A Software Reproduction of Virtual Memory for Deeply Embedded Systems

Conference Paper · May 2006

DOI: 10.1007/11751540_109 · Source: DBLP

6 authors, including:

Keun Soo Yim
Google Inc.
31 PUBLICATIONS **318** CITATIONS
SEE PROFILE

JeongJoon Yoo
Samsung Advanced Institute of Technology
31 PUBLICATIONS **74** CITATIONS
SEE PROFILE

Yeonseung Ryu
Myongji University, Yongin, South Korea
58 PUBLICATIONS **373** CITATIONS
SEE PROFILE

A Software Reproduction of Virtual Memory for Deeply Embedded Systems

Keun Soo Yim[a,*], Jae Don Lee[a], Jungkeun Park[a],
Jeong-Joon Yoo[a], Chaeseok Im[a], and Yeonseung Ryu[b]

[a]Computing Lab., Samsung Advanced Institute of Technology, Yongin 449-712, Korea
[b]Dept. of Computer Software, Myongji University, Yongin 449-728, Korea
*keunsoo.yim@samsung.com

Abstract. Both the hardware cost and power consumption of computer systems heavily depend on the size of main memory, namely DRAM. This becomes important especially in tiny embedded systems (e.g., micro sensors) since they are produced in a large-scale and have to operate as long as possible, e.g., ten years. Although several methods have been developed to reduce the program code and data size, most of them need extra hardware devices, making them unsuitable for the tiny systems. For example, virtual memory system needs both MMU and TLB devices to execute large-size program on a small memory. This paper presents a software reproduction of the virtual memory system especially focusing on paging mechanism. In order to logically expand the physical memory space, the proposed method compacts, compresses, and swaps in/out heap memory blocks, which typically form over half of the whole memory size. A prototype implementation verifies that the proposed method can expand memory capacity by over twice. As a result, large size programs run in parallel with a reasonable overhead, comparable to that of hardware-based VM systems.

Keywords. Embedded system, memory system, and heap management.

1 Introduction

Embedded computer systems are used in home and mobile consumer electronics. Specifically, embedded systems are used not only to control electric motors and LEDs but also to process multimedia data and communication signals. As the electronics are mass produced, it is important to reduce the hardware cost of the embedded systems. An embedded system consists of a microprocessor, code and data memories, and I/O devices. The microprocessor and I/O devices have the irreplaceable capabilities which make us hard to remove them from the system. As a result, memory devices are often directed as an optimization point for lowering the *hardware cost*.

Optimizing the memory size has another benefit in terms of *power consumption*. In near future, the electronics will utilize various physical statuses to work more intelligent. For collecting and controlling the statuses, tiny embedded systems will be used. An example of these tiny systems is micro sensors [1], which are installed in various points where power cable is not reachable. These sensors thus cannot but rely on

battery energy. In contrast, they are requested to work as long as lifetime of the electronics or buildings installed, typically more than ten years. It is sometime impossible to check and replace the battery of all distributed sensors. Thus, efficient management of given battery energy is important. Among the hardware components of a sensor, the energy use of a microprocessor, a code memory (e.g., ROM), and I/O devices can be reduced significantly while they are not being used. On the other hand, a data memory (e.g., DRAM) dissipates a large amount of energy although it is idling due to its refresh operation for all bit lines. Thus, reducing the data memory size is an important design objective in the tiny embedded systems.

There exist many techniques that can reduce the data memory size. Most of them need extra hardware, which are not applicable to the tiny systems. For example, virtual memory (VM) system needs a memory management unit (MMU) and translation lookaside buffer (TLB) [2, 3]. However, MMU and TLB are not supported in a majority of embedded processors (over 90%) [4] due to the following three reasons. One is that the VM hardware is difficult to design as well as forms large silicon area if designed. Another is that as VM hardware checks addresses generated by every instruction, a significant amount of energy is dissipated. The other is that TLB is a source of variable memory access time, making it hard to guarantee real-time constraints. Thus, recently a software reproduction of a portion of VM functionality was presented that is able to check stack overflow and resizes the stack adaptively [5].

