EMBEDDED SOFTWARE STREAMING VIA BLOCK STREAMING

A Dissertation
Presented to
The Academic Faculty

by

Pramote Kuacharoen

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
April 2004

EMBEDDED SOFTWARE STREAMING VIA BLOCK STREAMING

Approved by:

Professor Vincent J. Mooney III, Committee Chair

Professor Mostafa Ammar

Professor Biing-Hwang Juang

Professor Vijay K. Madisetti

Professor Karsten Schwan

Date Approved: 7 April 2004

To my mother

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to everyone who helped make this dissertation possible. First and foremost, I would like to thank my advisor, Professor Vincent J. Mooney III, for his patience and guidance during my graduate study at Georgia Tech. With his knowledge and experience, he has guided me to achieve my research objective. I truly admire him for all his encouragement he has given me throughout this process. He was always there when I needed him.

Second, I would like to thank my dissertation committee members, Professor Mostafa Ammar, Professor Biing-Hwang Juang, Professor Vijay K. Madisetti, and Professor Karsten Schwan, for their critical evaluation and valuable suggestions for fine tuning the focus of my dissertation.

Third, I would like to thank all members of the Codesign group for their friendship and support, especially, Mohamed A. Shalan and Tankut Akgul who are coauthors of a number of the conference papers. We had many good times together at conferences and had many interesting conversions during lunch times.

Finally, I would like to thank my family for their love and support throughout my entire graduate study. My mother always encourages me to work hard. To our beloved brother, I did not have much chance to spend time with you, but you will be missed forever and will always be in my heart.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Downloading software from a server usually takes a noticeable amount of time, that is, noticeable to the user who wants to run the program. However, this issue can be mitigated by the use of streaming software. Software steaming is a means by which software can begin execution even while transmission of the full software program may still be in progress. Therefore, the application load time observed by the user can be significantly reduced. Moreover, unneeded software components might not be downloaded to the device, lowering memory and bandwidth usages. 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 applications which can be run on demand. Software streaming also enables small memory footprint devices to run applications larger than the physical memory by using our software streaming technique.

Traditionally, an embedded application includes data inside its program file since the amount of data has historically been quite small. However, as embedded applications become more complex, the amount of data required may be larger. Therefore, embedding the data in the program file may become impractical. Thus, file I/O operations such as file read and file 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, file 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 short, this dissertation also addresses streaming of software programs which use significant amounts of file I/O.

In this dissertation, we present a streaming method we call *block streaming* to transmit stream-enabled applications, including stream-enabled file I/O. We implemented a tool to partition software into blocks which can be transmitted (streamed) to the embedded device. Our streaming method was implemented and simulated on an MBX860 board and 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 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. The experimental results also show that our implementation improves file I/O operation latency; in our examples, the performance improves up to 55.83X when compared with direct download.

CHAPTER I

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. Today, the user has to download the software and install it prior to use. 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 for 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 software.

1.1 Problem Statement

This research addresses long application load times when running software over a networked environment. The amount of time required to download large software from a server can be significant, thus delaying the execution of the software. This research also addresses utilization of resources such as bandwidth and memory; since many features downloaded to the device may not be used at all, these unused features waste the user's time, bandwidth, and memory. As a result, many embedded devices which have limited storage resources cannot support large applications.

The primary objective of the research is to implement a method for executing software on a device while transmission/streaming may still be in progress, reducing the amount of time from when the application is selected for download to when the application can be executed. Furthermore, the secondary objective is to reduce the occurrence of and the amount of time during which an application is suspended or stalled during execution due to missing code. The tertiary objective of the research is to provide support for small memory footprint embedded devices.

1.2 Contributions

In this dissertation, we present a new method for streaming software. 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.

The following items are contributions of this research.

- Block streaming allows the execution of stream-enabled software on a device even while transmission/streaming of the software may still be in progress. Thus, the user does not have to wait for the completion of download before running the program. As a result, the user will experience a relatively shorter application load time.

- Block streaming enables client devices, especially embedded devices, to potentially support a wider range of applications by more efficiently utilizing resources — such as memory and bandwidth — and more efficiently supporting dynamic download and execution of programs.

- Block streaming provides applications with the ability to access file data without obtaining the entire data file. Parts of the data in the data file can be manipulated while other parts of the file data are being transferred. As a result, applications can potentially access file data more quickly.

- Block streaming facilitates software distribution and software updates since software is directly streamed from the server. In case of a bug fix, the software developer can apply a patch at the server. The up-to-date version of the software is then always streamed to the client device.

- Block streaming has the potential to dramatically alter the way software is executed in the embedded setting where minimal application load time for newly selected applications is important to clients.

While security is a major concern for all modern computing systems, we do not cover security issues in this dissertation. However, we do present in Chapter 7.3 a few known approaches to security which we believe could be used in conjunction with this dissertation to implement secure software streaming.

1.3 Terminology for Software Streaming

In this section, we define terms which we use in this dissertation.

Application load time: The amount of time from when an application is selected for download to when the application can be executed.

Application suspension time: The amount of time from when an application is suspended due to missing code to when the application can be resumed.

Off-block branch: a branch instruction that may cause the CPU to execute an instruction in a different code block.

Softstream client: A device which executes a stream-enabled application.

Softstream protocol: A standard procedure for regulating data transmission between a softstream server and a softstream client.

Softstream server: A program which accepts and responds to requests from a softstream client.

Stream-enabled application: An application which can be executed while the transmission/streaming of the software may still be in progress.

Stream-enabled software: Software which can be executed while the transmission/streaming of the software may still be in progress.

Stream-enabling information: Data which is used to received and integrate stream units into a stream-enabled application.

Stream block: Contiguous executable code and/or data.

Stream unit: A payload which consists of the stream unit header and associated stream block.

User perceived application load time: The amount of time from when an application is selected to download to when the application can interact with the user.

1.4 Organization of the Dissertation

The dissertation is organized into nine chapters:

CHAPTER I: INTRODUCTION. This chapter provides a general overview of software streaming. The chapter also provides the terminology for software streaming and the contributions of the dissertation.

CHAPTER II: MOTIVATION AND PERSPECTIVE. This chapter describes some motivation for this dissertation. This chapter also describes the perspective of software streaming via block streaming.

CHAPTER III: RELATED WORK. This chapter reviews work related to software streaming, specifically, in the areas of conventional memory management and client/server computing. For conventional memory management, this chapter surveys overlaying, paging, and dynamic linking. For client/server computing, this chapter surveys methods for transferring files across the network and executing applications.

CHAPTER IV: SOFTWARE STREAMING VIA BLOCK STREAMING. This chapter explains the concept of our software streaming method. This chapter explains our software streaming client/server model and the protocol. Performance metrics used to evaluate our software streaming implementation are also discussed. This chapter also explains how software profiling is essential for software streaming.

CHAPTER V: STREAM-ENABLED PROGRAM FILES. This chapter describes the process of generating a stream-enabled program from its binary image by partitioning the program file into blocks and modifying the binary image. This chapter also describes the process of loading the code blocks into memory and linking the code blocks together by using a binary rewriting technique.

CHAPTER VI: STREAM-ENABLED FILE I/O. This chapter describes stream-enabled file I/O which enables data in the file to be used without downloading the entire file. The data file is divided into blocks and is profiled. Then, the server sends data blocks according to the transmission profile.

CHAPTER VII: BLOCK STREAMING PERFORMANCE ENHANCEMENT. This chapter describes a method to improve code locality by remapping binary code at the function level. Remapping binary code can reduce the occurrence of block misses and, hence, the number of software suspensions. This chapter also presents a method which enables small memory footprint embedded devices to support large applications which cannot be loaded into memory at once by using software streaming and allowing stream units to be removed.

CHAPTER VIII: EXPERIMENTS AND RESULTS. This chapter discusses the experiments and results of our software streaming implementation. The experiments were conducted on an MBX860 board and a hardware/software cosimulation platform.

CHAPTER IX: CONCLUSION. This chapter concludes the dissertation.

CHAPTER II

MOTIVATION AND PERSPECTIVE

Computing environments have evolved from one computer serving many people, e.g., a mainframe, to one computer serving one person, e.g., a personal computer (PC). Current and future computing environments have many computers serving one single person. In the future, computing resources will become ubiquitous. Many embedded devices and sensors will be used to support everyday activities [1]. An evolution of computing environments are illustrated in Figure 1.

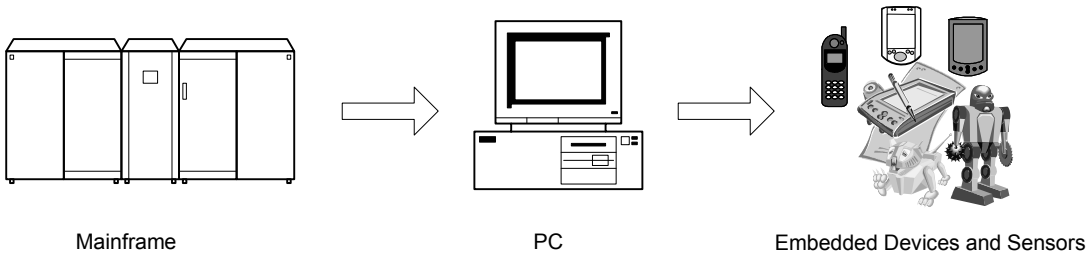

Figure 1: An evolution of computing environments.

Software distribution has also changed over the past 40 years. Initially, software was predominantly packaged and was delivered through postal mail or was sold at stores. This still happens today although not as frequently. As the Internet becomes more prevalent, software can be easily downloaded or can be run remotely. However, future computing environments will demand a new approach for distributing software. One possible method to deliver software to future embedded devices is through software streaming. A possible evolution of software distribution is shown in Figure 2.

Consider a scenario where a user utilizes a portable device to download an application from a remote server and executes the application. As the availability and use of computing resources becomes more and more ubiquitous, this scenario is likely to

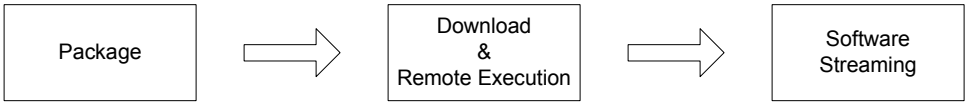

Figure 2: A possible evolution of software distribution.

become quite common. The user may have to wait a long time to execute a cutting edge application which the user has selected for the first time.

The problem is the user demands small embedded devices — such as cellular phones and personal digital assistants which have limited resources — to run many applications. However, a small embedded device cannot, by definition, hold thousands or more of the latest applications. Furthermore, the user does not want to wait for a long time for an application to start. A long wait time may be overcome by streaming program files. With streaming support, the application can start running without requiring all program code to be downloaded. Furthermore, most embedded programs are written in a manner similar to desktop programs and thus use a significant amount of file I/O. If the application is suspended for a long time while reading a data file, the user perceives that the application is loaded slowly. Therefore, we need to stream data files as well. With the combination of program file streaming and data file streaming techniques, the user can interact with the application more quickly, and small embedded devices can potentially run many more applications as if they were all fully downloaded and installed already.

Software to be streamed must be modified before it can be streamed over a transmission medium. The software must be partitioned into parts for streaming. We call the process of modifying code for streaming *software streaming code generation*, and we perform this modification after normal compilation. After modification, the application is ready to be executed after the software unit containing the program entry point is loaded into the memory of the device. In contrast to downloading the whole program, software streaming can improve the 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 memory, the needed part must be transmitted in order to continue executing the application. The application load time can be adjusted by varying the block sizes and the amount of blocks transferred before the application can be executed. The download time for the subsequent blocks must be considered to avoid suspension of the application 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. By using software profiling, the application suspension due to block misses can potentially be avoided.

In short, we propose streaming of embedded software as a new paradigm for software delivery.

CHAPTER III

RELATED WORK

There are two main areas which are related to this research, namely, conventional memory management and client/server computing. We survey conventional memory management techniques which cope with memory being scarce. We also discuss client/server computing which deals with transferring data across a network and executing applications stored at a server.

3.1 Conventional Memory Management

In this section, we present the conventional memory management techniques of overlaying and paging. First, we describe overlaying which is used to manage an application's memory without any assistance from an operating system (OS). Then, we present paging which is typically implemented in an OS.

3.1.1 Overlaying

For systems that have limited physical memory and that lack hardware support for memory management, a technique called overlaying can organize a large program and data in such a way that various modules can be assigned to the same memory space [34], [35]. The programmer writes an overlay manager which dynamically loads the instructions and data into memory as needed at any given time. When instructions are needed and memory is not available, the overlay manager evicts a block of instructions and loads the instructions into space that was previously occupied by the evicted instructions. Overlays do not require any special support from the operating system. The overlay manager reads a needed module from the program file on the

disk into memory and jumps to execute the code. The programmer must have complete knowledge of the structure of the program, its code, and its data structures in order to design the overlay structure properly.

3.1.2 Paging

Most virtual memory systems use a technique called paging to manage memory [37]. The paging technique permits the virtual address space of a process to be noncontiguous. When a program generates a virtual address, the virtual address goes to a memory management unit (MMU) instead of the memory bus. Then, the MMU translates the virtual address into a corresponding physical address and puts the physical address to the memory bus. The virtual address space is divided up into blocks, called pages, all of the same size. A range of physical memory can be mapped to multiple pages using a page table. The pages which are not currently being used can be swapped out while the needed page can be swapped in. Therefore, the paging technique allows the execution of programs which use more memory than the available physical memory.

3.1.3 Dynamic Linking

Dynamic linking refers to the method of deferring the linkage of external modules until after the load module has been created. Thus, the load module contains unresolved references to other programs. These references can be resolved either at load time or run time. For load-time dynamic linking, all unresolved references in the program are resolved. If the target module is not loaded, it must be loaded. The references are modified to the loaded locations. Since linking is done before program execution, it may take some time before the program can be executed. With run-time dynamic linking [9], [10], some of the linking is postponed until execution time. Therefore, the program execution can start sooner. However, when the program refers to an external reference, the target module must be loaded for the program to continue.

If the external reference is encountered again, there is no need to reload the target module. Run-time dynamic linking also minimizes the work of the linker since the only external references linked are those actually encountered at least once during execution.

3.2 Client/Server Computing

In a networked environment, a client device can request certain software from a 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. The download (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 the following subsection, we discuss several methods for executing an application stored at the server.

3.2.1 Direct Download

One of the simplest means to transfer files is to use direct download (DD). In DD, 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.

An application which may consist of program files and data files can also be downloaded from a server. When the entire application is downloaded, the user typically installs the application before running it. The time to download the software can be excessive for large programs and unacceptable to users, resulting in, for example,

users who refuse to update their software or who refuse to even download new software applications that they might otherwise desire (i.e., in the absence of excessive download times).

There are a few variants of DD. For example, a classic implementation of DD is the File Transfer Protocol (FTP) [28]. The FTP client downloads the entire file from the server. Another variant is Java's FileInputStream [18]. In FileInputStream, when an application reads a file, a Java Virtual Machine (JVM) downloads the entire file from a server via HTTP [8]. If the file is large, the download time is long and the application is suspended until the download is finished.

3.2.2 Remote Execution

Remote execution is a service which provides a user on a client access to remote applications. With remote execution, the applications are stored on a different machine and the local machine does not store or cache the applications. The user issues commands to the remote machine to execute commands or run programs. The remote machine sends output to display at the client. The remote applications are usually not highly interactive by nature; otherwise, network latency becomes an issue as applications are typically too slow to react to user input. Another drawback in remote execution is the server may not efficiently support all clients due to the server's limited computing resources. The server can be bombarded with requests. Hence, the server may not response to a particular client in a timely manner.

3.2.2.1 X Window System

The X Window System [27], [32], [46] provides a mechanism to run an X application (X client) remotely. The X server runs at a local computer, collecting input from the keyboard or mouse and accepting commands from an X client at a remote computer. The X server is also responsible for displaying graphics. When the user interacts with an X application, the user input is sent to the X client. The X client handles the user

input and sends commands to the X server to update the screen. In a high latency network, communication between the X server and the X client is very slow. The response time may be too high for an interactive application so that the application becomes unresponsive. Furthermore, an application which generates a huge amount of data may experience performance degradation [42]. Therefore, the X Window System only works well in low latency, high bandwidth networks such as local area networks.

3.2.2.2 Common Gateway Interface

Common Gateway Interface (CGI) [2], [5] is a standard for external gateway programs to interface with information servers such as Web servers. Typically, the user uses a program such as a Web browser on the local machine to send input to the server. The server runs a CGI program and sends the output from the CGI program back to the client. The client displays the output on the screen. For example, suppose the user wants to search a book title in a database using a Web browser. The Web browser sends a query to the server. After receiving the query, the server runs a CGI program and sends the query result back to the client. In this case, the server can be overloaded since it can handle only a certain amount of simultaneous requests. Besides processing time, there is also overhead associated with invoking a CGI script [43]. Therefore, CGI may not work well for a highly interactive application.

3.2.3 Virtual File Systems

A virtual file system (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).

3.2.3.1 Andrew File System

Andrew File System (AFS) [4] 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. A callback mechanism is used for cache consistency [38]. The file server keeps track of which clients have cached copies and notifies the client to invalidate its copy when the file is modified. Caching reduces traffic between the client and the server. When, for example, a client runs an application, the application is automatically cached. For subsequence calls, the client uses the cached version of the application. Since the client uses its local copy, the client does not have to communicate with the server. Therefore, no traffic is generated and the server is not interrupted to satisfy requests for commonly used files. When a new version of the application is installed, the server calls back all clients having cached the application. On the next execution of the application, the client realizes that the local copy is inconsistent with the one stored on the server and fetches the new version. 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.

3.2.3.2 Network File System

Network File System (NFS) [13], [31], [33] employs a client/server model for file sharing on a network. NFS uses remote procedure calls to manipulate files. Using the NFS protocol, a client can access file systems located on the server. The client uses memory to cache file attributes [3]. With the small memory-only cache, more network traffic is generated. As a result, NFS can support only a small number of clients. NFS works well in a local area network environment.

From an application perspective, the file systems are accessed as if they are local on the client machine. An advantage this environment is that one copy of the program can be shared by many users. The user also has access to many additional programs, which otherwise would not be available due to the fact that the combined total memory requirement for all the programs exceeds the local disk size. Moreover, NFS enables a diskless client (a machine that does not have a local disk) to mount a remote file system. NFS would be a possible solution for many embedded devices which do not have a local disk. The diskless client uses the server as a storage device. The embedded devices may run a version of embedded Linux which supports NFS. Then, the embedded devices can mount a remote file system using NFS. However, performance is a different issue. Unlike our method, NFS uses only file caching not profiling. We explain how file profiling can improve file access performance in Section 8.3.

3.2.3.3 A Low-Bandwidth Network File System

LBFS [25] is a network file system designed for low-bandwidth networks. The LBFS file server divides files into chucks and indexes the chucks by a hash value. Similarly, the LBFS client also indexes a large persistent file cache. The client is assumed to have enough cache to contain a user's entire working set of files. LBFS reduces bandwidth requirements by exploiting inter-file similarities. To avoid transmitting the redundant data over the network, LBFS identifies chucks of data that the recipient already has in other files. For example, a file written by an application often contains a number of segments in common with the previous version of the file. Therefore, there is no need to send the entire file back to the server when only a portion of the file is changed. Only the modified parts are sent back to the server.

LBFS only deals with reducing data transfer between the server and the client once the data is already transferred. The main contribution is to reduce bandwidth

by sending only the modified chucks. However, the method does address how quickly the file can be used initially at the client.

3.2.3.4 Method and System for Executing Network Streamed Application

Eylon et al. [7] 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 as 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, overhead from a virtual file system may be too high for some embedded devices to support. Unlike the method in [7], our block streaming method does not need virtual file system support.

3.2.4 Incremental Software Delivery

The incremental software delivery (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.

3.2.4.1 Java Applet Implementation

In Java applets, the typical process of downloading an 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 [18], [21], [44]. 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, allows one request (e.g., for a class file) per connection [26]. 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 [45], which in turn unfortunately can increase the application load time. Furthermore, very large Java applets exist [41], which definitely increase the application load time.

While using 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 [8], 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 in which the application can start running as soon as the first executable block is loaded into memory. Chapters III, IV, and V describe our implementation of software streaming via block streaming in detail.

3.2.4.2 Software Caching

Huneycutt et al. [14] propose a method to solve an extreme case of limited resource issues for networked embedded devices such as distributed sensors and cell phones; the solution is achieved by implementing software caching. In this system, the memory on the device acts as cache for a more powerful server. The server sends a section of instructions or data to the device. When the device requires code outside the section, a software cache miss occurs. The server is then requested to send the needed code. Dynamic binary rewriting is required to properly supply the code to the device which

incurs time overheads of at least 19% compared to no caching [14]. The program will also suffer frequent suspensions if software cache misses constantly occur. Our method allows software code to be streamed in the background which may reduce software suspension due to code misses.

3.2.4.3 Streaming Modules/Functions

Raz, Volk and Melamed [29] 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 our block streaming approach (see Chapter 4), this issue (unpredictable suspension time) can be lessened by streaming fixed-size blocks.

Krintz et al. [16] propose a non-strict form of mobile program execution by overlapping execution with transfer. Java is used as an execution environment. In the proposed model, a class file is partitioned into two parts, namely, global data with in the class and function code with its local data. The global data must be transferred first. Then each function can be sent while the class file is being executed. During the execution of a Java program, if a function is called but it or its data have not completed transferring, the program is stalled until the function code or data has transferred.

3.2.4.4 Liquid Software

Hartman et al. [12] introduce the idea of dynamically moving functions in a network. Their implementation uses location-independent code or mobile code which runs on a heterogeneous platform. Using mobile code, the execution state of one computer

can be moved to another. However, so-called *liquid software* must execute efficiently on each computing machine. Therefore, a very fast compiler (a compiler which can compile liquid software as fast as it can be transmitted over a network) is needed on every machine to translate the mobile code into native code. Liquid software requires every machine on the network to have high processing power.

3.2.4.5 Streaming Source Code

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 [6]. 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 device.

3.3 Summary

In this chapter, we provided an overview of conventional memory management and client/server computing. We will apply and extend several memory management techniques in our work. The related work in the area of client/server computing can help solve issues such as application load time, application suspension time due to missing a particular piece of code or data, how frequently the application suspensions occur (as opposed to the amount of time spent resolving a particular occurrence), low bandwidth, and limited memory. However, none of the prior work covered in this chapter solves these issues together simultaneously. For example, Java reduces application load time, but Java classes are sent when they are needed, increasing

the occurrence of application suspensions. Moreover, Java assumes that the client has enough memory to store the entire program. Function level streaming methods described in Section 3.2.4.3 may reduce the occurrence of application suspensions, but the methods also assume that the client has enough memory to load the entire program. On the other hand, software caching described in Section 3.2.4.2 assumes that memory is limited so the device memory acts as a cache for the server, increasing the occurrence of application suspensions. Thus, prior approaches tend to help solve one problem (e.g., limited local memory) at a cost of exacerbating another problem (e.g., occurrence of application suspensions). However, modern embedded devices require these issues to be solved together for the devices to effectively support the ever growing number of applications demanded by users. Therefore, our work aims to address these issues together, improving all of them simultaneously in a coordinated fashion.

