Access Pattern Based Stream Buffer Management Scheme for Portable Media Players

Hee Jin Ahn, Seongjin Cho, Hyunik Na, and Hwansoo Han

Abstract — Limited amount of main memories are often available on mobile phones and portable media players. In multitasking environments, however, programs often need to wait for pages to be loaded to the main memory, even for the pages once loaded but swapped out due to memory shortage. Such page fault mechanism provided by the operating system often becomes the main hindrance to the uninterrupted playback of multimedia players on portable electronic devices. In this paper, we propose a memory management scheme that helps multimedia players perform with a consistent quality, when other processes are concurrently running on memory-limited environments. We devise a profile-based memory allocation scheme for multimedia stream buffers with the adoption of page prefetching and pinning. Using periodic monitoring points of main loops, we profile access patterns of memory, analyze the meaningful access patterns for stream buffers and provide the access pattern information for custom memory management. Our scheme helped reduce the number of page faults in heap. Reductions in page faults for a multimedia player are 60–80% under various memory-limited execution environments.

Index Terms — access pattern-based memory management, stream buffer access pattern, portable media player.

I. INTRODUCTION

Memory-limited embedded systems, such as mobile phones and portable media players, are now very widely used. Despite their small memory size, we often need to run multiple applications concurrently. Virtual memory provides an application program with the illusion that it has a large contiguous memory, while in fact the memory is physically fragmented and may even overflow out to disk storage. Virtual memory enables several programs to run concurrently even if the total amount of memory needed is greater than the physical memory available. In multitasking environments, programs often need to wait for pages to be loaded on the main memory, particularly for the pages which were on the main memory once but later swapped out due to memory shortage. This page fault handling is often the main hindrance to seamless execution.

Multimedia players are quite sensitive to page fault behaviors and even small paging delays may severely disrupt their playback function. Seamless execution is the most important factor that needs to be satisfied in multimedia players. In multitasking environments with a limited memory, however, it is hard to guarantee smooth playback. In this paper, we propose a scheme that helps multimedia players perform better and execute with consistent quality over other programs.

We occasionally need to use the same input many times for a multimedia player, when we play favorite video clips multiple times. Our scheme is particularly helpful in this situation. We profile a program with an input for the first time and apply our optimized memory management based on the memory access pattern for the subsequent playbacks. A simple-minded approach is to record the address sequence of all the accessed pages and bring them into memory before they get needed. For multimedia players, stream buffers are dynamically allocated according to the changing timing constraints from run to run. Thus, different address traces are generated even if we run the same program twice with the same input.

In this paper we introduce the notion of stream, memory allocations with the same backtrace of function calls and memory accesses to the allocated memory. For the same input, the traces of memory addresses may be different, but the traces of memory offsets within a stream are similar. We analyze profiled stream access patterns, and apply our new memory allocation and management scheme with prefetching and pinning at runtime according to the analyzed pattern information. Our method helped reduce the number of page faults in heap, and provided better execution quality for multimedia players under memory-limited environment.

II. MONITORING POINTS FOR REAL-TIME CONSTRAINTS IN MULTIMEDIA PLAYERS

Multimedia players are required to execute evenly over the time to produce quality playback. Suppose, for example, a real-time application spends most of its execution time in a loop which processes the main task of the application. The real-time constraint of the application can be represented as the maximum limit of the execution time for an iteration of the loop, that is, the period of the loop. We will refer to this kind of loop as real-time critical loop. For most multimedia players, the real-time critical loop would be the loop that decodes and displays each frame and the real-time constraint can be represented with the execution time of one real-time critical loop iteration.
Actually, there can be many loops with real-time constraints in a program. This motivates us to define the term real-time critical loop more strictly. Fig. 1 depicts the properties of real-time critical loops listed in the following.

- A real-time critical loop has relatively many iterations. We cannot place a definite numerical restriction on the number of iterations, but a real-time critical loop is supposed to be used as a unit to examine paging behavior, thus needs to be reasonably fine-grained. Outermost loops with just a couple of iterations need to be excluded.
- A real-time critical loop is executed throughout the whole program run. As mentioned above, the loop is used as a unit of program examination, thus always needs to be in execution while the program is running.
- The time interval between iterations of a real-time critical loop must be relatively uniform.
- There is only one critical loop in a program. Even if there are more than two loops satisfying three conditions above, we need to specify only one of them for analysis purposes.

