EFFICIENT EXECUTION OF LARGE APPLICATIONS ON PORTABLE AND WIRELESS CLIENTS

PRAMOTE KUACHAROEN*
School of Applied Statistics, National Institute of Development Administration
Bangkapi District, Bangkok, 10240, Thailand

VINCENT J. MOONEY III
Associate Professor, School of Electrical and Computer Engineering
Adjunct Associate Professor, College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA

VIJAY K. MADISETTI
Professor, School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA

Wireless and embedded portable devices, such as cell phones and PDAs, place a premium on storage and communications bandwidth. The communications channel itself is prone to outages as well. However, users are expecting much more capability from these devices, including the ability to run business applications (e.g., Oracle), play video games, and also to perform a variety of business functions. We propose, design, and show results of new technology, called block-streaming, that allows large (in code size) applications to run effectively on wireless and portable devices in memory and bandwidth constrained modes. This technology allows software applications to execute correctly but in a smaller footprint, and interestingly enough with a higher degree of user satisfaction, due to minimization of delays and retransmissions.

INTRODUCTION

As the availability and use of computing resources become more and more ubiquitous, a scenario where a user utilizes a portable device to download applications from remote servers and executes the downloaded applications is likely to become quite common. Today, such a user would typically have to wait a long time to execute a cutting-edge application which he/she had selected for the first time. This is a problem as users also demand small embedded devices — such as cellular phones and personal digital assistants which have limited resources — to run many applications concurrently. With

* This work was performed when the author was in the Ph.D. program at the Georgia Institute of Technology.

limited storage resources on the device, keeping all features of all applications loaded in memory may not be possible.

A long wait time may be overcome by using a software streaming method [1], [2], [3], [4] which allows the execution of stream-enabled software on a device even while the transmission of software may still be in progress. In other words, the software can be executed while it is being streamed instead of requiring the user to wait for the completion of the entire software's download. We introduced a software streaming method called "block streaming" in [2]. Block streaming reduces application load time (the amount of time from when the application is selected to download to when the application can be executed). Block streaming can also reduce bandwidth utilization and memory usage since unneeded software code may not be sent to the client devices. However, our initial work does not address the situation where client memory is not sufficient to store all needed code. Furthermore, a large potion of the application code may be executed only once. This application code can be removed from memory to allow needed code and/or data to be streamed to the client device. Therefore, we present a novel method to manage client memory.

We apply two techniques, namely, code transformation and stream unit removal to allow an application which is larger than the available memory to be executed as described in the following two sections.

CODE TRANSFORMATION

In [2], we present 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 (e.g., memory) available to the program. However, in this section, we introduce techniques which may improve performance and may reduce resource usage by statically transforming the program binary image.

Determine Function Boundaries

One drawback to dividing a binary image into equally-sized 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 function 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 1 shows that occurrence of application suspensions is reduced when the block boundary is match closely with function boundary.

Example 1: Figure 1(a) shows that function fn2() is split with part of the function in the first block (2 instructions) and the rest 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 1(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 1(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. □

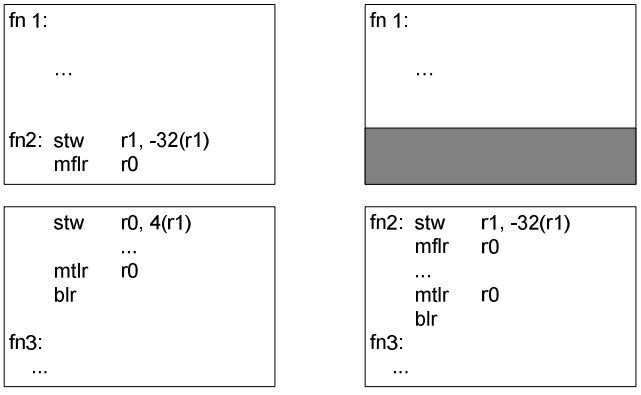

Figure 1. Enforcing block boundaries. (a) A function is placed into separate blocks. (b) The block boundaries are matched with function boundaries.

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 is in the same order as the original

source code. After generating blocks for streaming, the order of function placement remains the same. Example 2 shows how the lack of spatial locality of reference degrades stream-enabled software.

Example 2: Suppose that a program file is divided into three blocks as shown in Figure 2. The functions are in the same order as they were written. The function fn1() calls fn5(), and the function fn5() calls fn7(). These functions 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. □

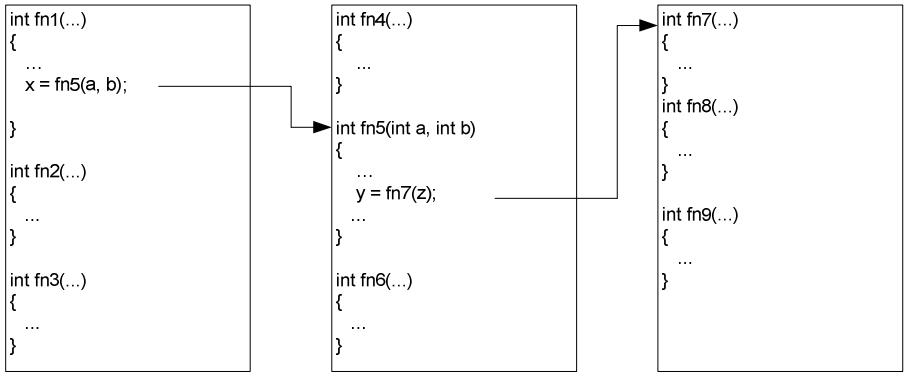

Figure 2. An example shows the program lacks block locality.