Data memory consists of three main areas: global, heap, and stack, and the heap forms over half of total data memory space. Thus, in this paper, we present a software reproduction of VM functionalities specifically for the heap. The reproduced functionalities include heap compaction, compression, and paging. We define a novel heap API used in cooperation with the existing API of *malloc* and *free*. The novel API is designed to achieve the following two. One is abstracting the physical address of an allocated heap block by a logical ID, and the other is identifying unused heap blocks or unused parts of a heap block. When the ratio of free memory space is lower than a specified threshold, the proposed method tries to compact and compress the identified unused memory parts. When both two methods are impossible or inefficient, paging is used that swaps out the least recently used one among the unused to a flash storage. To improve the performance, small size parts are ignored from the compression and paging operations. To verify the effectiveness of the method, we have developed a prototype system on an ARM-based embedded machine [6]. The experimental results verify that the proposed method can expand physical memory capacity significantly so that large size programs run concurrently with a reasonable overhead.

The rest of this paper is organized as follows. Section 2 reviews the previous works in embedded systems area. In Section 3, we describe both the overall organization and specific mechanisms of the proposed method. We then summarize the experimental results in Section 4. Finally, Section 5 makes a conclusion with future directions.

2 Related Work

This section summarizes the prior works reducing data memory size. The prior methods are classified into three categories by the type of memory data: global variable,

stack, and heap. First, Biswas *et al.* [6] reused unused parts of *global* variable area as a stack or a heap. In compile time, they identified dead or used parts of the global area during the execution of specific functions in a program. The dead parts are reused directly when a memory overflow occurs, and the unused parts are reused by preserving the original data through a data compression technique.

Second, Middha *et al.* [5] mimicked a *stack*-related portion of VM functionality in software. They put the inspection codes to the entry and exit points of all functions. In the entry, the codes check whether a stack overflow will occur in the function or not. If it predicts an overflow, it allocates a memory block and extends the stack into the allocated. In the exit point, it tests whether the stack was extended at the entry or not. If the stack was extended, it restores the stack pointer properly and then frees the allocated block. Thus, the stack space is managed in a space-efficient manner while preventing a stack overflow.

Third, in embedded systems, heap area however was seldom managed in a space-efficient manner although heap typically forms over half of the whole memory space. Fundamental methods, which manage heap area in a space-efficient manner, are described as follows. All the followings need a support of VM hardware.

Heap compaction: Heap frequently allocates memory blocks and frees the allocated. As a result, a large portion of heap area is wasted by both internal and external fragmentations [7]. Heap compaction addresses these fragmented areas by compacting the allocated blocks. As the physical address of blocks is changed after compaction, compaction typically requires the address translation mechanism of VM. Specifically, VM translates logical address to physical one. Thus, after the compaction, the virtual address can be modified to direct the changed physical address.

Paging: Embedded systems consist of a small size of data memory. For example, the size is 16KB in a washing machine and is between 512 bytes and 2KB in a sensor node [1]. This severely limits both the capacity and functionality of the systems. Paging can logically expand the memory space by evicting less-frequently used pages to storage and by allocating the obtained space for the other tasks. When the evicted page is accessed, the page is then reloaded into the memory. Typically, VM hardware (e.g., MMU and TLB) is used to implement this paging method [2, 3, 7].

In order to avoid a use of VM hardware, a software method was developed for code memory [10]. It divides a program code in a basic block and replaces all branch instructions that cross two basic blocks by a system call instruction in a similar way that dynamic linking method does [18]. Thus, the program generates a system call whenever it wants to jump another block. Then operating system (OS) checks whether the wanted is loaded in memory or not and loads the wanted to the memory if it is absent. The system call instruction is then changed to an original branch instruction for speed-up. However, it is not appropriate for data memory because data does not have a flow of control that it can intercept as a system call. Thus, this paper presents a software-defined paging system for data memory.

Data compression: Compression reduces the data size and thus expands the memory space. Specifically, OS divides main memory into two parts: uncompressed and compressed [8, 9]. When the physical memory space is not enough, OS compresses a page that has a weak locality and stores it to the compressed part. If compression and decompression can be done quicker than a paging operation to secondary storage, this

method improves the overall performance. Since paging is basically needed, this compression method also needs a support of VM. Although compression method was first developed for high-performance computers, it is also effective at embedded systems that have flash memory storages. This is because if the paging method is used then the lifetime flash memory is seriously shortened. Compression addresses the reduction of flash lifetime.

The above three features are supported in the proposed method without relying on any extra hardware device as described in Section 3.

