Demand Prepaging for Flash Memory Using Static Program Analysis

Dong-Yun Shin, Youngho Lee, Sung-Soo Lim
Kookmin University, Korea
elsdy@kookmin.ac.kr, buriburi@davinci.snu.ac.kr, sslim@kookmin.ac.kr

Abstract
This paper proposes prepaging scheme using static program analysis to improve the predictability of execution behaviors of applications in embedded systems equipped with flash memory. We built a static program analyzer and the RTOS-based reference platform equipped with flash memory implementing the demand prepaging scheme proposed in this paper. Evaluation results show that the proposed prepaging scheme shows competitive execution times and improved memory consumption and prepaging hit ratio results compared to sequential prepaging.

Keywords: Demand Prepaging, Flash memory, Static Program Analysis, Page Fault Ratio.

1 Introduction
As the applications of embedded systems are being diversified, the memory requirements and performance demands are being increased significantly. Therefore, the demand paging, which has been used for general purpose systems as a mandatory method to make memory usage efficient, starts to be used in embedded systems equipped with flash memory. One big drawback of using demand paging is in performance predictability: since the page fault event has history sensitive nature in virtual memory system, predicting exactly the hit/miss of every page access from the page table is not possible. Such drawback prevents the demand paging from being used in embedded real-time systems with firm or hard timing requirements.

One possible solution of improving demand paging performance would be prefetching the pages that are to be accessed in future and eliminating most of the page faults of future page accesses. One extreme solution of the prefetching would be shadowing which fetches all the pages of target application before run-time. However, shadowing would consume significant memory and cause tremendous startup time of every application. If we would like to avoid the impractical overhead of shadowing, we could prefetch small amount of pages before those pages are actually accessed making prefetching of pages distributed over the whole execution period. This effectively divides a whole program execution into two different phases: prefetching and running phases. By separating the prefetching stage and running stage of the applications, the predictability of execution performance of each program could be improved since we could predict the execution time of the program as the actual execution time excluding most of the page fault handling time. Figure 1 shows the separation of prepaging phase and task execution phase.

![Figure 1: Demand paging and prepaging](image_url)

Prefetching useful pages before the actual accesses of the pages requires identification of the useful pages that are expected to be fetched at run-time. This necessitates the analysis of useful page access sequences statically. This paper proposes a prepaging technique analyzing the page access sequence using a static program analysis method and applying the static analysis results to prefetching the pages that are analyzed as the potentially accessed pages at run-time. The proposed static program analysis technique transforms target applications into control flow graphs and extracts corresponding page
mapping information and the page access sequences from the program representations. The page access information is provided to the operating system for every application at loading time and the operating system could use the information to prefetch the useful pages reducing most of the future page faults.

Prepaging has been already proposed and used in general purpose operating systems running on the computer systems equipped with magnetic disk-based secondary storage. For such systems, the pre-pageing should be performed considering the positions of the pages on disk sectors and the disk header since the seeking times of disk access times are significant. In such systems, static program analysis-based pre-pageing would not be appropriate since the pages to be prefetched might spread over a number of sectors resulting in significant performance degradation due to the prefetching. However, in this paper, since we are assuming embedded systems using flash memory as the secondary storage, the static program analysis-based pre-pageing could be well applied.

One issue of using the pre-pageing is how much impact the pre-pageing would have on the performance of the system or on the performance of each application. The impact of pre-pageing on the performance would be in the total number of page faults, the execution times of target applications, and the amount of memory consumption. Though we could easily reason that applying pre-pageing separating pre-pageing phase and running phase of each application would improve the predictability of the application, characterizing the overall impact of applying the pre-pageing on the performance factors would be significantly important to give sufficient motivation of using the technique.

In order to characterize the performance impact of pre-pageing, we implemented the pre-pageing based on static program analysis results on an ARM9-based reference platform running Nucleus real-time operating system. The performance impact of the pre-pageing with static program analysis was evaluated by comparison study among the proposed pre-pageing scheme, sequential pre-pageing (i.e., prefetching certain amount of sequential pages at every prefetching phase), and basic demand paging (i.e., each page is fetched only when the page is actually demanded). The comparison results show that the prefetching based on static program analysis significantly reduces the page fault ratio and the total memory consumption while gives little impact on the total execution times.

