STREAM-ENABLED FILE I/O FOR EMBEDDED SYSTEMS

Pramote Kuacharoen
Department of Computer Science, School of Applied Statistics
National Institute of Development Administration (NIDA)
118 Seri Thai Rd., Bangkapi, Bangkok 10240, Thailand
pramote@as.nida.ac.th

Vincent J. Mooney III and Vijay K. Madisetti
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
{*mooney, vkm*}@ece.gatech.edu

ABSTRACT

As embedded applications become more complex, file I/O operations such as read and write become increasingly important. However, file I/O operation latency may be significant when the file is located remotely. File I/O operation latency may be reduced by the means of incremental data delivery. Using this method, the data is not necessarily transmitted in a linear order of the data in the file, but is preferably transmitted in the order in which the data is used. Therefore, the application can obtain needed data more quickly. Furthermore, transmission bandwidth and memory usage may be lowered since unneeded data may not be sent.

In this paper, we present a stream-enabled file I/O method which allows data files to be streamed to an embedded device. The experimental results show that our implementation improves file I/O operation latency; in our examples, the performance improves up to 4.95X and 55.83X when compared with network file system and direct download, respectively.

1. INTRODUCTION

Often an embedded application includes data inside its program file. For instance, a game application may embed data for rendering a scene in the program code. However, as the data becomes larger, embedding the data in the program file becomes impractical. Moreover, the data may change over time, which obsoletes the application embedding the old data. Therefore, it is preferable to keep the data sepa-

rate from the application code itself and supply on demand the needed data to the application.

Traditionally, many embedded applications perform file I/O operations to request needed data from a remote server. However, file I/O operations may take a significant amount of time when the file is located remotely. This may cause the application(s) to be suspended for a long time. Alternatively, only the needed data within a data file may be sent to the client device on demand. In this case, the suspension time may be shorter. However, if different data in the file is accessed frequently, the application may suffer from the network latency.

In this paper, we propose a method to send file data incrementally and to allow embedded applications to access the file while transmission may still be in progress. Similar to media streaming such as audio and video streaming, we allow data to be processed before the download of the entire file is completed. However, unlike media streaming, we allow random data accesses and updates. We also profile the data so that it may be more likely to be transmitted in the order in which the data is used. Our objectives are to reduce I/O latency (the time from when an embedded application performs a file I/O operation to when the embedded application can continue running) and to minimize resource utilization such as bandwidth and memory.

This paper consists of six sections. Section 2 reviews related work in the area of file I/O. Section 3 describes our stream-enabled file I/O implementation. Section 4 discusses experimental results. Finally, Section 5 concludes the paper.

2. RELATED WORK

There are many protocols used to transfer files across a network. These protocols are implemented to serve different purposes based on the needs and the limitations of client applications. In this section, we discuss existing work related to our research.

2.1 DIRECT DOWNLOAD (DD)

One of the simplest means to transfer files is to use Direct Download (DD). In the DD implementation, the server sends the entire file to the client. Usually, the

client application waits for the completion of the download before starting to process the data. Downloading a large file takes a significant amount of time via a typical network connection. Hence, the user may have to wait for some time before the data can be used. There are a few variants of the DD implementation. For example, a classic implementation of DD is File Transfer Protocol (FTP) [10]. The FTP client downloads the entire file from the server. Another variant is Java FileInputStream [4]. In FileInputStream, a Java Virtual Machine (JVM) downloads a file from a server via HTTP [2].