In Example 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 [6]. 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 are 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.

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.

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 3 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 3: Suppose that the client has to deallocate Block 2 in Figure 3(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 3(b) shows Block 1 after Block 2 is removed. The second instruction of Block 1 bne load2_1 is change to load Block 2. □

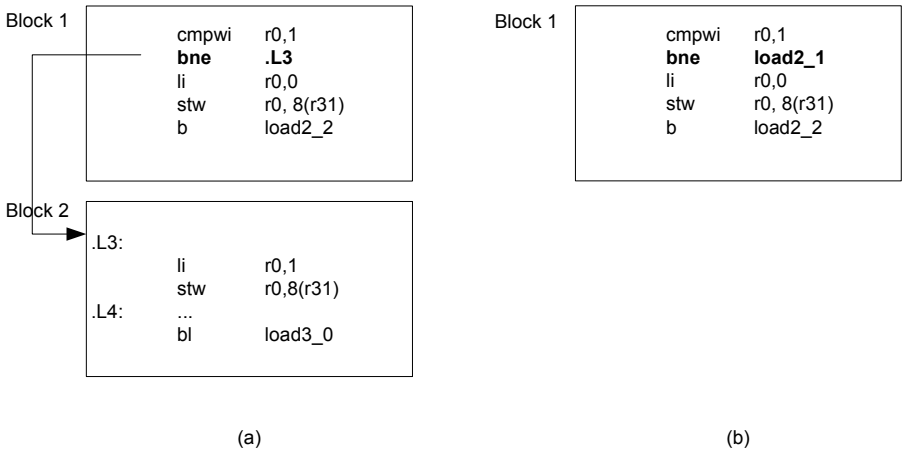

Figure 3. Unlinking. (a) Block 1 and Block 2 are linked together. (b) Block 2 is unlinked from Block 1.

To unlink a stream block, one needs to know all incoming off-block branches (branch instructions that may cause the CPU to execute instructions in different code blocks) to the block to be removed. Therefore, the additional off-branch information includes the number of incoming off-block branches and the branch numbers. 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.

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 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 [5]. 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. 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 4 illustrates the replacement of stream units.

Example 4: Figure 4 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. □

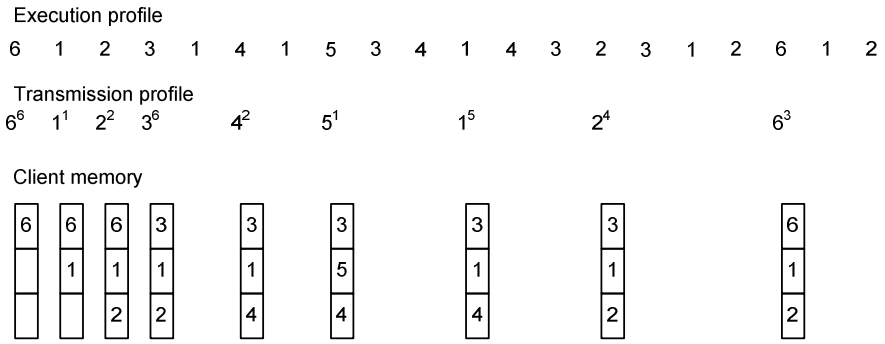

Figure 4. A transmission profile is created according to the minimum retransmission policy.

EXPERIMENTS AND RESULTS

We simulated a scenario where the user utilizes a portable device to download and play a game from a server. We assumed that the program size of 4 MB and the client has only 1 MB available memory. We divided the game into 256 16 KB blocks and the client memory into 64 16 KB blocks. We compared results from block streaming and demand loading. In demand loading, a block is sent when it is needed.

Table 1 shows the number of blocks transmitted and occurrences of application suspension for demanding loading and block streaming in a typical execution path of the game application. In this scenario, block streaming can potentially reduce the occurrence of application suspensions by 67.85%.

Table 1. Number of blocks transmitted and occurrences of application suspension for demanding loading and block streaming.

	Number of blocks transmitted	Occurrences of application suspension
Demand Loading	336	336
Block Streaming	336	108

CONCLUSION

Block streaming enables small memory foot print embedded devices to support applications larger than the available memory while reducing the occurrence of application suspensions. Our simulation shows that block streaming can potentially reduce the occurrence of application suspensions by 67.85% when compared with demand loading.

REFERENCES

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

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

[3] Lindholm, T. and Yellin, F., "The Java Virtual Machine Specification," 2nd edition, Reading, MA: Addision-Wesley, 1999.

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

[5] Silberschatz, A., Galvin, P., and Gagne, G., "*Applied Operating System Concepts,*" 1st edition, New York, NY: John Wiley, 2000.

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