Software Streaming via Block Streaming

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

Abstract

Software streaming allows the execution of stream-enabled software on a device even while the transmission/streaming may still be in progress. Thus, the software can be executed while it is being streamed instead of causing the user to wait for the completion of download, decompression, installation and reconfiguration. Our streaming method can reduce application load time seen by the user since the application can start running as soon as the first executable unit is loaded into the memory. Furthermore, unneeded parts of the application might not be downloaded to the device. As a result, resource utilization such as memory and bandwidth usage may also be more efficient. Using our streaming method, an embedded device can support a wide range of real-time applications. The applications can be run on demand. In this paper, a streaming method we call block streaming is proposed. Block streaming is determined at the assembly code level. We implemented a tool to partition real-time software into parts which can be transmitted (streamed) to the embedded de-vice. Our streaming method was implemented and simulated on a hardware/software co-simulation platform in which we used the PowerPC architecture. We show a robotics application that without our streaming method is unable to meet its real-time deadline. However, with our software streaming method, the application is able to meet its deadline. The application load time for this application also improves by a factor of more than 10X when compared to downloading the entire application before running it.

1. Introduction

Today's embedded devices typically support various applications with different characteristics. With limited storage resources, it may not be possible to keep all features of the applications loaded on an embedded device. In fact, some software components may not be needed at all. As a result, the memory on the embedded device may not be efficiently utilized. Furthermore, the application software will also likely change over time to support new functionality, and perhaps quite rapidly in the case of game software [1]. Today, the user has to download the software and install it. This means that the entire software must be downloaded before it can be run. Downloading the entire program delays its execution. In other words, *application load time* (the amount of time from when the application is selected to download to when the application can be executed) is longer than necessary. To minimize the application load time, the software should be executable while downloading. *Software Streaming* enables the overlapping of transmission (download) and execution of embedded software.

Embedded software to be streamed must be modified before it can be streamed over a transmission media. The software must be partitioned into parts for streaming. We call the process of modifying code for streaming *software streaming code generation*. We perform this modification after normal compilation. After modification, the application is ready to be executed after the first executable software unit is loaded into the memory of the device. In contrast to downloading the whole program, software streaming can improve application load time. While the application is running, additional parts of the stream-enabled software can be downloaded in the background or on-demand. If the needed part is not in the memory, the needed part must be transmitted in order to continue executing the application. The application load time can be adjusted by varying the size of the first block while considering the time for downloading the next block. This can potentially avoid the application suspension due to block misses. The predictability of the software execution once it starts can be improved by *software profiling* which determines the transmission order of the blocks.

In this paper, we present a new method for software streaming. The transmission of the software can be completely transparent to the user. The software is executed as if it is local to the device. Our streaming method can also significantly reduce the application load time since the CPU can start executing the application without downloading the entire program.

This paper consists of five sections. The first section introduces the motivation of the paper. The second section describes related work in the area of software streaming. In the third section, our method for streaming real-time embedded software is presented. We provide performance analysis of the streaming environment in the fourth section. Finally, in the fifth section, we present an example application and results.

1530-1591/03 $17.00 © 2003 IEEE

2. Related work

In a networking environment, a client device can request certain software from the server. A typical process involves downloading, decompressing, installing and configuring the software before the CPU can start executing the program. For a large program, download time can be very long. Typically, application load time refers to time between when the application is selected to run and the time when the first instruction of the software is executed. Transmission time of the software predominantly contributes to the application load time. Hence, a long download time is equivalent to a long application load time. A long application load time experienced by the user is an undesirable effect of loading the application from the network.

In Java applet implementation, the typical process of downloading the entire program is eliminated. 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 the server. If a Java class is not available to the Java Virtual Machine (JVM) when an executing applet attempts to invoke the class functionality, the JVM may dynamically retrieve the class file from the server [2], [3]. In theory, this method may work well for small classes. The application load time should be reduced, and the user should be able to interact with the application rather quickly. In practice, however, Web browsers make quite a few connections to retrieve class files. HTTP/1.0, which is used by most Web servers [4], allows one request (e.g., for a class file) per connection. Therefore, if many class files are needed, many requests must be made, resulting in large communication overhead. The number of requests (thus, connections) made can be reduced by bundling and compressing class files into one file [5], which in turn unfortunately can increase the application load time. While the transition to persistent connections in HTTP/1.1 may improve the performance for the applet having many class files by allowing requests and responses to be pipelined [6], the server does not send the subsequent java class files without a request from the client. The JVM does not request class files not yet referenced by a class. Therefore, when a class is missing, the Java applet must be suspended. For a complex application, the class size may be large, which requires a long download time. As a result, the application load time is also long, a problem avoided by the block streaming method which we will describe in the next section.

