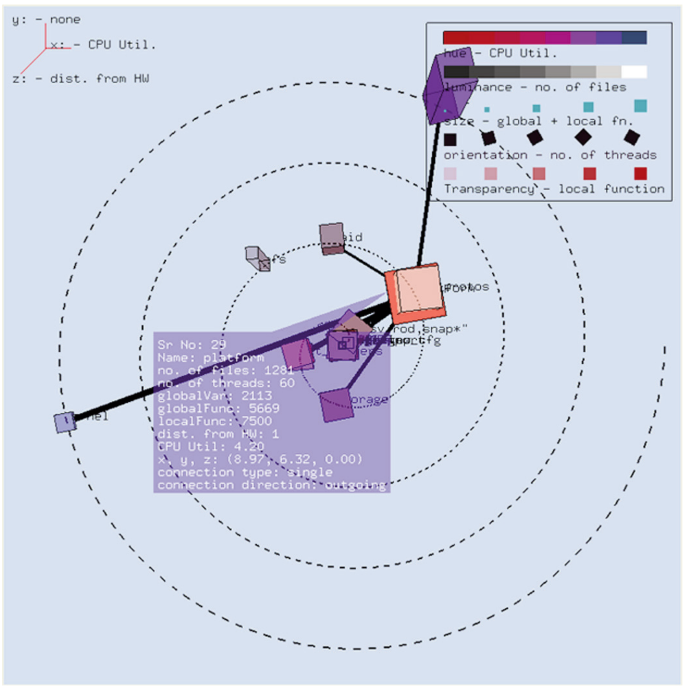
Interpretation

Figure 1 and Figure 2 were generated using the MDS technique. The placement of the software components shows how each component is related to every other component. The distance between any two components is inversely proportional to the number of references between them. The size of each glyph is directly proportional to the amount of code that resides in it and the hue represents the CPU utilization for the SPC1 benchmark: blue hues imply higher CPU utilization and red



```
Sr No: 29
Name: platform
no. of files: 1281
no. of threads: 60
globalVar: 2113
globalFunc: 5669
localFunc: 7500
dist. from HW: 1
CPU Util: 2.10
x, y, z: (0.30, -7.56, 0.00)
connection type: single
connection direction: outgoing
```

Figure 5: Visualization of global symbol cross-reference table and CPU utilization: spiral outgoing connections from the module platform, **SPC1** *benchmark.*



```
Sr No: 29
Name: platform
no. of files: 1281
no. of threads: 60
globalVar: 2113
globalFunc: 5669
localFunc: 7500
dist. from HW: 1
CPU Util: 4.20
x, y, z: (8.97, 6.32, 0.00)
connection type: single
connection direction: outgoing
```

Figure 6: Visualization of global symbol cross-reference table and CPU utilization: spiral outgoing connections from the module platform, **SFS** *benchmark.*

hues imply lower CPU utilization. These figures show that *platform* is the biggest component in the system and *kernel* is the busiest component for the SPC1 benchmark.

Figure 3 and Figure 4 are 2-D grid representations of the same software architecture displayed in Figure 1 and Figure 2, with one major difference: instead of using MDS for placing the glyphs, we map the number of global functions to the x-axis and the number of global variables to the y-axis. These figures show that in general, components with more global functions also have more global variables and vice versa.

In Figure 3 and Figure 4, most components are placed on a narrow diagonal area of the grid, which suggests that there exists an empirical global variable to global function ratio that all components appear to adhere to. We see that platform is the biggest component because it has the largest number of global variables and global functions. The kernel component is an outlier and has more global functions than global variables. This makes sense because kernel provides basic services to all other components and the mechanism by which other components access these services is by invoking global functions.

Figure 5 is a spiral representation of the software architecture displayed in the above images. Spiral representation is useful for viewing a single important attribute. In this case, CPU utilization is mapped to the radial position. The farther a component is from the center of the spiral, the higher the CPU utilization for SPC1. This image shows that the component with the highest CPU utilization is kernel. We also see that a small number of components are active during the benchmark because most of the components are clustered near the center.

Figure 6 is also a spiral layout, but it displays CPU utilization for the SFS benchmark. In this image we see that CPU utilization in nfs is significant, whereas in Figure 5, CPU utilization in target is significant. This is intuitively correct because SFS is a network filesystem benchmark that runs over the NFS protocol, whereas SPC1 is a storage-centric benchmark that runs over the SCSI initiator-target protocol.

We found that the MDS technique is useful for viewing the entire architecture as a single picture and comprehending the relationship of the various software components at a high level. 2-D grid layouts are better suited for isolating the effect of any set of two attributes, while spiral layouts allow us to focus on any single chosen attribute and study its impact on the software architecture.