2.2 VIRTUAL FILE SYSTEM (VFS)

VFS allows a client machine to mount directories located at server machines. The client machine can access files in the mounted directories as if they are local files. A typical flow for accessing a remote file is as follows: the user process invokes a system call (e.g., read), the kernel dispatches the command to the VFS and the VFS handles the request (for example, if the read command is issued, the VFS obtains the data either from the server or local cache and copies the data to user memory). Two well-known virtual file systems are Andrew File System (AFS) [1] and Network File System (NFS) [11]. AFS is a distributed file system with a common name space. Data are stored in volumes on AFS file server machines and accessed through a cache manager on AFS client machines. AFS is a large scale file system; there is only one AFS on the Internet. Every AFS cell is under the same AFS root directory. AFS may be too complex for a small embedded device. On the other hand, NFS is a workgroup file system which is designed to serve a small number of clients. NFS is supported by Linux which has been ported to many embedded devices. Therefore, NFS can be mounted on embedded devices. Unlike our method, both AFS and NFS use only file caching not profiling. We explain how file profiling can improve file access performance in Section 4.

2.3 INCREMENTAL SOFTWARE DELIVERY (ISD)

The ISD technique enables applications to run on a client device without having the whole program downloaded. The program code is incrementally delivered

while the application is running. The program code increments may be transmitted on demand or in the background.

A Java applet is an example of an on-demand ISD implementation; a Java applet can be run without obtaining all of the classes used by the applet. Java class files can be downloaded on demand from a server. If a Java class is not available to a JVM when an applet attempts to invoke the class functionality, the JVM may dynamically retrieve the class file from the server [4],[7]. The drawback of the on-demand ISD implementation is that the program code increments are only sent when the client requests them. Therefore, the application may suffer from being suspended due to missing program code. However, this issue can be mitigated by using a streamed ISD implementation which allows program code increments to be transmitted in the background. In the streamed ISD implementations [3],[5],[6],[12], the program is broken up into parts (increments) and then is profiled for transmission. The program code increments are sent according to the profile without waiting for the client to make a request. The program data is usually embedded in the program code and must be transmitted before the program code. As mentioned previously, when the data is large, embedding the data in the program is impractical.

Our method, on the other hand, enables data to be streamed to the client device while allowing an application to access data concurrently with transmission, instead of embedding data inside program file. Moreover, the application may need to update its data files.

3. STREAM-ENABLED FILE I/O

In some applications, data can be much larger than executable code. For an application using a large amount of data, the application usually reads data from files instead of embedding the data in the program file. Downloading the entire data before the application can start reading would cause the application to be suspended for a long time. Furthermore, the application may use only a subset of the data at one time. Therefore, requiring all data to be available to the application at once is typically unnecessary. Example 1 shows a quantitative comparison between downloading the entire file versus just downloading the needed data.

Example 1: Assume that a game application contains a 4 MB data file. However, the game application needs to process only 1 MB of the data (a single scene) before the user can begin to play the game. If we transfer the entire file over a 128-Kbps link, it will take over 262 seconds (approximately 4 minutes and 22 seconds). On the other hand, if we transfer 1 MB of data and allow the game application to start processing the data, it will take only approximately 65 seconds. Therefore, in this case, the game application can utilize the data needed for the scene 4X faster when compared to transferring the entire file. □

In addition, an application may also take some time to process the input data. By enabling the application to process part of the needed data while the rest of the data is concurrently being downloaded, the total amount of time required to access and process the needed data can be reduced. This is because the application usually has to wait for transmission. While waiting for transmission, the application can perform some computation on the data already loaded. Interleaving the transmission and the computation of data is faster than serializing data transmission and the computation.

Using the quantitative comparisons from Example 1, we propose a Stream-enabled file I/O (SIO) implementation which transfers files in blocks and allows the application to access files while transmission may still be in progress. The block transmission is not necessarily performed in a linear order of the blocks in the file, but is preferably performed in the order in which the blocks are used. Therefore, the application can obtain the needed data more quickly. Moreover, the transmission bandwidth and memory usage may be lowered since unneeded data may not be transmitted. We describe our implementation in the following subsections.

3.1 SIO PROTOCOL

SIO implements network file I/O using a client-server model. The design goal is to enable applications running on a client device to perform file I/O operations on files stored at a server. At the server, we divide files into blocks which we call *data blocks* and then create a transmission profile (information how to transmit data blocks) for each file. Please note that we assign an ID to each data blocks sequentially the same as the order in which they appear in the file, starting from 0.