Raz, Volk and Melamed [7] describe a method to solve long load-time delays by dividing the application software into a set of modules. For example, a Java applet is composed of Java classes. Once the initial module is loaded, the application can be executed while additional modules are streamed in the background. The transmission time is reduced by substituting various code procedures with shortened streaming stub procedures, which will be replaced once the module is downloaded. Since software delivery size varies from one module to another, predicting suspension time may be difficult. However, in block streaming, this issue can be avoided by streaming fixed-size blocks.

Eylon et al. [8] describe a virtual file system installed in the client that is configured to appear to the operating system as a local storage device containing all of the application files to be streamed required by the application. The application files are broken up into pieces called streamlets. If the needed streamlet is not available at the client, a streamlet request is sent to the server and the virtual file system maintains a busy status until the necessary streamlets have been provided. In this system, overheads from a virtual file system may be too high for some embedded device to support. Unlike the method in [8], our block streaming method does not need virtual file system support.

Software streaming can also be done at the source code level. The source code is transmitted to the embedded device and compiled at load time [9]. Although the source code is typically small compared to its compiled binary image and can be transferred faster, the compilation time may be very long and the compiler's memory usage for temporary files may be large. Since the source code is distributed, full load-time compilation also exposes the intellectual property contained in the source code being compiled [10]. Moreover, a compiler must reside in the client device at all times, which occupies a significant amount of storage space. This method may not be appropriate for a small memory footprint and slower or lower-power processor embedded devices.

3. Block streaming

In this section, we present software streaming via block streaming. The presented streaming method is lightweight in that it tries to minimize bandwidth overheads, which is a key issue for streaming embedded software. In our approach, the embedded application must be modified before it can be streamed to the embedded device. As shown in, the block-streaming process on the server involves (i) compiling the application source code into an executable binary image by a standard compiler such as gcc, (ii) generating a new binary for the stream-enabled application and (iii) transmitting the stream-enable application to the client device.

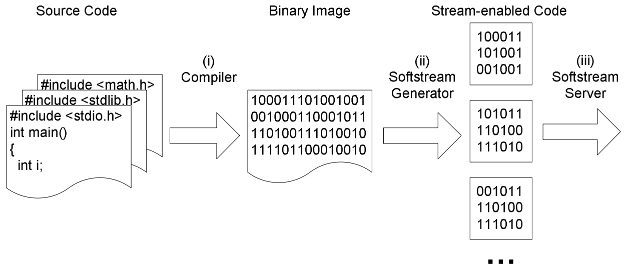


Figure 1. Server-side block-streaming process.

The process to receive a block of streamed software on the client device is illustrated in Figure 2: (i) load the block into memory and (ii) link the block into the existing code. This "linking" process may involve some code modification which will be described later.

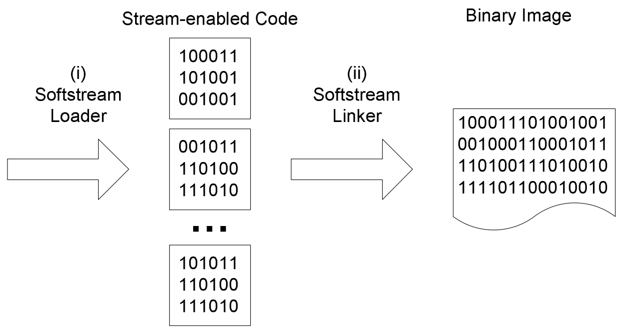


Figure 2. Client-side block-streaming process.

We define a code block to be a contiguous address space of data or executable code or both. A block does not necessarily have a well-defined interface; for example, a block may not have a function call associated with the beginning and ending addresses of the block, but instead block boundaries may place assembly code for a particular function into multiple, separate blocks. The compiled application is considered as a binary image occupying memory. This binary image is divided into blocks before the stream-enabled code is generated. The size of each block of the same application may be different. The block size may be determined from streaming parameters such as network speed.