3 Proposed Software Virtual Memory

This section describes the proposed software virtual memory (SWVM) [12]. Figure 1 shows the overall architecture of a SWVM-based software platform. Legacy applications still use a typical heap allocator. SWVM is based on the same allocator in order to ensure the backward compatibility. At the same time, translated applications use the novel SWVM interface to utilize compaction, compression, and paging features. The interface and detailed mechanism are given in Section 3.1 and 3.2, respectively.

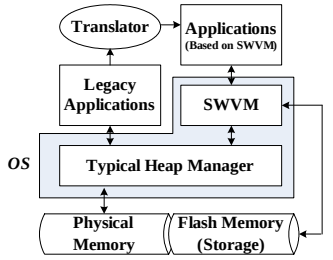

Fig. 1. System architecture.

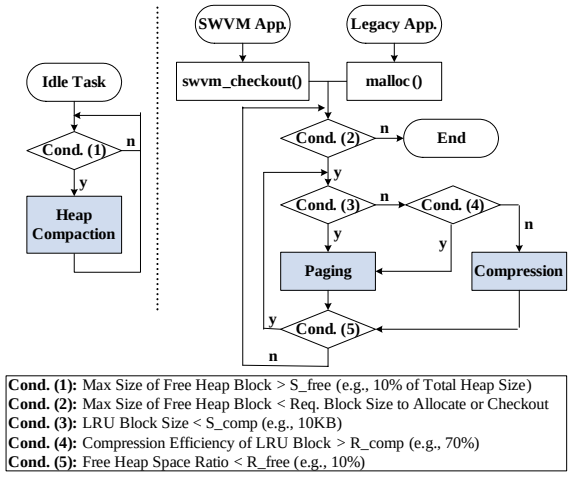

Fig. 2. Algorithm flowchart.

3.1. System interface

The novel interface described in code (1) enables the following two features. One is abstracting the physical address of an allocated heap block by a *handle ID*, and the other is identifying both unused heap blocks and unused parts of a heap blocks.

As an alternative of *malloc(), swvm_alloc()* is defined in which *block size* and *option* are used as parameter. *Option* is used to provide extra information to SWVM.

For example, it can specify that compression is preferred as the block typically has the high compression efficiency. *swvm_alloc()* returns a *handle ID* of a allocated block instead of a physical address of the block. *Handle ID* is used by *swvm_checkout()* that loads the specific part of the memory block to physical memory. The loaded part is specified by its *offset* and *length* parameters, and the return value is a *physical address* of the loaded part. In this way, applications access the allocated memory block. When the loaded part will not be used for a specific period of time, the applications can call *swvm_checkin()* that reclaims the loaded part. Repeatedly, *swvm_checkout()* and *swvm_checkin()* can be called in many times. Finally the allocated block is freed by *swvm_free()*.

```
int  id; char *pt; int c; /* 0<=c<length */
id = swvm_alloc(size, opt);
pt = (char *) swvm_checkout(id, offset, length);
*(pt + c) += value; /* read & write operations */
swvm_checkin(pt);
swvm_free(id);
```
(1)

Conceptually, *handle ID (id)* and *physical address (pt)* correspond to the virtual and physical addresses in VM, respectively. Thus, a mapping table between *handle ID* and *physical address* is needed in a similar way that VM does. The mapping table can be constructed in two ways according to the granularity of requesting checkout and checkin operations. One is when only whole memory block can be checked out and in. In this case, the mapping table is a one dimensional array whose size is maximum number of *handle ID*. Each element in the array stores both the *status* and loaded *physical address* of a corresponding memory block. The other is when a partial checkout is allowed that can checkout only a part or multiple parts of each memory block. This needs a two dimensional array where first-level array directs a custom data structure for each memory block. The custom data structure keeps the *status* of memory block and the *address* of loaded parts.

The legacy applications and middleware can be translated to SWVM-based applications. Both programmer or source code analyzing tool can change the *malloc()* and *free()* to *swvm_malloc()* and *swvm_free()* as well as insert *swvm_checkout()* and *swvm_checkin()* to the boundary where the heap blocks are used. The boundary can be accurately identified by using the data flow analysis [11]. If translation cost is high, we can selectively apply the translation to programs that need large size memory.