CHAPTER IV

SOFTWARE STREAMING VIA BLOCK STREAMING

Software streaming (softstream) is a method for overlapping transmission and execution of stream-enabled software. The stream-enabled software can run on a device even while the transmission/streaming of the software may still be in progress. Thus, a user does not have to wait for the completion of the software's download prior to starting to execute the software. Using software streaming, the user can experience that the application is loaded very quickly since the application can starting running as soon as the stream unit containing the program entry point is loaded into memory.

In this chapter we will introduce the basics of software streaming. In the following chapter (Chapter 5), we will focus on the case of streaming program files. One chapter later (Chapter 6), we will extend our streaming approach to data files as well. For now, however, let us discuss how clients request streamed applications from servers.

4.1 Softstream Client/Server Model

In our software streaming implementation, we use a client/server model as illustrated in Figure 3. A *softstream server* stores stream-enabled applications. A *softstream client*, on the other hand, runs *stream-enabled applications*. A stream-enabled application, composed of *stream units*, is an application which can be safely executed while it is being sent (streamed). Each stream-enabled application has a corresponding transmission profile which is used to determine the order of the transmission of the stream units.

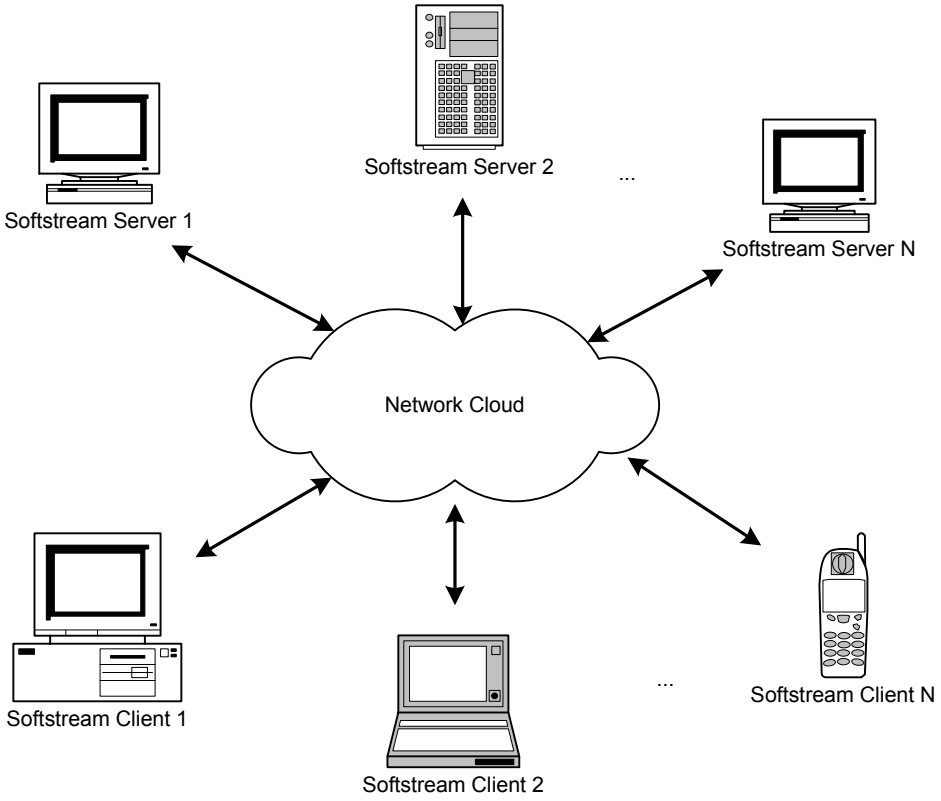

Figure 3: An exemplary computer network for software streaming.

When a softstream client requests a stream-enabled application, the softstream server sends the stream-enabled application to the softstream client. The softstream client loads the stream units into memory and links the stream units together. After loading the stream unit containing the program entry point, the softstream client can begin execution of the stream-enabled application even while the rest of the application is still in the process of being streamed. While the application is running, if a particular stream unit is needed but is currently missing, the softstream client sends a request to the softstream server to send the needed stream unit.

4.2 Stream-Enabled Software

Before a file can be transferred and used on a device (using our streaming method) the file must be modified and may be profiled as well (to optimize streaming performance). We treat program files and data files differently. For a program file, we need to

generate new binary images (including proper resolution of all branch instructions) from the original binary program image. However, for a data file, there are no branch instructions and so no new code needs to be generated. If profiled, each file has an associated transmission profile. Program file and data file issues are discussed in detail in Chapter 5 and Chapter 6, respectively.

We call a file ready to be streamed a *stream-enabled file* which is composed of stream units. As shown in Figure 4, a program file or a data file is divided into blocks, and then *stream units* are generated. A stream unit is composed of a *stream unit header* together with an associated *stream block*. A stream unit header contains information which is used to assemble stream blocks together at a softstream client, and a stream block is a data block or a new binary image code block generated from an original unmodified block of program code. The server sends the stream-enabled file by transmitting stream units. When the client receives a stream unit, the client assembles the stream unit into the stream-enabled file using the stream unit header in the stream unit.

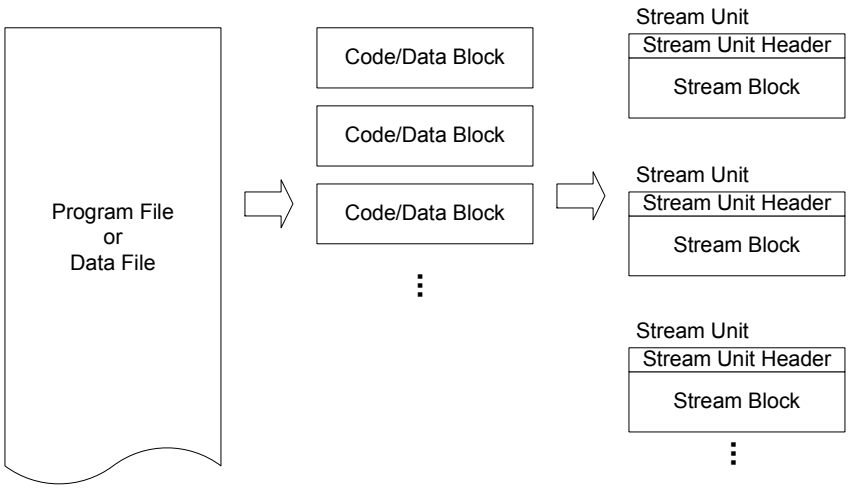

Figure 4: Generating stream units.

This dissertation focuses on single threaded applications. However, we do present in Chapter 7.3.3 some approaches to extend block streaming to support multi-threaded applications.

4.3 Softstream Protocol

The *softstream protocol* is an application-level protocol for software streaming via block streaming. The softstream protocol enables stream-enabled applications to be transferred and executed on a device. The softstream protocol is a request/reponse protocol, which means that a softstream client sends a request, and a softstream server responds to the request. The communication of the softstream protocol takes place over any reliable transport protocol layer such as Transmission Control Protocol (TCP) as shown in Figure 5(a), which means that error checking is performed by the transport and lower layer protocols. As shown in Figure 5(b), the client-side softstream protocol stack consist of three main layers: the *stream-enabled software* layer, *softstream assembly* layer (the *softstream loader/linker* and *stream-enabled file I/O* layer), and *softstream protocol* layer.

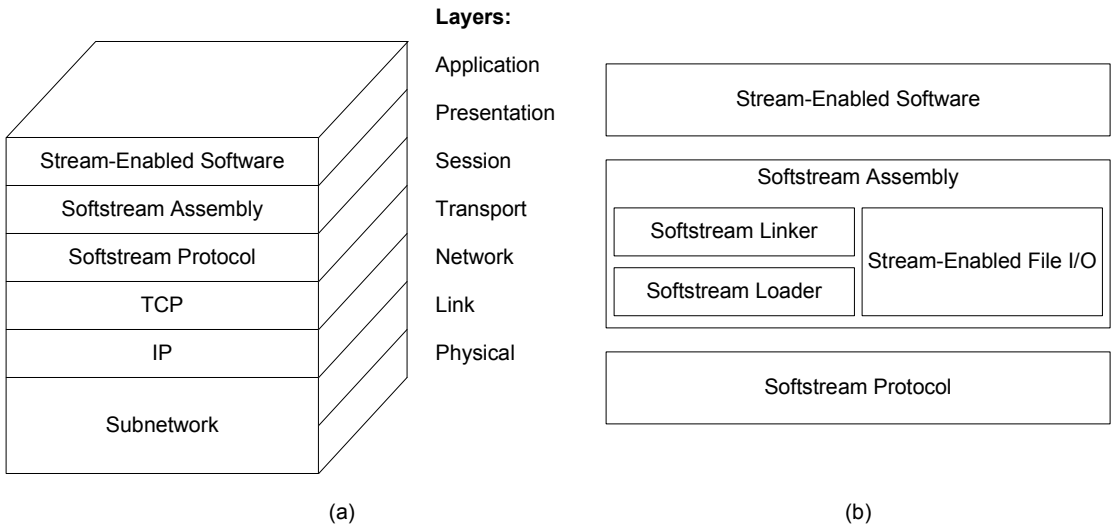

Figure 5: Client-side softstream protocol stack. (a) Softstream protocol stack in the OSI reference model. (b) Protocol stack detail.

The stream-enabled software layer is the application layer. A stream-enabled application is executed in this layer. Stream units are assembled together in the softstream assembly layer. Since there are significant differences between program files and data files, we handle them differently. For the program files, we need a

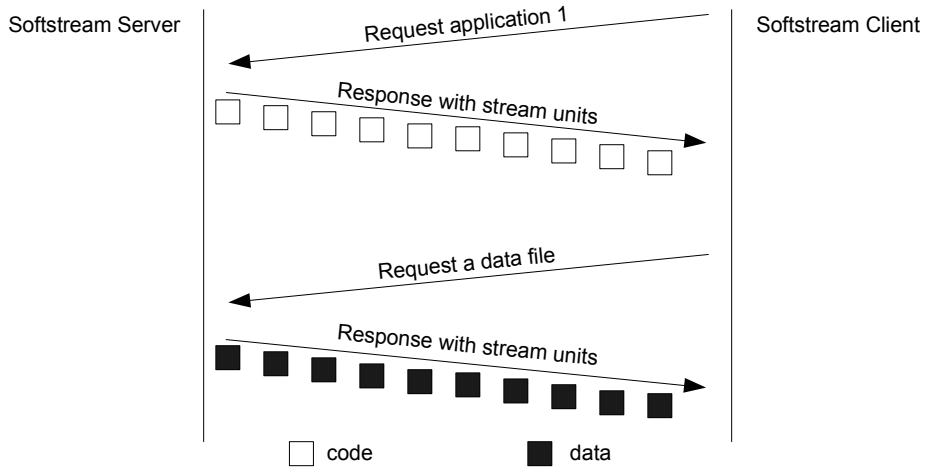

Figure 6: A simple client/server communication.

softstream loader and a softstream linker. However, for the data files, we use stream-enabled file I/O (to be described in detail in Chapter 6). The softstream protocol layer handles the communication between a softstream server and a softstream client. Example 4.3.1 shows a typical interaction between layers.

Example 4.3.1 A simple communication between a softstream server and a softstream client is shown in Figure 6. When the user runs a stream-enabled application, the softstream loader sends a request to the softstream server to stream the application using the softstream protocol. After receiving the request, the server sends stream units to the client. The softstream loader (Figure 5) stores the received blocks in memory using stream-enabling data. When the stream unit containing the program entry point is received, the stream-enabled application can be executed. The softstream linker assembles blocks as needed while the application is running. When data in a file is needed, the application performs File I/O operations through the stream-enabled file I/O layer. □

4.3.1 Softstream Message Format

In order to stream a stream-enabled application properly, we use a certain message format which we call a *softstream message*. Softstream messages can be classified into two main categories: request and response. A request message is sent by the client

for a particular service, and the server sends a response message. Figure 7 shows the softstream message format.

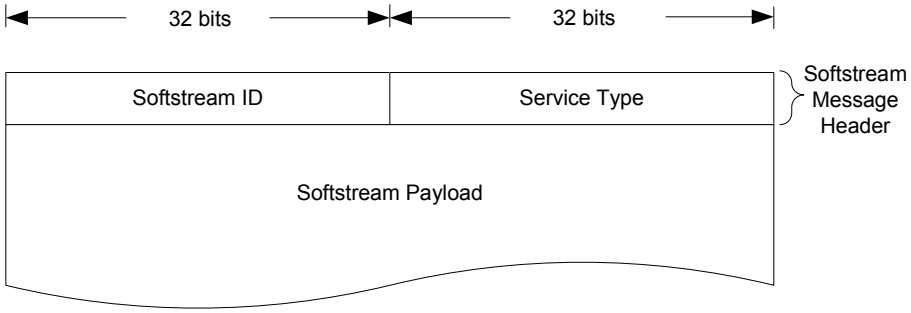

Figure 7: Softstream message format.

An example of a possible softstream payload in Figure 7 could be, in the case of a response message, a stream unit to a client sent by a server.

The three fields in the softstream message shown in Figure 7 have the following meaning.

Softstream ID

This 32-bit field in Figure 7 is called a softstream ID and serves as an identification corresponding to an open file on the softstream server. This field is zero when the client initiates a request to open a stream. Once the stream is open, the server assigns this value; each softstream ID assigned by a particular server can be guaranteed to be unique. However, if a softstream client connects to multiple servers, one server may assign the same softstream ID as another server. Therefore, concurrent stream-enabled files at the client may potentially have the same softstream ID. However, this will not cause any problem since the network connections between the softstream client and the servers are different (i.e., different sockets).

Service Type

This 32-bit field in Figure 7 specifies the type of the service. Figure 8 shows the structure of the service type field for a request message. The first 16 bits of the service type field specifies a stream operation type. There are three types of stream

operations, namely, read request (0x0001), write request (0x0002), and close request (0x0003). The read request stream operation attempts to obtain a stream unit. The write request stream operation attempts to write to a file. Finally, the close request stream operation advises the server to close the open stream. Note that while 16 bits of information are not strictly needed to specify the stream operation, we chose 16 bits in order to be easily addressable (i.e., as a halfword); however, if desired, less than 16 bits could be used.

The last 16 bits of the service type field contains a flow control field. The flow control field regulates how stream units are transmitted. We describe flow control options in Section 4.3.3.

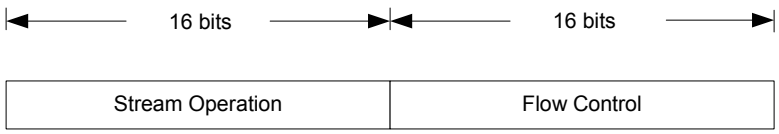

Figure 8: Structure of the service type field.

For a response message, all 32 bits of the service type field are used for stream unit removal; the details will be as described in Section 7.2.

Softstream Payload

This field contains a message body which is used to communicate between a softstream server and a softstream client. There are three types of softstream payloads for a request message, namely, a filename, an identification of the needed (requested) stream unit, and nothing (for a close request message). There are three types of softstream payloads for a response message, namely, file information for a requested program file, a stream unit, and an error message. Error messages include errors such such as access denied and file not found.

4.3.2 Transmission Profile

It is preferable to send stream units according to the order in which they are used. This can lower the occurrence of application suspensions due to missing stream units.

In order to achieve this, we profile a file to be streamed. Stream unit transmission can be viewed as a flow graph where a node represents a stream unit and an edge represents a time order of the transmission. We call this flow graph a *transmission flow graph*.

A *transmission profile* is simply a number presenting the first stream unit to be sent and an array containing numbers indicating the order of subsequent stream units to be sent. A number in an array presents the stream unit to be sent after the stream unit represented by the array index is sent. For example, if the value of the first element (index of 0) of the array is 1, the stream unit represented by 1 will be sent after the stream unit represented by 0. If an array element has a value of -1, the stream unit represented by the index is an end node. When an application is requested, the server sends the stream unit represented by the first number in the transmission profile. After the first stream unit is sent, the server uses the number of the first stream unit as an index to obtain the next number in the array. Then, the server transmits the stream unit represented by the obtained number. The process may be repeated until the obtain number is -1. For a candidate application, we typically define a few (usually less than five) common transmission profiles. If a stream unit is requested, the softstream server sends stream units according to the transmission profile.

Example 4.3.2 Figure 9 show a transmission profile. In this transmission profile, the first stream unit represented by 0 will be sent first. Then, we use 0 as index to obtain the value in the array. Using the number representing a stream unit as an index until obtaining -1 yields 1, 2, 4, 6 and 7. Therefore, stream units represented by these numbers will be sent sequentially. □

A profiling algorithm can be used to determine in which order stream units should be sent. Stream units can be sent based on a particular objective. If we want to save bandwidth and memory, we send only the stream units which appear along the

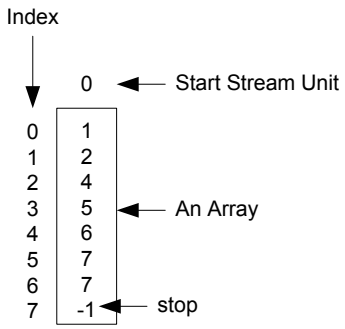

Figure 9: A Transmission profile.

most likely path in a transmission flow graph. If a stream unit in another path is needed, we interrupt the current sequence and start a new sequence from the needed/requested stream unit. On the other hand, if we want to potentially lower occurrence of application suspensions due to missing code/data, we send all stream units in a breadth-first fashion starting with the stream units which are closer to the root node in a transmission flow graph.

Note that more than one transmission sequence can be contained in the transmission profile. Then, if a request arrives for a block not in the initial transmission sequence, then either (i) a new sequence may be selected based on the new request or (ii) no sequence can be selected which means that only specifically requested blocks are streamed (this case will be described as a version of *on-demand* stream flow control in Section 4.3.3).

Example 4.3.3 An example of a transmission flow graph is shown in Figure 10. Suppose that SU 3 and SU 5 in Figure 10 contain features uncommonly used. To save bandwidth and memory for the typical case where SU 3 and SU 5 are not used, the transmission sequence would be SU 0, SU 1, SU 2, SU 4, SU 6, and SU 7 (the stream units SU 4 and SU 6 would never be sent). This specific transmission sequence is represented by the transmission profile of Figure 9 which is explained in Example 4.3.2. However, suppose that an atypical case occurs, i.e., the stream units needed are SU 0, SU 1, SU 3, SU 5 and SU 7 (SU 2, SU 4 and SU 6 are not needed). For this atypical case, suppose that the typical transmission sequence was being

followed with SU 2 completely loaded already at the point when this atypical case finishes SU 1 and calls for code in SU 3. Then the original stream sequence would be interrupted. In this case, SU 3 would be loaded, and then index 3 would be used in Figure 10 to determine the next stream unit to be loaded. Thus, the stream units loaded subsequent to SU 3 would be SU 5 and SU 7. □

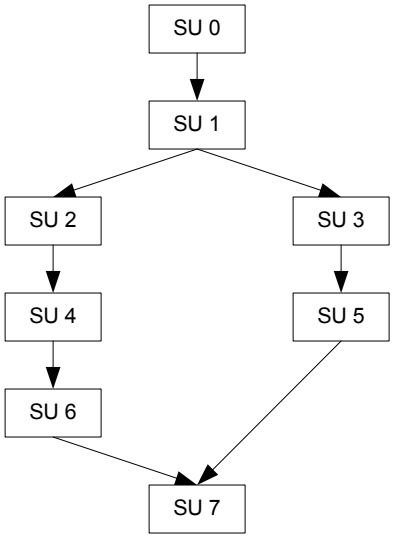

Figure 10: A transmission flow graph.

Initially, a transmission profile may be created manually by the programmer. When the application is being used, the server can collect the usage pattern to improve how stream units should be sent.

Please note that we do not propose a specific, unique algorithm to be used exclusively to create a transmission profile but instead provide support for any algorithm. In this spirit, while we have not formally and precisely defined what exactly constitutes a transmission flow graph, we have nonetheless give a few intuitive examples which can serve as a guide to possible ways to derive a transmission profile. Please note further that many other implementations of the core idea behind a transmission profile — namely, the core idea being to guide stream unit transmission — exist. We give here only one of many possible implementations of a transmission profile.

4.3.3 Softstream Flow Control

The softstream flow control techniques are used to regulate the amount of stream units sent to the client. There are two types of flow control, namely, continuous stream and on-demand stream. Depending on the availability of its resources, the client decides the type of flow control to be used by setting the last 16 bits of the service type field. The flow control techniques are described in the following subsections.

4.3.3.1 Continuous Stream

It is typically preferable that an application runs without interruption due to missing code/data. By allowing the stream units comprising the application to be transmitted in the background while the application is running, all program code/data can potentially arrive in memory prior to being needed. As a result, execution delay due to code/data misses is reduced, possibly to zero. Thus, the application is suspended less frequently or not at all. When a client requests continuous stream service, the softstream server sends stream units according to the corresponding transmission profile. If the program execution is different from the most likely transmission sequence (as predicted by the transmission profile), the client simply requests the missing block. Then, the server sends stream units with a sequence starting from the missing stream unit. In continuous stream flow control, stream units are sent from a start node to an end node along a predicted path, which means that some of the stream units may not be sent. Setting the last 16 bits of the service type field (described in Section 4.3.1) to 0x0000 advises the server to use continuous stream flow control.

4.3.3.2 On-Demand Stream

In some cases where bandwidth and memory are scarce resources, continuous stream flow control may not be suitable. Many stream units may not be needed by the application under particular conditions. For example, a multimedia application may use only one filter algorithm in a certain mode of operation or may provide features

which are rarely used. Memory usage is minimized when stream units are downloaded on-demand, i.e., only when the application needs to use the code in the stream unit. While downloading the requested code, the application will be suspended. In a multitasking environment, the stream unit request system call will put the current task in the I/O wait queue, and the operating system may switch to run other ready tasks. On-demand stream flow control allows the user to decrease resource usage at a cost of potentially increasing the number of application suspensions.

On-demand stream flow control does not imply that the client must request a single stream unit at a time. The client may request a particular number of stream units at one time. However, the client does not have to wait until all requested stream units are loaded into memory; the client can continue execution as soon as the first stream unit is loaded. Other stream units will be sent in the background according to the application's transmission profile.

To use on-demand stream flow control, a client sets the last 16 bits of the service type field to the number of stream units the client can store when the client requests a file. Note that for on-demand stream flow control, the client can specify up to $2^{16} - 1$ units to be transmitted; otherwise, continuous stream unit will be used. If needed in the future, this field can be expanded to use addition bits (e.g., 32 instead of 16).