The paper is organized as follows: Section 2 summarizes related work in pre-pageing and static program analysis. Section 3 describes the proposed pre-pageing scheme based on static program analysis. Section 4 shows experimental results with the implemented reference platform for the evaluation. Section 5 concludes the paper.

2 Background

Demand paging has been considered as a necessary technique in virtual memory-based general purpose computer systems. Recently, the demand paging starts to be considered in embedded systems as the complexities and the sizes of the embedded systems increase [3] [4].

Pre-pageing has been proposed to reduce the number of page faults during run-time [11]. The pre-pageing schemes until now have been basically demand pre-pageing which pre-fetches pages that are potentially accessed in future every time page fault occurs [5] [6] [7] [8] [9] [10] [12]. Since the conventional secondary storage used in computer systems has been disk, the pre-pageing has been proposed to utilize and hide the seeking time of disk accesses. Therefore, the conventional pre-pageing schemes and the pre-pageings already implemented in the existing operating systems are usually sequential pre-pageing along with the sectors in the target disk storage. The selection of pages to be fetched is determined completely depending on the location of the pages on the disk.

In the conventional pre-pageing, the pages to be fetched are determined dynamically depending on the pages that are currently accessed and the positions of the pages. One technique that is proposed to use static program behavior information requires user provided information on the program structure and a mathematical model is used to identify the candidate pages [12]. The approach proposed in this paper is different from the existing technique in that we use static program analysis technique which extracts accurate information for the page access sequences and the page access sequence information is inserted into executable program images to be utilized by operating system loader [1][2].

3 Pre-pageing Using Static Program Analysis

This section describes how the static program analysis technique is used by identifying the page sequences to prefetch. The purpose of static program analysis is to accurately identify the pages to be potentially accessed at run-time and build the page access sequence information to be used by OS loader and scheduler to be used in page prefetching stage. This section first explains the static program analysis technique in detail and then describes how the static program analysis results are used in pre-pageing at run-time.

3.1 Static program analysis

The objective of using static program analysis in our demand pre-pageing system is to identify the sequence of pages that will be requested during execution of user programs and informing the
operating system of the sequence of potentially requested pages. The most important evaluation factor of static program analysis for demand prepaging is the accuracy of the sequence of potentially accessed pages. In this work, we first have used a simple control flow-based analysis and making correspondences between each basic block in the control flow graph and pages to be fetched from flash memory. The mapping between the basic blocks and the corresponding pages is performed by investigating the addresses of fetched instructions and calculating the corresponding page numbers using the page size information: the appropriate offset is applied reflecting the location in the memory space for the programs to be loaded.

Figure 2 shows how we utilize static program analysis for demand prepaging. Target executable files are fed into our simple static program analyzer who analyzes the page mappings for the basic blocks of the target executables. Each basic block is linked to pages that contain the instructions included in the basic block. By making correspondence between each basic block and pages, the sequence of pages to be potentially loaded into memory from secondary storage could be identified.

The control flow information and the page sequence information provided by our static program analyzer are integrated into binary information and inserted into the target executable file by a separate tool. The additional information for prepaging is included in the attribute field of ELF executable format. The expanded form of executable will be loaded by the operating system to build an internal data structure to track the currently accessing page and potentially useful pages. The operating system should be slightly modified to recognize the page sequence information and perform the demand prepaging using the information.

As an example of page mapping based on control flow analysis, Figure 3 shows a control flow graph with basic block numbers and corresponding pages with the mapping between each basic block and pages.

As shown in Figure 3, a page could contain more than one basic block or a single basic block could be mapped to more than one page. By analyzing the control flow and corresponding page mapping information, we could identify the sequence of potential page accesses during run-time.