After the application is divided into blocks, exiting and entering a block can occur in three ways. First, a branch or jump instruction can cause the CPU to execute code in another block. Second, when the last instruction of the block is executed, the next instruction can be the first instruction of the following block. Third, when a return instruction is executed, the next instruction would be the instruction after calling the current block subroutine, which could be in another block. We call a branch instruction that may cause the CPU to execute an instruction in a different block an *off-block* branch. All off-block branches to blocks not yet loaded must be modified for streaming to work properly. (Clearly, the reader may infer that our approach involves a form of binary rewriting.)

The method for transferring blocks to the client device is explained in detail in Section 3 of [11]. Once the first block of an application is downloaded, the Softstream Loader shown in Figure 2 loads block to the allocated memory and stores the address of the block in the block look-up table. After the Softstream Loader finishes, the block-streamed application begins execution. Execution continues fine until an off-block branch is taken. Figure 3 shows the sequence when a modified off-block branch is first taken. The Softstream Loader is invoked to load the needed block. If the needed block is not fully in memory, the needed block is, if not already done previously, requested to be streamed. The block lookup table is used to check if the block is in the memory. After the needed block is in memory, the instruction which invokes the Softstream Loader routine is modified to jump to the proper location. The program execution is then resumed.

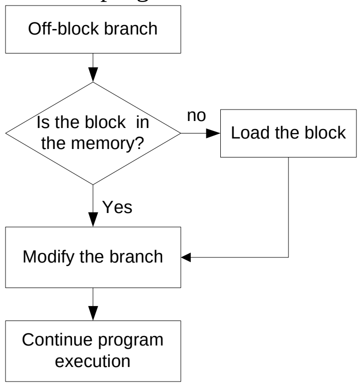


Figure 3. Softstream Loader flow chart.

3.1. Embedded software streaming code generation

In our approach, the stream-enabled embedded software is generated from the executable binary image of the software. The executable binary image of the software is created from normal compilation. Stream-enabled code generation can be done statically or dynamically. Static code generation is performed before the software is requested by the user. Generating the stream-enabled code statically does not contribute to load-time overhead, once created, since the stream-enabled software is always ready to be transmitted. On the other hand, dynamic stream-enabled code generation is done while the software is being streamed. This may add some overhead. However, in dynamic stream-enabled code generation, the streaming can be more adaptive to environmental conditions such as network congestion as the code generator modifies off-block branch instructions and creates corresponding information for Softstream Loader.

Example 1: Consider the if-else C statement in Figure 4. The C statement is compiled into corresponding PowerPC assembly instructions. The application can be divided so that the statement is split into different blocks. Figure 4 shows a possible split.

```
if (i==1)
  i=0;
else
  i=1;
```

Block 1

```
      cmpwi 0,01
      bc 4,1, .L3
      li 0,0
      stw 0, 8(31)
      b .L4
```

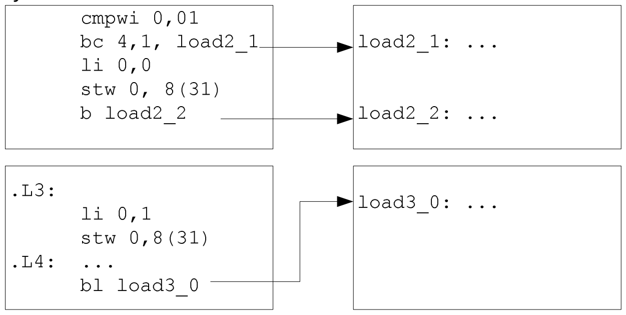
Block 2

```
.L3:
      li 0,1
      stw 0,8(31)
.L4:
      ...
```

Figure 4. C code and corresponding PowerPC assembly.

In this simple example, the first block (Block 1) contains two branch instructions, each of which could potentially jump to the second block. If the second block (Block 2) is not in memory, this application will not work properly. Therefore, these branch instructions must be modified. All branch instructions that may cause the CPU to execute instructions from different blocks are modified to invoke the appropriate loader function if the block is not in memory. After the missing block is loaded, the intended location of the original branch is taken. Figure 5 shows Block 1 and Block 2 from Figure 4 after running the software streaming code generation program on the application code. Branch instructions `bc 4,1, .L3` and `b .L4`, as seen in Figure 4, are modified to `bc 4,1, load2_1` and `b load2_2`, respectively, as seen in Figure 5.