Table 1 illustrates softstream message header values and explanations. If a client sends a request message to stream an application using a continuous stream flow control, the client sets the softstream ID field to zero, the service type field to 0x00010000, and the softstream payload to the application name (see the first row of Table 1). On the other hand, if a client sends a request message to stream an application using a on-demand stream flow control, the client assigns the service type field to 0x0001nnnn, where *nnnn* represents the number of stream units to be transmitted (see the second row of Table 1). When an application open a a file for writing, the service type field is set to 0x00020000 (third row of Table 1). When the requested file is open, the

softstream ID field is used to represent the file. If a client closes a file, the client transmits a request message with the file's softstream ID field assigned by the server and a service type field of 0x00030000 (4th row of Table 1). While running an application, if a required stream unit is not memory, the client sends a request message with a service type field of 0x00010000 and with softstream payload field set to the required stream unit (5th row). The client can request a certain number of stream units by setting the service type field to 0x0001nnnn, where *nnnn* is the maximum number of stream units that can be stored at the client (6th row). When an application writes to an open file, the client sends a message with the service type field of 0x00020000 (7th row). For a server's first response message to a client requesting a file, the server sends in the message payload either (i) the file information or (ii) an error message. An error message is marked with a service type field of 0xFFFFFFFF (8th row). Finally, the server can send a response message with a stream unit in the softstream payload field; in this case, the value of the service type field is set the value of the stream unit ID to be replaced, i.e., in this case, the value of *nnnnnnnn* specifies which stream unit to be replaced (8th row).

4.4 Summary

Software streaming enables an application to run while application transmission may still be in progress. However, the application must be modified before it can be transferred and executed on the client device. A transmission profile should be generated and is a crucial part of software streaming since the server sends stream units according to the transmission profile. Therefore, the transmission sequence affects the performance of the application. The process of transforming an application into a stream-enabled application is discussed in Chapter 5 (for the application program itself) and Chapter 6 (for any file I/O required by the application program).

Table 1: Softstream message header values and explanations.

Softstream ID	Service Type	Softstream Payload	Explanation
0	0x00010000	File info	Continuous stream request to start a new stream
	0x0001nnnn	File info	On-demand stream request to start a new stream with nnnn units
	0x00020000	File info	Open file to write request
$\neq 0$	0x00030000		Close stream request
	0x00010000	Stream unit ID	Continuous stream request starting from the stream unit ID in the payload
	0x0001nnnn	Stream unit ID	On-demand stream request starting from the stream unit ID in the payload, up to nnnn stream units can be sent
	0x00020000	Data info	Write request
	0xnnnnnnnn	Info	Server response

CHAPTER V

STREAM-ENABLED PROGRAM FILES

Software to be streamed must be modified before it can be streamed over a transmission medium. The software must be partitioned into stream units 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 stream unit containing the program entry point is loaded into the memory of the device. In contrast to downloading the whole program, software streaming can reduce application load time. While the application is running, additional parts of the streamed software can be downloaded in the background (continuous streaming) or on-demand. If a required stream unit is not in memory, the needed stream unit must be transmitted in order to continue executing the application.

5.1 Stream-Enabled Program File Overview

Our implementation of software streaming presented here tries to optimize application performance. Our target platform is an embedded device. It turns out that for a program file the number one issue for proper streaming is the suitable handling of branches in the program file. In particular, information about the branch instructions in a program file is generated both at a server and at an embedded client device. Generating branch information at the server can reduce client processing time since such code generation is done off-line (from the perspective of the client). On the other hand, generating branch information at the client can save bandwidth if, as a result, the server does not have to send an excessive amount of branch information (the

branch information forms a part of the stream unit header described in Section 4.2). Processing time is a key issue for a low processing-power embedded processors, and bandwidth is a key issue for transferring data across a network since transfer time is proportional to data size.

Throughout this chapter, we use the Motorola PowerPC microprocessor architecture [24] in our implementation. However, the techniques illustrated here can be easily applied to other architectures.

5.1.1 Client/Server Processes

As shown in Figure 11, the software streaming process on the server involves (i) compiling the application source code into an executable binary image by a standard compiler such as GCC [11], (ii) dividing the executable binary image into code blocks, (iii) generating a new binary for the stream-enabled application, and (iv) creating a transmission profile. The four steps (i-iv) can be done off-line. The server uses the transmission profile to determine transmission order of stream units to a client.

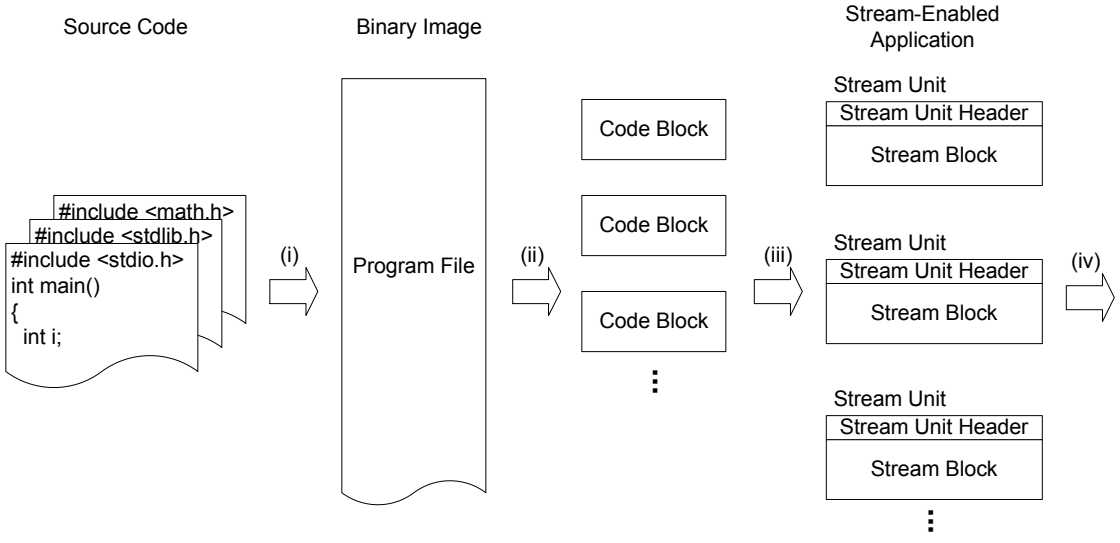

Figure 11: Server-side software-streaming process.

The process to receive a block of stream-enabled application on the client device is illustrated in Figure 12: (i) loading a stream block into memory and (ii) linking the

stream block into the existing code. This linking process will most likely involve some code modification(s) which will be described later in Section 5.3 and in Section 5.4.

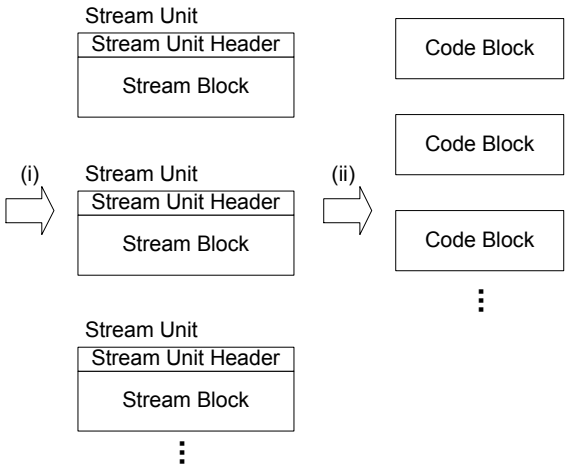

Figure 12: Client-side software-streaming process.

5.1.2 Softstream Generator

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. 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 (stream block) is generated. Each block is generated into a stream unit. The size of each block of the same application may be different. However, in our current implementation, we use a fixed block size. The block size may be determined from streaming parameters such as network speed. Example 5.1.1 illustrates that a binary image is divided into fixed-size blocks.

Example 5.1.1 The box on the left in Figure 13 represents the compiled binary image of an application, and the boxes on the right represent the binary image of the application broken

up into blocks. For instance, if equal block size is configured, a 10 MB robotic exploration application can be broken into ten 1 MB blocks. □

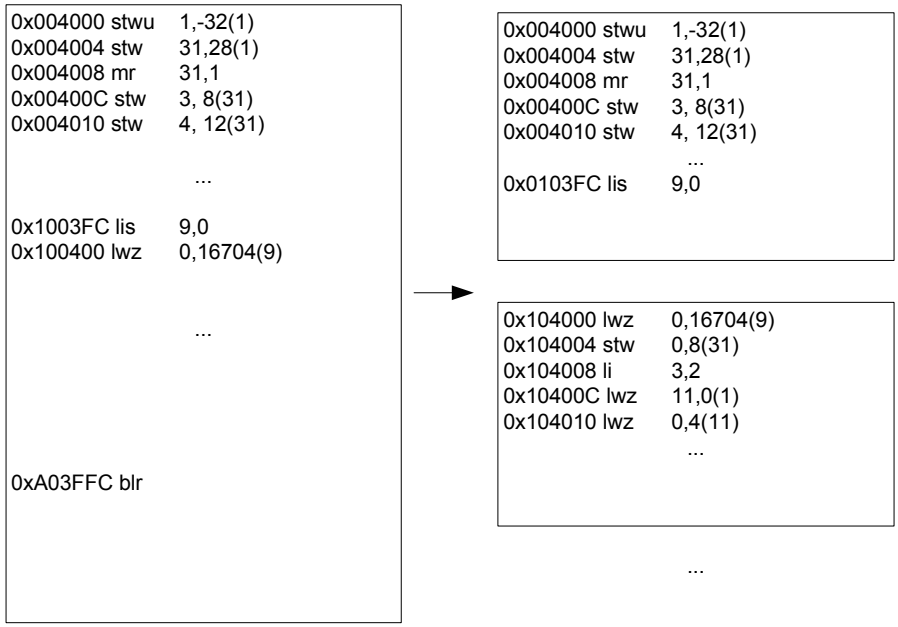

Figure 13: The binary image of an application is broken up into blocks.

After the application is divided into blocks, exiting and entering a block can occur in four ways. First, a branch or jump instruction can cause the processor 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. Fourth, when an exception has occurred, the exception handling code may be in a different 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 the application to work properly.

We aim to generate stream blocks in such a way that they are relocatable; if we are successful, a generated stream block can be placed anywhere in memory. As a result, a client does not have to allocate memory space for the entire program in

order to execute part of the program code. Furthermore, the client can dynamically allocate memory for a stream block as needed.

5.1.3 Stream-Enabled Program File Request

A softstream client requests a stream-enabled program file by sending a read request message to a server. The client assigns the softstream ID field to zero, the first 16 bits of the service type field to 0x0001, and the last 16 bits of the service type field to a value depending on the preferred flow control. The client also sends the unique filename (name of the desired program) in the softstream payload field.

In order for the server to properly send (and for the client to properly receive) stream units for a stream-enabled application, some basic information is required. This stream-enabled application information shown in Table 2 consists of application size, program entry point, number of stream units, number of off-block branches, and stream block size. Each field is a 32-bit integer. The softstream loader/linker software at the client device uses this information to load and to link stream units together. Code generation techniques are discussed in Section 5.2.

Table 2: Stream-enabled application information.

Field	Explanation
Application Size	The size of the program code
Program Entry Point	The location of the first instruction in the program to be executed
Number of Stream Units	The amount of stream units in the application
Number of Off-Block Branches	The amount of off-block branches in the application
Stream Block Size	The maximum stream block size in a stream unit

Example 5.1.2 The stream-enabled application information of a 10 MB robotic exploration application in Example 5.1.1 is illustrated in Figure 14. Since the application is divided into 10 equal-sized blocks, the number of stream units is 10 and the maximum stream block size is 1 MB. The number of off-block branches for this particular application is 30. Finally, the

program entry point is at the address 0x4000. When the stream block with an ID of 0 is loaded, the program can start running since the stream block contains the program entry point. □

Application Size:	10 MB
Program Entry Point:	0x4000
Number of Stream Units:	10
Number of Off-Block Braches:	30
Stream Block Size:	1 MB

Figure 14: Stream-enabled application information.

The softstream server responds to the client request by sending a softstream message with a softstream ID and the stream-enabled application information stored in the softstream payload field. When a stream block is needed, the client sends a request message by setting the softstream ID field to the received softstream ID and softstream payload to the needed stream block ID.

5.1.4 Softstream Loader/Linker

To transmit an application using our approach, the softstream server daemon running on the server first transmits the stream-enabled application information of Table 2 to the client device. After the softstream loader receives the stream-enabled application information, the softstream loader sets up a *stream block lookup table* as shown in Figure 15, where N is the number of stream units. The stream block lookup table has *stream block ID* and *address* entries. Stream block IDs are assigned sequentially as they appear in the stream-enabled file, starting from zero. If the address entry is 0xFFFFFFFF, the stream block is not yet loaded into memory.

Software streaming code generator (*softstream generator*) software on the server generates a stream unit header for each stream unit. The stream unit header provides the softstream loader/linker with the information to handle stream block misses and to link stream blocks together. Each stream unit must be encapsulated with a softstream message header (as shown in Figure 7 of Section 4.3) before the stream unit can be transmitted. The stream unit (a softstream payload of a response message) format

Stream Block ID	Address
0	0xFFFFFFFF
1	0xFFFFFFFF
2	0xFFFFFFFF
...	...
N-1	0xFFFFFFFF

Figure 15: Stream block lookup table.

for program files is illustrated in Figure 16. For program files, the stream unit header consists of *stream block ID*, *off-block branch information size*, *stream block size*, and *off-block branch information*. The off-block branch information size field in Figure 16 indicates the amount of off-block branch information in the stream unit in bytes. The stream block size field represents the amount of the program code/data in the stream unit in bytes (note that even though we divide a program file into fixed-sized blocks, the last block may not be the same size as the others). The off-block branch information contains data which is used to link stream blocks together. We will describe how to generate off-block branch information in Section 5.2.3.

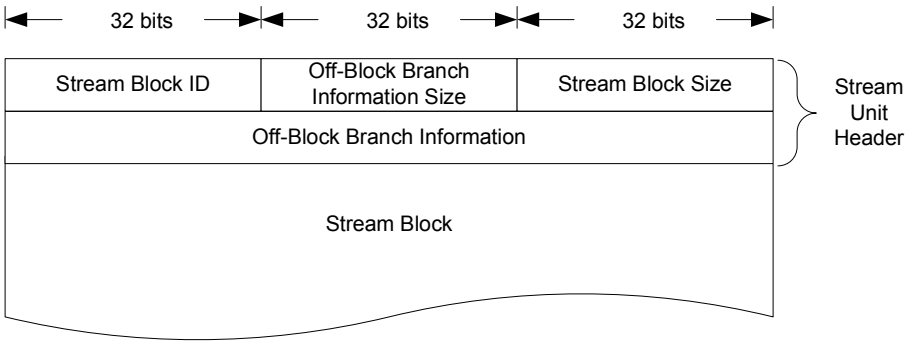

Figure 16: The stream unit format for program files.

Example 5.1.3 The robotic exploration application in Example 5.1.1 and Example 5.1.2 has program code size of 10 MB. The application is divided into 1 MB blocks. Each block is generated into a stream unit with an ID corresponding to the stream block ID which the stream unit encapsulates. The stream unit generated from the first block is illustrated in Figure 17.

Since the stream block is the first block generated, the stream block ID generated for this block is 0. For this first stream unit, the off-block branch information size for this stream unit is 16 bytes, and code size is 1 MB ($= 2^{20} =$ 0x00100000). □

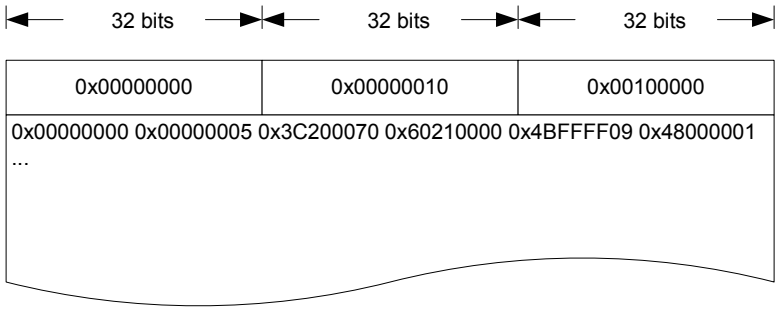

Figure 17: The first stream unit of the robotic exploration application.

Once the stream block containing the program entry point (e.g., the stream unit with ID=0) is downloaded, the softstream loader stores the stream block in the memory space allocated for the stream block using a dynamic memory management technique and saves the address of the stream block in the stream block lookup table. The softstream loader will be discussed in detail in Section 5.3.

After the softstream loader finishes assembling the stream block containing the program entry point, the application begins execution. Application execution continues until an off-block branch is taken. Figure 18 shows the sequence when a modified off-block branch is first taken. The softstream linker uses the stream block lookup table (Figure 15) to check if the needed stream block is already in memory. If the needed stream block is not fully in memory, the needed stream block is, if not already done previously, requested to be streamed. After the needed stream block is in memory, the off-block branch instruction is modified to jump to the proper location. The program execution is then resumed. We will discuss the softstream linker in detail in Section 5.4.

A stream-enabled application at a client consists of two threads: a softstream loader thread and an application thread. In this dissertation, we only support a single

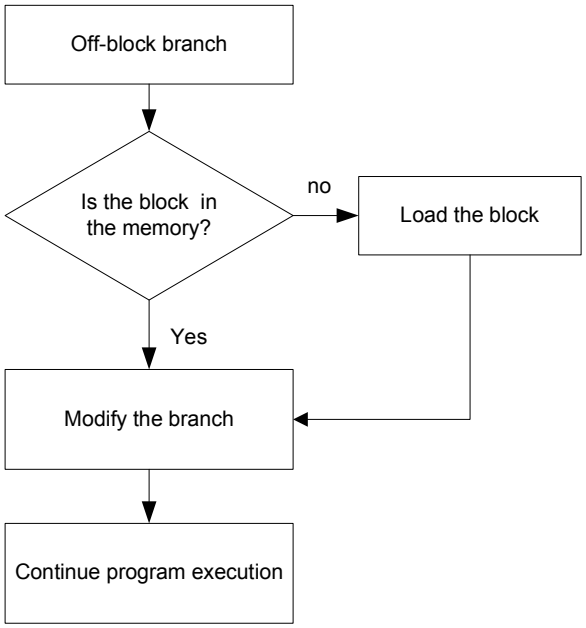

Figure 18: Block loader flow chart.

threaded application (however, in Chapter 7.3.3 we do suggest some approaches to extend to support multi-threading). The application thread writes the needed stream block ID to a shared variable, and the softstream loader thread reads the variable and requests the needed stream block. The softstream loader thread communicates to a softstream server using the transport layer, e.g., through a TCP socket.

5.2 Stream-Enabled Code Generation

In our approach, stream-enabled embedded software is generated from an executable binary image of the software application to be streamed. The executable binary image of the software is created from a standard compiler such as GCC. The following sections describe how to generate a stream-enabled application in detail.

5.2.1 Preventing the Execution of Non-existing Code

As mentioned previously in Section 5.1.2, exiting and entering a block can occur in four ways: (i) after executing a branch or jump instruction, (ii) after executing the last instruction in the block, (iii) after executing a return instruction, or (iv) after an

exception has occurred. In order to prevent the CPU from executing code incorrectly, these off-block branch instructions are modified. For the first case (i), a branch instruction which can potentially cause the processor to execute code in another block not yet loaded into memory is modified to load the block containing the needed code as illustrated in Example 5.2.1. For the second case (ii), suppose the last instruction in a block is not an unconditional branch or a return instruction. If left this way, the processor might execute the next instruction. To prevent the processor from executing invalid code, an instruction is appended by default to load the next block; an example of this is shown at the end of Example 5.2.1. For the third case (iii), no return instruction ever needs to be modified if blocks are not allowed to be removed from memory since the instruction after the caller instruction is always valid (even if the caller instruction is the last instruction in the block upon initial block creation, an instruction will be appended according to the second case above). However, if blocks are allowed to be removed, all return instructions returning to another block must be modified to load the other block (i.e., the caller block) if needed. For the last case (iv), we require all exception handling code to be streamed before allowing the execution of any block(s) containing the corresponding exception instruction. Therefore, exception instructions are not modified since the entire exception handling code is already loaded prior to any execution of the exception instruction. Note that we can easily extend our approach to stream exception handling code by modifying the interrupt table to jump to the branch table to load the needed code.

Example 5.2.1 Consider the if-else C statement in Figure 19. 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 19 shows a possible split.

In this small 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

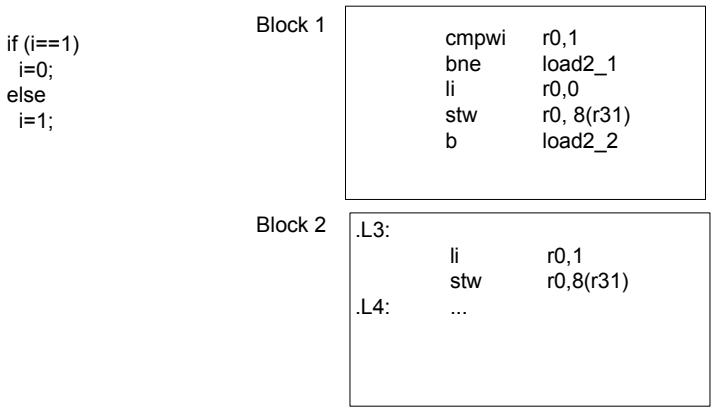

Figure 19: C code and corresponding PowerPC assembly.

modified. All off-block branches are modified to invoke the appropriate loader function if the block to which the branch jumps is not in memory. After the missing block is loaded, the intended location of the original branch is taken. Figure 20 shows Block 1 and Block 2 from Figure 19 after running the software streaming code generation program on the application code. Branch instructions `bne .L3` and `b .L4`, as seen in Figure 19, are modified to `bne load2_1` and `b load2_2`, respectively, as seen in Figure 20.

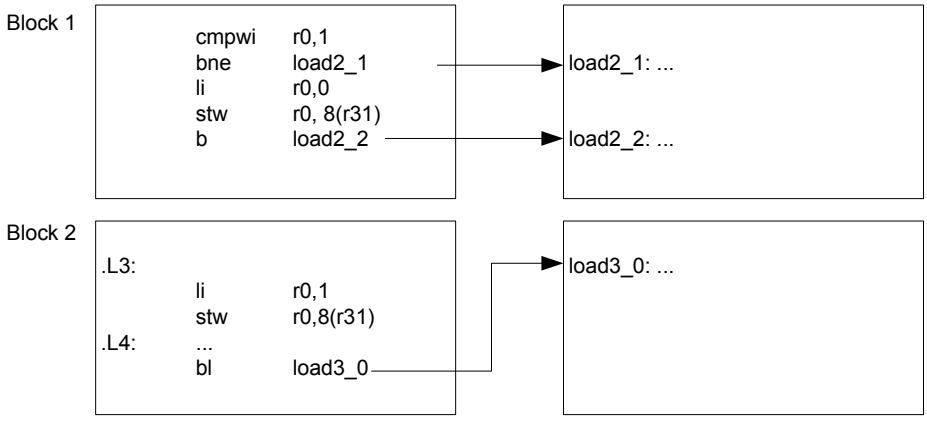

Figure 20: Block 1 and Block 2 after the stream-enabled code generation.

