Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory* †

Chanik Park‡ Junghee Lim§ Kiwon Kwon§ Jaejin Lee§ Sang Lyul Min§

§School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
{junghee,kiwon,jlee}@aces.snu.ac.kr, symin@dandelion.snu.ac.kr

‡Memory Division, Samsung Electronics Co., Ltd.
Hwasung-City, Gyeonggi-Do 445-701, Korea
ci.park@samsung.com

ABSTRACT

In this paper, we propose a novel, application specific demand paging mechanism for low-end embedded systems with flash memory as secondary storage. These systems are not equipped with virtual memory. A small memory space called an execution buffer is allocated to page an application. An application-specific page manager manages the buffer. The manager is generated by a compiler post-pass and combined with the application image. Our compiler post-pass analyzes the ELF executable image of an application and transforms function call/return instructions into calls to the page manager. As a result, each function of the code can be loaded into memory on demand at run time. To minimize the overhead of demand paging, code clustering algorithms are also presented. We evaluate our techniques with five embedded applications. We show that our approach can reduce the code memory size by 33% on average with reasonable performance degradation (8-20%) and energy consumption (10% more on average) for low-end embedded systems.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design Studies; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, optimization*; D.4.2 [**Operating Systems**]: Storage Management—*secondary storage, storage hierarchies, virtual Memory*

General Terms

Algorithms, Management, Measurement, Performance, Design, Experimentation

Keywords

Compilers, Post-pass optimization, Clustering, Paging, Heterogeneous memory, SRAM, Flash memory, Embedded systems.

1. INTRODUCTION

Flash memory is widening its user base in mobile embedded systems not only for data storage but also for code storage due to its non-volatility, solid-state reliability, and low power consumption. Typically, flash memory is categorized into two types: NOR and NAND. NOR type is particularly well suited to code storage and execute-in-place (XIP) applications that require high-speed random access [9]. While NAND flash does not lend itself to XIP applications due to its sequential access and long access latency, it is good for data storage due to its lower cost per bit and smaller cell organization (i.e., high density) [9]. As a result, NAND flash memory has been widely used in cellular phones and portable memory cards in consumer electronics.

A typical memory architecture for low-end embedded systems, such as low-end cellular phones and memory card controllers, consists of code storage (i.e., NOR flash memory, EEPROM, or Mask ROM), for bootstrapping and firmware execution, and low-power SRAM for working memory and data storage. To design such a low-end embedded system, reducing the size of SRAM has been a major concern because SRAM is a cost and power bottleneck. Another important design consideration is that the firmware code should be easy to update because bug fixes

*This work was supported in part by the Ministry of Education under the Brain Korea 21 Project in 2004, by the Ministry of Science and Technology under the National Research Laboratory program, by the Institute of Computer Technology at Seoul National Universy, and by the IT SoC Promotion Group of the Korean IT Industry Promotion Agency under the Human Resource Development Project for IT SoC Key Architects.

†Correspondence to jlee@aces.snu.ac.kr.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

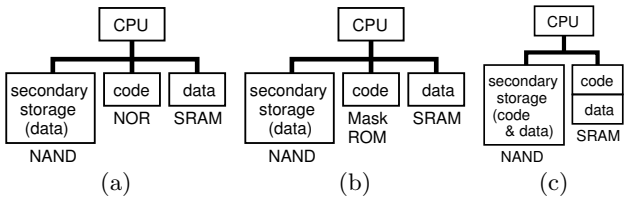


Figure 1: Different memory architectures. (a) NOR+SRAM (b) Mask ROM+SRAM (c) NAND+SRAM (shadowing)

or specification changes should be reflected to the system in a timely manner at a low cost.

Recently, the size of the firmware is increasing from tens of KB to hundreds of KB as new features, such as security functions, are added to the system. As a result, an efficient memory management for code has become a key consideration in designing this type of embedded system.

Figure 1 shows the current trends of memory architectures in embedded systems with NAND flash memory. Figure 1(a) shows an architecture that uses NOR flash memory for code storage and SRAM for working memory (**NOR+SRAM**). This architecture gives moderate performance and power consumption with ease of firmware update. However, its cost is high due to the NOR flash memory. The second architecture in Figure 1(b) (**MROM+SRAM**) reduces the system cost by replacing the NOR flash memory with cheaper mask ROM, but its difficulty of update may cause nonrecurring engineering costs in the case of firmware bugs or functionality upgrades. The more advanced solution (**NAND+SRAM**) is shown in Figure 1(c). Neither NOR flash memory nor mask ROM is used in this architecture. It uses NAND flash memory for code storage with a shadowing technique[19]. Copying the whole firmware code into SRAM offers the best performance possible at run time, but slows down the boot process. Even though **NAND+SRAM** seems to meet performance requirements and it is easy to update firmware, additional SRAM is necessary to store both data and code. This is not desirable because it increases cost and power consumption. Among others, the most critical problem in existing memory architectures is that they cannot accommodate the increasing code size without redesigning existing systems. This is a big hurdle to time-to-market delivery.

To solve the problems occurring in **NAND+SRAM** architectures, we propose a novel demand paging technique that can be implemented in a compiler post-pass (i.e., at link time or on binary images). Our technique targets low-end embedded systems without virtual memory or MMU hardware support.

In our demand paging, a small size of SRAM space is reserved to execute code. The application is divided into multiple segments. Each code segment in the application is loaded on demand into the reserved space and runs when it is needed during execution. Intuitively, in order to utilize the limited memory space, we can take advantage of dynamic loading techniques such as overlays[14]. However, this requires manual efforts by programmers to reorganize the structure of the application. Moreover, restructuring the application is impossible when the maximum size of the call chain in the application is bigger than the memory space.