When a client requests a file, the SIO server sends the file information (e.g., file size and block size) and streams data blocks to the client according to the transmission profile. At the client, when the application opens a file, the SIO client sends a request to receive the file from the server and uses the file information to construct file status information. The file status information contains a data block table. Each entry in the data block table is an address field for storing the location of the data block. However, if the address is invalid (i.e., the address value of 0xFFFFFFFF), the data block is not yet loaded; otherwise, the data block is in memory. The offset of the address field is the block ID of the data block. Figure 1 shows an example of a data block table. When the SIO client receives a data block, the corresponding address field entry is updated to the location where the data block is stored. For example, in Figure 1, when data block with an ID of 1 is received, the SIO client loads the block into memory starting at 0x00010400 and updates the address field of the entry at offset 1 with the starting location of the data block.

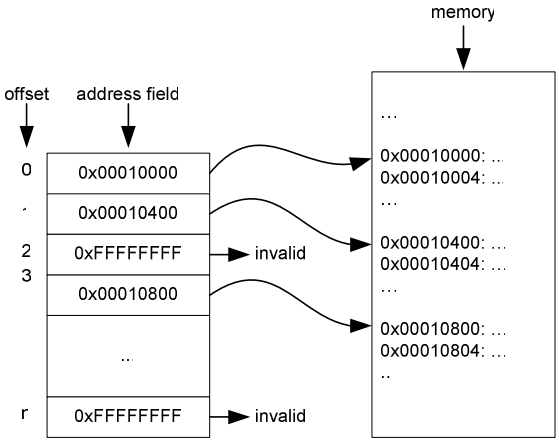

Figure 1. Data block table.

By using a data block table, SIO function calls can determine whether or not the data block is available. If the block is missing, the SIO client sends a request to the SIO server for the needed block. Otherwise, the application can access the data in the block. For limited storage devices, unneeded data blocks can be removed by changing the address fields of the blocks to an invalid address and freeing the memory occupied by the unneeded data blocks. If a block is later needed, it will be requested for retransmission.

In our current implementation, we only allow a server to transmit fixed size blocks. However, the data block containing the end of file can be smaller than others.

4. DATA PROFILING

Data profiling is used to predict the runtime file access behavior of the application in order to stream data accordingly. Data which are most likely to be used first should be streamed first. Using data profiling, the data miss rate and application suspension time due to missing data can be significantly reduced. Furthermore, data which is not needed by the application may not be sent at all, saving memory and bandwidth. In this section, we discuss profiling of data files.

A data file is profiled at the block level. However, we do not rearrange data within a block or across multiple blocks; we leave the data structure inside the file unchanged. Thus, since block boundaries are at points dictated by the fixed sizes (e.g., all blocks of size 4 KB), a data structure may be split across two blocks. In this case, in order to obtain the entire data structure, both blocks must be streamed. After the file is divided into blocks, we predict the order that the program uses the data blocks and then we create a flow graph for data transmission. A node (vertex) of the graph represents the data block and an edge linking two nodes indicates the possible flow of the transmission. We also assign a weight to each edge of the flow graph; the higher the weight is, the higher probability that the block corresponding to the next node will be sent. Then, we create a transmission profile of the file by traversing the flow graph. Data blocks are sent according to the profile. The profile of the file may be dynamically or statically updated based on the statistical usage of the file collected from the actual file access pattern of the application.

One of the simplest access patterns is a sequential access pattern; the application reads the data file sequentially from start to finish. Profiling a sequentially-accessed file is still beneficial. For instance, as mentioned in Example 1, the game application needs to process only 1 MB of data to allow the user to play the game; therefore we can profile the data in such a way that the first scene is sent to the client with a higher data rate from the subsequent scenes. In this way, the subsequent

scenes will be sent while the user is playing the first scene without utilizing resources at such high rate as the first scene. When the user advances to the next scene, the data may be ready at the client device.