Since the last instruction of Block 2 is neither an unconditional branch nor a return instruction, the instruction `bl load3_0` will be appended to Block 2 after it is loaded from Figure 19 to load the subsequent block. The instruction `bl load3_0` can be replaced later by the first instruction of the block after Block 2 if the client preserves code continuity, which means that the client allocates memory large enough to store the entire application. □

Branch instructions can be categorized into conditional branch instructions and unconditional branch instructions. A conditional branch has two paths: a target path and a fall-through path. The execution follows the target path if the condition is satisfied. Otherwise, the execution follows the fall-through path. On the other hand, for an unconditional branch, the execution always follows the target path. Both conditional and unconditional branches may have a flag causing the return address (the address after the branch instruction) to be saved in the link register (the PowerPC architecture uses the link register to save a return address). Usually, the return address is saved when a function or a subroutine is invoked. When the target path is followed, the address of the instruction after the branch instruction is saved as the return address. After executing a return instruction, the processor executes the instruction at the return address. Essentially, the return instruction can be classified as an unconditional branch. If the target address of a branch is outside the block which contains the branch, we modify this branch. In the next two subsections, we will discuss in detail how our approach handles branches.

5.2.1.1 Handling Static Branches

A static branch is a branch whose target address is encoded in the instruction. A static branch instruction can be conditional or unconditional. The target address can be an absolute address or a relative address.

Consider the 32-bit PowerPC conditional branch format in Figure 21. BD, the fourth field from the left in Figure 21, is an immediate field specifying a 14-bit signed two's complement integer that is concatenated on the right with 0b00 (note that '0b' indicates a number in binary format) and sign-extended to 32 bits. If the absolute address bit (AA), the fifth field from the left in Figure 21, is 0b0, the immediate field BD represents an address relative to the current instruction address. On the other hand, if AA is 0b1, the immediate field BD represents an absolute address.

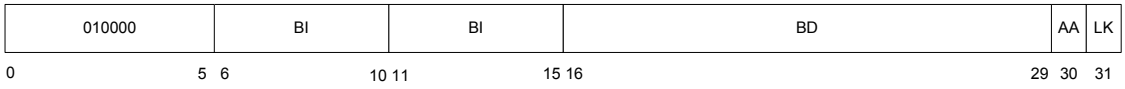

Figure 21: The 32-bit PowerPC conditional branch instruction format.

The 32-bit PowerPC unconditional branch instruction format is shown in Figure 22. LI, the second field from the left in Figure 22, is an immediate field specifying a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 32 bits. As previously explained, depending on the value of AA, the immediate field LI can be a relative address or an absolute address.

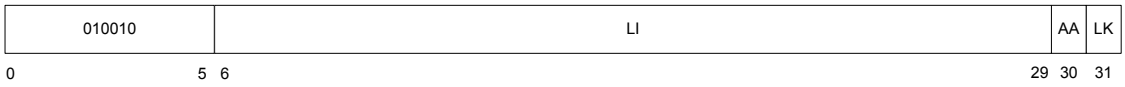

Figure 22: The 32-bit PowerPC unconditional branch instruction format.

In order to determine whether or not a branch is an off-block branch, the branch target address is calculated as shown in Listing 1.

Get_Branch_Target_Address() takes as input a static branch instruction with its instruction address and returns the branch target address. Conditional branch target address calculation is shown in lines 2 to 6, and unconditional branch target address calculation is shown in lines 8 to 12. The *EXTS()* function in Listing 1 returns the sign-extended value of the input, and $\|$ is a concatenate operation. If the target address does not belong the current block, the branch is an off-block branch. The address field (BD or LI) of the off-block branch instruction is modified (via binary rewriting) so that when the branch is taken, it goes to the loader function to load the target block.

5.2.1.2 Handling Dynamic Branches

A dynamic branch is a branch which does not have the branch target address encoded in the instruction. The branch target address is usually stored in a particular register or can also be in memory. For example, when a function pointer is used, the address

Listing 1 *Get_Branch_Target_Address():* Determining the branch target address.

Input: A branch instruction B and the current instruction address CIA
Output: The target address T
begin

```
 1 if B is a conditional branch then
 2   if B is an absolute address branch then
 3     T = EXTS(BD‖0b00)
 4   else
 5     T = CIA + EXTS(BD‖0b00)
 6   end if
 7 else
 8   if B is an absolute address branch then
 9     T = EXTS(LI‖0b00)
10   else
11     T = CIA + EXTS(LI‖0b00)
12   end if
13 end if
```

end

of the function is loaded to the link register before the function is called. When a dynamic branch is taken, the next instruction loaded is from the address in the link register. (Note that if no special link register exists, then the register used in place of the link register needs to be used instead.) Example 5.2.2 shows dynamic branches which are used to call a function using a function pointer and to return back to a return address.

Example 5.2.2 A generic function is a function which operates on a variety of objects. Usually, a generic function requires another function which manipulates the passed objects. In Figure 23, a sort function (shown partly in both C and PowerPC assembly) sorts objects in a certain order. The sort function requires three parameters: an array of objects of the same size, the size of the object, and a pointer to a function used to compare objects. When the sort function is called, the caller return address is saved in the link register. Inside this function, the function used to compare objects can only be identified during runtime. Therefore, the sort function contains a dynamic branch to the function used to compare objects. The return

instruction to the caller function is also a dynamic branch since the return address is saved in a link register.

```
void sort(void *base, int size, int (*comp)(void *, void *))
{
1   mflr        r0                      Save the return address
 ...
 result = comp(&base[i], &base[j]);

2   mtlr        r5                      Load the address of the function into LR
3   lwz         r3,0x0(r12)        }    Load the function parameters
4   lwz         r4,0x0(r13)        }
5   bclrl       0x14,0x0                Call the function

 ...
6   mtlr        r0                      Load the return address into LR
7   blr                                 Return back to the caller function
}
```

Figure 23: A sort function using a function pointer as a parameter.

A typical step for calling a function using a function pointer is shown in assembly instructions in Figure 23. First, the address of the function pointer is loaded into the link register as shown in line 2. Note that the address of the function which is used to compare objects is stored in $r5$. Second, the values to be passed are loaded as shown in line 3 and line 4. Finally, the function is called using a dynamic branch as shown in line 5.

For returning back to the caller function from the sort function, the caller return address is saved in $r0$ (line 1) and is later loaded into the link register (line 6); finally, the return instruction is executed (line 7). After the execution of the return instruction, the instruction at the address stored in the link register will be executed. □

A dynamic branch must be intercepted to determine if the block containing the branch target address is loaded already into memory. Whether or not the block containing the branch target is loaded already into memory is verified by checking the designated register storing the branch target address. In order to first verify if a dynamic branch target address leads to a stream block already in memory, we modify the dynamic branch so that when the dynamic branch is taken, it goes to a

dynamic branch interceptor function. If the stream block is in memory, the dynamic branch interceptor function redirects to the intended location, and the program can continue its execution. Otherwise, the program will be suspended, the stream block required for execution will be requested, and, after loading the needed stream block, the dynamic branch interceptor function jumps to the proper code location.

Figure 24 shows the format of the 32-bit PowerPC conditional branch to the link register. If the branch is taken, the branch target address is the address stored in the link register. This dynamic branch has a similar format as the branch format in Figure 21. Therefore, we replace this dynamic conditional branch by binary rewriting with a conditional branch which leads to the location of the dynamic branch interceptor function and is redirected to the intended location. In the similar fashion, we replace a dynamic unconditional branch with an unconditional branch.

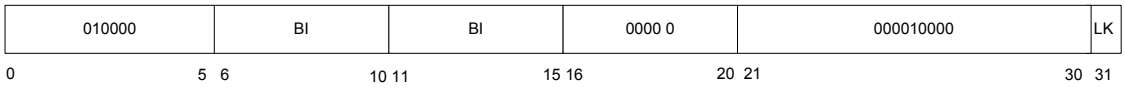

Figure 24: The format of the 32-bit PowerPC conditional branch to the link register.

Example 5.2.3 shows how dynamic branches are modified.

Example 5.2.3 Figure 25 illustrates the result of modifying the dynamic branches (line 5 and line 7) of the sort() function in Figure 23 to jump to the dynamic branch interceptor function. □

For fixed-sized blocks, the information needed for redirecting a dynamic branch can be calculated as shown in the following equations. Note that all operations are integer operations.

$$
\begin{aligned}
stream\ block\ ID &= (target\ address - first\ address) \div stream\ block\ size \\
offset &= (target\ address - first\ address)\ \%\ stream\ block\ size \\
redirect\ address &= stream\ block\ address + offset
\end{aligned}
$$

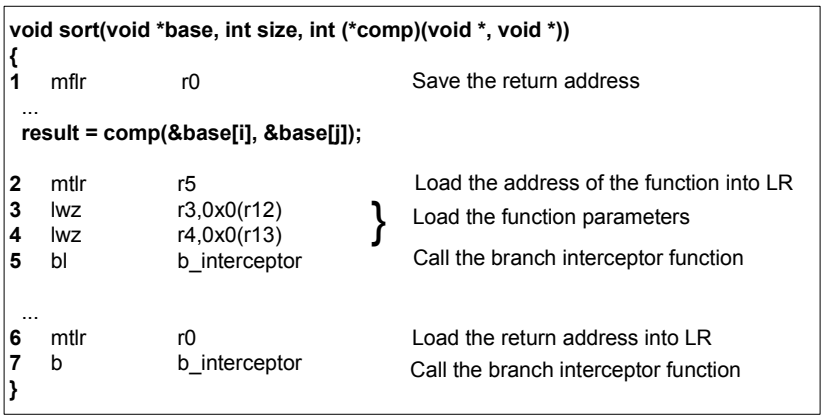

Figure 25: A sort function using a function pointer as a parameter.

where

stream block ID is the stream block ID of the branch target address,

target address is the address stored in the link register,

first address is the location of the first instruction in the program,

stream block size is the size of all stream blocks,

offset is the offset location of the instruction within the stream block,

redirect address is the intended address of the dynamic branch,

stream block address is the location of the stream block.

Example 5.2.4 Suppose that the robotic exploration application in Example 5.1.1 uses the sort function in Example 5.2.2. We modify the sort function as in Example 5.2.3. When the instruction at line 5 in Figure 25 is executed, the branch interceptor function will be called. The branch interceptor function obtains a value from the link register. Suppose that the value of the link register is 0x00105024. The target address is the value in the link register, i.e., the target address is 0x00105024. Since the stream block size is 1 MB and we assume we start from memory address 0x00000000 (i.e., *first address* = 0x00000000), the target stream block ID is 1, and the offset is 0x5024. If the stream block with stream block ID of 1 is not in memory,

the stream block will be requested. If we load the stream block with an ID of 1 in to memory starting from 0x00200000, the redirect address is 0x00205024. □

5.2.2 Coping with Non-Interruptible Sections

Some applications may contain one or more non-interruptible (critical) sections which are used to access shared resources such as data and devices. This means that while the processor is executing a non-interruptible section, the processor cannot accept interrupts. Usually, a non-interruptible section contains an instruction to disable interrupts and one or more instructions to enable interrupts. Example 5.2.5 illustrates a non-interruptible section.

Example 5.2.5 Figure 26 shows the μC/OS-II [17] OSSchedUnlock() function which contains commands to disable and to enable interrupts using OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL(), respectively. In this example, the interrupts are disabled at line 5 and are enabled at lines 11, 16, and 21, depending on a certain condition. The OSSched() function is not a part of the non-interruptible section. □

A non-interruptible code section should not be suspended due to missing software components since downloading software from the network can be very long and unpredictable. Furthermore, if we were to allow an application to enter a non-interruptible section not yet fully loaded, then the following could occur: the application could need to execute code in a stream unit not yet loaded, however, the softstream loader would not be able to receive any stream unit because the interrupts are disabled in a non-interruptible section! Therefore, we stream any non-interruptible section in its entirety to the client device before the non-interruptible section can be entered.

A programmer may write a non-interruptible section to perform a non-interruptible task. However, a compiler may put a non-interruptible section anywhere in a program file. Since we generate a stream-enabled application from its binary image

```
1    void OSSchedUnlock (void)
2    {
3       if (OSRunning == TRUE)
4          {                                          /* Make sure multitasking is running */
5             OS_ENTER_CRITICAL();
6             if (OSLockNesting > 0)
7                {                                    /* Do not decrement if already 0 */
8                   OSLockNesting--;                  /* Decrement lock nesting level */
9                   if ((OSLockNesting | OSIntNesting) == 0)
10                     {                             /* See if scheduling re-enabled and not an ISR */
11                      OS_EXIT_CRITICAL();
12                      OSSched();                    /* See if a higher priority task is ready  */
13                   }
14                else
15                   {
16                      OS_EXIT_CRITICAL();
17                   }
18             }
19          else
20             {
21                OS_EXIT_CRITICAL();
22             }
23       }
24   }
```

Figure 26: The μC/OS-II OSSchedUnlock function.

(i.e., we have may not have any source code or other additional information), we need to locate non-interruptible sections. Listing 2 shows pseudo code for obtaining non-interruptible sections in a program. *Get_CS()* takes as input an assembly program P and starts searching for non-interruptible sections by searching for interrupt disable instructions. *Get_CS()* stops when all interrupt enable instructions are found (lines 3 to 30).

A non-interruptible section may have one disable interrupt instruction and many interrupt enable instructions. Therefore, all branches must be followed to reach all corresponding interrupt enable instructions. For an unconditional branch, we follow the branch target path (lines 8 to 12). However, for a conditional branch, we save the address of the branch target address and proceed with the the branch fall-through path (lines 13 to 15). The branch target path will be added later. If we encounter an interrupt disable instruction while constructing a non-interruptible section, that location may already be covered (line 16 to 19) or will be included (line 20 to 22).

Listing 2 *Get_CS():* Obtaining non-interruptible (critical) sections.

Input: A program P
Output: The non-interruptible sections in program P
begin

```
 1 repeat
 2   Add the address of an interrupt disable instruction to the address list
 3   repeat
 4     Get an address from the address list
 5     Set the current address and the start address to the obtained address
 6     repeat
 7       Read the instruction from the current address
 8       if Encountered an unconditional branch then
 9         Set the end address to the address of the branch instruction
10         Add the address range to the current critical section
11         Advance to the branch target address
12         Set the start address to the branch target address
13       else if Encountered a conditional branch then
14         Add the branch target address to the address list
15         Advance to the next address
16       else if Encountered an interrupt disable instruction then
17         if Already covered then
18           Set the end address to the previous address
19           Add the address range to the existing critical section
20         else
21           Mark the interrupt disable instruction as covered
22           Advance to the next address
23         end if
24       else
25         Advance to the next address
26       end if
27     until An interrupt enable instruction is found
28     Set the end address to the address of the interrupt enable instruction
29     Add the address range to the current critical section
30   until The address list is empty
31 until All interrupt disable instructions are covered
```

end

Example 5.2.6 Figure 27 shows assembly code containing a non-interruptible section. This code is input to $Get\_CS()$. The first instruction of Figure 27 disables all interrupts and is discovered by line 2 of Listing 2, thereby allowing the loop starting at line 3 to be entered. The conditional branch `bne .L3` on line 3 of Figure 27 is discovered by line 13 of Listing 2 with its target address stored as indicated in line 14. In this case, line 15 examines the fall-through address associated with the found branch instruction. The next interesting instruction is `b .L4` on line 6 of Figure 27; this branch is discovered by line 8 of Listing 2. Since `b .L4` is an unconditional branch, this discovery delimits the first part of the non-interruptible section as consisting of lines 1-6 in Figure 27. In a similar fashion, the second part of the non-interruptible section is found to consist of lines 11-12 of Figure 27 (with the discovery of an enable interrupt command at line 12 causing an end to this second part). Finally, the third (and final) part of the non-interruptible section is found to be lines 7-9. The result list of address lines corresponding to this non-interruptible section consist of lines 1-9 and 11-12 of Figure 27. □

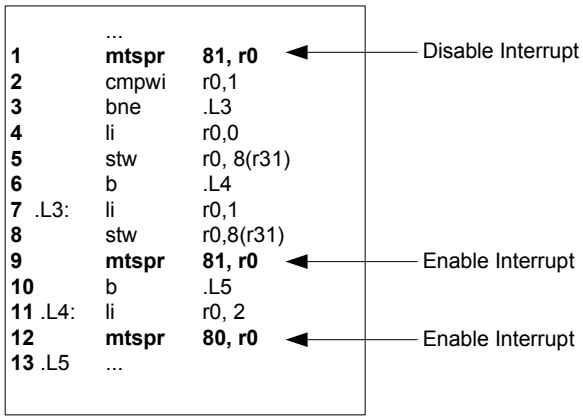

Figure 27: Assembly code containing a non-interruptible section.

Note that *Get_CS()* only obtains non-interruptible sections with no dynamic branches. If the program contains a non-interruptible section utilizing dynamic branches, we assume this information is provided by the programmer in some unambiguous format.

5.2.3 Generating Off-Block Branch Information

In order to link a stream block B into an application, we only need the locations of all off-block branches. The stream unit header with which B arrived contains a list of the locations all off-block branches in B. The instruction of an off-block branch in B can be obtained from its location.

Now consider the following question: how can we find out the stream block ID of the block containing the target address of an off-block branch? For each static off-block branch instruction, we can compute the branch target address which can be used to determine the needed stream block ID. However, for each dynamic branch, we use a dynamic branch interceptor (described in Section 5.2.1.2) to obtain the needed stream block ID at runtime. In this way, the softstream linker will compute the needed stream unit ID when an off-block branch is taken. We also sequentially assign each off-block branch with a number, starting from zero. Therefore, as illustrated in Figure 28, the off-block branch information consist of the start branch number (i.e., the branch number assigned to the first off-block branch appearing in this code block), the number of off-block branches, and series of off-block branch locations (addresses).

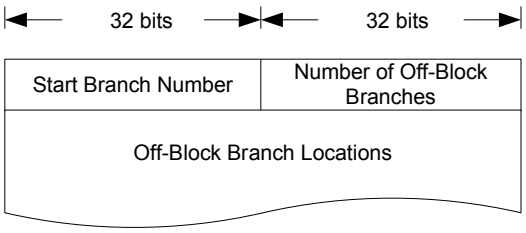

Figure 28: Off-block branch information.

Off-block branch information contains the data which is used to assemble stream blocks together at a softstream client. In block streaming, we generate off-block branch information at a server so that a client does not have to examine every instruction in a stream block before the client can load the stream block. The client has to perform code modification during load time. Therefore, we have to generate a

sufficient amount of off-block branch information for the client to link stream blocks together.

5.2.4 Stream Block Placement

In Figure 11, step (iii) shows a code block being transformed into a stream block with associated stream unit header. We have introduced the stream unit header details in Section 5.1.4 and Figure 16, describing in subsequent sections how each field of the stream unit header is determined. If the original binary image is position independent, the stream blocks generated will also be position independent. A position independent stream block can be loaded anywhere in memory. However, if the original code is not position independent, the embedded program data must be placed in the exact memory locations as originally compiled in order to guarantee correct execution (especially when considering data loads hard coded address). For the rest of this dissertation, we assume position independent compiled assembly code.

5.3 Load-Time Code Modification

When a stream unit is received, a client loads the associated stream block into memory and updates the location of the stream block in the stream block lookup table. Since the stream block is generated by the softstream generater relocatable or position independent as explained in Section 5.2.4, the client does not have to allocate memory large enough to store the entire application initially, but instead the client can dynamically allocate memory for stream blocks as needed. Then, the client uses the off-block branch information to integrate the stream block into the application: all off-block branches must be modified so that when any branch is taken, proper code will be executed. Each off-block branch address location and actual instruction are saved in a *branch information table*. Each off-block branch is modified such that when the off-block branch is taken, it goes to specific *branch loader code*. The branch

loader code contains instructions to load branch numbers and to invoke the softstream linker.

Assuming that an off-block branch instruction can jump to the branch loader code, the modified version of the off-block branch instruction can be calculated as follows.

$$
\begin{aligned}
bc &= (((BLC + n \times 12) - L) \ \& \ 0x0000FFFC) \mid (I \ \& \ 0xFFFF0003)) \\
b &= (((BLC + n \times 12) - L) \ \& \ 0x03FFFFFC) \mid (I \ \& \ 0xFC000003))
\end{aligned}
$$

where

bc is the modified conditional off-block branch instruction,

L is the location of a off-block branch instruction,

BLC is the address branch loader code,

n is the off-block number,

I is the off-block branch instruction,

b is the modified unconditional off-block branch instruction.

Note that an off-block branch incurs 3 instructions in the branch loader table and each instruction occupies 4 bytes. Therefore, we multiply the branch number by 12.

Example 5.3.1 Suppose that the block loader code is located at 0x00000100, the unconditional off-block instruction is at 0x00004000, branch number is 4, and the instruction is 0x4800005D. Substituting all values into the equation to compute b, we obtain the modified unconditional off-block branch instruction 0x48FFC131. □

Figure 29 shows a possible branch loader code and branch information table for an application. Each field in the branch loader code consists of an instruction to save register *r3*, an instruction to load the branch number into *r3*, and an instruction to branch to the softstream linker function. In the softstream linker function, we use register *r3* to pass the branch number to the softstream linker. This is why we save

the value of the register *r3* prior to using *r3*; at the end of the linker function, we restore *r3* from the stack (*r1* is the stack pointer). Each field of the branch information table consists of the offset of the off-block branch and the instruction of the off-block branch. These values will be used during run-time linking of the block.