Monitoring points are inserted in the loop head of the real-time critical loop and used as units of monitoring and analyzing paging behavior for the program. We can assign various functionalities to monitoring points. A monitoring point performs a specific task depending on the kind of analysis or optimization. For the simplest example, a monitoring point can record elapsed time during program execution, to see if the program satisfies its real-time constraints.

III. Monitoring Paging Behavior

There can be various reasons why a program could not satisfy its real-time constraints, but the most probable situation would be page faults. In such case, the incurring disk I/O delays the access to fetch the needed pages from the disk. Our page swap monitor analyzes paging behavior of real-time applications at monitoring points. Page swap monitor can help answer these kinds of questions:

- Does the program satisfy its real-time constraints? If not, what is the reason?
- Which processes are created and terminated? How many page frames did they have in memory in the meantime?
- How many page faults and page swap-outs occurred? Where did they occur in the address space of the program?

The main module of page swap monitor is a kernel logging module inside the operating system kernel. Currently, we support x86 and ARM architectures.

A. Kernel Modification for Paging Behavior Log

To get information about page-ins/outs of processes, we need to know when the processes are created and terminated. Thus, we leave logs at each stage of process life cycle. When several processes share their memory area, they share their virtual memory structure, so their paging behavior must be considered together. Threads are common example of such case, since most of modern operating systems implement a thread with a light-weight process. We regard the group of processes sharing their memory as one process, and leave logs of their paging behavior as a whole. Only the creation of the first process and the termination of the last process in the group are logged.

Since many processes may share a virtual memory structure, operating system kernel maintains the links from process descriptors to virtual memory structures, but the reverse links do not exist. Our page swap monitor, however, needs to find out the corresponding process when a page fault occurs. Thus, we added the reverse links from virtual memory structure to process descriptor so that we can refer to process descriptors from virtual memory structure.

The page fault handler checks if page fault address belongs to the process address space by looking up the virtual memory structure. If the memory region access rights match the access type that caused the exception, the page fault handler is invoked to allocate a new page frame. The page fault handler indicates major fault if it succeeded in allocating a page frame with appropriate disk I/O or it indicates minor fault if the page fault has been handled without disk I/O. When a page fault occurs, we can refer to the process virtual address where the page fault is generated, and we obtain the type of page fault from the return value of the page fault handler. From the process descriptor and virtual memory structure, our page swap monitor can locate the address where the fault was generated.

There are several cases why a page is swapped out and its page frame is reclaimed. The two main reasons for memory reclamation are low-on-memory reclamation, which is activated when a memory allocation fails, and periodic reclamation, which is activated by kernel swap daemon to reserve free memory for later use. When the kernel tries to reclaim a page, it first unmaps the page from the page table and checks if the page can be actually forced out to disk. When reclaiming a page, to the kernel unmaps the page mapping in the page table, and here we can get the virtual address of the page, and the process descriptors it belongs to. Then the page may or may not be swapped out to disk after several checks are applied. Thus when a page table mapping is cleared, the page may or may not be swapped out to disk. Page swap monitor detects both cases, and leaves a different log message for each of them.
B. Monitoring Points Implementation

For our page behavior monitoring, we must specify one focused process. The focused process is the process we want to examine whether it satisfies its real-time constraints. For a focused process, page swap monitor assumes that a monitoring point is inserted at the head of the real-time critical loop as shown in Fig. 2. Actual implementation of monitoring point is a system call, \( MP() \), which we implemented for the purpose of kernel information logging. Each time a monitoring point is called, our page swap monitor recognizes it as the progress of the focused process. When this system call is invoked, kernel leaves a log of the elapsed time of the focused process and the kernel information of other concurrently running processes. Later, we can inspect if the focused process satisfies its real-time constraints by using these logged messages.