Conclusions and Future Work

We use perceptual visualizations that harness the strengths and avoid the limitations of low-level human vision. Individual software components are presented using graphical glyphs that vary their spatial position, color, and texture properties to encode a component's attribute values. The result is a display that allows viewers to rapidly and accurately analyze, explore, compare, and discover within the software architecture. We have presented a method for automated software architecture visualization using raw source code as input and described three different graphical layouts that help in understanding different aspects of the software architecture. Our visualization technique is not restricted to architecture data and it can be applied to other 2-D relationship matrices. In the future, we would like to conduct user studies to evaluate the advantages and disadvantages of our multiple graphical layouts for visualizing software architecture.

Glossary

Glyph. Simple 3-D geometric graphical object.

SPC1. A sophisticated performance measurement workload for storage subsystems. SPC1 is a block-oriented benchmark that simulates the demands placed upon online, nonvolatile storage in a typical server class computer system [13].

SFS. Measures NFS file server throughput and response time. SFS is an NFS benchmark that provides a standardized method for comparing performance across different vendor platforms [14].

NFS. A network file system protocol originally developed by Sun Microsystems in 1984, allowing a user on a client computer to access files over a network as if the network device were attached to its local disks.

SCSI. Small computer system interface, a set of standards for physically connecting and transferring blocks of data (typically of 512 bytes) between computers and peripheral devices.

References

1. Architectural description languages. 2006. http://www.sei.cmu.edu/str/descriptions/adl_body.html (accessed 5/15/2006).
2. Auer, M., Graser, B., and Biffl, S. 2003. An approach to visualizing empirical software project portfolio data using multidimensional scaling. In *Proceedings of the IEEE International Conference on Information Reuse and Integration*. Las Vegas, NV. IEEE Systems, Man, and Cybernetics Society. 504-512.
3. Borgatti, S. P. 1997. Multidimensional scaling. http://www.analytictech.com/borgatti/mds.htm.
4. Carlis, J. V. and Konstan, J. 1998. Interactive visualization of serial periodic data. In *Proceedings of the 11th ACM Symposium on User Interface Software and Technology*. San Francisco, CA. 29-38.
5. Garlan, D., Monroe, R. T., and Wile, D. 2000. Acme: Architectural description of component-based systems. 47-67.
6. Garlan, D. and Shaw, M. 1993. An introduction to software architecture. In V. Ambriola and G. Tortora Eds, *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, Singapore. 1-39.
7. Holt, R. and Pak, J. Y. 1996. GASE: Visualizing software evolution-in-the-large. In *Proceedings of the 3rd Working Conference on Reverse Engineering* (*WCRE'96*). Washington, DC. IEEE Computer Society. 163.
8. Holten, D. 2006.Interactive software visualization within the reconstructor project (reconstructor: Reconstructing software architectures for system evaluation purposes). In *Proceedings of Visual Analytics Science and Technology Symposium*. Baltimore, MD.
9. Kazman, R. and Carrière, S. J. 1999. Playing detective: Reconstructing software architecture from available evidence. *Automat. Softw. Engin.* 6, 2. Kluwer Academic Publishers. 107-138
10. Knodel, J., Muthig, D., Naab, M., and Zeckzer, D. 2006.Towards empirically validated software architecture visualization. In *Proceedings of the ACM Symposium on Software Visualization*. New York, NY. 187-188.
11. Sawant, A. P. and Bali, N. 2007. DiffArchViz: A tool to visualize correspondence between multiple representations of a software architecture. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (*VISSOFT'07*). Banff, Canada. 121-128.
12. Sawant, A. P., Vanninen, M., and Healey, C. G. 2007. PerfViz: A visualization tool for analyzing, exploring, and comparing storage controller performance data. In *SPIE-IS&T Visualization and Data Analysis 6495*, 07. San Jose, CA. 1-11.
13. SPC benchmark-1 (SPC-1) official specification. 2006. http://www.storageperformance.org/specs/SPC-1_v1.10.1.pdf (accessed 3/10/2007).
14. SPEC SFS97 R1 V3.0—standard performance evaluation. 2006. http://www.spec.org/sfs97r1/ (accessed 3/10/2007).
15. Understand for C++. 2006. http://www.scitools.com/products/understand/cpp/product.php (accessed 3/10/2006).
16. Young, F. W. 1985. Multidimensional scaling. *Encyclopedia of Statistical Sciences 5*.

Biographies

Amit Prakash Savant (amit.sawant@ncsu.edu) *is a PhD student in the Computer Science Department at North Carolina State University. His research interests include computer graphics, scientific visualization, information visualization, visual perception, and databases. Web site:* http://www4.ncsu.edu/~apsawant.

Naveen Bali (naveen.bali@netapp.com) *is a performance engineer, and a member of the technical staff in the performance engineering group at Network Appliance, Inc. His research interests include performance optimization and visualization.*