The proposed technique is implemented in our post-pass optimizer, SNACK-pop (**S**eoul **N**ational university **A**dvanced **C**ompiler tool **K**it - **p**ost-pass **op**timizer). SNACK-pop transforms each function call/return instruction in an application's ELF binary to a call to the small page manager that then becomes part of the application itself. The page manager always resides in SRAM. It checks if the target segment of the branch instruction resides in SRAM. If so, it just passes control to the target in SRAM. Otherwise, the page manager loads the code segment from the NAND flash memory to SRAM and passes control to the target in SRAM.

With our compiler-assisted demand paging technique, code memory size can be significantly reduced with reasonable performance degradation and energy consumption. As a result, we can build a cost-effective memory system for low-end embedded systems with limited resources.

The rest of this paper is organized as follows. Section 2 describes the overall framework of our approach. Section 3, presents our approach in detail. Section 4 describes the evaluation environment. Section 5 discusses the experimental results obtained. Section 6 lists related work. Finally, Section 7 concludes the paper.

2. OVERALL FRAMEWORK

In this section, we describe the basic idea of our demand paging technique implemented in the compiler's post-pass. Before we get to this issue, we look into the organization and basic operations of the NAND flash memory.

2.1 Background: NAND Flash Memory

NAND flash memory consists of a fixed number of blocks, where each block has 32 pages and each page consists of 512 or 2048 bytes of main data and 16 or 64 bytes of spare data (Figure 2).

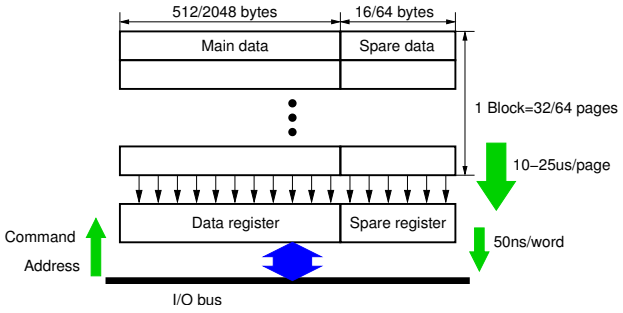


Figure 2: The organization of NAND flash memory.

Read/write operations are performed on a page basis. In order to read a page, a read command and an address are fed into NAND flash memory through I/O pins. After 10-25us of delay, the requested page is loaded into data and spare registers. Thereafter, 8 or 16-bit data is sequentially

fetched from data and spare registers every 50ns. Consequently, an efficient random access to the data is impossible. This is partly why shadowing is required to store code in NAND flash memory, and why it is not commonly used for code storage.

In order to support primitive operations of NAND flash memory, such as reset, read, write, and erase, a software driver (flash driver) is needed. Each flash operation consists of a sequence of commands and addresses. The flash driver always resides in SRAM due to its criticality. Our demand paging technique extensively uses the page-based flash read operation.

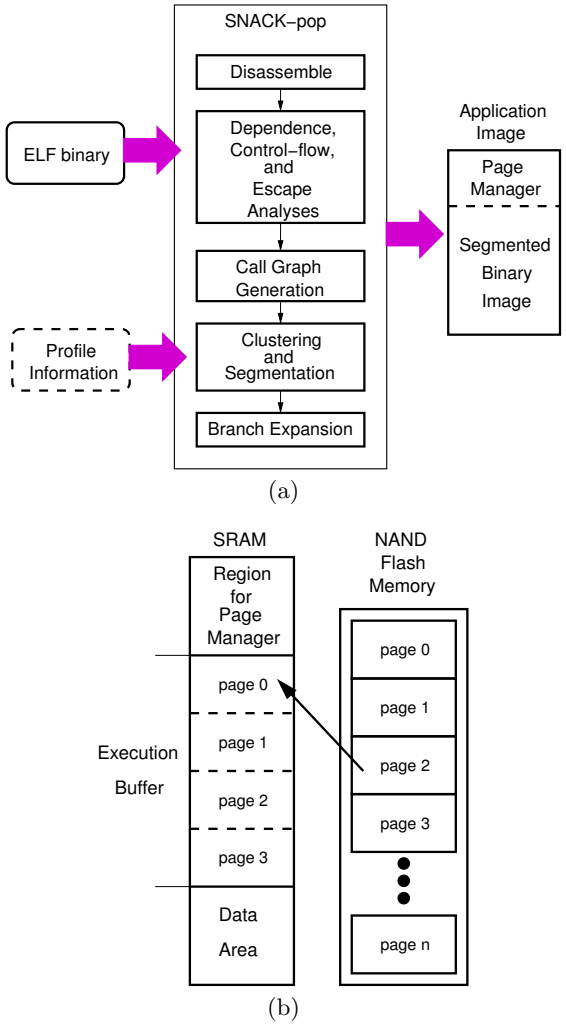


Figure 3: Our framework. (a) SNACK post-pass optimizer. (b) SRAM memory layout.

2.2 The Framework

Figure 3(a) shows the overall framework of our approach. An ELF binary with symbolic information is fed into the SNACK post-pass optimizer. It performs conservative data-dependence, control-flow, and escape analysis using the information provided by the ELF executable format[1]. In the escape analysis phase, it detects addresses of constant data in the code section that are passed as an argument to a function. The constant data in the code section will be copied and clustered together with the function that receives the addresses, or they will be moved to a fixed region in the page manager later. Then, SNACK-pop generates a static call graph for the application. The static call graph can be annotated with additional profiling information. The functions in the application are clustered together into segments using the call graph information. The size of each segment is a multiple of 512B or 2KB that is the same as the page size in the NAND flash memory. Finally, SNACK-pop converts function calls/returns in the image to the calls to the page manager. The page manager code and the segmented code together become the final executable image.

Every control transfer instruction between functions in the original image is transformed into a call to the page manager. The page manager checks if the target function is found in the *execution buffer*. The execution buffer is the region in the SRAM where paging occurs. The SRAM memory layout is shown in Figure 3(b). If the target function is not in the execution buffer, the manager invokes the flash driver to load the target code from the NAND flash memory to the execution buffer. The execution buffer consists of multiple pages. Its page size is equal to the page size of the NAND flash memory. The size of the execution buffer varies as performance requirements vary. The page manager resides in SRAM to handle page faults. It contains page fault handling routines and the flash driver. Data area in the SRAM is for the program stack and heap. Its size is known at link time.