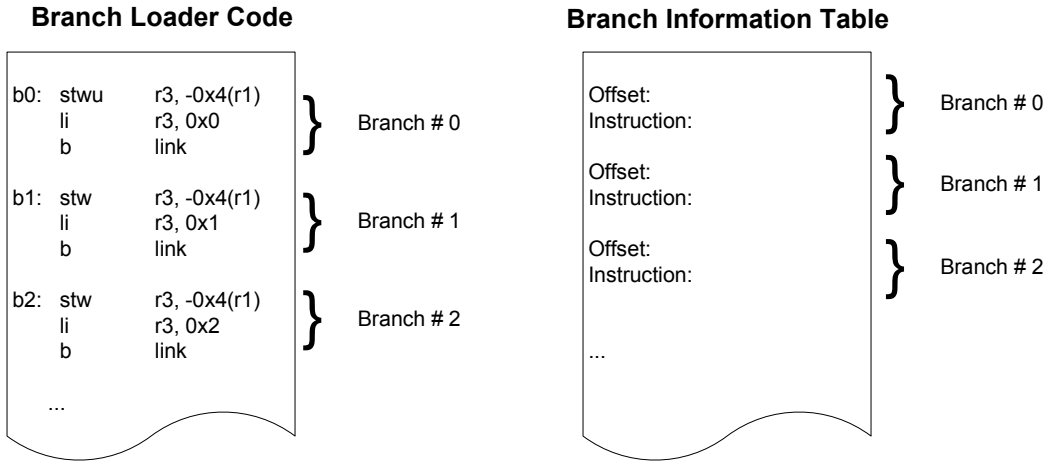

Figure 29: Branch loader code and branch information.

5.4 Run-time Code Modification

Recall that, as described in Section 5.3, all off-block branch instructions are modified to branch to an appropriate branch loader code to load the needed stream block before proceeding to the target. After loading the needed stream block, the corresponding off-block branch is modified to jump to the target location instead of invoking the loader function the next time the off-block branch is re-executed. Although run-time code modification (run-time binary rewriting) 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 stream block is in memory, but instead the modified branch instruction branches to the exact appropriate code location.

Example 5.4.1 Suppose that the software from Figure 19 is modified, resulting in the software streaming code generated as illustrated in Figure 20. Further suppose that the streaming

code is running on an embedded device. When an off-block branch is executed and is taken, the missing block must be made available to the application. Figure 30 shows the result after the first 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 30).

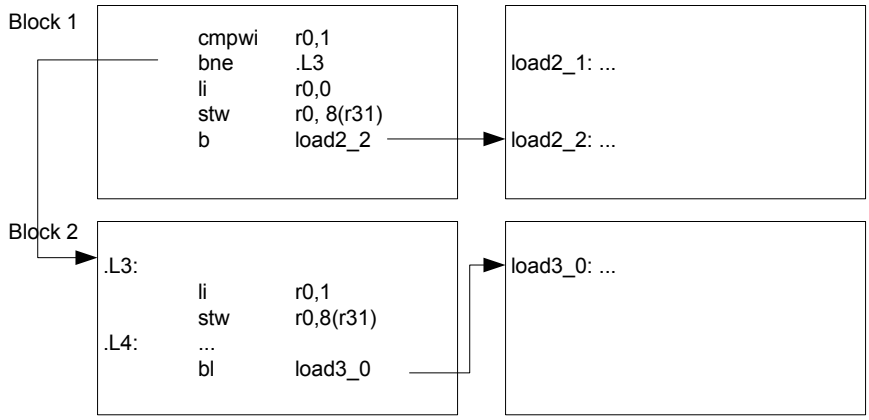

Figure 30: Runtime code modification.

The run-time code modification only changes the instruction which issues the request. Other branches (such as `b load2_2` in Figure 30) 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. □

Listing 3 *Link():* Run-time Linking.

Input: Branch Number i, Branch Information BI
Output: None
begin
1 Save registers used in this function
2 Obtain the branch instruction from the branch information table
3 Determine the needed block ID
4 **if** The needed block is not loaded **then**
5 Request the needed block
6 Wait until the block is loaded
7 **end if**
8 Compute the branch target address
9 Rewrite the off-block branch to jump to the branch target address
10 Restore registers
11 Jump to the branch target address
end

Listing 3 shows the pseudo code for run-time linking when an off-block branch is taken for the first time. The off-block branch will jump to a corresponding address in the branch loader code. The branch number is loaded and is passed to the *Link()* (softstream linker) function. In the softstream linker function, registers which are used in this function are saved (line 1). Since the branch number is passed as a parameter, the *Link()* function obtains the branch information from the branch information table (line 2). From the original off-block branch instruction stored in the branch information table, we can determine the stream block ID of the needed stream block (line 3). If the needed stream block is not loaded into memory, the needed stream block will be requested and the program is suspended until the stream block is loaded (lines 4 to 7). After the needed stream block is loaded, we calculate the branch target address (line 8) and modify the off-block branch to jump the intended location (line 9). The used registers are restored (line 10) and the function jumps to the intended location.

5.5 Program Profiling

It is extremely unlikely that the application executes its code in a linear address ordering. Therefore, the stream units should preferably be streamed in the order of code execution. This can minimize the occurrence of application suspensions resulting from missing code blocks. Program profiling can help determine the order of code to be streamed in the background and/or on-demand. When a code miss occurs, the associated missing code block will be requested. When a stream unit is requested, the transmission order of stream units will restart from the requested stream unit. A software execution path can be viewed as a traversal of a control flow graph such as the graph illustrated in Figure 31. When the program execution flows along a certain path, the streaming can potentially be conducted accordingly. Example 5.5.1 shows an example how a program file may be profiled. A path prediction algorithm

is necessary for background streaming to minimize software misses. Since we do not have an automatic tool to profile program files, we manually predict the software execution path. However, if developed, an automated tool could be used and could effectively interface to our streaming methodology.

Example 5.5.1 Suppose that a program file is divided into eight blocks, and stream units are created. As illustrated in Figure 31(a), for an execution path consisting of stream blocks B0, B6, B1, B2, B3, B1, B4, B1, B5, B3, B4, B1, B4, B3, B2, B3, B1, B2 and B6 in the given order, assuming that stream blocks are not removed from the client memory, we can send stream units containing stream blocks B0, B6, B1, B2, B3, B4, and B5. A stream unit containing the stream block B7 need not be streamed since it is not in the execution path. If the stream block B7 is needed, the client will request stream block B7, and a stream unit containing B7 will be sent by the server. In short, for this execution path, a transmission profile (containing transmission sequence B0, B6, B1, B2, B3, B4, and B5 as shown in Figure 31(b)) provides low occurrence of application suspensions. □

5.6 Performance Metrics

A stream-enabled application has many components that individually contribute to performance. However, in performance analysis, we only measure the components which are affected by software streaming. Thus, the performance metrics are soft-stream overhead, application load time, and application suspension time.

5.6.1 Softstream Overhead

The process of generating a stream-enabled file adds additional information (a soft-stream message header shown in Figure 7 and a stream unit header shown in Figure 4) which we call *stream-enabling information*. This stream-enabling information is added

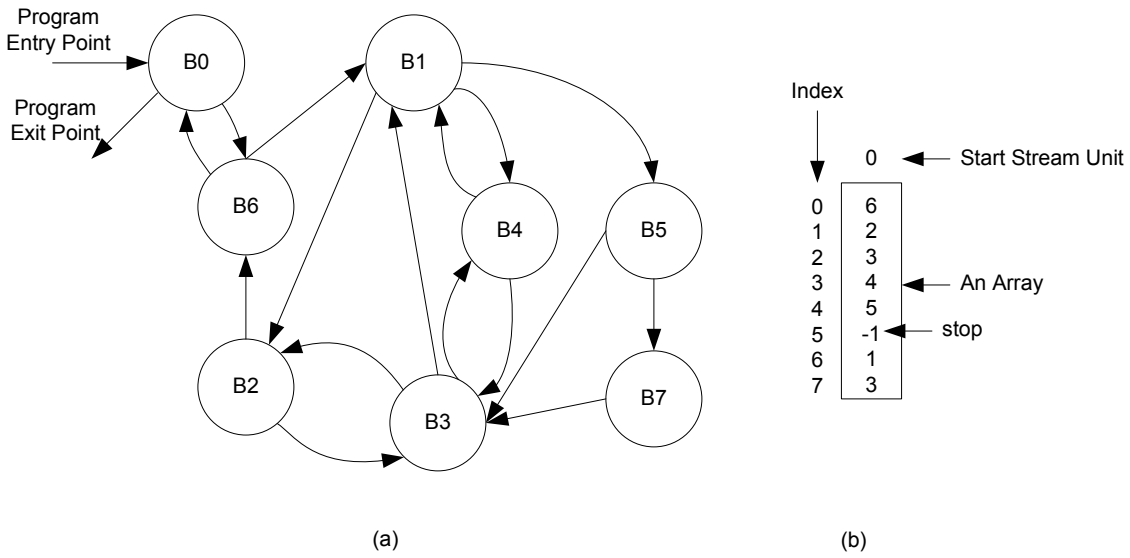

Figure 31: Program Profiling: (a) Control flow graph of the software (note all edges have associated conditions not shown). (b) A transmission profile.

to the original file to enable the file to be streamed. The stream-enabling information added constitutes the *softstream overhead*, potentially resulting in additional bandwidth utilization, memory usage, and processing time. Thus, softstream overhead comprises three specific types of overhead: *bandwidth overhead*, *memory overhead*, and *processing time overhead*.

5.6.1.1 Bandwidth Overhead

When stream-enabling information is sent to a client over a network, additional bandwidth is utilized. Therefore, the more stream-enabling information is added, the higher bandwidth is needed. Generally, since the client processor can process data received from a network more quickly than the network can supply new data, the client can generate part of the off-block branch information, instead of transmitting all off-block branch information. Specifically, the client can perform load time code modification, generating a branch information table and branch loader code for the application. However, the off-block branch information should be sufficient so that

the client does not spend an excessive amount of time processing generating the off-block branch information. Furthermore, in a network which does not provide quality of service (QoS), the amount of time for the client to obtain a stream unit is very difficult to predict. Thus, the amount of the stream-enabling information should be carefully considered.

5.6.1.2 Memory Overhead

The size of stream-enabling information transmitted by the softstream server is not necessarily the same as the size of *stream-execution information* stored in the client memory. Stream-execution information for an application consists of stream block table, branch information table, and branch loader code. In particular, the softstream message header is dropped and is not individually stored at the client. The stream unit header is used to generate perform load time code modification and generate an entry in the branch information table; after initial use, the stream unit header may dropped (erased). Memory overhead is memory space occupied by stream-execution information.

5.6.1.3 Processing Time Overhead

After a softstream client receives a stream unit, the softstream client uses the stream-enabling information to assemble the stream unit into the application. Furthermore, during program execution, if an off-block branch is taken for the first time, the soft-stream client has to check if the stream unit containing the target address of the off-block branch is in memory. Then, appropriate modification of stream blocks is performed. We call the time the softstream client uses to handle (to verify, to modify, and to load) stream units *processing time overhead*.

5.6.2 Application Load Time

Application load time is the time between when the application program is selected to run and when the CPU executes the first instruction of the selected application. Application load time is proportional to the total size of stream units needed before the application can begin execution and is inversely proportional to the speed of the transmission medium. The application program can start running earlier if the application load time is lower. Application load time can be calculated as illustrated in the equations below.

$$t_{application\ load\ time} = \sum_{i=0}^{n_s} t_{stream\ unit\ download\ time}$$

where

$t_{stream\ unit\ download\ time}$ is the time to download a stream unit,

n_s is the number of stream unit(s) needed before the application can begin execution.

Note that $t_{stream\ unit\ download\ time}$ may be difficult to measure since network and server conditions are unpredictable. However, if we ignore all unpredictable factors as well as ignore the time used in requesting an application or a stream unit, we can exclusively use the amount of time to transfer a stream unit to estimate the time to download a stream unit. The transfer time of a stream unit can be computed as follows:

$$t_{transfer} = \frac{u\ bytes}{r\ bps} \times \frac{8\ bits}{1\ byte}$$

where

u is the size of the stream unit in bytes,

r is the bandwidth of the transmission medium in bps.

Example 5.6.1 illustrate how application load time is estimated.

Example 5.6.1 Suppose we need to stream 4 stream units over a 128 bps connection before the application can start running, and the sizes of stream units are 4124 bytes, 4132 bytes, 4140 bytes, and 4156 byes. The estimate application load time is 1.03 seconds. □

5.6.3 Application Suspension Time

An application suspension occurs when the next instruction that would be executed in normal program execution is in a stream unit yet to be loaded or only partially loaded into memory. The application must be suspended until the required stream block is streamed into memory. The worst case suspension time occurs when the needed stream unit is still at the server and the client has to send a request to receive the needed stream unit; in this case the suspension time is the time to download the entire stream unit (note that we assume we can ignore the relatively tiny amount of time used in requesting the stream unit). The best case occurs when the stream unit is already in memory. Therefore, given our assumptions, the suspension time is between zero and the time to download the entire needed stream unit as shown in the following equation. The time to download a stream unit depends on many factors such as the network and server conditions which are unpredictable. However, application suspension time is also proportional to the stream unit size. The application developer can vary the stream unit size to obtain an acceptable application suspension time in ideal conditions if a block miss occurs.

$$t_{suspension} \begin{cases} = & 0, & \text{when the stream unit is loaded} \\ \leq & t_{stream\ unit\ download\ time}, & \text{when the stream unit is not fully loaded} \end{cases}$$

where $t_{stream\ unit\ download\ time}$ is the time to acquire the needed stream unit.

For applications involving many interactions, low application suspension time 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 [35]. 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 stream unit misses occur.

5.7 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, stream block size and program execution path. Without knowledge about the environment for software streaming, the performance of the system can be suboptimal. For instance, if the stream block size is too small and the processor can finish executing the first stream block faster than the transmission time of the second stream block, the application must be suspended until the next stream block is loaded. This would not perform well in an interactive application. Increasing the stream block size of the first stream block will delay the initial program execution, but the application may run more smoothly.

The most obvious overhead is the code added during the stream-enabling code generation step for block streaming. For block streaming, each off-block branch adds four bytes of extra information to the original code. The off-block branch information (added code) for off-block branches is the offset of the instruction which occupies four bytes. The original code and the off-block branch information are put together into a stream unit. A stream unit must be loaded before the CPU can safely execute the code. Therefore, the added off-block branch information incurs both transmission and memory overheads.

Table 3 shows bandwidth overhead and memory overhead for an off-block branch. The bandwidth overhead is four bytes and the memory overhead (assuming no fragmentation) is 20 bytes. The bandwidth overhead is the off-block branch information (the location of the branch). We use the off-block branch information to generate additional off-block branch information during load-time code modification. For each off-block branch, we generate 12 bytes (three instructions) for the branch table and eight bytes for the branch information. Therefore, the memory overhead is higher than the bandwidth overhead. Increasing the stream block size may reduce total overheads for a stream unit since the stream block may contain less off-block branches. However, a larger stream block size increases the application load time and the application suspension time since a larger stream block takes longer to be transmitted.

Table 3: Stream-enabled program softstream overhead for a stream unit.

Type of overhead	Overhead per off-block branch	Overhead for stream block b
Bandwidth	4 bytes	$4 \times n_b$ bytes
Memory	20 bytes	$20 \times n_b$ bytes

Note that in Table 3 n_b is the number of off-block branches in stream block b.

Processing time overhead is 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.

5.8 Summary

One of the crucial steps for enabling program files for streaming is the code generation step. At this step, we determine what is best for an application by considering the trade-offs of the overheads. The amount of off-block branch information generated directly affects the performance of the application.

Load-time code modification is done for each block received by using the off-block branch information. At this step, the block is linked to the branch table. When the branch is executed and the target path is taken, the CPU executes an instruction located at the branch table. We perform run-time code modification at this time. If the needed stream block is not in memory, the client requests the needed stream block from the server. After the needed stream block is in memory, linking between stream blocks actually occurs. Then, the program can continue its execution.

In the next chapter, we will describe stream-enabled file input/output which is a block streaming approach for data file.

CHAPTER VI

STREAM-ENABLED FILE INPUT/OUTPUT

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

Traditionally, many embedded applications download the entire data file before starting using the data. However, downloading the entire data file may take a significant amount of time when the file is large and is located remotely. This may cause the applications to be suspended for a long time. In this chapter, we present a method to send file data incrementally and to allow embedded application to access the data file while transmission may still be in progress.

6.1 File Transfer

An application may use only a subset of a required data file at a particular time. Therefore, requiring all data to be available to the application at once is typically unnecessary. Example 6.1.1 shows a quantitative comparison between downloading the entire file versus just downloading the needed data.

Example 6.1.1 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. As shown in Example 6.1.2, interleaving the transmission and the computation of data is faster than serializing data transmission and the computation.

Example 6.1.2 Suppose that the game application in Example 6.1.1 reads 1 MB of data over a 128-Kbps link to render the first scene, and the game application is limited by the computing capability of the processor which enables the application to process data at a rate of 64 KB/s. As shown in Figure 32, downloading a 1 MB block takes approximately 65 seconds and processing it takes 16 seconds. Thus, the user has to wait 81 seconds. On the other hand, by downloading 4 KB of data at a time and concurrently allowing the game application to process the data after a 4 KB block is loaded, the user has to wait only 65 seconds to begin to play, assuming that the game application reads a few bytes at a time. The game application can finish rendering the scene 1.25X faster when the data is transmitted in 4 KB blocks as compared to when the entire 1 MB of data is sent at once. Note that we omit context switching time because the

context switching time is typically small and thus its effect on the result is negligible, and we also ignore protocol overhead. □

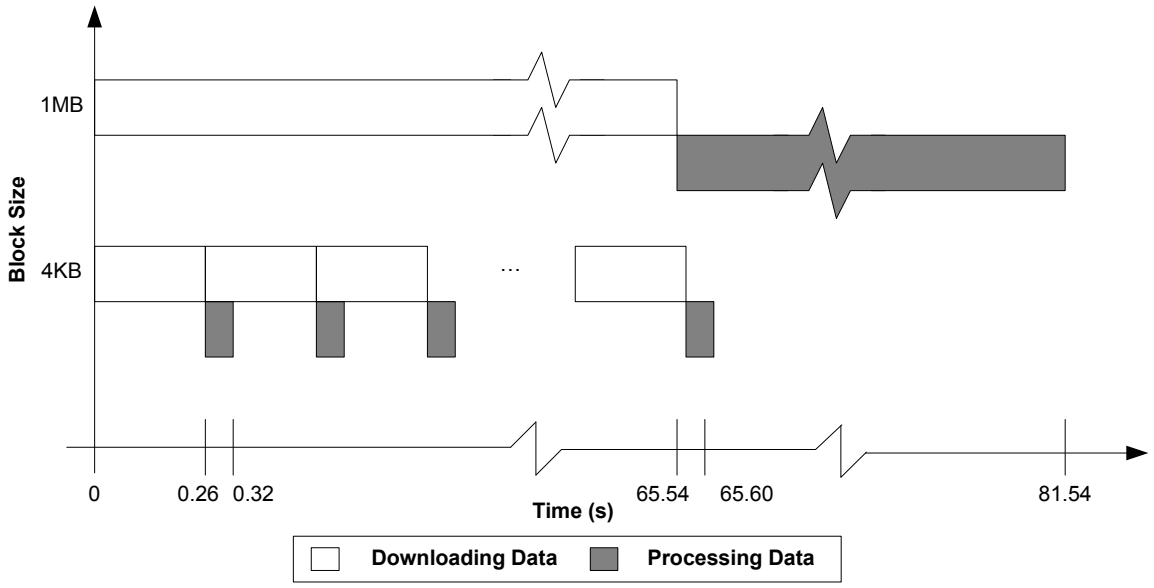

Figure 32: Downloading and processing data for 1MB of data streamed by block sizes of 1MB and 4KB.

Using the quantitative comparisons from Example 6.1.1 and Example 6.1.2, 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. Also, the transmission bandwidth and memory usage may be lowered since unneeded data may not be transmitted. We describe our implementation in the following subsections.

6.2 SIO Messages

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 on how to transmit data blocks) for each file. We assign an ID to each data block sequentially in the same order in which they appear in the file, starting from 0. When a client requests a file, the softstream server sends the file information (e.g., file size and block size) and streams data blocks to the client according to the transmission profile.

The stream unit format for a data block is shown in Figure 33. A softstream message can be classified as a request message or a reply message. For SIO, a client sends a request message when a file is open, closed, read, or written. The server sends reply messages to respond to the client request. When the client requests a file to be streamed (open), the the client sends the filename. When the client requests a needed stream block, the client sends the stream block ID or the stream unit ID. When the client write data into a file, the client sends offset, data size, and data. The offset field represents the location in which the data will be written. The data size is the size of the data in the data field. The server sends stream units in response to the client request. When the missing block is requested, the server sends the stream unit ID, data size, and data. The server also sends typical file information such as access status (granted or denied), file size, and block size.

Figure 33: The stream unit format for data blocks.

At the client, when the application opens a data file, if the data file has not yet been requested, the SIO client sends a request to receive the data file from the server and uses the data file information to construct data file status information. The data file status information contains a *data block table*. Each entry in a data block table is an address field storing the location of the data block associated with that entry. However, if the address is invalid (we use the address value of 0xFFFFFFFF

to indicate an invalid address), the data block is not yet loaded; otherwise, the data block is in memory. Figure 34 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. Example 6.2.1 illustrates how a data block table is updated.

Example 6.2.1 In Figure 34, when the stream unit containing a data block with stream block ID of 1 is received, the SIO client allocates the needed memory space. Suppose that the allocated memory space starts at 0x00010400. The SIO client loads the data 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. □

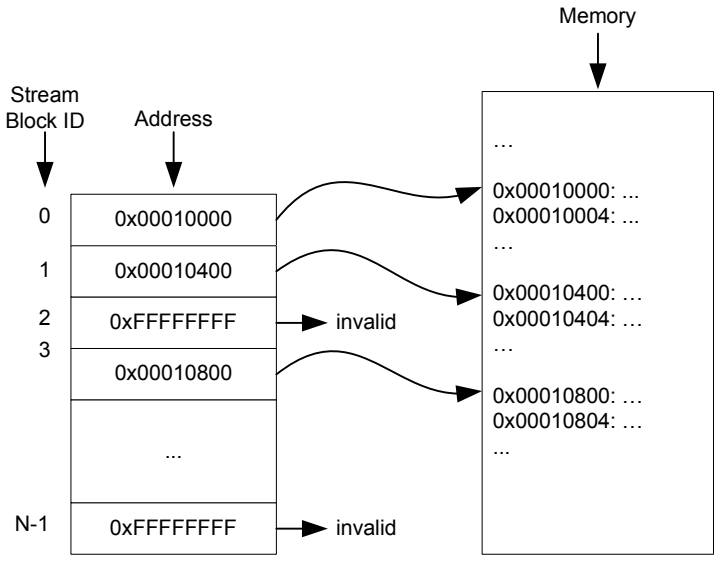

Figure 34: Data block table.

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, an unneeded data block can be removed by changing the address field of the block to an invalid address and freeing the memory

occupied by the unneeded data block. If the removed data block is later needed, it will be requested for retransmission. In SIO, we only allow a server to transmit fixed size blocks to lower processing time. However, the data block containing the end of file can be smaller than others.