One naïve prepaging would be sequential prepaging which fetches the certain number of pages sequentially at every page fault or page prefetch time. Since the sequential prepaging depends on the temporal and spatial locality for the performance improvement by prepaging, it does not require any static program analysis. Many modern operating systems are already using the sequential prefetching of pages. Though the sequential prefetching could be beneficial, since sequentially prefetched pages would contain unnecessary pages resulting in excessive prefetching time and space consumption, resource and time constrained real-time embedded systems could not conveniently use the sequential prefetching. In order to improve the hit ratio of prepaging as much
as possible, we need to identify the potentially accessed pages by static program analysis. The benefit of using static analysis for demand prepaging highly depends on the accuracy of the page sequence analysis results. When we use just the static control flow information without any dynamic execution behavior information in identifying the potential page accesses, there is possibility of fetching redundant or useless pages that will not be accessed; this is caused by fetching the pages included in the execution path that is not feasible in dynamic execution but included in the control flow. In order to eliminate the effect of infeasible execution path, we need to perform data flow-centric analysis such as abstract interpretation or symbolic execution. Such data flow-centric analysis would significantly increase analysis complexity in both analysis time and memory space aspects. In this work, instead of applying the data flow-centric analysis methods (though such analysis could be ultimate solutions to improve the accuracy), we combine dynamic execution behavior information and the page sequences to be prefetched and continuously refine the pages to prefetch at run-time. This could be good reconciliation between the data flow-centric static program analysis maximizing the accuracy of the analysis and the naïve control flow-based analysis purely depending on the static control flow program representation. Figure 4 shows how we could refine the candidate pages to fetch during run-time as the currently accessing pages are updated.

3.2 Prepaging

The parameters to be determined in performing prepaging include 1) degree of prepaging: how many pages are pre-fetched on a single page fetch, 2) page selection policy: which pages are selected to prefetch and 3) memory space for prefetched pages: how many prefetched pages can be maintained. We ignore the last factor by assuming that there is enough memory space to maintain the prefetched pages and thus the pages are not swapped out by fetching new pages. This assumption is to simplify the problem of eliminating the page replacement issues in this work. We have a plan to extend our work to consider the effect of page replacement. For the second factor, page selection policy, we use either our static program analysis-based approach or conventional sequential prefetching. One of our objectives is to compare our control flow-based page selection with conventional sequential page prefetching. The first factor, degree of prepaging, is parameterized ranging from 1 page to 15 pages. As the degree of prepaging increases, the higher hit rate of prefetched pages trades off both the increased burst memory transfer and higher memory consumption.

Since one of the main objectives of the prepaging would be improving predictability of the performance of demand paging by separating prefetching phase and page access phase, the method how to separate the two phases would be important. In real-time systems with fixed priority scheduling policy, we could apply the well known scheduling technique to the system where the prefetching phase constitutes one separate task with sufficiently high priority to fetch potentially accessible pages; the amount of the pages to prefetch and the length of CPU time to be allocated to the prefetch task would be the factors to be determined before the system is built. We consider such factors as the orthogonal issues to the objective of our current paper focusing on the impact of prepa-
Figure 5: Static program analyzer for prepaging

ging on the various performance factors; therefore, in this paper, we just assume that the page prefetching is performed when page fault occurs. In other words, the prefetch phase is not separated as a task with fixed scheduled time, but prefetch is performed every time when the currently accessed page is not included in the previously prefetched pages. This is called demand prepaging. Figure 5 shows the behavior of demand prepaging. The prepaging is performed at every page fault for certain fixed number of pages. Therefore, the prepaging phase occurs not regularly. According to the original idea of prepaging where we schedule the prepaging as a fixed period task in real-time systems, we could build a periodical task set using prepaging with improved predictable performance as shown in Figure 1.

Figure 6: Demand prepaging

The prepaging is performed at every page fault in the following steps: First, the currently accessed page is compared with the control flow graph and page mapping information included in the program executable image to find the start page to be prefetched. Second, the candidate pages are selected from the control flow and page mapping information traversing the control flow nodes. Third, the pages are prefetched from the secondary storage (flash memory in this paper) for the number of pages according to the prepaging degree configured initially. During prefetching, since we assume that the memory space to store prefetched pages is infinitive, the page replacement to spare the space for the prefetched pages is not needed.