Our approach focuses on reducing the code space size in the SRAM. The entire segmented image resides in the NAND flash memory before it runs.

3. COMPILER – ASSISTED DEMAND PAGING

In this section, we elaborate on our compiler assisted demand paging techniques.

3.1 Segmentation

The compiler post-pass transforms the binary image of an application into an executable form that is suitable for demand paging. Code sections in the image are divided into segments. Each segment consists of single or multiple pages. The page size is the same as the page size of the NAND flash memory. Each segment satisfies the following:

- Its size is a multiple of the page size, and it contains at least one page.
- It contains at least one function. A function cannot be contained across different segments.

For example, the binary image as shown in Figure 4(a) consists of 7 functions A, B, C, D, E, and F. This image can be segmented into the image shown in Figure 4(b). Note that the size of the segmented image is bigger than the original because of the internal fragmentation caused by segmentation.

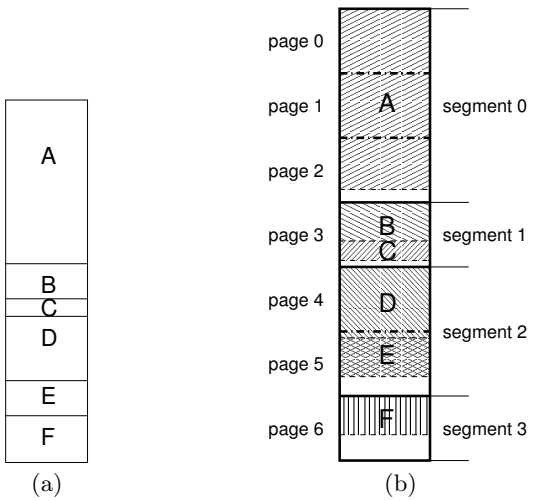


Figure 4: The original image and the image after segmentation. (a) The original image. (b) The image after segmentation.

Page Table

Index	Segment Index	Buffer Page Index
0	0	0
1	0	1
2	0	2
3	1	3
4	2	-1
5	2	-1
6	3	-1

(a)

Buffer Page Table

Index	Page Index
0	0
1	1
2	2
3	3

(b)

Segment Table

Index	Start	End	Size
0	0	3	3
1	3	4	1
2	4	6	2
3	6	7	1

(c)

Figure 5: Tables maintained by the page manager. (a) Page Table. (b) Buffer Page Table. (c) Segment Table.

3.2 Page Manager and Execution Buffer

The page manager contains a page table, a buffer page table, and a segment table. The tables for the application in Figure 4 are shown in Figure 5.

A page table entry consists of two fields, a segment index and a buffer page index. The segment index field is the index of the segment to which an image page belongs. The buffer page index field is set to -1 if the image page does not reside in the execution buffer. If the image page exists in the buffer, the field is set to the corresponding buffer page index. Because the image after segmentation contains 7 pages, there are total 7 entries in the page table in Figure 5(a).

A buffer page table entry contains the image page index that has been loaded in the corresponding buffer page. In Figure 5(b), there are a total of 4 SRAM pages allocated in the execution buffer for paging.

A segment table entry consists of three fields, the indexes of beginning and ending image pages of the segment and the number of pages in the segment. Because the image after segmentation contains 4 segments, there are total 4 entries in the segment table in Figure 5(c).

The execution buffer is shown in Figure 6. It consists of the physical pages allocated in SRAM and a pointer to the next available page (*PNAP*) in the buffer. *PNAP* contains the index of the first buffer page available for paging. We assume a *round-robin* page replacement policy. The smallest replacement unit for paging is a segment. If buffer pages are needed for loading a new segment, and if *PNAP* points to the end of the execution buffer or the number of pages from *PNAP* to the end of the buffer is less than the number of pages in the segment, then *PNAP* is wrapped around and points to the first page of the buffer. Then, the pages from the first buffer page are overwritten with the segment, and *PNAP* is set to the index of the page next to the loaded segment.

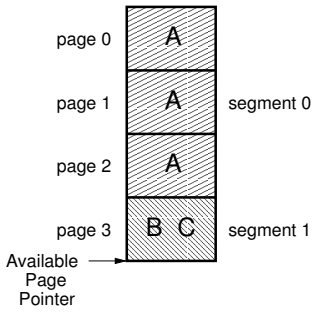


Figure 6: The execution buffer.

3.3 Page Management

Whenever the page manager receives the target address a of a function call/return, which is an absolute address in the segmented image, it does the following (Figure 7):

1. If there is a return address r_{buffer} in the linkage register, it converts r_{buffer} to the corresponding absolute address r in the segmented image and saves it to the linkage register. Note that the return address r_{buffer} is a physical address in the buffer (SRAM).

2. It computes the corresponding absolute page index p using a,

$$p = (a - base) >> log_2 P$$

where *base* is the starting address of the image, and P is the page size.

3. It accesses the page table with p. If the value b of the buffer page index field is less than 0 (this means that a page miss occurs),

   (a) With the value s of the segment index field of the page table entry p, it accesses the segment

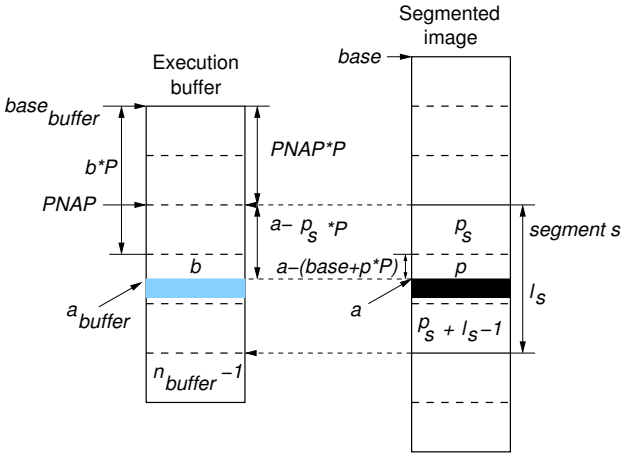


Figure 7: Converting the target address in the image to the address in the buffer.

table. It obtains the starting absolute page index p_s and the length l_s of the segment s.