When the data access pattern is unordered and unpredictable, profiling may be difficult or impossible. However, if the data access pattern is known or has a certain characteristics, we profile the file so that the data will be sent in a similar fashion to the data access pattern or the characteristics. Therefore, the application would have access to data more quickly.

5. EXPERIMENTAL RESULTS

We implemented a stream-enabled file I/O method in C. We tested our implementation on an MBX860 board [8] running Linux version 2.4.21 [14]. As shown in Figure 2, the MBX860 board is connected to a local area network via a 10-Mbps Ethernet port. However, the server is located in a different subnet which means the traffic is routed through network devices such as routers and switches.

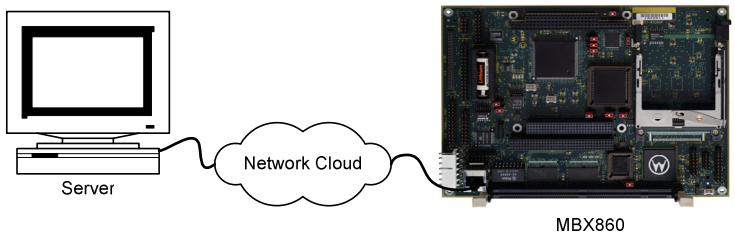

Figure 2. Experimental setup.

We also used the Linux Traffic Shaper (shapercfg version 2.2.12) to regulate the connection speed. In all experiments, we configured the connection speed to 128 Kbps. Then, we measured the performance of our SIO and compared the results with the results obtained by NFS and DD. For the DD, we implement a version of DD using a TCP socket to download the entire file first and then allow the application to access the data.

Experiment 1: Reading data file using various benchmarks. In this experiment, we created four benchmarks, namely, Seq, Rand 1K, Stat and BSearch to test the performance of SIO, NFS and DD. These four benchmarks simulate typical

activities for reading data from files. The Seq benchmark sequentially reads a 1-MB data file with a minimal amount of data processing; the data is read and assigned to a variable. This benchmark simulates applications which read an entire file into memory. The Rand 1K benchmark randomly reads 1 KB of data from a 1-MB data file. Since data access is random, this benchmark tests the performance under such circumstances when the application's data accesses are unordered and unpredictable. The Stat benchmark calculates various statistical values of data in a 1-MB data file. This benchmark simulates applications which interleave reading and processing data. Finally, the BSearch benchmark finds a specific value in a 1-MB file whose data is sorted in ascending order. This benchmark tests the performance of reading data files which have a known, non-sequential data access characteristic.