Suppose the last instruction of a block is not a branch or a return instruction. If left this way, the CPU will automatically execute the first instruction of the following block. To prevent the CPU from executing code of a non-existing block, an instruction is appended by default to load the next block. For example, as shown in Figure 5, the instruction `bl load3_0` will be appended to Block 2 from Figure 4 to load the subsequent block. The instruction `bl load3_0` can be replaced later by the first instruction of the block after Block 2 in order to preserve code continuity. □



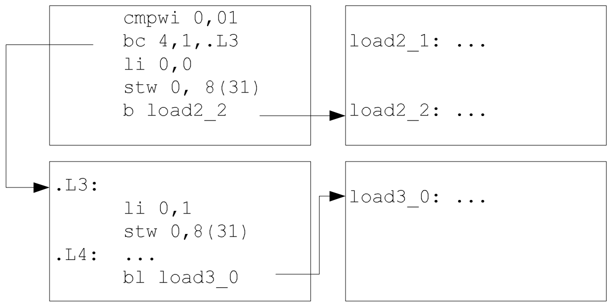
```
      cmpwi 0,01
      bc 4,1, load2_1          load2_1: ...
      li 0,0
      stw 0, 8(31)
      b load2_2                load2_2: ...

.L3:
      li 0,1                   load3_0: ...
      stw 0,8(31)
.L4:  ...
      bl load3_0
```

Figure 5. Block 1 and Block 2 after the stream-enabled code generation.

3.2. Runtime code modification

After an off-block branch is taken and the block to which the branch leads is loaded into memory, the off-block branch instruction will be modified to jump to a correct address location, instead of invoking the same loader function. Although runtime code modification introduces some overhead, the application will run more efficiently if the modified branch instruction is executed frequently. This is because, after modification, there is no check as to whether or not the block is in memory, but instead the modified branch instruction branches to the exact appropriate code location.

Example 2: Suppose that the software from Figure 4 is modified using the software streaming code generated as illustrated in Figure 5 and is running on an embedded device. When an off-block branch is executed and taken, the missing block must be made available to the application. Figure 6 shows the result after the branch to load Block 2 is replaced with a branch to location .L3 in Block 2 (look at the second instruction in the top left box Figure 6). The runtime code modification only changes the instruction which issues the request. Other branches remain untouched even if the corresponding block is already in memory. If such a branch instruction is executed to load a block already in memory, the branch instruction will be modified at that time. □



```
      cmpwi 0,01
      bc 4,1,.L3               load2_1: ...
      li 0,0
      stw 0, 8(31)
      b load2_2                load2_2: ...

.L3:
      li 0,1                   load3_0: ...
      stw 0,8(31)
.L4:  ...
      bl load3_0
```

Figure 6. Runtime code modification.

4. Performance analysis

In order to take advantage of software streaming, the streaming environment must be thoroughly analyzed. The streaming environment analysis will likely include CPU speed, connection speed, block size and the program execution path. Without knowledge about the environment for software streaming, the performance of the system can be sub-optimal. For instance, if the block size is too small and the CPU can finish executing the first block faster than the transmission time of the second block, the application must be suspended until the next block is loaded. This would not perform well in an interactive application. Increasing the block size of the first block will delay the initial program execution, but the application may run more smoothly.

4.1. Performance metrics

Overheads:

The obvious overhead is the code added during the stream-enabled code generation step for block streaming. For the current implementation, each off-block branch adds 12 bytes of extra code to the original code. The stream-enabling code (added code) for off-block branches consists of (i) the ID of the needed block, (ii) the original instruction of the branch, and (iii) the offset of the instruction. Since each field (i-iii) occupies four bytes, the total is 12 extra bytes added. The original code and the stream-enabling code are put together into a streamed unit. A streamed unit must be loaded before the CPU can safely execute the code. Therefore, the added stream-enabling code incurs both transmission and memory overheads. Increasing the block size may reduce these overheads. However, a larger block size increases the application load time since the larger block takes longer to be transmitted.

Runtime overheads are associated with code checking, code loading and code modification. Code loading occurs when the code is not in the memory. Code checking and code modification occur when an off-block branch is first taken. Therefore, these overheads from the runtime code modifications eventually diminish.