(b) If $PNAP + l_s > n_{buffer}$, $PNAP$ is set to the first buffer page index (i.e., 0), where n_{buffer} is the number of buffer pages.

(c) It accesses the buffer page table to obtain the page table indexes of the image pages residing in the buffer pages from $PNAP$ to $PNAP+l_s-1$. It sets to -1 the buffer page index fields in the page table using the indexes (i.e., the corresponding pages are invalidated).

(d) The segment s is loaded from the flash memory to the execution buffer pages $PNAP$ to $PNAP + l_s - 1$. The page table entries from p_s to $p_s + l_s - 1$ and the corresponding buffer page table entries are updated.

(e) The buffer address a_{buffer} that corresponds to the absolute address a is obtained by,

$$a_{buffer} = (a - p_s * P) + (base_{buffer} + PNAP * P),$$

where $(a - p_s * P)$ is the offset of the target in the segment, and $base_{buffer}$ is the starting address of the first buffer page.

(f) $PNAP$ is updated to point to the next available buffer page $PNAP + l_s$.

(g) Set pc to a_{buffer} (i.e., a jump to the target occurs).

4. Otherwise (if a page hit occurs, $b \geq 0$), it computes the corresponding buffer address a_{buffer}.

(a) $a_{buffer} = (a - (base + p * P)) + (base_{buffer} + b * P)$, where $a - (base + p * P)$ is the offset of the target in the image page p, and $base_{buffer} + b * P$ is the physical address of the buffer page that contains the image page p.

(b) Set pc to a_{buffer} (i.e., a jump to the target occurs).

3.4 Call/Return Translation

When the segment to which the target address of a function call/return belongs has been paged out from the buffer, it is impossible to jump to the target address. Consequently, each function call/return that has the target address in another segment must be translated into a call to the page manager. The absolute target address in the image is passed to the page manager. The page manager loads the segment that contains the target address to the execution buffer.

```
...                          ...
                             BL       L_foo
BL      foo          ⇒       ...
...          L_foo:          STMFD    sp!, {r0, lr}
                             MOV      r0, function_addr
                             LDR      pc, manager_func_BL
```

(a)

```
...                          ...
                             B        L_foo
B       foo          ⇒       ...
...          L_foo:          STR      r0, [sp, #-4]
                             MOV      r0, function_addr
                             LDR      pc, manager_func_B
```

(b)

```
...                          ...
                             B        L_lr
MOV     pc, lr       ⇒       ...
...          L_lr:           LDR      pc, manager_func_lr
```

(c)

```
...                          ...
                             B        L_rx
MOV     pc, rx       ⇒       ...
...          L_rx:           STR      rx, [sp, #-4]!
                             LDR      pc, manager_func_rx
```

(d)

Figure 8: Branch translation. (a) When the call uses the linkage register lr for return. (b) When the call does not use the linkage register lr for return. (c) Return (d) The pattern of the function calls that use function pointers.

However, there are some exceptions. The post-pass optimizer divides the calls into two categories depending on their types.

The target of the call is in the same segment. In this case, we do not need to translate the function call into a call to the page manager. Whenever the function call of this type occurs, the page that contains the target address resides in the execution buffer. However, before the call occurs, the return address must be translated into the absolute address in the image and saved in the linkage register. The extra code for this translation can be located in the space due to internal fragmentation in the segment, or the page manager can handle the translation. Whenever a return occurs, the page manager uses the absolute address saved in the linkage register to find the corresponding location in the execution buffer.

The target of the call is not in the same segment. Each call of this type is translated into a call to the page manager. The target address is passed to the page manager. The page manager takes an appropriate action described in Section 3.2. The extra code generated by the translation is located in the space due to internal fragmentation in the segment.

Figure 8(a) and (b) show the translation when the target is not in the same segment. When the call is of type "BL foo", the register $r0$ and the linkage register lr are saved on the stack (Figure 8(a)) before the page manager is called. The register $r0$ is used to pass the address of the function "foo". After the page manager is called, it retrieves the saved value of lr from the stack and converts the physical return address to the absolute address in the image. After $r0$ is restored from the stack, lr is set to the absolute address. Then, control is transferred to the function "foo" from the page manager. When the call is of type "B foo" (Figure 8(b)), its translation is similar to the case of "BL foo", but it does not modify lr.

Function returns. When the callee returns control to the caller, it is not guaranteed that the caller resides in the execution buffer. This means that the post-pass optimizer always needs to translate a function return to a page manager call. Figure 8(c) shows the translation. The page manager loads the target segment in the execution buffer using the value of lr, and control is passed to the target address in the buffer.

Function pointers. Figure 8(d) shows call patterns using function pointers (i.e., pc is set to the value of some register rx, not lr). Before the page manager is called, rx is saved on the stack. The page manager retrieves the value of rx (the physical target address) from the stack, converts it to an absolute address, and then loads the target segment in the execution buffer using the absolute target address. Then, control is passed to the target in the buffer (see Section 3.2 for more detail).

3.5 Constant Data

When a chunk of constant data in the code section is accessed by multiple functions in different segments, the chunk is copied into a space at the end of the page manager (i.e., the data chunk is pinned in the page manager region in the execution buffer). This is because the constant data chunk must be in the execution buffer whenever one of the functions accesses the data. There are two cases when we pin a chunk of constant data.

Different functions *directly* access the data. In this case, we just copy the chunk to the pinned region in the page manager.