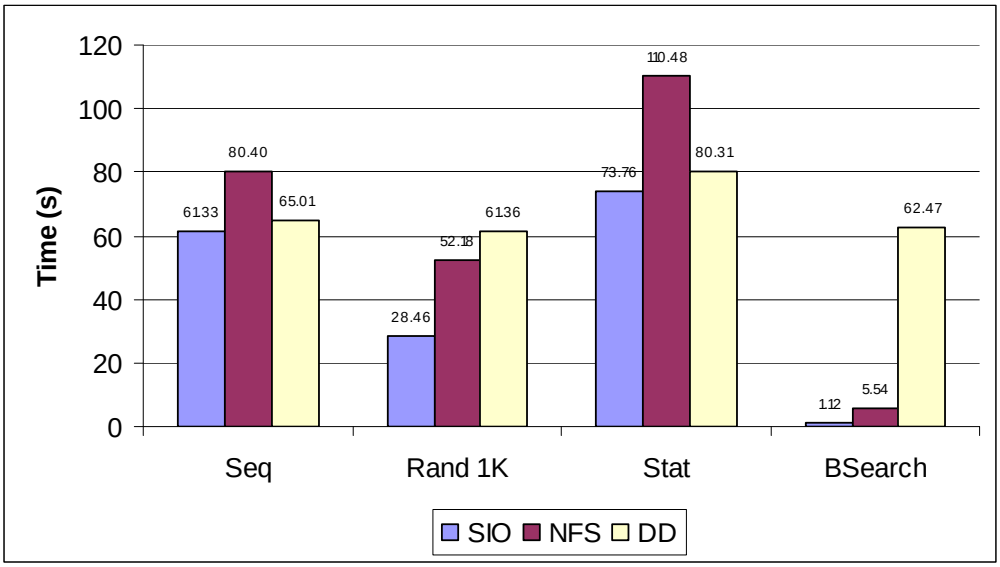

Figure 3. File I/O Performance Comparisons

The performance comparison of SIO, NFS and DD using these four benchmarks is shown in Figure 3. Figure 3 shows the time taken to stream and process a 1-MB data file used by each benchmark. For the Seq benchmark, SIO is 1.31X faster than NFS. However, SIO performs almost the same as DD since the whole file is transmitted and data processing is minimal. For Rand 1K benchmark, SIO is 1.83X and 2.16X faster than NFS and DD, respectively. In this benchmark, a subset of the data is required at the client. Obviously, downloading the whole file takes longer. For Stat benchmark, SIO outperforms NFS and DD. Even though the whole file is

needed, SIO allows computations while the file is being transferred. For BSearch benchmark, SIO is 4.95X and 55.83X faster than NFS and DD. The performance of SIO is much better than both implementations because SIO uses data file profiling as described in Section 4. Data file profiling predicts which data block is needed first. Therefore, the server will stream the block accordingly.

Experiment 2: Data acquisitions. In this experiment, we measured the amount of time the application takes to acquire a certain amount of data from a 1-MB file over a 128-Kbps connection and we also compared our implementation with NFS and DD. The data is read sequentially from the beginning of the file until the required amount of data is acquired. Note that, in this experiment, we do not process data; we read the data and store it in memory. The results are plotted in Figure 4.

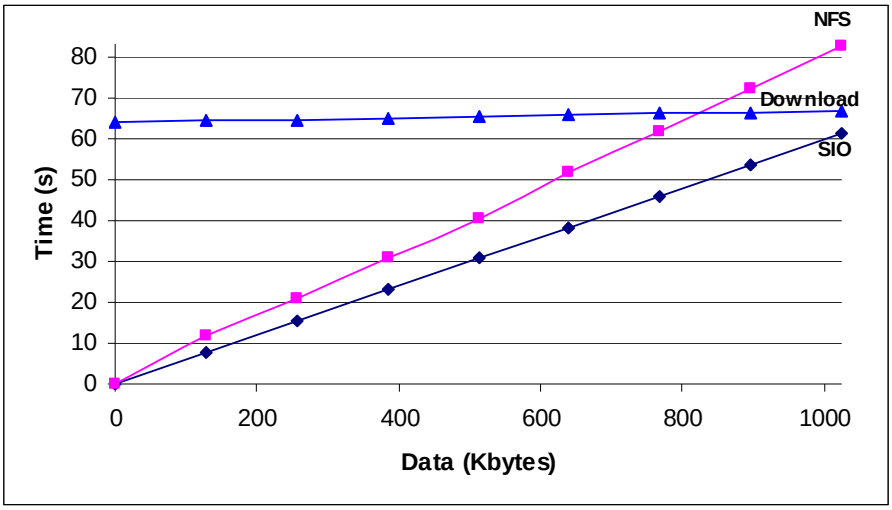

Figure 4. Time to acquire data from a 1-MB file.

For the DD implementation, the amount of time to acquire data varies only slightly with data size since the entire file must be downloaded independent of the amount of data needed. In contrast, SIO and NFS implementations allow the application to process the data after a subset of the data is loaded. Therefore, the amount of time to acquire a particular amount of data for both implementations depends on the size of the data. In other words, the amount of time the application takes to acquire data is proportional to the size of the data. However, the amount of time to acquire data via SIO approaches the amount of time to download the entire file as the size of

data approaches the size of the file while the amount of time to acquire data via NFS is more than the time to download the entire file.