Application load time:

Application load time is the time between when the program is selected to run and when the CPU executes the first instruction of the selected program. The application load time is directly proportional to the size of data and is inversely proportional to the speed of the transmission media. The program can start running earlier if the application load time is lower. The application load time can be estimated as explained in detail in [11].

Application suspension time:

Application suspension occurs when the next instruction that would be executed in normal program execution is in a block yet to be loaded or only partially loaded into memory. The application must be suspended while the required block is being streamed. The worst case suspension time occurs when the block is not in memory; in this case the suspension time is the time to load the entire block which is the transmission time of the streamed block. The best case occurs when the block is already in memory. Therefore, the suspension time is between zero and the time to load the whole block. Application suspension time is proportional to the streamed block size. The application developer can vary the block size to obtain an acceptable application suspension time if a block miss occurs.

For applications involving many interactions, low suspension delay is very crucial. While the application is suspended, it cannot interact with the environment or the user. Response time can be used as a guideline for application suspension time since the application should not be suspended longer than response time. A response time which is less than 0.1 seconds after the action is considered to be almost instantaneous for user interactive applications [12]. Therefore, if the application suspension time is less than 0.1 seconds, the performance of the stream-enabled application should be acceptable for most user interactive applications when block misses occur.

4.2. Background streaming

In some scenarios, it may be a better solution if the application can run without interruption due to missing code. By allowing the components or blocks to be transmitted in the background (background streaming) while the application is running, the needed code may be in memory prior to being needed. As a result, execution delay due to code misses is reduced. The background streaming process is only responsible for downloading the blocks and does not modify any code.

4.3. On-demand streaming

In some cases where bandwidth and memory are scarce resources, background streaming may not be suitable. Certain code blocks may not be needed by the application in a certain mode. For example, the application might use only one filter in a certain mode of operation. Memory usage is minimized when a code block is downloaded on-demand, i.e., only when the application needs to use the code block. While downloading the requested code, the application will be suspended. In a multitasking environment, the block request system call will put the current task in the I/O wait queue and the RTOS may switch to run other ready tasks. On-demand streaming allows the user to trade off between resources and response times

4.4. Software profiling

It is extremely unlikely that the application executes its code in a linear address ordering. Therefore, the blocks should preferably be streamed in the most common order of code execution. This can minimize delays due to missing code. Software profiling can help determine the order of code to be streamed in the background and on-demand. When the software execution path is different from the predicted path, the order of background streaming must be changed to reflect the new path. A software execution path can be viewed as a control/data flow graph (CDFG) as illustrated in Figure 7. When the program execution flows along a certain path, the streaming will be conducted accordingly. For instance, if the execution path is through software block B3, B5 and B6 can be streamed in the background. B4 will not be streamed at all. A path prediction algorithm is necessary for background streaming to minimize software misses. The block size for the paths which are unlikely to be taken can be determined according to the appropriate suspension delay. Currently, we manually predict the software execution path.

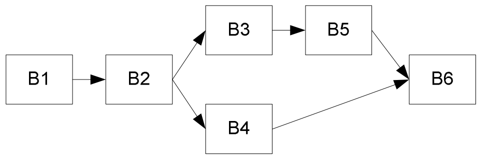

$$\begin{tikzcd} & & B3 \arrow[r] & B5 \arrow[dr] & \\ B1 \arrow[r] & B2 \arrow[ur] \arrow[dr] & & & B6 \\ & & B4 \arrow[urr] & & \end{tikzcd}$$

Figure 7. CDFG of a software execution path.

5. Simulation Results

An adaptive system application below was simulated using hardware/software co-simulation tools described in [11]. We use the block-streaming method to transmit the software from the server to the client device.

In robot exploration, it is impossible to write and load software for all possible environments that the drone will encounter. The programmer may want to be able to load some code for the robot to navigate through one type of terrain and conduct an experiment. As the robot explores, it may roam to another type of terrain. The behavior of the robot could be changed by newly downloaded code for the new environment. Furthermore, the remote robot monitor may want to conduct a different type of experiment which may not be programmed into the original code. The exploration would be more flexible if the software can be sent from the base station. When the robot encounters a new environment, it can download code to modify the robot's real-time behavior. The new code can be dynamically incorporated without reinitializing all functionality of the robot.