Different functions *indirectly* access the data. When this occurs, the address of the data is passed to the function. The post-pass optimizer performs a conservative escape analysis to detect such a case. Then, it copies the escaping constant data to the pinned region in the page manager.

3.6 Optimization by Pinning Segments

There are two significant forms of overhead in our segmented paging mechanism. One is extra branching and computation due to the page manager calls. The other is page misses that occur when the segment requested is not in the execution buffer. If we pin some segments in the execution buffer and make them never page out, these types of overhead will be reduced significantly. Since resident segments are never paged in and out, the function call/return targets in the resident segments do not need to be translated into calls to the page manager, which reduces the branching overhead. Moreover, if the resident segments are frequently accessed, the total number of page misses will be significantly reduced. To determine the segments to be pinned, we use static call graph information, dynamic profiling information, or user hints in the case of time-critical code.

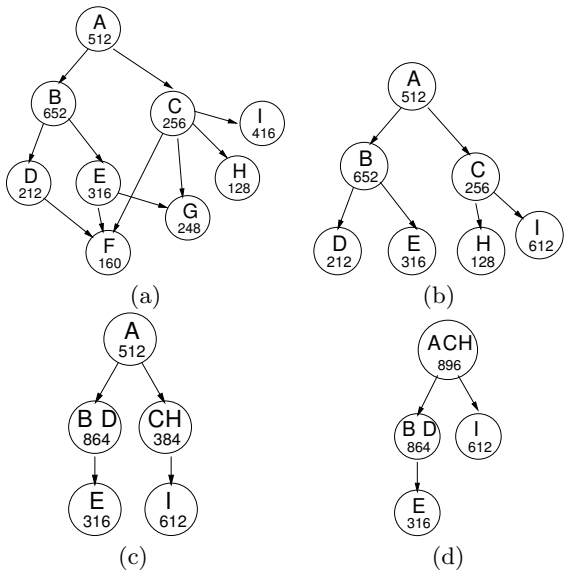


Figure 9: Clustering of functions using a static call graph. (a) The original static call graph. (b) After determining the functions to be pinned in the resident segment. (c) After merging each single-parent leaf node to their parent node. (d) After merging a node n that has no parents and its child whose only parent is n.

3.7 Clustering Techniques

In this section, we describe function clustering techniques used in our segmented paging. We can think of a variety of clustering techniques to improve the locality of segments in the execution buffer. We use three different clustering techniques in this paper.

3.7.1 *Basic Clustering* (Base)

We cluster functions in order of their occurrence in the original image. Each segment contains exactly one function. However, if internal fragmentation of the segment is big enough for the next adjacent function, we include it in the current segment. Note that the size of a segment is a multiple of the page size.

3.7.2 *Static Clustering* (Static)

We cluster functions into segments using the static call graph generated from the original image. Let max_seg_size be the maximum segment size allowed for clustering and $max\_res\_func\_size$ be the maximum size of a function that can be pinned in the resident segment.

1. Each leaf node whose size is less than $max\_res\_func\_size$ and whose number of parents is greater than one is clustered into the resident segment in some order until the resident segment is full. We remove the leaf node from the call graph.

2. Each single-parent leaf node is merged to its parent if the size of the parent plus its size is less than max_seg_size. Repeat this step until there is no change in the call graph.

3. Starting from the root node of the call graph, if a node n has no parents, merge its child c whose only parent is n to n itself. The children of c becomes the children of n and n becomes a parent of them. Repeat this step until there is no change in the call graph.

4. Each node in the call graph becomes a segment.

For example, we have a static call graph shown in Figure 9(a). The labels in each node represent the names of the functions contained in the node and the size of the node in bytes. Let max_seg_size be 1024 bytes and $max\_res\_seg\_size$ be 512 bytes. In the first step, the functions F and G are placed in the resident segment (Figure 9(b)). In the second step, D is merged to B and H is merged to C (Figure 9(c)). Finally, CH is merged to A in the third step (Figure 9(d)). The nodes BD, ACH, E, and I in the final call graph become three segments in the segmented image.

If a function (called a header function) and all its descendants are in the same segment, and there is no call to the descendants from outside the segment, the calls from the header function to the descendants do not need to be translated to the page manager calls. The same is true for the returns from the descendants.

3.7.3 *Clustering with Profiling Information* (Profile)

Functions can be clustered with segments based on profiling information (i.e., using the dynamic number of calls to each function). After profiling, functions are sorted according to the number of calls to them in decreasing order and placed in a list. Let $max\_res\_seg\_size$ be the size of the resident segment. The function with the highest number of calls and all its descendants in the call graph are clustered to the resident segment if their total size is less than $max\_res\_seg\_size$. Then, they are removed from the sorted list. The same process goes on with the next highest function in the list until there is no room left in the resident segment. For the remaining functions in the list, we use the basic clustering technique (Base).

Application	Source	Code Size (SRAM)
	Description	
(*Quicksort*)	MIBench	3544B (4KB)
	A quick sort implementation.	
(*Dijkstra*)	MIBench	3586B (4KB)
	An implementation of Dijkstra's shortest path algorithm.	
(*Sha*)	MIBench	2432B (3KB)
	An implementation of the secure hash algorithm that produces a 160-bit message digest.	
(*ADPCM-enc*)	MediaBench	1624B (2KB)
	An implementation of the speech compression algorithm for adaptive differential pulse code modulation.	
(*ADPCM-dec*)	MediaBench	1624B (2KB)
	A decompresser of *ADPCM-enc*.	
(*Bitcount*)	MIBench	12420B (12KB)
	Tests bit manipulation capabilities of a processor by counting the number of bits in an array of integers.	
Combine	Synthetic	13684B (14KB)
	A multi-function program in which *Quicksort*, *Dijkstra*, *Sha*, *ADPCM-enc*, *ADPCM-dec*, and *Bitcount* are combined together.	
FFT	MIBench	15776B (16KB)
	An implementation of discrete Fourier transformation.	
Epic	MediaBench	32196B (32KB)
	An image data compression utility based on the bi-orthogonal critically-sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder.	
Unepic	MediaBench	23268B (23KB)
	A decompresser of*Epic*.	
MP3 (Mad)	MIBench	36912B (37KB)
	A decoder for the MP3 audio data format.	