4 Experimental Results

The effect of prepaging using static program analysis information is evaluated on a real platform running a real-time operating system equipped with demand paging and NAND flash memory. Demand prepaging using static program analysis information is compared with demand prepaging for sequential pages.

4.1 Experiments set up

The hardware platform used for our experiments is a reference platform based on S3C2410 (ARM920T core-based) processor and equipped with 64MB of SDRAM and 1Gbits of NAND flash memory. The operating system used in the platform is Nucleus Plus, a commercial real-time operating system widely used in handheld devices. Though original version of Nucleus Plus kernel does not implement demand paging, a demand paging module implemented by Seoul National University is used for base platform. We modified the original implementation of demand paging to include our demand prepaging algorithm using static program analysis results.

The implementation of the static program analyzer is based on Diablo which is an existing link-time optimizer from Paris Research Group, Ghent University, Belgium. The control flow construction part of Diablo is extracted to form the front-end of our static program analyzer [13]. Our static program analyzer has the structure shown in Figure 6. Control flow analysis and page mapping analysis are performed after extracting necessary information from the ARM/ELF executable format. Since we assume that the page size is 4 KB, the page mapping analysis is performed calculating the start addresses of pages from the beginning of the code segment in ARM/ELF executables.

In order to evaluate the various performance factors, an event logging and time stamp-based performance analysis method is used. The event logging system known as Linux Trace Toolkit is transplanted onto Nucleus Plus in our experiment platform [14]. The event logging system collects predefined events with their timestamps in a specified memory space. The predefined events include page
fault detection, fault handler start, fault handler done, and page load done; each log has event identifier and the corresponding timestamp. The logged events are later analyzed through offline performance analyzer running on PC.

4.2 Benchmarks

In order to evaluate the effect of our prepaging scheme, five different synthetic benchmark programs are composed so that the benchmark programs could show diverse page access behaviors. The reason why we use synthetic benchmark programs instead of using standard benchmark programs is to make the impact of page access sequence behaviors of user programs on efficiency of prepaging stringent. The synthetic benchmark programs to show extreme cases of page access patterns maximizing the impact on the prepaging are shown in Figure 7. In each benchmark program control flow, the number of pages mapped to each basic block (region) is also shown in the left side of each graph.

Figure 7(a) is the simplest program with a simple loop where the single region contains 17 pages. Figure 7(b) contains three different paths in a loop
where each iteration constitutes different path; the regions contain 6, 11, and 6 pages, respectively. Figure 7(c) shows a complicated benchmark program where there are 63 pages and every iteration goes through different execution path. Depending on the input values, the execution path is selected among 16 different paths where certain paths include significantly more pages than other paths (i.e., if region 8 is selected after region 7, the on-going path would contain significantly more pages than the path directly going through region 0 right after region 7). Figure 7(d) and Figure 7(e) use same control flow but contain different number of pages: while Figure 7(d) shows the control flow where the number of pages differs significantly depending on the paths selected, Figure 7(e) shows the control flow where the number of pages is similar among the paths. We call the five synthesized benchmark programs as SYNTH1, SYNTH2, SYNTH3, SYNTH4, and SYNTH5, respectively. In our experiments, we provide sample input data to the synthetic benchmark programs so that the programs show mostly extreme behaviors in page accesses.

4.3 Evaluation results

We performed experiments to compare various performance factors among static program analysis-based prepaging, sequential prepaging, and demand paging. The performance factors include the total memory consumption, the total execution time, and prepaging hit ratio. Figure 8 shows the total memory consumption for the five benchmark programs. Since demand paging only consumes the memory space to store pages that are fetched on demand, the memory consumption for the demand paging is minimal among the schemes. Static program analysis-based prepaging shows less memory consumption than sequential prepaging for all the benchmark programs. The gap between the static program analysis-based prepaging and the sequential prepaging increases as the prepaging degree increases since the possibility of prefetching unnecessary pages in sequential prepaging also increases. When the benchmark program has execution path alternating between different paths frequently, the static program analysis-based prepaging shows much improved performance.