Experiment 3: Data utilization rate. In this experiment, we measured the amount of time its takes to process a 1-MB file using various data utilization rates (the rates that the application uses data) over a 128-Kbps connection. The intention of the experiment is to show the effects of the data utilization rates on the amount of time required to process all data when reading a file which must be transferred over a certain connection speed. The results are illustrated in Figure 5.

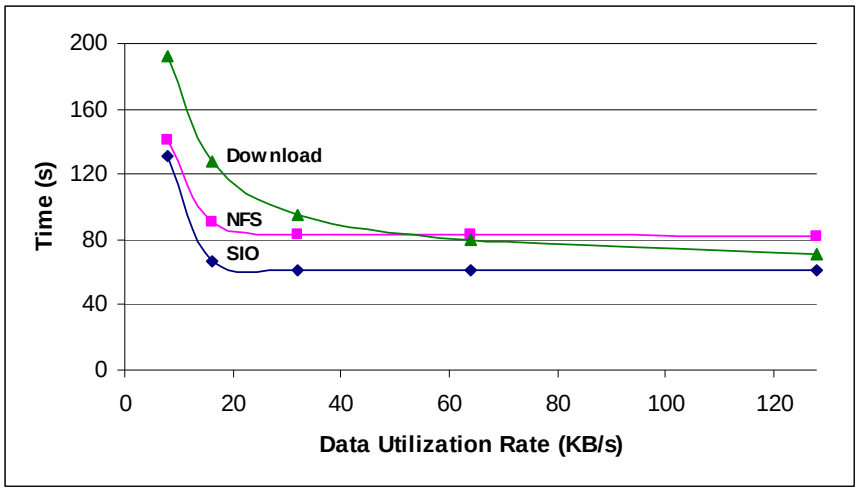

Figure 5. Data Utilization Rate vs. Time

SIO implementation outperforms both NFS and Download implementation. When the data utilization rate is less than connection data rate, the application spends more time processing the data after all data is transfer to the client. However, when the data utilization rate is greater than connection data rate, total time is dominated by transfer time.

Experiment 4: Combining stream-enabled file I/O and stream-enabled program file. In this experiment, we combined block streaming for program file method (SPG) and block streaming for data file method (stream-enabled file I/O). Then, we compared the user perceived application load time (the amount of time from when the application is selected to download to when the application can interact with the user) with the user perceived application load times obtained when run-

ning the application via SPG, NFS and DD. In the SPG implementation, we embedded all data in the program code.

Therefore, all data must be streamed first. We created a simple game application which has a program file of size 512 KB and a data file of size 1 MB. The game application contains four scenes, and each scene is rendered using 256 KB of data. The code needed for rendering the scene occupies 128 KB of memory. The user can start playing the game after the first scene is rendered. The amount of time the user has to wait before playing the game is shown in Figure 6.