Since we need to leave logs in various parts of the operating system kernel, we implemented \textit{Pagelog} and \textit{Klog} within the kernel, which enables the kernel to leave log messages in a convenient and efficient way. The logging function of Pagelog is invoked in various places of the kernel. Then, Pagelog categorizes these logs, formats message strings, and passes the message strings to Klog. By printing log messages to a file within a separate kernel thread, Klog performs the actual logging. When Pagelog starts, Pagelog scans the list of all processes and adds the processes occupying more than 1% of the whole memory into the list of interested processes. For the focused process, we leave detailed log messages of its paging behavior. For the page fault of the focused process, we record the memory address, type of memory region the fault address belongs to (e.g., code, data, heap, or stack), and the page fault type (e.g., major or minor). For other interested processes, we just leave one summarized log message for several consecutive page faults.

For the monitoring point of the focused process, we leave a log message of the elapsed time and monitoring point number, along with the number of pages in memory of each process in the list of interested processes. Starting from the second line of the format, the number of pages in the main memory for each interested process is written. The number of lines depends on the number of the interested processes. The detailed format of log messages are summarized in Table I.

For our page behavior monitoring, we must specify one focused process. The focused process is the process we want to examine whether it satisfies its real-time constraints. For a focused process, page swap monitor assumes that a monitoring point is inserted at the head of the real-time critical loop as shown in Fig. 2. Actual implementation of monitoring point is a system call, \( MP() \), which we implemented for the purpose of kernel information logging. Each time a monitoring point is called, our page swap monitor recognizes it as the progress of the focused process. When this system call is invoked, kernel leaves a log of the elapsed time of the focused process and the kernel information of other concurrently running processes. Later, we can inspect if the focused process satisfies its real-time constraints by using these logged messages.

Since we need to leave logs in various parts of the operating system kernel, we implemented \textit{Pagelog} and \textit{Klog} within the kernel, which enables the kernel to leave log messages in a convenient and efficient way. The logging function of Pagelog is invoked in various places of the kernel. Then, Pagelog categorizes these logs, formats message strings, and passes the message strings to Klog. By printing log messages to a file within a separate kernel thread, Klog performs the actual logging. When Pagelog starts, Pagelog scans the list of all processes and adds the processes occupying more than 1% of the whole memory into the list of interested processes. For the focused process, we leave detailed log messages of its paging behavior. For the page fault of the focused process, we record the memory address, type of memory region the fault address belongs to (e.g., code, data, heap, or stack), and the page fault type (e.g., major or minor). For other interested processes, we just leave one summarized log message for several consecutive page faults.

For the monitoring point of the focused process, we leave a log message of the elapsed time and monitoring point number, along with the number of pages in memory of each process in the list of interested processes. Starting from the second line of the format, the number of pages in the main memory for each interested process is written. The number of lines depends on the number of the interested processes. The detailed format of log messages are summarized in Table I.

The operating system kernel uses an internal routine to log various kinds of debug messages or information. Unfortunately, some messages can be skipped, when a large amount of messages are delivered at once. Because the internal log function uses a circular buffer to store the messages, messages can be skipped, or even worse, can be truncated in the middle. Thus, our Klog enables reliable logging of paging events. Klog tries not to skip messages if possible, and when it cannot, records the number of skipped messages for later reference. Since Klog prints a whole log message at once, messages are not truncated in the middle. Moreover, Klog messages can be written to a separate file, unlike the existing internal log function.

When paging events occur, Pagelog writes log messages to the circular buffer of Klog. The disk writing thread of Klog starts when kernel boots, waits for log messages to be written in its buffer, and writes them to disk when messages are written. To solve the problem of the existing log function stated above, Klog checks if there is enough space when it writes a message to the buffer. Otherwise, Klog just skips the message and updates the statistics of skipped messages.