6.3 SIO Function Calls

We implemented SIO function call primitives, namely, *sio_open()*, *sio_close()*, *sio_lseek()*, *sio_read()* and *sio_write()*. These function calls are very similar to the file I/O system calls. The applications must use SIO function calls to perform SIO operations. The *sio_open()* function sends a request to the server to open a file. If the file can be opened, it returns a file descriptor (file number) to the application. The file descriptor is an integer used to represent the opened file. The SIO server starts streaming data blocks after the open request is received. The *sio_close()* function deallocates the file descriptor and updates the data if the file is modified. The *sio_lseek()* function sets the file pointer to a specified offset in the file. The value in the file pointer is the current location in the file. The *sio_read()* function attempts to read a specified amount of data. The SIO client checks if the data is available at the client device. If not, the data block containing the needed data will be downloaded before the application can continue. The *sio_write()* function attempts to write a specified amount of data. The *sio_write()* function has two modes: write-through and delayed-write. In the write-through mode, the SIO client sends data to the SIO server first and returns control back to the application. On the other hand, in the delayed-write mode, SIO may delay sending the data to the server.

The following descriptions are the internal design of the SIO functions.

- *int sio_open(const char *path, int oflag)*: When *sio_open()* is invoked, the client sends a request message to stream a file. The softstream ID is set to zero and the service type is set according to *oflag*. The server assigns a softstream ID which

can be used as a file descriptor. Then, the server sends back file information such as access mode, file size, and block size. The client uses this information to construct file status information and returns a file descriptor back to the application. The file descriptor is the identification of the opened file. The application uses the file descriptor to communication with SIO function calls. The server can start streaming the file.

- *int sio_close(int fildes)*: The application requests the closing of the open file associated with *filedes*. Using the softstream protocol (Section 4.3), the client sends a message with the value of service type of 0x00000000. When the server receives this message, the server closes the stream.

- *int sio_read(int fildes, void *buf, int nbyte)*: The *sio_read()* function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*. The client checks if the block(s) containing the needed data is/are in memory. If not, the client sends a request to stream the needed block(s). The needed data may be located in different blocks. When the server receives a block request, it sends the requested block to the client. In continuous stream service, if the server is still sending stream units according to a certain order in the corresponding transmission profile, the server ends the previous streaming order and starts a new streaming order starting from the requested block. As a result, block misses may be reduced. The client also keeps track of the position in the file. The *sio_read()* function returns the amount of data read from the file.

- *int sio_write(int fildes, const void *buf, int nbytes)*: The *sio_write()* function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with file descriptor *fildes*. The client sends the write message to the server. The payload for the write request consists of the offset in the file, the

size of the data, and the data. The *write()* function return the amount of data written.

- *int sio_lseek(int fildes, int offset, int mode)*: The *lseek()* function set the current position in the file associated with file descriptor *fildes*. The file pointer is set according to the value of *mode*. The *mode* value allows the application to flexibly set the current position in the file. If *mode* is 1, the file pointer is set to *offset* bytes. If *mode* is 2, the pointer is set to the current location plus *offset*. If *mode* is 3, the pointer is sent to the size of the file plus offset. If the position of the file pointer is in a block not loaded in memory, the client requests the server to stream the block. The *sio_lseek()* function return the position of the file pointer.

6.4 SIO Support by Modifying the Application Binary Image

An application can use SIO function calls by including SIO files in the source code. However, source code of an application might not be available. Therefore, we support this application by modifying the applications' binary images. In a typical system, an application invokes a system call to perform file I/O. A system call is a call to the kernel for a specific function which controls a device or executes a privileged instruction. Usually, when a system call is invoked, the system call function loads a system call number into a register and generates an interrupt. The kernel uses the number in the register to distinguish which service is requested. After finishing the system call service, the control is handed back to the application.

In order to modify an application's binary image so that the application can use SIO functions, we identify the system calls which are used to handle file I/O and replace the system calls with the SIO functions. The process for modifying binaries to use SIO functions is illustrated in Example 6.4.1.

Example 6.4.1 Figure 35 shows a subset of a program using the uClibc [40] library. In *main()*, the system calls *open()* (line 3) and *write()* (line 5) are called to open a file and write to it. When *open()* (__libc_open) is called, it manipulates the passed parameters and jumps to __syscall_open (line 18). The __syscall_open code loads system call number 5 into the register *r0* (line 13) and jumps __uClibc_syscall (line 14). The __uClibc_syscall code generates an interrupt by using the *sc* instruction. Similarly, when the *write()* (__libc_write) is invoked (line 5), the __libc_write loads system call number 4 to register *r0* and jumps to __uClibc_syscall (line 10).

```
1  10000284 <main>:
2  ...
3  100002a8:      48 00 12 69        bl      10001510 <__libc_open>
4  ...
5  100002c0:      48 00 12 41        bl      10001500 <__libc_write>
6  ...
7
8  10001500 <__libc_write>:
9  10001500:      38 00 00 04        li      r0,4
10 10001504:      48 00 00 dc        b       100015e0 <__uClibc_syscall>
11
12 10001508 <__syscall_open>:
13 10001508:      38 00 00 05        li      r0,5
14 1000150c:      48 00 00 d4        b       100015e0 <__uClibc_syscall>
15
16 10001510 <__libc_open>:
17 ...
18 1000157c:      4b ff ff 8d        bl      10001508 <__syscall_open>
19 ...
20
21 100015e0 <__uClibc_syscall>:
22 100015e0:      44 00 00 02        sc
23 100015e4:      4c 83 00 20        bnslr
24 100015e8:      48 00 00 04        b       100015ec <__syscall_error>
```

Figure 35: A subset of a program using uClibc file I/O.

This program can be modified to use SIO functions by locating the code which generates an interrupt. We can insert code to check if the system call is to handle file I/O. The inserted code will redirect to an appropriate SIO function call. Otherwise, a system call interrupt will be generated. We can also trace back to the location from which the system call is invoked by locating the branch instruction which calls a particular libc function; then, we modify the the

branch instruction to jump directly to the SIO functions. For example, line 3 and line 5 will be modified to branch to *sio_open()* and *sio_write()*, respectively. □

6.5 Using Stream-Enabled File I/O for Program Files

Chapter 5 illustrates a method to stream a program file by modifying its binary image. The necessary off-block branch information is generated off line. The objective of the softstream generator is to save the client processing time. The client does not have to generate all off-block branch information. However, if the client has high computing power, we can use our stream-enabled file I/O technique to stream the program file.

Using stream-enabled file I/O, a program file is treated as a data file. The program file is divided into blocks, and a transmission profile is created by analyzing the program flow. When the program file is requested, the server sends stream units containing the program file blocks according to the transmission profile. However, when a stream unit arrives at the client, a version of softstream loader which has softstream generator capabilities generates off-block branch information and code from the stream unit and the softstream linker integrates the stream-enabled block into the application. In other words, the stream-enabled code generation step is done at the client.

Another advantage for using stream-enabled file I/O may occur when the client supports demand paging. The size of the block would be the same as the size of the page. There is no needed to modify branches, since if a branch target address is in a page not yet allocated, a page fault occurs. When the page fault occurs, the operating system can read the page using stream-enabled file I/O. Using a transmission profile to send the program file, the page fault rate can be reduced.

6.6 Data Profiling

Data profiling is used to predict the run-time file access behavior of an application in order to stream data accordingly. Data which is most likely to be used first will be streamed first. Using data profiling, the data miss rate and the application suspension time due to missing data can be significantly reduced. Furthermore, data which is not needed by the application for a certain execution 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 structures 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 a 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 based on an algorithm. 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 as shown in Example 6.6.1.

Example 6.6.1 As mentioned in Example 6.1.1, 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 data rate higher than the rate for the subsequent scenes. In this way, the subsequent scenes will be sent while the user is playing the first scene without utilizing resources as much as the first scene. When the user advances to the next scene, the next required 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 characteristic, we profile the file so that the data will be sent in a similar fashion to the data access pattern or characteristic. Therefore, the application would have access to data more quickly. Example 6.6.2 shows a profiling method for a file which has a well-known data access characteristic.

Example 6.6.2 Suppose that an application searches for a specific value in a file which stores data sorted in ascending order. The application uses a binary search algorithm which has an efficiency of O(log N), where N is the number of data elements in the file. Since the data access characteristic of the application is according to the binary search algorithm, we can profile the file according to the binary search algorithm as illustrated in Figure 36. In this example, we divide the file into 10 blocks labeled from 0 to 9 (Figure 36(a)) and create a transmission flow graph (Figure 36(b)). Each edge is assigned a weight value based on the probability of the blocks corresponding to next level nodes to be sent. We also show two possible transmission profiles based on depth-first (Figure 36(c)) and breadth-first (Figure 36(d)) graph traversing algorithms. Since the number of blocks is even we pick block 4 to be the middle block. A depth-first transmission profile sends the data blocks starting from the root node in a transmission flow graph, then to each child node in the same branch until reaching the leaf

node before starting another branch. The breadth-first transmission profile, on the other hand, sends the data blocks starting from the root node, then to each child node on the same level before advancing to another level.

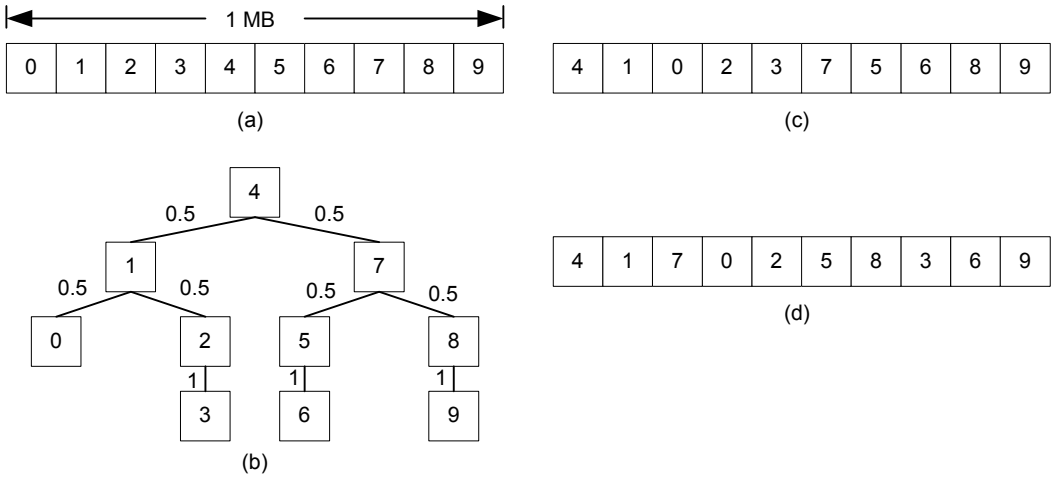

Figure 36: Profiling Data: (a) The file is divided into blocks. (b) The corresponding transmission flow graph. (c) Depth-first transmission profile. (d) Breadth-first transmission profile.

Suppose that the depth-first transmission profile is used and the data we are looking for is in block 5. According to the transmission profile, we send block 4 and then block 1. Suppose further that, while sending block 1, the request for block 7 comes in. We send block 7, block 5 and then block 6. If we profile the file to save the client memory, we wait until the application sends another request since we reach a leaf node. Otherwise, we send block 8 and block 9. At the client device, the application finds that the data is located in block 5. For this scenario, if we chose to save the client memory with the continuous stream flow control, we only send five blocks of data instead of sending the entire file (10 blocks), saving 50% of memory. Note that the continuous stream flow control does not mean that all blocks must be sent; the continuous stream flow control means all blocks in a certain path in the transmission profile is sent until the end of the path is reached. □

Please note that we do not propose a specific, unique algorithm to be always used to create a transmission profile but instead provide support by specifying how the data structure in the transmission profile for any algorithm, such as a depth-first graph traversing algorithm, which the user desires to implement. For example, to create a transmission profile, the user may use an algorithm to discover a path that maximizes the sum of edge weights traversed in the transmission flow graph.

6.7 Performance Analysis

Since we divide a data file into blocks to create a transmission profile, we do not add any data to the original file. Therefore, the size of the file remains the same. However, when the server sends a block to the client, the server adds a stream unit ID and the data size. As shown in Table 4, the bandwidth overhead is only eight bytes. At the client, we keep track of data blocks using a data block table. Therefore, as shown in Table 4, the memory overhead is only four bytes (assuming no fragmentation). Both bandwidth and memory overheads are relatively small. For example, bandwidth and memory overhead for a 1-KB block is only 0.78% and 0.39%, respectively. Note that if we include the softstream header, the bandwidth overhead would be 16 bytes.

Table 4: Stream-enabled file I/O overhead for a stream block.

Type of overhead	Overhead per stream block
Bandwidth	8 bytes
Memory	4 bytes

Runtime overhead occurs every time the data is accessed. This is because the *sio_read()* function checks if the block containing the data is in memory. If the data is in the same block, reading one byte or a few bytes incurs the same overhead.

6.8 Summary

File I/O operation latency may be significant when the file is located remotely. The latency can increase significantly if the entire file is required to be downloaded before a file operation can begins. A large file can take a considerable amount of time to download to the client. However, using our stream-enabled file I/O method, the client can access data in the file while it is being transferred, which may reduce the file I/O operation latency.

In SIO, we divide a file into blocks and create a transmission profile. The server sends the blocks according to the transmission profile. The sequence of transmission can be interrupted and another can be started if the client operates in a different mode. The application at the client uses the SIO function calls to perform I/O operations.

In the next chapter, we will discuss several techniques to improve block streaming performance and to allow block streaming to be used on a limited memory embedded device.

CHAPTER VII

BLOCK STREAMING PERFORMANCE ENHANCEMENT

This chapter examines some more advanced techniques related to stream-enabled code generation, further enhancing the performance of stream-enabled software. First, we discuss a code transformation technique for software streaming. Then, we introduce a method which enables a small memory footprint embedded device to support a large application, i.e., larger than the available memory, by profiling and allowing stream units to be removed from the client's memory. The code transformation scheme and profiling are done at the server while the removal of stream units is done at the client.

7.1 Code Transformation

Chapter 5 discusses a stream-enabled code generation method, which divides the program binary image into blocks before generating stream-enabled code. The program binary image is used as it is, without considering other issues such as performance and resources. However, this section introduces techniques which may improve performance and may reduce resource usage by statically transforming the program.

7.1.1 Determining Function Boundaries

One drawback for randomly dividing a binary image into blocks is that some of the code in a particular block may not be used. For instance, consider the case where the first instruction of a function is the last instruction in block. For this case, perhaps only one instruction of the entire block (the last instruction) may be needed if the

other function(s) in the block are never called. As a result, memory and bandwidth are not efficiently used. Moreover, when the function is called and the block is not in memory, we have to stream two blocks for the function to work. However, by obtaining the size of each function, the block boundaries can be enforced to more closely match with function boundaries.

Example 7.1.1 shows that occurrence of application suspensions is reduced when the block boundary is match closely with function boundary.

Example 7.1.1 Figure 37(a) shows that function *fn2()* is split with part of the function in the first block (2 instructions) and the rest is in the second block. When *fn2()* is called and is not in memory, we request the first block and call the function. The application may be interrupted shortly thereafter because it needs the rest of the function code to return back to the caller. The second block may be streamed in background or on-demand. When the second block is loaded, the application continues its execution. In this scenario, the application is interrupted twice, and we have to send two blocks. If *fn2()* is put in the second block, we only have to send one block, saving memory and bandwidth. Moreover, the occurrence of application suspensions is reduced.

As illustrated in Figure 37(b), *fn2()* is placed in the second block. If client memory is allocated into fixed size blocks corresponding to fixed sized code blocks, this method creates internal block fragmentation which wastes client memory. For example, the first block of Figure 37(b) contains eight bytes of unused memory space. Therefore, the amount of wasted space must be taken into consideration before matching the function boundaries with the block boundaries. If the wasted space is too large, it may be better to leave the function in different blocks. □

Function boundaries can be determined using the symbol table if the binary image format is in Executable and Linking Format (ELF). Figure 38 shows a sample

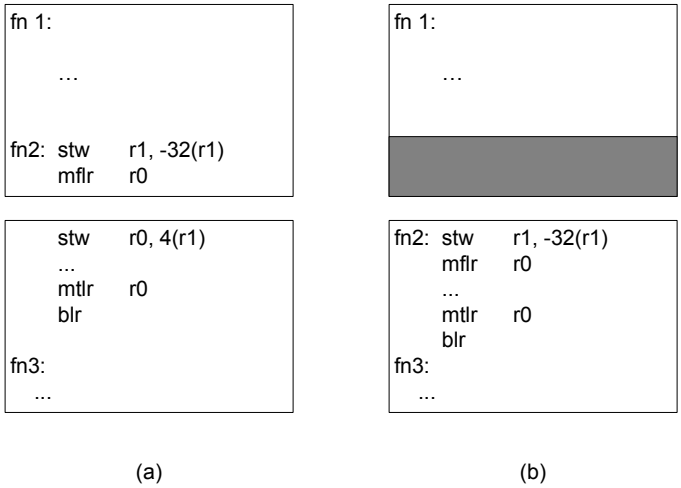

Figure 37: Enforcing block boundaries. (a) A function is placed into separate blocks. (b) The block boundaries are matched with function boundaries.

symbol table. The value field gives the value of the associated symbol, depending on the context. For a function, this field is the address of the first instruction of the function. The type field indicates that the symbol is associated with a function or other executable code. The function belongs to section name *.text*. The size field gives the size information associated with the symbol. Finally, the name field is the name of the symbol. In our case, it is the name of the function. Using the symbol table, we obtain the function location and size of the function. Hence, we can easily determine the boundaries of the function using the symbol and can match block boundaries with function boundaries.

Value	Type	Section Name	Size	Name
10003504	F	.text	00000078	__pthread_find_self
1000d77c	F	.text	00000008	geteuid
100093dc	F	.text	00000018	pthread_mutexattr_getkind_np
1000d4d8	F	.text	00000008	__syscall_pread
1000d47c	F	.text	00000008	__syscall_llseek
1000f1ac	F	.text	0000005c	memmove
10006424	F	.text	00000180	pthread_exit
100080c0	F	.text	000002b0	sem_timedwait
1000d87c	F	.text	00000008	munmap
10027e08	O	.sdata	00000004	__pthread_manager_thread_bos
1000d8cc	F	.text	00000020	sched_getparam
1000e6b4	F	.text	000000dc	_uintmaxtostr
1000d398	F	.text	0000004c	_longjmp
...				

Figure 38: A sample symbol table.

Symbols in the program binary image take storage space, especially if the program is stored in memory such as ROM or Flash memory. For this reason, the symbols in program binary image may have been discarded (e.g., using *GNU strip*). As a result, function boundaries information cannot be obtained from the symbol table. However, we can determine the function boundaries from examining the program code.

The binary code of a function generated by a compiler usually has a single function entry point and a single function exit point. The function entry point is the branch target address of the branch instruction which calls the function. When this branch instruction is executed, the return address (address after the branch instruction) is saved. The function exit point is at the return instruction. The function boundaries are at the function entry point and the function exit point. The code which is not covered by the function caller instruction is either dead code or functions which are called using function pointers.

Another simple technique to obtain function boundaries is to look for return instructions by scanning from the first instruction to the last instruction. The first instruction of the program code is the function entry point, and the function exit point is at the first return instruction encountered. The instruction after the return instruction is the next function entry point. All function boundaries are obtained at the end of the program.

Note that we do not consider the case of a point where a function invokes another function at a function exit point. This is because when the invoked function is done, the program returns back to the current function. Furthermore, the invocation location is irrelevant in determining function boundaries.

7.1.2 Remapping Functions

A programmer usually writes an application in such a way that functions with a similar purpose are put in together in a file. Functions are typically placed randomly

within a file. When compiled, the binary code of the functions are in the same order as the original source code. After generating blocks for streaming, the order of function placement remains the same. Example 7.1.2 shows how the lack of spatial locality of reference degrades stream-enabled software.

Example 7.1.2 Suppose that a program file is divided into three blocks as shown in Figure 39. The functions are in the same order as they were written. The function *fn1()* calls *fn5()*, and the function *fn5()* calls *fn7()*. These function are in separate blocks. When the function *fn1()* is invoked and is not in memory, the block containing *fn1()* will be requested and will be loaded. The function *fn1()* is interrupted quickly because *fn5()* is not in memory causing the block containing *fn5()* to be loaded. The function *fn5()* is also interrupted to load the block containing *fn7()*. Therefore, we need three blocks for *fn1()* to complete its operation. □

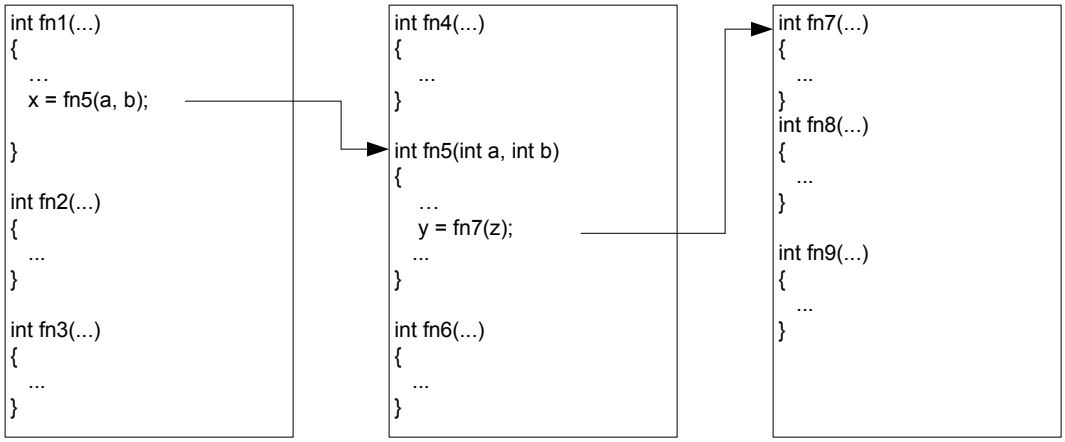

Figure 39: An example shows the program lacks block locality.

In Example 7.1.2, we can remap *fn1()*, *fn5()*, and *fn7()* so that they are in the same block. We only need one block for *fn1()* to complete its operation without being interrupted due to missing code. Remapping functions according to execution paths improves the locality of reference.

Programs often spend 80 or 90 percent of their time in 10 to 20 percent of the code [44]. The frequently used code should be packed together to improve temporal locality of reference of the stream block since the code will be executed more often. If the client has limited memory, stream blocks are removed from memory before other needed blocks is loaded. The stream blocks may be requested more frequently if functions in a program is arranged randomly. However, remapping frequently used functions together may reduce occurrence of application suspensions due to missing stream blocks, since the temporal locality of reference is improved. Therefore, we remap the frequently used functions together.