3.2. Management polices

In the proposed interface, if blocks or parts of blocks are not checked out, this means that they are not in a use (namely, unused state). The unused can be compacted, compressed, or swapped out, and they can be decompressed or swapped in before checking them out. However, when a block or its part is in an unused state for a specific period of time, we classify the block as a least recently used (LRU) block where the above three methods are applied in SWVM. This reduces an overhead of decompression and swap in operations because of the fact that programs typically exhibit a strong tendency in the memory access pattern. For example, a block used in the near past has a high probability to be accessed in the near future (namely, temporal local-

ity). Based on this, *swvm_checkin()* inserts its parameter memory block (or part) to a global LRU queue with the current system time.

SWVM utilizes the LRU queue for expanding the effective memory capacity. Figure 2 gives an overall algorithm. The idle task in OS performs heap compaction if the maximum size of free heap block is lower than the threshold value S_{free}, e.g., 10% of total heap space size. The idle task selects the largest free heap block. It then tries to relocates the two blocks physically adjacent to the free block. This can be done only when the adjacent blocks are in the unused state. The relocated space is selected based on the best-fit policy. It then merges the free block and the space pre-used by the relocated blocks. This repeats the same procedure until the adjacent blocks are being used for other purposes. If the relocation and merging is not possible, the idle task selects next largest free block and repeats the same until it is preempted by another higher priority tasks.

On the other hand, compression and paging operations are triggered on demand if the maximum free heap block size is lower than the size of requested heap block in *swvm_checkout()* and *malloc()* functions. We pop a least recently used block from the LRU queue. If its size is smaller than S_{comp} (e.g., 10KB), it would have a low compression efficiency, degrading the value of compression. Thus, in this case, paging is used. Paging also is used when its compression efficiency is lower than R_{comp} (e.g., 70%). When a heap overflow occurs, we perform both compression and paging until the free space ratio becomes larger than R_{free} (e.g., 10%). Here, we prefer compression than paging by applying paging only if compression results in a low space benefit. This is because paging faces the fore said endurance problem in flash memory. Users can specify their preference of paging and compression for a particular heap block by using *option* in *swvm_alloc()*.

There are many compression algorithms. In SWVM, dictionary-based algorithms (e.g., Ziv-Lempel and X-Match [8]) are used because they provide the high compression efficiency for memory data and perform (de)compression quick. Paging needs a swap area in flash memory, and there are two methods for constructing the swap area. One is directly assigning a physical partition, and the other is using a file provided by flash file systems. The partitioning method typically incurs lower processing overhead than the file system-based method. However, partitioning results in a wear-leveling problem in flash memory because the number of performed erase operations in blocks in the swap partition would be different from that performed in other partitions in the same flash memory.

Figure 3 shows a state transition diagram of a heap block in SWVM. When a block is allocated, it is logically allocated but is not physically neither allocated nor initialized. When it is checked out, SWVM allocates it physically. Then, if it is checked in, it is maintained in an unused state and can be changed to the swapped out or compressed mode when heap overflow occurs. Blocks in the unused state can transit to the used and free states by *swvm_checkout()* and *swvm_free()*, respectively.

In SWVM, checkout granularity is either a whole heap block or parts of a heap block. If a part of a heap block is checked out, we call it a finer-grained checkout, which is based on four basic cases depicted in Figure 4. In Figure 4, a block is allocated in step (1) and a part of it is requested to be checked out in step (2), which is called as ‘subset’ case as a subset of a heap block is checked out. In step (3), the part

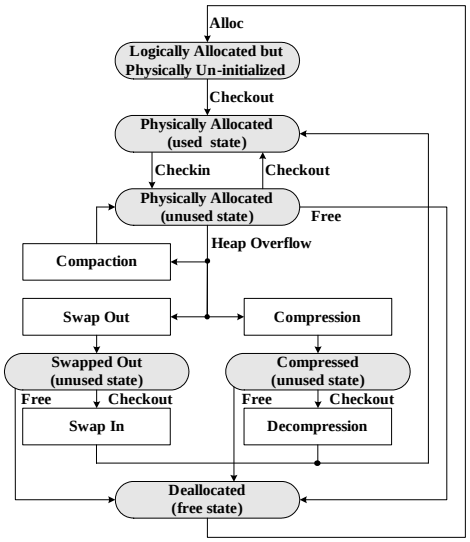

Fig. 3. State transition diagram of a heap block.