In this application, we used the block streaming method to transmit the software to the robot. A binary application code of size 10MB was generated. The stream-enabled application was generated using our Softstream code generation tool. There were three off-block branch instructions on average in each block. The software was streamed over a 128Kbps transmission media. Table I shows average overheads of added code per block and load time for different block sizes. The average added code per block is 36 bytes (due an average in each block of three off-blocks branches each of which adds 12 bytes). This overhead is insignificant for block sizes larger than 1KB. The load times were calculated using only transmission of the block and the streamed code. We did not include other overhead such as propagation delay and processing.

Table I. Simulation results for block streaming.

Block size (bytes)	Number of blocks	Added code/block	Load time (s)
10M	1	0.00%	655.36
5M	2	0.00%	327.68
2M	5	0.00%	131.07
1M	10	0.00%	65.54
0.5M	20	0.01%	32.77
100K	103	0.04%	6.40
10K	1024	0.35%	0.64
1K	10240	3.52%	0.06
512	20480	7.03%	0.03

Without using the block streaming method, the application load time of this application would be over 10 minutes (approximately 655 seconds). If the robot has to adapt to the new environment within 120 seconds, downloading the entire application would miss the deadline by more than eight minutes. However, if the application were broken up into 1MB or smaller blocks, the deadline could be met. Even the strict deadline is not crucial, the block streaming method reduces the application load time by more than ten times for the block sizes of 1MB or less. The load time for block size of 1MB is approximately 65 seconds whereas the existing method application load time is more than 655 seconds.

If the software profiling predicts the software execution path correctly, the application will run without interruption due to missing blocks; the subsequent-needed blocks can be streaming in the background while the CPU is executing the first block. However, if the needed block is available at the client, the application must be suspended until the needed block is downloaded. If the size of the missing block is 1KB, the suspension time is only 0.06 seconds. As mentioned previously, this suspension delay of less than 0.1 seconds is considered to be almost instantaneous for user interactive applications. While our sample application is not a full industrial-strength example, it does verify the block streaming functionality and provides experimental data.

6. Conclusion

Embedded software streaming allows an embedded device to start executing an application while the application is being transmitted. We presented a method for transmitting embedded software from the server to be executed on a client device. Our streaming method can lower load-time delay, bandwidth utilization and memory usages. We verified our streaming method using a hardware/software co-simulation platform for the PowerPC architecture, specifically for MPC 750 processors.

7. Acknowledgements

This research is funded by the State of Georgia under the Yamacraw initiative [16]. We acknowledge donations received from Denali, Hewlett-Packard Company, Intel Corporation, LEDA, Mentor Graphics Corp., SUN Microsystems and Synopsys, Inc.

8. Reference

[1] R. Avner, “Playing GoD: Revolution for the PC with Games-on-Demand,” *Extent Technologies*, http://www.gamesbiz.net/keynotes-details.asp?Article=248.

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

[3] J. Meyer and T. Downing, *Java Virtual Machine*, California: O’Reilly & Associates, Inc., 1997, pp. 44-45.

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

[5] P. S. Wang, *Java with Object-Oriented Programming and World Wide Web Applications*, California: PWS Publishing, 1999, pp. 193-194.

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

[7] U. Raz, Y. Volk and S. Melamed, *Streaming Modules*, U.S. Patent 6,311,221,October 30, 2001.

[8] D. Eylon, A. Ramon, Y. Volk, U. Raz and S. Melamed, *Method and System for Executing Network Streamed Application*, U.S. Patent Application 20010034736, October 25, 2001.

[9] G. Eisenhauer, F. Bustament and K. Schwan, “A Middleware Toolkit for Client-Initiate Service Specialization,” Operating *Systems Review*, vol. 35, no. 2, 2001, pp. 7-20.

[10] M. Franz, “Dynamic Linking of Software Components,” *Computer*, vol. 30, pp. 74-81, March 1997.

[11] P. Kuacharoen, V. Mooney and V. K. Madisetti, “Software Streaming via Block Streaming,” Georgia Institute of Technology, Atlanta, Georgia, Tech. Rep. GIT-CC-02-63, 2002.

[12] W. Stallings, *Operating Systems*, 2nd ed., New Jersey: Prentice Hall, 1995, pp. 378-379.

[13] Synopsys Inc., http://www.synopsys.com

[14] Seamless CVE™, http://www.mentor.com/seamless

[15] Mentor Graphics XRAY® Debugger, http://www.mentor.com/embedded/xray

[16] Yamacraw, http://www.yamacraw.org