IV. PROFILING MEMORY ALLOCATIONS AND ACCESSES

In this section, we present the concept \textit{memory stream}, which is a collection of memory allocations that are used for the same purpose within a program. We also discuss how we profiled and analyzed memory access patterns of each stream. Based on the pattern information of each stream, we propose two new memory management schemes. The first scheme is the stream-based memory allocation, which enables streams to use memory more efficiently. The second scheme is the memory prefetching and pinning based on the stream pattern information. For the purpose of the pattern analysis and our new memory management schemes, we exploit the information we gather from the monitoring points in real-time critical loops. The whole process of our memory management scheme is illustrated in Fig. 3.
Multimedia applications extensively use the heap area, and perform active page swap-ins and swap-outs in the heap area when there is not enough memory. One of the reasons for this massive heap usage is that they use buffering to encode or decode various formats of data. For example, in a movie player, to make movies play in the screen smoothly, the player needs to decode a few frames ahead and store them in buffers. Hereafter, we will use the term buffer as a region of memory allocated in heap for temporary usage. In memory-limited multitasking environments, the majority of page faults are generated from heap area. To reduce page faults, prefetching buffers in the same stream are allocated sequentially (i.e. the \( k \)-th buffer in address), the behavior of \texttt{malloc()}, \texttt{realloc()}, and \texttt{free()} by specifying appropriate hook functions. We define our own memory allocation functions where we leave a log message, which is a pair of the backtrace of the memory management function and the memory address. In the analysis phase later, we group memory allocations by the backtrace of the current function call sequence. The depth of the backtrace can be fairly deep, so we only use a few last traces. According to our experiment, the access pattern of a stream does not change much, even if we increase the depth of backtrace from 3 to 30. For the experimental results we will present in the later section, we used 7 as the depth of backtrace.

We also profile memory accesses during monitoring point intervals. At each monitoring point, we disable memory accesses to the entire heap area. During the execution of loop body, segmentation faults occur whenever a page of the heap area is first accessed. By using our own page fault handler, we leave a log message for the profile of the accessed page address. In this way, we can obtain the set of accessed pages per each monitoring point interval.

V. ACCESS PATTERN-BASED MEMORY MANAGEMENT

To discover the access pattern of each stream, we implemented our pattern analyzer. The pattern analyzer first parses the profile data obtained from the profile and groups the memory accesses by the stream. To determine which stream a given memory access belongs to, our pattern analyzer maintains stream-specific information such as all the buffers allocated and freed within each stream. The final step of the pattern analyzer is to classify streams into two patterns.

For the purpose of pattern analysis, we use the notion of offset rather than address. Offset is, assuming all buffers in a stream are allocated sequentially (i.e. the \( k \)-th buffer in the stream is followed by the \( k+1 \)-th buffer in address), the
memory address within the series of sequential buffers, provided that the start address of the first buffer is zero. An example of stream and offset is depicted in Fig. 4. Offsets are more valuable characteristics than addresses. While addresses are not fixed among multiple runs of a program even with the same input, offsets are relatively fixed over multiple runs. Thus, our pattern analyzer performs its analysis on the offsets, not the real addresses.

**Fig. 4.** Stream and offset: multiple buffers allocated in a scattered address space compose a stream. The offset of the stream is the byte address from the beginning of the stream, when buffers are consecutively placed.

![Diagram of Stream and Offset](image)

**A. Two Types of Stream Access Patterns**

For some streams, a few buffers are allocated at the start of the program and they are constantly accessed over the all monitoring point intervals. The buffers, which belong to such streams, show almost the same accessed offsets throughout the execution as shown in Fig. 5. We refer to this kind of pattern as constant access pattern. To detect constant access pattern, we compute the ratio of overlapping pages to all accessed pages in adjacent monitoring point intervals (MPs). The higher the ratio is, the higher the probability of constant access pattern becomes. We may compute the ratio of overlapping pages within k adjacent monitoring intervals, loosening the criterion for the constant access pattern. For a constant pattern stream, we store only the backtrace of memory allocation and the type of the stream pattern denoting that this stream is a constant pattern stream. For constant pattern streams, we pin their buffers as soon as they are allocated.

Other streams, where accessed offsets vary as execution proceeds, are classified as interval access pattern. Fig. 6 shows an example of such stream. If such streams are frequently accessed, we collect the accessed offset ranges for each monitoring interval and summarize them as offset intervals of accessed pages. These pieces of the access pattern information are later used for prefetching at run time.

**B. Memory Allocation Scheme**

We implement a new set of dynamic memory management functions to achieve better performance for streams by exploiting their access pattern information. From the profile data, we select only the frequently accessed streams and apply our new memory management functions for them. When `malloc()` is called, we intercept the call by using `malloc_hook` mechanism and inspects its backtrace of function call to find out which stream it belongs to. If it belongs to frequently accessed streams, we allocate memory via our new memory allocation function. Otherwise, the original `malloc()` handles the allocation. Other memory management functions are also handled in a similar way.