Table 1: Applications used. *Combine* is a combination of *quicksort*, *dijkstra*, *sha*, *ADPCM-enc*, *ADPCM-dec*, and *bitcount*.

4. EVALUATION ENVIRONMENT

4.1 Applications

We evaluate our scheme using 10 embedded applications from MI bench [6] and Media bench [8]. The descriptions of the applications are summarized in Table 1. To reflect the situation of realistic embedded mobile applications to our evaluation, we converted the file I/O routines in the applications to the routines that access memory.

Since *Quicksort*,*Dijkstra*, *Sha*, *ADPCM-enc*, *ADPCM-dec*, and *Bitcount* have code size that is too small to evaluate our demand paging schemes, we combined them and made one synthetic application, *Combine*. In *Combine*, each of these 6 applications is executed (called) once. *Combine* represents multi-function embedded applications that execute one function at a time.

4.2 Simulation Environment

The evaluation is done using simulation. Our simulation environment is ARMulator[2]. The architecture modeled is ARM7 without caches. For the ARM7 architecture, ARMulator gives 100% cycle accurate simulation results. The clock is 20Mhz and the SRAM access time is 50ns in the simulation runs. The NAND flash read latency in Section 2.1 is fully considered in the simulation. The latency to read one page (512 bytes) from NAND cell to page register is 10us. Reading 256 16-bit words from the page register to SRAM takes 256 × 50ns = 12.8us.

5. EXPERIMENTAL RESULTS

5.1 Performance

Figure 10 compares the execution time of each application under different paging schemes with various SRAM sizes: basic demand paging (*Base*), demand paging with static clustering (*Static*), and demand paging with profiling information (*Profile*). The SRAM size varies from (page manager size + resident segment size + the maximum non-resident segment size) to the code size in the original image. For each application, the bars represent the execution time under specific SRAM size. No bar for one of the clustering schemes means that we cannot run the segmented image under the scheme due to the code size. The bars are normalized to the execution time of the application when the entire image fits in the SRAM (i.e., without paging). Figure 10 also shows the page hit rate of each application under different clustering schemes.

Base slows down each application significantly. It suffers from many page misses and indirect page manager call overhead. However, it reduces the SRAM size by about 50% compared to the size required by the original image. In particular, *Base* is very effective for *MP3*. This is because *MP3* has well divided phases in its running sequence.

Static performs very well for all but *FFT*. Except for *FFT*, its performance degradation is less than 20%. For *Epic* and *Unepic*, performance degradation is less than 10%. We see that the static clustering technique significantly reduces the number of indirect page manager calls and page misses. *Profile* performs almost equally well with *Static*. This is also due to the reduction of the number of indirect page manager calls and page misses by pinning some segments the area in SRAM.

Our paging schemes degrade the performance of *FFT* a lot because of page misses. Many page misses occur because *FFT*'s working set requires that the SRAM size be equal to the original image size. This type of application is not well suited to our demand paging scheme. However, for applications with well divided phases, such as *Combine*, *Epic*, *Unepic*, and *MP3*, our paging scheme is very effective.

Table 2 summarizes the smallest SRAM size with less than 20% slow down for each application with our compiler-assisted demand paging techniques. For *FFT*, we choose an SRAM size equal to the original image size and we do not apply demand paging.

Application	Original Code Size	SRAM size with demand paging (ratio to the original)	
Combine	14KB	10KB	(71%)
FFT	16KB	16KB	(100%)
Epic	32KB	16KB	(50%)
Unepic	23KB	18KB	(78%)
MP3	37KB	22KB	(59%)
Average	24.4KB	16.4KB	(67%)

Table 2: The smallest SRAM size with less than 20% slow down.

5.2 Effects of Page Manager Calls

Figure 11 shows the number of page manager calls executed for each application for each scheme. The number is normalized to the number in *Base*. *Static* has the smallest number due to clustering related functions together in a segment and pinning the functions with multiple parents in the resident segment. This is why *Static* often performs better than *Profile*. *Profile* is able to reduce the number of page manager calls significantly by pinning some functions in the resident segment. Note that control does not need to go through the page manager if the target is in the resident segment.

Application	Normalized Energy Consumption
Combine	1.16
FFT	11.77
Epic	1.08
Unepic	1.02
MP3	1.14

Table 3: Energy consumed by each application at the SRAM size specified in Table 2.

5.3 Energy Consumption

In this paper, we focus on the energy consumed by the CPU and memory components: SRAM and NAND flash memory. In addition, we consider only the energy consumed by the CPU and memory when they are active because energy-aware embedded systems can selectively place idle components into lower power states[16, 15].

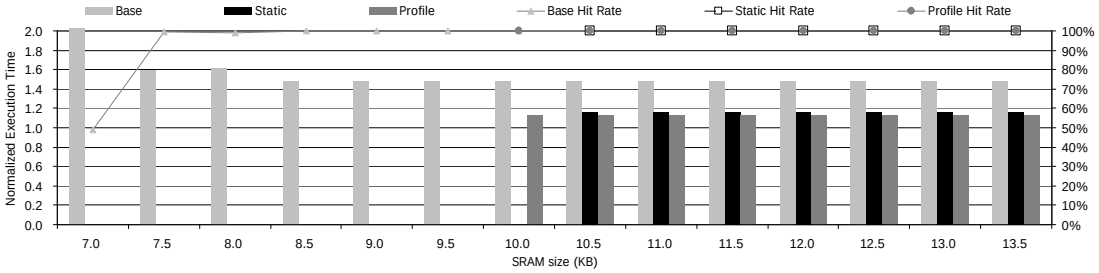
In the case of memory components, the number of memory accesses is directly proportional to the active power specified in the data sheet provided by the vendor[4, 13]. Thus, we can estimate the energy consumption E using the following:

$$E = E_{cpu} + E_{memory}$$

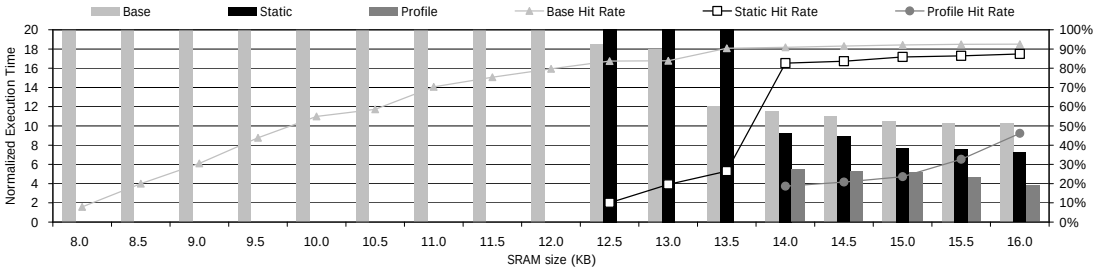
$$E_{cpu} = t_{active} \times P_{cpu}$$

$$E_{memory} = N_{access} \times E_{active}$$

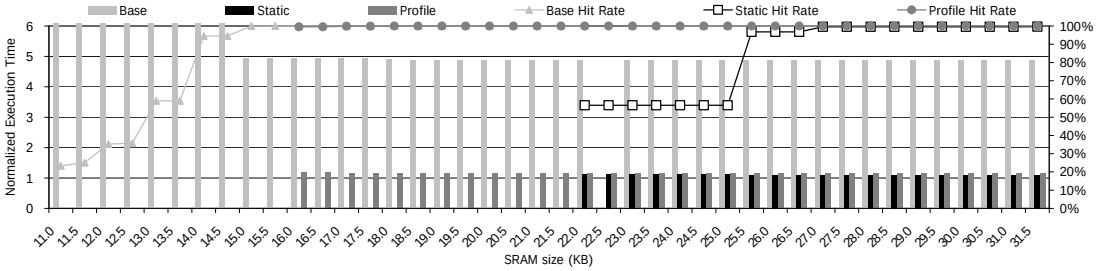
where t_{active} is the total amount of time when it executes instructions, and P_{cpu} is the average power consumed by



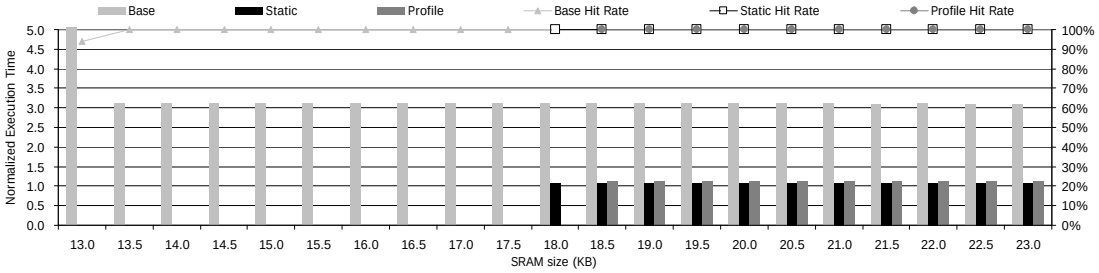
(a) *Combine*



(b) *FFT*



(c) *Epic*



(d) *Unepic*

(e) *MP3*

Figure 10: Execution time and page hit rates for different clustering schemes.

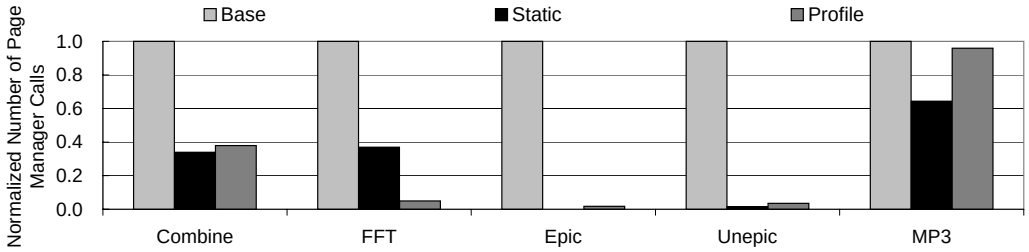


Figure 11: The number of page manager calls executed.

the CPU in this period. E_{active} is the average energy consumed by one memory access, and N_{access} is the number of read and write operations to memory. P_{cpu} and E_{active} are obtained by the data sheet provided by the vendor[4, 13]. N_{access} and t_{active} can be obtained from simulation runs with ARMulator for each application.

Table 3 shows the estimated energy consumption using the formula. To compute the numbers, we used the SRAM size shown in Table 2 with our best demand paging schemes. The energy consumption is normalized to the case of the entire original image running on SRAM. The energy consumption of *FFT* is high with our demand paging schemes. This is due to prolonged execution time caused by the demand paging. The system designer can run this type of application without applying demand paging and with the SRAM of the original code size when it is necessary. Other than that, our demand paging consumes about 10% more energy, on average, than without paging.

5.4 Real-Time Constraints

It is not easy to distinguish time-critical code from the given binary image. A straightforward solution is to make programmers provide the post-pass optimizer with some hints or pinning information of the time-critical code sections. The sections are locked in SRAM and not to be paged when the application is running. On the other hand, the system designer can choose the SRAM size and the paging schemes depending on their cost, energy, and real-time requirements using the performance data in Figure 10.

Overall, we see that, with compiler-assisted demand paging, we can reduce the SRAM size by 33% on average for most of the applications, with 8-20% of performance degradation and 10% of more energy consumption compared with no demand paging. This shows that our demand paging is very effective at reducing the code memory size in low-end embedded systems.