In order to remap functions, we analyze the application at the function level since the source code may not be available. Then, we create a program call graph which represents the program flow. The binary image is rearranged based on its program call graph to improve spatial locality. Functions which are potentially executed in a proximate time frame will be placed in a proximate memory location. Common functions are also placed in a proximate memory location. After rearranging functions, the stream-enabled application can be generated by dividing the binary image into blocks and generating stream units. A transmission profile of the stream-enabled application is also generated using a profiling approach.

7.1.3 Generating Fixed Size Stream Units

In this section, we discuss a method to generate fixed size stream units and to allow stream blocks to be loaded to anywhere in memory and to be removed from memory. However, note that a conditional branch has a limited range to where the branch can jump. If a conditional branch is an off-block branch, the conditional branch limits where the stream blocks can be loaded.

Recall the 32-bit PowerPC conditional branch format in Figure 21. If a conditional branch is an off-block branch, the first instruction of the corresponding stream-enabling code can only be located in address from 0x00000000 to 0x0000FFFC or in address having displacement less than or equal to 32764 (0x00007FFC) bytes (8191 instructions) from the off-branch instruction. Furthermore, the distance between the off-block branch instruction and the branch target address cannot be more than 32764 bytes apart. Therefore, location of the blocks are restricted. However, this issue can be mitigated by inserting code in the block so that the conditional branch jumps to the a longer branch and is redirected to the stream-enabling code. Using an unconditional branch, the next instruction may be located in address from 0x00000000 to 0x03FFFFFC or in address having displacement less than or equal to 3,554,428 (0x01FFFFFC) bytes (8,388,607 instructions) from the branch instruction. Example 7.1.3 shows a short range branch (a conditional branch) is rewritten to jump to an inserted unconditional branch instruction.

Example 7.1.3 In the original program, Block 1 and Block 2 are close enough for the conditional branch `bne .L3` to reach address `.L3` in Block 2. However, as illustrated in Figure 40(a) by the "X" over the arrow, Block 1 and Block 2 could be placed in memory so far apart that the branch target address (`.L3`) is to far to be reached (i.e., further away then 8191 instructions). Therefore, in Figure 40(b), we insert a branch instruction `b .L3` which is within the reach of the conditional branch `bne .L3`; in this way, the desired jump is achieved in two hops (two jumps). □

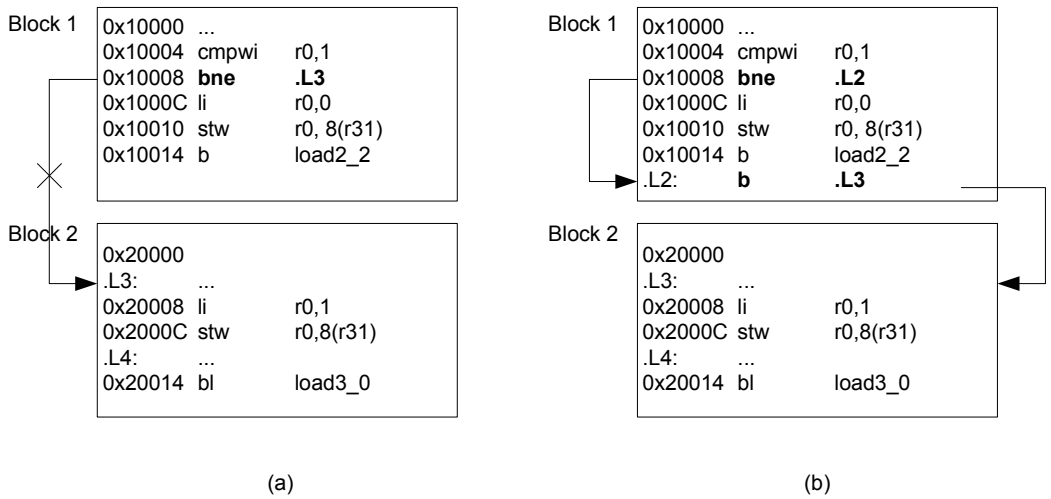

Figure 40: Rewriting a short range branch. (a) The branch target address is too far for the conditional branch. (b) An unconditional branch is inserted to redirect the conditional branch.

For the software streaming implementation which allows blocks to be removed, the block size must be fixed to guarantee avoidance of memory fragmentation. Inserting code after dividing the binary image into blocks results in blocks with different sizes. Fixed size blocks can be generated by taking inserted code into account as shown in Listing 4.

In Listing 4, we start by locating all conditional branch locations with one complete pass over the program P (line 1). Then we enter a loop where we first obtain the block starting the first instruction of the program (line 3). Some of the instructions in this block may be removed later in the process. Then, we scan for a conditional off-block branch in the block. For each conditional off-block branch, we remove an instruction from the block (line 6) and replace the removed instruction with a unconditional branch to jump to the intended location (line 7). Next, we write a conditional off-block branch to jump to the added branch (line 8). We have to fixed all affected branches. The process repeat until the end of the program is reached.

Listing 4 *Generate_Fixed_Size_Blocks():* Generating fixed size blocks.

```
Input: A program P
Output: List fixed size stream units S
begin
 1 Locate all conditional branches
 2 repeat
 3   Get a fix block from the remaining code starting for the low address
 4   repeat
 5     if The conditional branch in the block an off-block branch then
 6       Remove the last instruction and put it back in the remaining code
 7       Add an unconditional branch which jumps the original branch target address
 8       Rewrite the conditional branch to jump to the added instruction
 9       Fix all affected branches
10     end if
11   until All conditional branch in the the current block are covered
12   Add the current block to S
13 until The end of the program is reach
end
```

7.2 Stream Unit Removal

For a client which has limited memory, removing stream blocks from memory is essential in order to support an application larger than the available memory. When a stream block is received, it is linked to the application. Therefore, when the stream block is removed, it must be unlinked. If the stream block is needed later, it will be requested.

7.2.1 Unlinking Mechanism

Unlinking is a reverse process of linking. All the branches which jump to the stream block to be removed must be unlinked. Example 7.2.1 shows unlinking a block using binary rewriting. Note that we can avoid run-time binary rewriting altogether by not performing run-time code modification. However, the code would not perform efficiently if the branch is taken frequently since stream-enabling code performs code checking and redirects to the proper location.

Example 7.2.1 Suppose that the client has to deallocate Block 2 in Figure 41(a) to make room for a new stream block. Since the second instruction of Block 1 (`bne .L3`) jumps to

Block 2 if the condition is satisfied, we have to modify this instruction to jump to the branch table. When the modified instruction is later executed, Block 2 will be requested if it is not in memory. Figure 41(b) shows Block 1 after Block 2 is removed. The second instruction of Block 1 (`bne load2_1`) is change to load Block 2. □

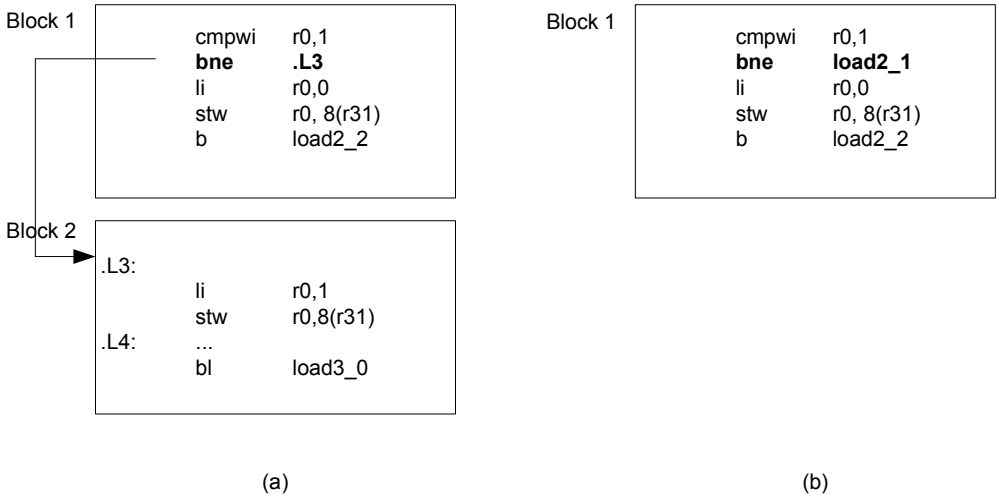

Figure 41: Unlinking. (a) Block 1 and Block 2 are linked together. (b) Block 2 is unlinked from Block 1.

To unlink a stream block, one needs to know all incoming off-block branches to the block to be removed. Therefore, the additional off-branch information includes the number of incoming off-block branches and the branch numbers as described in Section 5.2.3. Using the branch numbers, we locate the instructions which may jump to the removed block. Then, we modify (unlink) the branches to jump to the corresponding locations in the branch table.

7.2.2 Stream Unit Replacement

At the server, we create a program flow graph for the application. The client allocates memory to store stream blocks. When the client requests the application, the client sends the maximum number of stream blocks that the client can allocate. The last 16 bits of the service type field is set the the maximum number of stream blocks

(on-demand stream flow control). The server creates a transmission profile for the application based on the maximum number of stream blocks. The objective is to minimize the number of retransmissions. Therefore, we can create a transmission profile based on an optimal replacement algorithm described in [34]. As a result, a stream block that will not be used for the longest period of time will be replaced first. First, we can apply an optimal replacement algorithm along the predicted program execution path. Then, we can apply the optimal replacement algorithm along other paths. The stream block to be replaced with the requested block is indicated the service type field of the softstream header. When the program execution is as according to the predicted execution path, the number of retransmission will be minimal if we apply the optimal replacement algorithm. Example 7.2.2 illustrates the replacement of stream units.

Example 7.2.2 Figure 42 shows an example of a transmission profile according to the optimal replacement algorithm for a client with a maximum number of blocks of three. A superscript number indicates which stream block to be replaced. If the superscript number is the same as the stream block number, that stream block can be placed in an available memory block. When the client requests the application, the first three stream blocks are sent. Then, the client requests stream block 3, the server sends stream block 3 and advises the client to replace stream block 6, because stream block 6 will not be used until reference 18, whereas stream block 1 will be used at 5, and stream block 2 at 14. Stream block 4 can be sent to replace stream block 2 without waiting for the client request since stream block 2 will not be used until reference 14. When stream block 4 is needed, it will potentially be in memory, reducing occurrence of stream block misses. In the example, if we only requested a single stream block at a time based the optimal replacement algorithm, we would have nine occurrences of stream

block misses. However, with block streaming, we can potentially reduce occurrences of stream block misses to six since stream block 1, stream block 2 and stream block 4 are sent without waiting for the request from the client. □

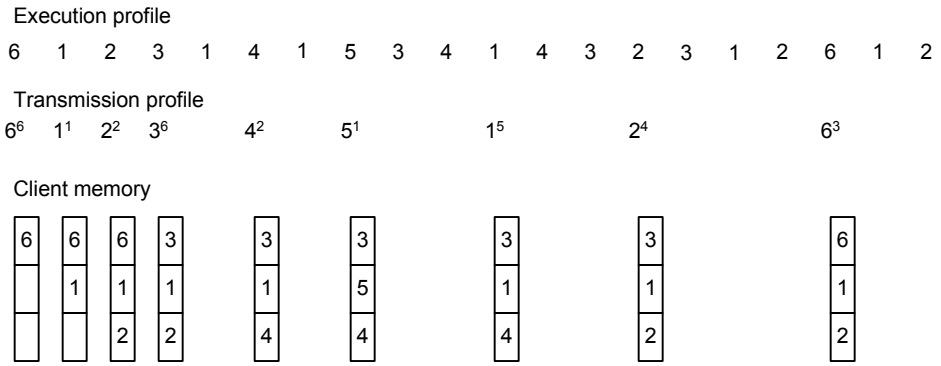

Figure 42: A transmission profile is created according to the minimum retransmission policy.

7.3 Security Issues

One very important area not addressed in this dissertation is security. Security risks may arise while transmitting stream units (network security) or while executing a stream-enabled application (computer security and thread-safe). We briefly discuss these security issues in the next two subsections.

7.3.1 Network Security

The softstream protocol is a layer above a connection-oriented protocol such as TCP which may not be secure. However, the softstream protocol may be implemented so that it will be a layer above a secure layer such as the secure socket later (SSL) [30]. The secure layer provides network security by encrypting and authenticating data. Therefore, stream-enabled applications will have lower network security risks.

7.3.2 Computer Security

A stream-enabled application could possibly be written such a way that it contains malicious code. In the public domain such as the Internet, we suggest that stream-enabled applications should be requested from a trusted server. Furthermore, since we use a binary rewriting technique, program space is writable. A stream-enabled application may intentionally modify other applications. A possible solution is to provide memory protection so that a stream-enabled application cannot write to another application's memory space.

7.3.3 Thread-Safe

We implemented a stream-enabled application using two threads: one for downloading stream units (the softstream loader thread) and the other for running the application (the stream-enabled application thread). When a stream block miss occurs, the stream-enabled thread writes the stream block ID to a shared variable notifying the softstream loader thread to request the stream unit. We can extend block streaming to support multi-threading stream-enabled applications by protecting the shared variable using a semaphore or a mailbox. If stream units are allowed to be removed from memory, the softstream loader/linker can potentially be extended to be thread-safe.

7.4 Summary

The performance of stream-enabled applications can be improved by both statically transforming code and profiling. Furthermore, small memory footprint embedded devices can run many applications as if they were fully loaded. The application can be executed even though the program size is much larger than the actual physical memory.

In the next chapter, we will discuss the block streaming experiments and results.

CHAPTER VIII

EXPERIMENTS AND RESULTS

We verified our software streaming implementation on a hardware/software cosimulation platform and on an MBX860 board. The stream-enabled program file implementation was tested on both a simulation and a board environment, while the stream-enabled file I/O was tested only on a board environment (the MBX860 board).

8.1 Experimental Setup

In this section, we describe our hardware/software cosimulation environment and our board environment used. The hardware/software cosimulation environment is used to debug our stream-enabled profile file method since we can easily examine registers and memory whereas the board environment is used to measure the actual performance which includes all overhead.

8.1.1 Simulation Environment

We implemented a tool for breaking up executable binary images in PowerPC assembly into blocks and generating corresponding stream units ready to be streamed to the embedded device. We simulated our block streaming for program file method using hardware/software co-simulation tools which consist of Synopsys® VCS™ [36], Mentor Graphics® Seamless CVE™ [20], and Mentor Graphics XRAY® Debugger [20]. Among the variety of processor instruction-set simulators (ISSes) available in Seamless CVE™, we chose to use an ISS for the PowerPC 750 architecture. As illustrated in Figure 43, the simulation environment is a shared-memory system. Each processor

runs at 400 MHz while the bus runs at 83 MHz. A stream-enabled application runs on the main processor. The I/O processor is for transferring data.

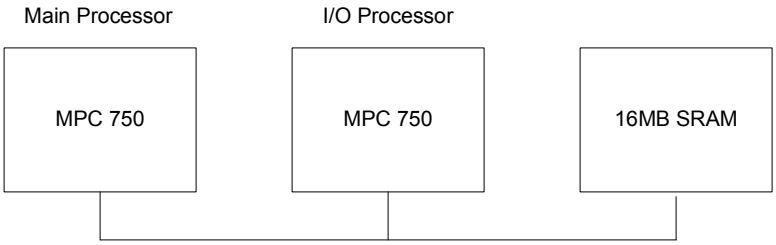

Figure 43: Simulation environment.

8.1.2 MBX860 Broad

For the board setup, we used an MBX860 board [22], [23]. The MBX860 board consists of a PowerPC 860 processor with a 4 MB DRAM, a 2 MB Flash, and a 10 Mbps Ethernet port. The board runs Linux version 2.4.21-ben2 [19]. As shown in Figure 44, 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. We also used the Linux Traffic Shaper [39] (shapercfg version 2.2.12) to regulate the connection bandwidth. Note that the Linux Traffic Shaper does not regulate connection bandwidth at the exact configured bandwidth.

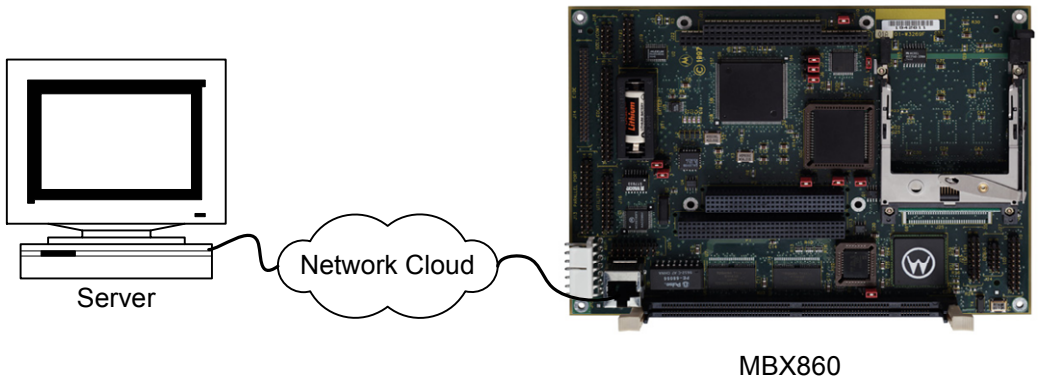

Figure 44: Experiment setup.

8.1.3 Code Size

Table 5 shows softstream program code sizes in number of lines of C. As illustrated, in Figure 45, he softstream server program and softstream generator program are on a softstream server. As shown in Figure 45, the softstream client program, softstream loader/linker program, and SIO function call code are on a client device.

Table 5: Softstream programs.

Program	Size (C lines)
Softstream server	$\approx$ 3400
Softstream client	$\approx$ 1400
Softstream generator	$\approx$ 2200
Softstream loader/linker	$\approx$ 1300
SIO function calls	$\approx$ 1500

Figure 45: Code location.

8.2 Stream-Enabled Program Files

In the following two subsections, we used our block streaming for program file method to transmit the stream-enabled robotic exploration application from the server to the robot and obtained the simulation results and the board results.

8.2.1 Simulation Results

In a real case example, the Mars Pathfinder did not function properly because of priority inversion. When JPL engineers discover the bug, a short C program was

uploaded to the robot to solve the problem [15]. We simulated similar scenario where a robot needed a new code to react to a new environment as described below.

In robot exploration, it may be 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 could be sent from the base station. When the robot encounters a new environment, it can download code to modify the robot's behavior. The new code can be dynamically incorporated without reinitializing all functionality of the robot.

In this robot application, we use the block streaming for program file method to transmit the software to the robot. A binary application code of size 10 MB was generated. The stream-enabled application was generated using our softstream code generation tool. The binary image is divided into blocks for streaming. There were three off-block branch instructions on average in each block. The software was streamed over a 128 Kbps transmission media. Table 6 shows average bandwidth overhead of added code per block and load time for different block sizes. The average added code per block is 36 bytes (due to an average in each block of three off-block branches each of which adds 4 bytes, 4 bytes for the start branch number, 4 bytes for the stream-enabled information size, 4 bytes for the code size, 4 bytes for the stream unit ID, 4 bytes for softstream ID, and 4 bytes for flow control). This overhead is insignificant for block sizes larger than 1 KB. The load times were calculated using only transmission of the stream unit. The results in Table 6 do not include

other overhead such as propagation delay (network latency) and processing. If other overhead was included, the load time would be larger.

Table 6: Simulation results for stream-enabled program files.

Block size (bytes)	Total # of blocks	Bandwidth overhead/block	Load time (s)
10M	1	0.0003%	655.36
5M	2	0.0007%	327.68
2M	5	0.0017%	131.07
1M	10	0.0034%	65.54
0.5M	20	0.0069%	32.77
100K	103	0.0352%	6.40
10K	1024	0.3516%	0.64
1K	10240	3.5156%	0.07
512	20480	7.0313%	0.03

Without using the block streaming method, the application load time 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 1 MB or smaller blocks, the deadline could be met. Even if 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 1 MB or less. The application load time for block size of 1 MB is approximately 65 seconds whereas the existing method application load time is more than 655 seconds.

While our sample application is not a full industrial-strength example, it does verify the block streaming functionality and provides experimental data.

8.2.2 MBX860 Board Results

We also ran the software on a MBX860 board to measure the application load times for various block sizes. The results are illustrated in Figure 46. The measurements include all overhead such as processing and network latency. Note the stream blocks

of size 5 MB and 10 MB cannot be loaded into memory on the board since they are too large. We overwrote part of the code which is not used for starting the program. However, the load time measurements are accurate. Using block streaming with stream block size of 1 MB, the robot application can start new code 10X faster than when downloading the entire 1 MB code.

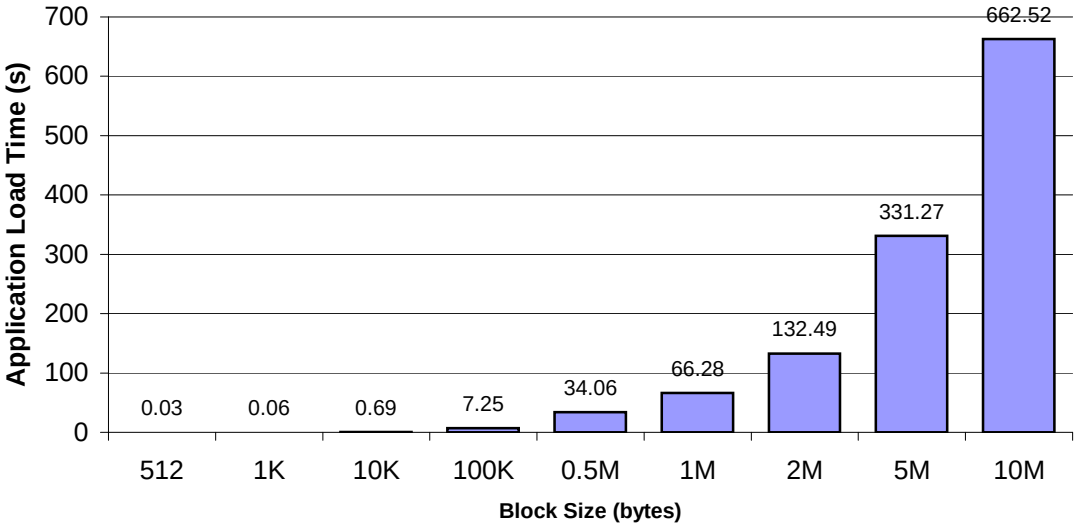

Figure 46: Application load time vs. block size for connection speed of 128 Kbps.

We increased the speed of network connection to 1 Mbps to its effect on application load time. The results are illustrated in Figure 47. When the network connection speed increases to 1 Mbps, the deadline of 120 seconds is met by all the block sizes, including downloading the entire file. However, streaming the application with stream unit size of 1 MB, the application can start running 10X faster than downloading the entire file. What if the robotic had to adapt to the new environment within 60 seconds? Downloading the entire application would cause the robot to miss its deadline by more than 13 seconds while streaming with stream unit size of 1 MB would make the deadline.