Inside the new allocation function, allocated buffers for each stream is separately managed. When we allocate the first buffer in a stream, we map the minimum number of pages that are needed to serve the requested buffer size. From the second allocation in the same stream, we allocate the requested buffer from the free space of the last mapped pages. New pages are mapped and appended only when the free space of the stream is not enough. In case the newly appended pages are adjacent to the last page of the stream, we coalesces two chunks of pages to increase the efficiency of page maintenance. If the allocated chunks of memories for a stream are fragmented too much, many wasted space will exist at the ends of the chunks. To minimize this wasted memory, we map the first buffer of each stream sufficiently separated from other streams so that a stream can occupy continuous range of address space as much as possible.

**Fig. 6.** Interval access pattern: accessed offsets in a stream vary from a monitoring point interval to another. Each accessed offset in an interval is represented with offset interval of accessed pages.

![Diagram of Interval Access Pattern](image)
We use a slightly different memory allocation scheme for the streams with constant access pattern. For these streams, a few buffers (mostly one buffer) are allocated at the start of the program and accessed frequently during the whole execution without being freed. Considering this access pattern, we can pin the corresponding pages of these buffers during the whole execution and prevent these pages from being swapped out to disk. Moreover, we treat all the constant access streams as one big stream and consecutively allocate buffers for them to avoid the wasted residual memory at the end of the last page of each constant access stream. Since the buffers are usually allocated at the beginning of the program for these streams, we know the occupied size of each stream in advance. Thus, we can even allocate them at once.

C. Page Prefetching and Pinning Scheme

Based on access pattern information, we can apply page prefetching and pinning to reduce the page faults of multimedia applications. Page prefetching and pinning scheme is implemented with two modules: page access advisor and page prefetching/pinning module, which are shown in Fig. 3. The page access advisor module controls which pages to prefetch, pin, and unpin at run time. All the decisions are made according to the page access pattern of streams. At profiling phase, we only keep the access patterns for frequently accessed streams. Thus, the page access advisor only affects on the pages of those frequently accesses streams. Frequent accessed streams are selected by using such criterion that the proportion of the monitoring point intervals where the stream is accessed among all the monitoring point intervals is higher than a predefined threshold. When a program begins its execution, the page access advisor reads the access pattern information for streams and gives instructions to the page prefetching/pinning module at each monitoring point.

For streams with constant access pattern, we can just pin the corresponding pages of buffers as soon as they are allocated. Thus, page access advisor does not take any action for them except initial page pinning. For streams with interval access pattern, we use the information of the offset intervals for accessed pages at each monitoring point interval to guide the page prefetching/pinning module. For each monitoring interval, the page access advisor computes the in-list, which lists up the pages that are newly needed, and the out-list, which lists up the pages that are not needed any more.

The page prefetching/pinning module reads ahead the needed pages to avoid page faults. It also pins the prefetched pages to prevent them from being swapped out to disk. Prefetching and pinning are always performed one after another, which means that the prefetched pages are always pinned. For the pages that are no more accessed, page unpinning is performed to allow those pages to be swapped out to disk. At every monitoring point, the module requests the page access advisor for the in-list two monitoring point intervals ahead. Since it takes time to read the pages from the disk into the main memory, we start prefetching a few monitoring point intervals ahead. In our experiment, we prefetch two monitoring point intervals ahead, but this can be tuned to achieve better performance on different environments. Then the prefetched pages are pinned until the pages appear on the out-list. When they are listed in the out-list, we unpin those pages to allow them to be swapped out when necessary.

For the purpose of both prefetching and pinning, we use mlock() system call. This system call locks the specified address space in main memory, preventing that memory from being paged out to the disk swap area. If the specified pages are not resident in the main memory, the system call reads the corresponding pages into the main memory and locks them in the main memory. Since mlock() blocks the current thread until the pages are all loaded, we perform this prefetching and pinning operation in a separate thread. To unpin pages, we use munlock() system call. This system call has the opposite effect, unlocking pages in the specified address range. After unpinning system call, all the pages in the specified memory range can be moved out to the external swap space.