6. RELATED WORK

To obtain better code memory space utilization, much work has been done and the results have been used practically. The idea of code overlay techniques is to load code and data in memory when needed[14]. The user needs to explicitly implement the overlay structure in the application. It may be very difficult for users to obtain deep knowledge of the application program structure. Moreover, if the function call depth is large and exceeds the amount of memory allocated to the application, it is impossible to run the application with the conventional overlays. Our compiler-assisted demand paging is automatic and can run large programs in limited physical memory. It does not depend on the call depth of the application.

Virtual memory management in a typical operating system is an automatic technique that allows code to execute in the physical memory smaller than the code size. The main advantage of virtual memory is that programs can be much larger than the physical memory. However, this technique requires operating system or hardware (MMU) support for efficient implementation. This approach may not be suitable for low-end embedded systems with limited resources.

Park *et al.* [12] proposed NAND execute-in-place (XIP) to allow direct code execution from NAND flash memory with cache controller support. In this approach the additional cost incurred by the hardware cache controller may not be acceptable to low-end embedded systems. Park *et al.* [11] characterized conventional paging mechanisms for energy consumption when paging was applied to embedded applications stored in NAND flash memory. They proposed an energy-aware page replacement policy for the applications stored in NAND flash memory. Their approach requires virtual memory or hardware (MMU) support. Tomiyama and Yasuura [18] proposed code placement techniques that reduce instruction cache miss rate. They used integer linear programming and profiling information (traces) to place code in main memory in such a way that the instruction cache misses are minimized. Our code clustering techniques use both static and dynamic call-graph information and pin some code sections in memory.

Recently, compiler techniques that exploit software-exposed speed-differentiated memory (a.k.a. scratch-pad memory) were proposed[3, 17, 20, 7, 10, 5]. Scratch-pad memory is part of heterogeneous memory in the system. Avissar and Barua [3] proposed an automatic data partitioning algorithm between heterogeneous memory units to improve performance. Steinke *et al.* [17] proposed a selection algorithm using a compiler. The algorithm selects beneficial program parts and variables when they are placed in scratch-pad memory to save energy. Verma *et al.* [20] proposed an algorithm that can be applied to embedded systems with caches in addition to scratch-pad memory in order to reduce energy consumed by instruction memory. Jain *et al.* [7] introduced methods to improve cache predictability and performance using application-specific cache replacement mechanisms based on the information obtained by software assistance. Panda *et al.* [10] introduced techniques to partition on-chip memory into

scratch-pad memory and cache. They assign critical data in the program into the scratch-pad memory to improve performance. Francesco *et al.* [5] proposed a mechanism that exploits direct memory access (DMA) engines and application programmer's interfaces (APIs) to reduce memory copy cost occurred in transferring data between heterogeneous memory units. Our approach is similar to theirs in that we focus on embedded systems with heterogeneous memory. However, our goal is to run code that is bigger than the available memory without significant performance degradation and with comparable energy consumption.

7. CONCLUSIONS

This paper introduced a compiler-assisted demand paging technique for low-end embedded systems. This technique is essential to low-end embedded systems with limited memory and no operating system or hardware support.

Our experimental results showed that the proposed approach is cost-effective and promising. Our approach reduces code memory size by 33% on average for most of the applications with 8-20% performance degradation and 10% more energy consumption on average. The embedded system designer can choose the SRAM size and the paging schemes we proposed depending on their cost, energy, and real-time requirements.

In the near future, we plan to extend our work to support demand paging in embedded systems with memory management units or a real time operating system that does not have virtual memory. Also, we plan to investigate more sophisticated code clustering techniques for demand paging.

8. REFERENCES

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[2] ARM. *ARM Developer Suite Version 1.2: ARM Debug Target Guide*. ARM Limited, 2001.

[3] Oren Avissar and Rajeev Barua. An Optimal Memory Allocation Scheme for Scratchpad-Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[4] Samsung Electronics Co. NAND Flash Memory and SmartMedia Data Book. 2002.

[5] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In *The 41st Design Automation Conference (DAC 2004)*, pages 238–243, June 2004.

[6] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Aommercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, December 1998.

[7] Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. Software-assisted Cache Replacement Mechanisms for Embedded Systems. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer Aided Design*, pages 119–126, 2001.

[8] Chunho Lee, Miograg Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[9] M-Systems. Two Technologies Compared: NOR vs. NAND. White Paper, 91-SR-012-04-8L, Rev 1.1, July 2003.

[10] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[11] Chanik Park, Jeong-Uk Kang, Seon-Yeong Park, and Jin-Soo Kim. Energy-Aware Demand Paging on NAND Flash-based Embedded Storages. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED 2004)*, August 2004.

[12] Chanik Park, Jaeyu Seo, Sunghwan Bae, Hyojun Kim, Shinhan Kim, and Bumsoo Kim. A Low-Cost Memory Architecture with NAND XIP for Mobile Embedded Systems. In *ISSS+CODES 2003: First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, October 2003.

[13] Samsung Electronics Co. http://www.samsung.com/products/semiconductor/asic. 2004.

[14] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley and Sons, Inc., 2003.

[15] Tajana Simunic, Luca Benini, Peter Glynn, and Giovanni De Micheli. Dynamic Power Management for Portable Systems. In *Proceedings of the 6th International Conference on Mobile Computing and Networking*, pages 22–32, 2000.

[16] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-Efficient Design of Battery-Powered Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):15–28, 2001.

[17] Stefen Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Design, Automation and Test in Europ Conference and Exposition (DATE 2002)*, pages 409–417, Feb 2002.

[18] Hiroyuki Tomiyama and Hiroto Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *IEEE Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[19] Toshiba America Electronic Components, Inc. Cost Savings with NAND Shadowing Reference Design with Motorola MPC8260 and Toshiba CompactFlash. 2002.

[20] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Design, Automation and Test in Europ Conference and Exposition (DATE 2004)*, pages 1264–1269, Feb 2004.