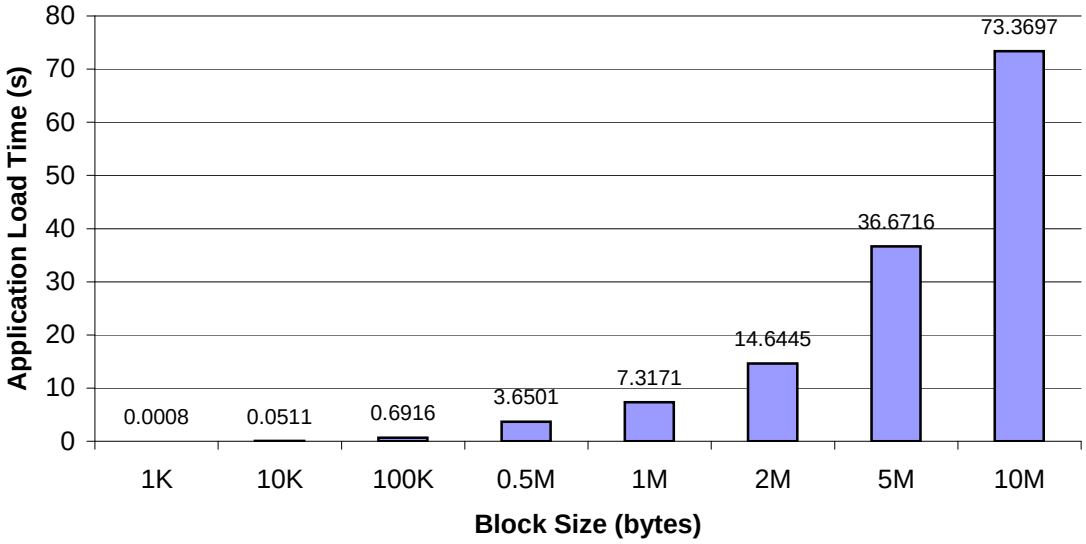

Figure 47: Application load time vs. block size for connection speed of 1 Mbps.

8.3 Stream-Enabled File I/O

We implemented a stream-enabled file I/O method in C. We tested our implementation on an MBX860 board which was set up as described in Section 8.1.2. 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 using NFS version 3 which comes with Linux version 2.4.21 and DD. For DD, we implement a version of DD using Linux TCP/IP 1.0 for NET4.0 socket to download the entire file first and then allow the application to access the data. DD is capable of downloading a file and put it in non-contiguous memory space.

8.3.1 Reading a 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 single 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 the data in a 1 MB data file. This benchmark simulates applications which interleave reading and processing of 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.

The performance comparison of SIO, NFS and DD using these four benchmarks is shown in Figure 48. Figure 48 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 since SIO streams data while NFS sends data upon request. However, SIO performs almost the same as DD since the whole file is transmitted and data processing is minimal. For the 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. DD takes the longest time since the whole file must be downloaded whereas SIO and NFS allow data to be access without obtaining all data. For the Stat benchmark, SIO outperforms NFS and DD. Even though the whole file is needed, SIO allows computations while the file is being transferred.

For the 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 the data file profiling approach described in Section 6.6. Specifically, we use a sequential transmission profile for the Seq, Rand 1K, and Stat benchmarks. The data blocks are sent in the same order as they appear in the file. When the client requests a data block, the server sends the requested block and the blocks after the requested block. However, we used a breadth-first transmission profile for the BSearch benchmark. Data file profiling is used to predict which data block is needed first. Therefore, the server will stream the block according to the transmission profile.

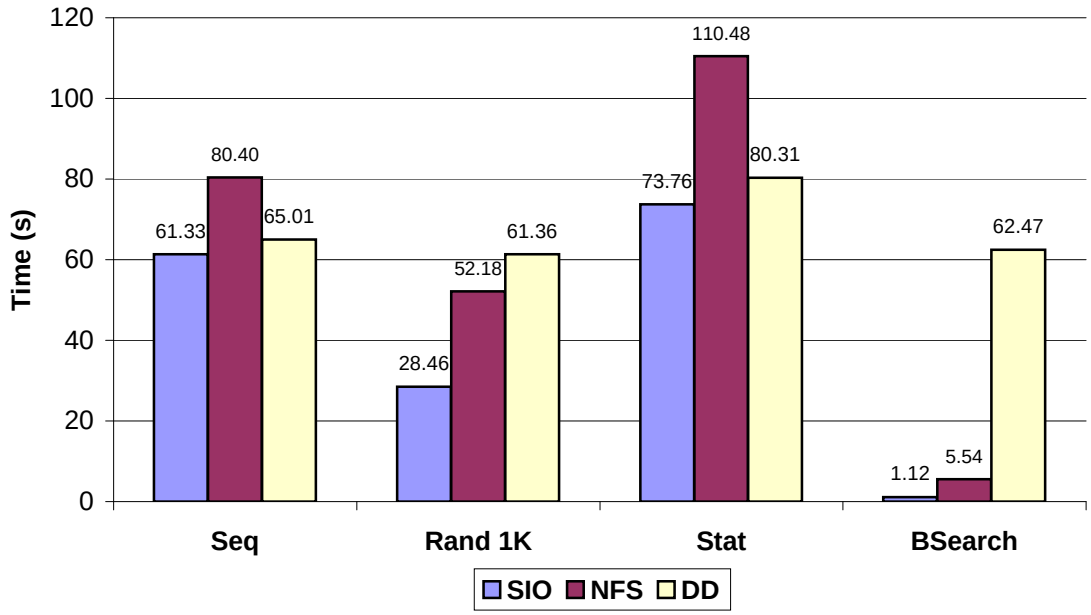

Figure 48: File I/O performance comparisons.

8.3.2 Data Acquisition

In this experiment, we measured the amount of time a game application takes to acquire a certain amount of data from a 1 MB file over a 128 Kbps connection and 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 49.

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

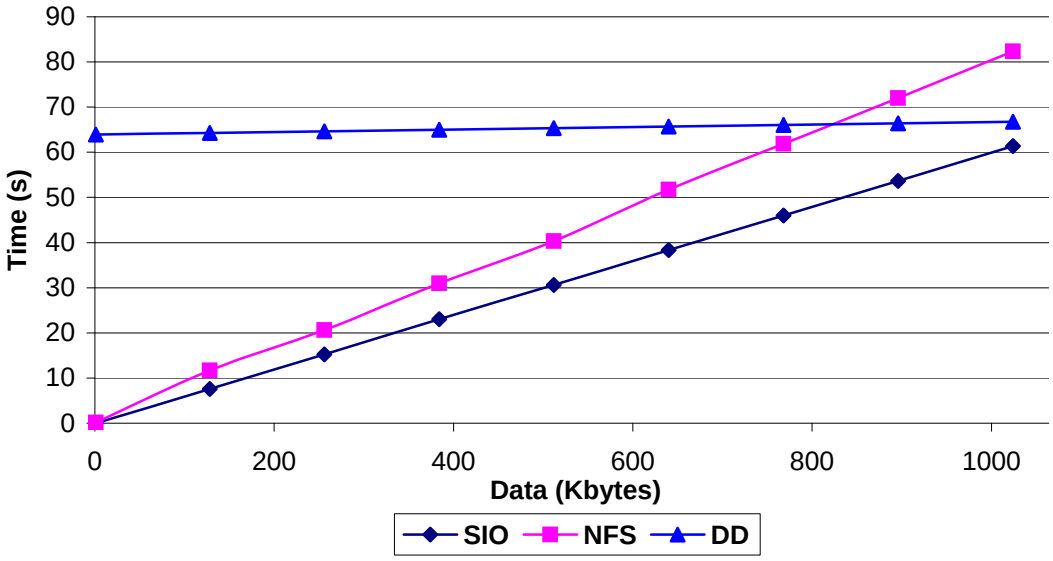

Figure 49: Time to acquire data from a 1 MB file.

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. The primary reason for this difference is NFS has a high overhead for transferring data. The high overhead of NFS is due to the file system support provided.

8.3.3 Data Utilization Rate

Data utilization rate measures how fast data (in KB) is consumed on average (per second) by the client. In this experiment, we measured the amount of time it takes to process a 1 MB file using various data utilization rates over a 128 Kbps connection. The intention of the experiment is to show the effects of the data utilization rate on the amount of time required to process all data when reading a file which must be transferred over a link at a particular connection speed. We used a different data utilization rate to simulate different types of data. For example, encrypted data is consumed at a different rate than unencrypted data since the encrypted data needs to be decrypted before it can be used. The results are illustrated in Figure 50.

The experiment shows that SIO outperforms both NFS and DD. When the data utilization rate is less than the connection data rate, the application spends more time

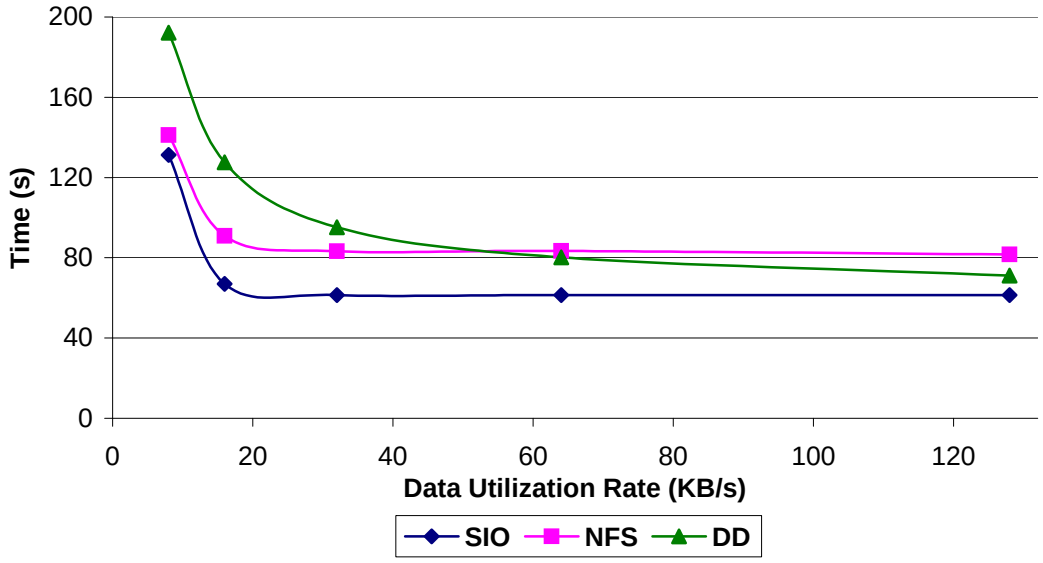

Figure 50: The amount of time it takes to process a 1 MB file with various data utilization rate.

processing the data after all data is transfer to the client. However, when the data utilization rate is greater than the connection data rate (which in our case is 16 KB/s), total time is dominated by transfer time. In this experiment, SIO performs better (i.e., processes the 1 MB file faster) than both NFS and DD for every data utilization rate measured. The main reason for this in the case of NFS is that SIO streams data automatically instead of waiting for a request. The main reason for this in the case of DD is that SIO allows the overlapping of transmission with computation.

8.4 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 running 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 51.

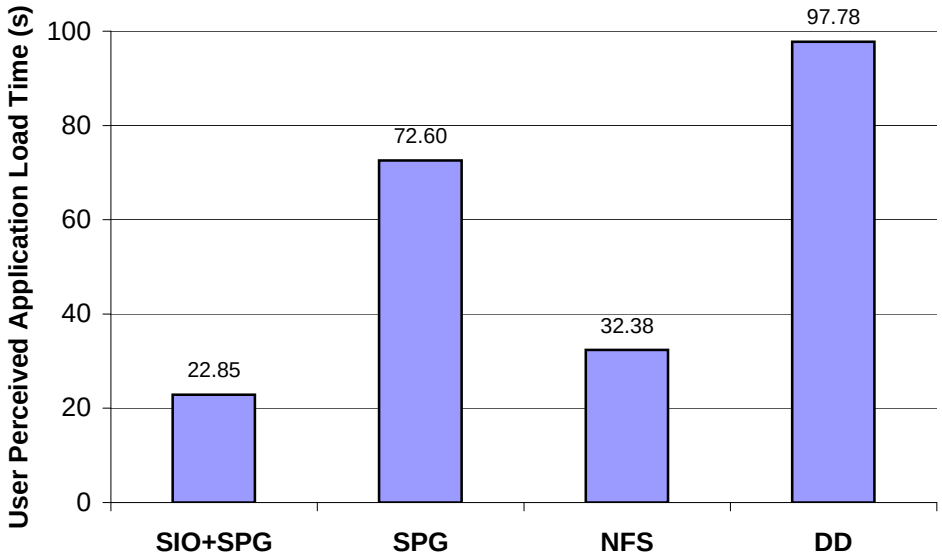

Figure 51: The amount of time the user has to wait before playing the game.

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.

8.5 Summary

Our embedded software streaming implementation reduces the application load time significantly depending on the size of the application and the size of blocks while having minuscule overhead. Our stream-enabled file I/O also performs well for various benchmarks. With the combination of stream-enabled program file and stream-enabled file I/O implementations, a stream-enabled application can achieve a short application load time and a short file I/O latency.

CHAPTER IX

CONCLUSION

Software streaming via block streaming allows a device to start executing an application while the application is being transmitted. We presented a method for transmitting embedded software from a server to be executed on a client device. Our streaming method can lower application load time, bandwidth utilization and memory usage. We verified our streaming method using an MBX860 board a hardware/software co-simulation platform for the PowerPC architecture, specifically for MPC 750 processors. In our experiment, a robotics application that without our streaming method is unable to meet its deadline. However, with our software streaming method, the application is able to meet its deadline. The application load time for the application also improves by a factor of more than 10X for the stream block size of 1 MB when compared to downloading the entire application before running it.

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 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, software streaming enables client devices, especially embedded devices, to support a wide range of applications by efficiently utilizing resources. Software streaming also enables small memory footprint embedded devices to run

applications larger than the physical memory. Using our approach, the user can experience a relatively short application load time. Additionally, our method facilitates software distribution and software updates since software is directly streamed from the server. In case of a bug fix, the software developer can apply a patch at the server. The up-to-date version of the software is always streamed to the client device. Finally, software streaming has the potential to dramatically alter the way software is executed in the embedded setting where minimal application load time is important to clients.

REFERENCES

[1] Abowd, G., Atkeson, C., Bobick, A., Essa, I., MacIntyre, B., Mynatt, E., and Starner, T., "Living laboratories: the future computing environments group at the georgia institute of technology," in *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems*, pp. 215–216, 2000.

[2] Barron, D., *The World of Scripting Languages*. Chichester, NY: Wiley, 2000.

[3] Callaghan, B., *NFS Illustrated*. Reading, MA: Addison-Wesley, 2000.

[4] Campbell, R., *Managing AFS: the Andrew File System*. Upper Saddle River, NJ: Prentice Hall PTR, 1998.

[5] Dwight, J., Erwin, M., and Niles, R., *Using CGI*. Indianapolis, IN: Que, 1997.

[6] Eisenhauer, G., Bustamante, F., and Schwan, K., "A middleware toolkit for client-initiated service specialization," *Operating Systems Review*, vol. 35, no. 2, pp. 7–20, 2001.

[7] Eylon, D., Ramon, A., Volk, Y., Raz, U., and Melamed, S., "Method and system for executing network streamed application," *U.S. Patent Application 20010034736*, Oct. 2001.

[8] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., *Hypertext Transfer Protocol – HTTP/1.1, RFC 2616*. The Internet Engineering Task Force, June 1999. http://www.ietf.org/rfc/rfc2616.txt?number=2616.

[9] Fong, P. and Cameron, R., "Proof linking: modular verification of mobile programs in the presence of lazy, dynamic linking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, pp. 379–409, Oct. 2000.

[10] Franz, M., "Dynamic linking of software components," *Computer*, vol. 30, pp. 74–81, Mar. 1997.

[11] GCC, http://gcc.gnu.org.

[12] Hartman, J., Manber, U., Peterson, L., and Proebsting, T., "Liquid software: a new paradigm for networked systems," Tech. Rep. 96-11, Department of Computer Science, University of Arizona, Tucson, AZ, June 1996.

[13] Herman, D., *UNIX System V NFS Administration*. Englewood Cliffs, NJ: PTR Prentice Hall, 1993.

[14] Huneycutt, C., Fryman, J., and Mackenzie, K., "Software caching using dynamic binary rewriting for embedded devices," in *Proceedings of International Conference on Parallel Processing*, pp. 621–630, 2002.

[15] Jones, M., "What happened on mars?," Dec. 1997. http://www-2.cs.cmu.edu/afs/cs/user/raj/www/mars.html.

[16] Krintz, C., Calder, B., Lee, H., and Zorn, B., "Overlapping execution with transfer using non-strict execution for mobile programs," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 159–169, 1998.

[17] Labrosse, J., *MicroC/OS-II: the real-time kernel*. Lawrence, KS: R & D Publications, 1999.

[18] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*. Reading, MA: Addison-Wesley, second ed., 1999.

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

[20] Mentor Graphics Corp., http://www.mentor.com.

[21] Meyer, J. and Downing, T., *Java Virtual Machine*, pp. 44–45. Cambridge, MA: O'Reilly, 1997.

[22] Motorola, Inc., *Data Sheet: MBX 860*. http://mcg.motorola.com/us/ds/pdf/ds0134.pdf.

[23] Motorola, Inc., Tempe, AZ, *MBX Series Embedded Controller Installation and Use*, MBXA/IH1 ed., July 1997. http://mcg.motorola.com.

[24] Motorola, Inc., Tempe, AZ, *PowerPC Microprocessor Family: the programming environments for 32-bit microprocessors*, MPCFPE32B/AD ed., 1997. http://mcg.motorola.com.

[25] Muthitacharoen, A., Chen, B., and Mazières, D., "A low-bandwidth network file system," in *Proceedings of ACM Symposium on Operating Systems Principles*, pp. 174–187, 2001.

[26] Nahum, E., Barzilai, T., and Kandlur, D., "Performance issues in WWW servers," *IEEE/ACM Transactions on Networking*, vol. 10, pp. 2–11, Feb. 2002.

[27] Pavlidis, T., *Fundamentals of X programming: graphical user interfaces and beyond*. New York, NY: Kluwer Academic, 1999.

[28] Postel, J. and Reynolds, J., *FILE TRANSFER PROTOCOL (FTP), RFC 959*. The Internet Engineering Task Force, Oct. 1985. http://www.ietf.org/rfc/rfc0959.txt?number=959.

[29] Raz, U., Volk, Y., and Melamed, S., "Streaming modules," *U.S. Patent 6,311,221*, Oct. 2001.

[30] Rescorla, E., *SSL and TLS : designing and building secure systems*. Boston, MA: Addison-Wesley, 2001.

[31] Santifaller, M., *TCP/IP and NFS: internetworking in a UNIX environment*. Reading, MA: Addison-Wesley, 1991.

[32] Scheifler, R. and Gettys, J., *X Window System: core and extension protocols: X version 11, releases 6 and 6.1*. Boston, MA: Digital Press, 1997.

[33] 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, Apr. 2003. http://www.ietf.org/rfc/rfc3530.txt?number=3530.

[34] Silberschatz, A., Galvin, P., and Gagne, G., *Applied Operating System Concepts*. New York, NY: John Wiley, first ed., 2000.

[35] Stallings, W., *Operating Systems: internals and design principles*. Upper Saddle River, NJ: Prentice Hall, fourth ed., 2001.

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

[37] Tanenbaum, A., *Operating Systems: design and implementation*. Upper Saddle River, NJ: Prentice Hall, second ed., 1997.

[38] Tobbicke, R., "Distributed file systems: focus on Andrew File System/Distributed File Service (AFS/DFS)," in *Proceedings of IEEE Symposium on Mass Storage Systems*, pp. 23–26, 1994.

[39] Traffic Shaper, Red Hat, http://www.redhat.com.

[40] uClibc, http://www.uclibc.org.

[41] van der Linden, P., *Not just Java: a technology briefing*. Mountainview, CA: Sun Microsystems Press, second ed., 1999.

[42] van der Werff, M., de Grijp, M., Vrind, S., and Haverkort, B., "The X Window System over ISDN–a performance study," in *Proceedings of Teletraffic Symposium*, pp. 1/1–1/7, 1993.

[43] Venkitachalam, G. and cker Chiueh, T., "High performance Common Gateway Interface invocation," in *Proceedings of IEEE Workshop on Internet Applications*, pp. 4–11, 1999.

[44] Venners, B., *Inside the Java Virtual Machine*. New York, NY: McGraw-Hill, 1998.

[45] Wang, P., *Java with Object-Oriented Programming and World Wide Web Application*, pp. 193–194. Pacific Grove, CA: Brooks/Cole Pub., 1999.

[46] X.Org Foundation, http://www.x.org.

PUBLICATIONS

This dissertation is based on and extends the work and results presented in the following publications:

[1] Kuacharoen, P. and Mooney, V., "Block streaming for embedded systems," to be published in *Proceedings of the Mobility Conference & Exhibition*, Aug. 2004.

[2] Kuacharoen, P., Mooney, V., and Madisetti, V., "Software streaming via block streaming," in the book *Embedded Software for SoC*, edited by Jerraya, A., Yoo, S., Verkest, D. and Wehn, N., Boston, MA: Kluwer Academic Publishers, pp. 435–448, Sep. 2003.

[3] Kuacharoen, P., Mooney, V., and Madisetti, V., "Software streaming via block streaming," in *Proceedings of the Design Automation and Test in Europe*, pp. 912–917, Mar. 2003.

[4] Kuacharoen, P., Mooney, V., and Madisetti, V., "Methods and systems for transmitting application software," *U.S. Patent Application 20040006637*, Jan. 2004.

The following publications are related but not covered in this dissertation:

[1] Akgul, B., Mooney, V., Thane, H., and Kuacharoen, P., "Hardware Support for Priority Inheritance," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 246-254, Dec. 2003.

[2] Kuacharoen, P., Shalan, M., and Mooney, V., "A configurable hardware scheduler for real-time systems," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 96–101, June 2003.

[3] Kuacharoen, P., Akgul, T., Mooney, V., and Madisetti, V., "Adaptability, extensibility, and flexibility in real-time operating systems," in *Proceedings of the EUROMICRO Symposium on Digital Systems Design*, pp. 400–405, Sep. 2001.

[4] Akgul, T., Kuacharoen, P., Mooney, V., and Madisetti, V., "A debugger RTOS for embedded systems," in *Proceedings of the 27th EUROMICRO Conference*, pp. 264–269, Sep. 2001.

[5] Kuacharoen, P., Akgul, T., Mooney, V., and Madisetti, V., "Dynamic operating system," *U.S. Patent Application 20030074487*, Apr. 2003.

[6] Akgul, T., Kuacharoen, P., Mooney, V., and Madisetti, V., "Debugger operating system for embedded systems," *U.S. Patent Application 20030074650*, Apr. 2003.