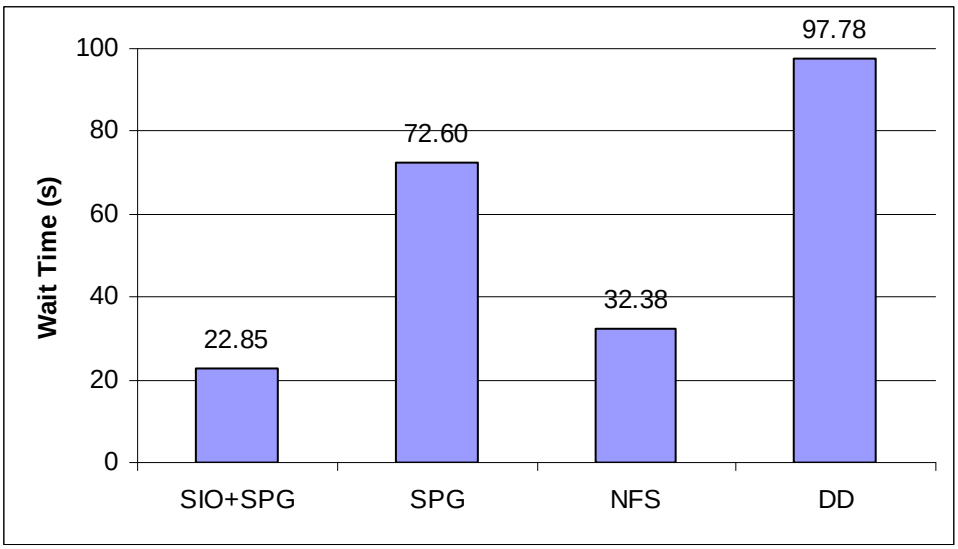

Figure 6. The amount of time the user has to wait before playing the game.

Using a combination of SIO and SPG (SIO+SPG), the user can start playing games 3.18X more quickly than using SPG alone, 1.42X more quickly than using NFS, and 4.28X more quickly than using DD. For the SPG implementation, all data must be streamed before the needed program code. Therefore, the game application can start rendering the scene when all data and the needed program code are loaded. As a result, SPG significantly underperforms NFS. If the game application were implemented using SPG for the program file and DD for the data file, the performance would still be bounded by file I/O.

6. CONCLUSION

File I/O operations may be accelerated using our stream-enabled file I/O method. The application can access the data more quickly since the data is likely to be transmitted in the order in which it will be used. We presented a method for transmitting a data file from a server to a client. We tested our implementation using an MBX860 board running an embedded Linux. The experimental results show that our implementation outperforms the other comparative methods; in our examples, the performance improves up to 4.95X and 55.83X when compared with network file system and direct download, respectively.

Advantageously, with the combination of program file streaming and the new data file streaming techniques introduced in this paper, the user can interact with the application more quickly, and small embedded devices can run many more applications as if they were fully downloaded and installed already. In addition, the embedded device may overcome memory limitations since unneeded code and data may not be sent to the device.

7. REFERENCES

[1] Carnegie Mellon University, "AFS Reference Page," http://www-2.cs.cmu.edu/afs/andrew.cmu.edu/usr/shadow/www/afs.html.

[2] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T., "Hypertext Transport Protocol — HTTP/1.1," RFC 2616, The Internet Engineering Task Force, June 1999.

[3] Huneycutt, M. H., Fryman, J. B., Mackenzie, K. M., "Software Caching using Dynamic Binary Rewriting for Embedded Devices," *Proceedings of International Conference on Parallel Processing*, August 2002, pp. 621-630.

[4] Lindholm T. and Yellin F., *The Java Virtual Machine Specification*, 2nd ed., Massachusetts: Addison-Wesley Publishing Company, 1999, pp. 158-161.

[5] Krintz, C., Calder, Brad, Lee H. and Zorn B., "Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998, pp. 159-169.

[6] Omitted for blind review.

[7] Meyer, J. and Downing, T., *Java Virtual Machine*, California: O'Reilly & Associate, 1997, pp. 44-45.

[8] Motorola, Inc., "MBX Datasheet," http://mcg.motorola.com/us/ds/pdf/ds0134.pdf.

[9] Nahum, E., Barzilai, T. and Kandlur, D. D., "Performance Issues in WWW Servers," *IEEE/ACM Transactions on Networking*, vol. 10, no. 1, pp. 2-11.

[10] Postel, J. and Reynolds, J., "File Transfer Protocol (FTP)," RFC 959, The Internet Engineering Task Force, October 1985.

[11] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M. and Noveck, D., "Network File System (NFS) version 4 Protocol," RFC 3530, The Internet Engineering Task Force, April 2003.

[12] Raz, U., Volk, Y. and Melamed, S., "Streaming Modules," U.S. Patent 6,311,221, July 22, 1998.

[13] Silberschatz, A. and Galvin, P. B., *Operating System Concept*, 4th ed., Massachusetts: Addison-Wesley Publishing Company, 1994, pp.253-255.

[14] The Linux Kernel Archives, http://www.kernel.org.