In each of these processes, the page access advisor provides the page access information in terms of stream offset. Thus, the given stream offsets need to be translated to the page addresses in the virtual address space. When we allocate buffers for a stream, we keep a mapping table for virtual address lookup, which is sorted by stream offset. This map enables a fast lookup of the metadata for allocated buffers and their page addresses can be quickly found as a result.

VI. EXPERIMENTAL RESULTS

We performed our experiments on a modified kernel, which is extended with our page swap monitor to measure various paging statistics. When we boot the machine, we set the total main memory to 100MB in order to simulate a memory-limited environment for portable media players. The multimedia player we used in our experiments is a movie player called MPlayer version 1.0rc1 [11]. MPlayer has a frame-drop option so that the player is allowed to skip a few frames to maintain A/V sync on slow systems. Since we profile memory accesses based on complete evaluation of all frames, we chose not to use the -framedrop option. The playing time of the sample movie is around one minute. We use MPEG 2/4 for the audio codec and H.264 for the video codec. Compared to heap area, code and read-only data are small but frequently accessed. Thus, we pin those areas for the real-time constraints. To simulate memory-limited multitasking environments, we concurrently execute a memory scanning program along with the movie player. The memory scanning program repeatedly sweeps through the preset size of memory to further limit the available memory for MPlayer.

With two concurrently running programs, we measured the number of page faults on code, data, heap, and stack segments. We performed our experiment with varying the size of memory scan from 60MB to 75MB to investigate how our pattern-based stream buffer management performs under a
tight memory condition. Fig. 7 shows the numbers of page faults occurred from MPlayer. The results presented here are the averages of 10 runs for each memory scan size. Page faults occurred mostly on heap, since we pinned the pages of code and read-only data, which are most frequently accessed ones. From the 65MB scan size and up, the original MPlayer begins to suffer from increasing number of page faults. Meanwhile, the MPlayer, which uses our pattern-based stream buffer management, does not suffer too much from page faults. The amount of the page faults are reduced by 60~80% for the scan sizes of 65MB and up.

Fig. 7. Page faults of MPlayer under a tight memory condition. Numbers of page faults are separately measured for stack and the others with a concurrently running memory scan program (60~75 MB scan sizes)

Another measure for the quality of movie player execution is the playing time, when no frame is dropped. Since we run MPlayer without the -framedrop option, MPlayer decodes all the frames even if it fails to meet the real time constraints. Fig. 8 shows the playing times of MPlayer with different scan sizes of 60~75MB for the memory scanning program. Our pattern-based stream buffer management has a slight overhead to prefetch, pin, and unpin pages every iteration of the real time critical loop. The playing time with 65MB scan size shows almost equal playing time to the original MPlayer, even though the number page faults is reduced. For 70~75MB scan sizes, our pattern-based stream buffer management provides better execution environment for MPlayer by greatly reducing the page faults. The playing time close to the original playing time implies that MPlayer will have a better chance to decode more frames when the frame drop option is turned on. Consequently, the quality of movie play will be better than the original MPlayer.

VII. RELATED WORK

There has been a large body of literature devoted to reading data ahead before they are actually needed to hide disk I/O delay. File prefetching is a main technique by predicting upcoming file accesses based on previous history. Some of the works used data compression method to prefetch data into cache [1], [2]. Curewitz, Krishana, and Vitter [1], targeting OODB system, suggested a prefetcher based on the Lempel-Ziv data compressor, while Kroeger and Long [2] used Prediction by Partial Match (PPM) method to track sequences of file access events. Griffieon and Appleton [3] used the probability graph to tackle the problem. Later, Kroeger and Long [4] extended their scheme to Partitioned Context Modeling (PCM) and Extended Partition Context Modeling (EPCM), and also extended their experiments to disk prefetching. Cho and Cho [5] suggested page premapping and page prefetching scheme, using hints that reflect the sequence of page references and the page fault behavior of a program.