Figure 9 shows the total execution times for the five benchmark programs for static program analysis-based prepaging, sequential prepaging, and demand paging. We should note that the total execution time includes the prepaging time which might be additional cost to fetch additional pages other than the faulty page. Therefore, the execution times for
prepaging with certain prepaging degree configurations show worse than the execution times

Figure 9: Total execution time

Figure 10: Prepaging hit ratio
of demand paging. However, as explained earlier, since the major purpose of using pre-paging would be improving the predictability of performance by separating prefetching phase and actual page access phase, the increase of the total execution times would not be critical issue. As shown in the figure, the increase of the total execution times when applying static program analysis-based pre-paging is not very significant compared to sequential pre-paging. Since sequential pre-paging fetches pages which are sequentially located in secondary storage without considering the necessity of the pages, the execution times also increase significantly as the amount of pages to prefetch increases.

As shown in the figure, depending on the pre-paging degree, static program analysis-based pre-paging shows better execution times than the execution time of demand paging since the number of pages to fetch match the size of region (pages) to access in the future.

Figure 10 shows the results of comparing pre-paging hit ratio. The pre-paging hit ratio means the probability of the number of pages that are actually accessed among the prefetched pages. As shown in the figure, the pre-paging hit ratio of static program analysis-based pre-paging is significantly better than the sequential pre-paging. In this comparison, the demand paging is not shown since the fetched page by demand paging should be accessed in future resulting in always 100% of hit ratio. Figure 10(d) and Figure 10(e) shows significant decreases of pre-paging hit ratio compared to other bench-mark programs since those benchmark programs have alternating execution paths that could increase possibility of fetching unnecessary pages. However, the static program analysis-based pre-paging also shows significantly better hit ratio than the sequential pre-paging for the benchmark programs, SYNTH4 and SYNTH5.

5 Conclusion

In this paper, a pre-paging scheme is proposed using static program analysis technique. A static program analysis technique is used to extract the candidate pages to be prefetched and the page access information is included in executable program images and the OS loader refers to the page access information to prefetch the pages to be accessed in the future. The main purpose of using pre-paging in embedded real-time systems would be improving the predictability of application execution performance. While such improvement would be naturally obtained by using pre-paging, the impact on the various performance factors when using pre-paging should be characterized.

We performed evaluation using a reference board based on ARM9 running Nucleus operating system where the pro-posed pre-paging scheme is implemented. A static program analyzer was implemented to extract page access information of target tasks.

The evaluation results reveal that the proposed pre-paging scheme shows improved better memory consumption, execution times, and pre-paging hit ratio than the sequential pre-paging that has been used conventional in various existing operating systems.

Acknowledgement

This work was supported in part by research program 2006 of Kookmin University in Korea and in part by Seoul R&D Program (NT070124C092852).

Reference


**Dongyun Shin**
He is a researcher on an PhD. degree in the embedded system laboratory in the school of computer science in the Kookmin University in Korea. His research interests include embedded system, mobile communication device, computer architectures, real time system. He received double BS degrees in computer science and information communication from Kookmin University in 2005, and get M.S degree in Computer Science from Kookmin Univ. in 2007. He is a student member of IEEE.

**Young-Ho Lee**
He received the B.S. and M.S. in Computer Science from Kookmin University, Korea in 2005 and 2007, respectively. He is currently a Ph.D. candidate in Computer Science and Engineering from Seoul National University. His research interests include embedded systems, multi-core architecture, and Network-on-Chip architecture. He is a student member of ACM.

**Sung-Soo Lim**
He is a full-time lecturer in School of Computer Science, Kookmin University, Korea. His research interest includes embedded mobile systems, real-time systems, and computer architecture. He received BS, MS, and PhD degrees in Computer Science and Engineering from Seoul National University. He was the Chief Technical Officer in PalmPalm Technology, Inc. in Korea from 1999 to 2004 and a visiting researcher in real-time system laboratory, University of Illinois at Urbana-Champaign in USA from 2000 to 2001.