Mowry, Demke, and Krieger [6] proposed a technique to efficiently prefetch data in scientific out-of-core applications. In their scheme, the compiler provides the crucial information about future access patterns based on array access pattern of a program, and the OS accepts dynamic prefetch hints generated by the compiler at runtime.

Patterson, Gibson, Ginting, Stodolsky, and Zelenka [7] showed how to use application-disclosed access patterns to expose and exploit I/O parallelism and to dynamically allocate file buffers among competing demands of prefetching hinted blocks, caching hinted blocks for reuse, and caching recently used data for unhinted accesses. Based on Patterson et al.’s system, Chang and Gibson [8] suggested an approach of using idle processor cycles to dynamically analyze application and predict its future I/O accesses. Coupled with an aggressive hint-driven prefetching system, they speculatively pre-execute the application’s code in order to discover and issue hints for its future read accesses.

Kaplan, McGeoch, and Cole [9] examined prepaged allocation and its interaction with two demand prepaing parameters: the degree, which is the number of extra pages that may be fetched at each page fault, and the predictor that selects which pages to prepage. Based on the result, they provided a description about when demand paging should be used.
Yet all of these researches are not specific for multimedia applications. Sbeyti, Niar, and Eeckhout [10] proposed a simple Pattern-Driven Prefetching (PDP) mechanism based on address access pattern of multimedia applications. The patterns observed are based on the notions of the inter-miss stride (memory address stride between two misses) and the inter-miss interval (number of cycles between two misses). According to the patterns being detected, PDP initiates prefetch actions to anticipate future accesses and hide memory access latencies. Sbeyti et al. targets on reducing the number of cache misses, which is the reason why they focused on fine-grained address access pattern, while we try to reduce page faults focusing on pattern of accessed pages between specified intervals, which is relatively coarse-grained.

VIII. CONCLUSION

In memory-limited multitasking environments, paging delays can be the main hindrance to seamless execution of multimedia players. Previous researches mainly focused on file prefetching or cache-level prefetching, and there was hardly any scheme specialized to reducing paging delays in multimedia players. In this paper we proposed a new memory management scheme based on the notion of stream, a set of heap-allocated buffers with the same backtrace of function calls. We first profiled memory allocations and memory accesses. Then, we used the profile data to analyze the access pattern of each stream. The access patterns we categorized streams are: constant access pattern and interval access pattern. According to the access pattern, we separately manage the memory for each stream. We also use prefetching, pinning and unpinning according to the pattern advices. Our experiment with multimedia players, our schemes show better execution quality by reducing page faults. Our scheme is particularly useful when the memory is a highly competitive resource.

ACKNOWLEDGMENT

We would like to thank Dr. Chanik Park and other members at Samsung Electronics Corp., who first brought up the page fault problem of multimedia applications to us. Many comments and discussion with them guided us to better evaluate the issues and pursue our solution.

REFERENCES


Heejin Ahn received the BS and the MS degrees in computer science from KAIST in 2006 and 2008. After graduation, she joined Samsung Advanced Institute of Technology, Korea as a member of research staff. Her research interests include compiler techniques for multimedia processing and reconfigurable architectures for consumer electronics.

Seongjin Cho received the BS and the MS degrees in computer science from KAIST in 2007 and 2009. After graduation, he joined Google, Korea as a member of technical staff. His research interests are in the field of multimedia processing, in particular multimedia codec to exploit new computer architectures such as general purpose computing on GPU and compiler techniques to optimize applications for embedded systems in conjunction with operating system supports.

Hyunik Na received the BS and the MS degrees in computer science from KAIST in 1999 and 2007. He is currently a PhD student at KAIST. Prior to his graduate study at KAIST, he served as a senior engineer at multiple venture companies including TmaxSoft and icube. His research interests are in the field of embedded computing, in particular compiler techniques to exploit link-time opportunities and special memory hierarchies of embedded processors for optimized code generation.

Hwansoo Han (M’03) received the BS and the MS degrees in computer engineering from Seoul National University, Korea in 1993 and 1995, and the PhD degree in computer science from the University of Maryland at College Park in 2001. He is currently an associate professor at Sungkyunkwan University. Previously, he served as an associate professor at KAIST and as a senior engineer at Intel. His research interests include compiler technology for high-performance computing, embedded computing, and secure computing.