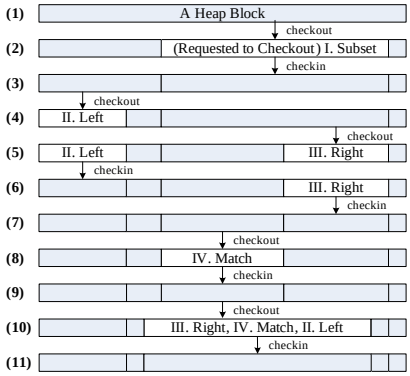

Fig. 4. Example of fine-grained checkout.

is checked in and this eventually produces three physical chunks for the heap block. In step (4), another part is checked out, which we call 'left' case as it is a left of a physical chunk. Then, in step (5), another part is again checked out, which is the 'right' part of a physical chunk. SWVM allows multiple parts of a heap block to be checked out simultaneously. Similarly, in step (8), an exactly matching part of a physical chunk is checked out, and we call this 'match' case. Based on these four basic cases 'subset', 'left', 'right', and 'match', when a checkout operation is requested to several physical chunks, we divide the operation to each physical chunk, perform individually in each chunk, and finally merges them. Step (10) shows an example where 'left', 'match', and 'right' cases are adopted in a single checkout operation. This is a conceptual description of SWVM management. In practice, SWVM also take into account the state of each physical chunk described in Figure (3).

4 Performance Evaluations

This section describes both the experimental setup and results. In the experiments, we used an ARM9 machine with 64MB of DRAM and 1GB of flash storage. To evaluate the performance accurately, we developed a prototype of SWVM on Samsung Multi-platform Kernel (SMK), a customizable real-time OS for embedded consumer electronics. SMK uses the Doug Lea's heap allocator [13] and Transactional File System version 4 (TFS4 [14]) as a flash file system. The code size of prototype is 2150 lines.

We designed a benchmark program that emulates various heap usage patterns according to its input parameters. *Total size* means the total size of allocated heap blocks, *init method* means a method used for initializing the blocks (e.g., random, constant, and none), *checkout unit* means the granularity of performing checkout in

SWVM (entire block: *coarse-grained*, part of a block: *fine-grained*), *access ratio* means the ratio of accessed heap block size over *total size*, and *locality* means the access locality of the blocks (e.g., temporal locality).

The benchmark program first allocates heap blocks where the block size is random generated within a value from 40KB to 200KB. It initializes the allocated blocks according to *init method*. Then it randomly accesses the allocated blocks according to the given parameter values, imitating both sequential and random memory access patterns. In this step, SWVM checks out an entire block and the exact part of the block to be accessed if *checkout unit* is coarse and fine, respectively. Finally, it frees all the allocated blocks.

First, we measured the execution time of each step in the benchmark program when *total size* is smaller than the physical memory size. Table 1 compares the execution time of SWVM and a typical method, the Doug Lea's allocator. Overall execution time of SWVM is comparable to that of the typical method. Specifically, in the allocation step, SWVM provides the faster speed as it logically allocates the requested heap blocks. In the init step, random is used as *init method*. The typical generates random values and writes them to heap blocks, while SWVM checkouts heap blocks, generates random values, writes the values to the blocks, and eventually checkins the blocks. Thus, SWVM works more however its execution time is slightly shorter than that of the typical. This is because both checkout and checkin operations are quite fast (e.g., 27ms if *total size* is 10MB) and in the experiment SWVM faced fewer cache conflict misses. SWVM allocates small metadata blocks in the heap, and this shifts the beginning address of the other heap blocks (like, cache coloring method in slab allocator [16]), potentially lessening the cache conflict misses.

Table 1. Memory system performance under random memory access pattern.

Method	Total Size MB (%)	Allocation Time (ms)	Init. Time (ms)	Access Time (ms (%))		Dealloc. T (ms)	
				CU=Coarse	CU=Fine	Coarse	Fine
Typical	1 (2)	1	140	74 (1.0)		1	
	10 (17)	3	1,397	739 (1.0)		2	
	58 (100)	13	8,098	4,275 (1.0)		11	
	59 (101)	Overflow	Overflow	Overflow		Overflow	
SWVM	1 (2)	1	138	73(1.0)	154(2.1)	0	1
	10 (17)	2	1,373	722(1.0)	1,550(2.1)	4	11
	58 (100)	10	7,967	4,202(1.0)	40,618(9.5)	23	198
	64 (110)	11	15,613	27,652(5.8)	61,639(13.0)	30	230
	70 (120)	12	25,322	37,214(7.2)	67,161(13.0)	33	238
	87 (150)	15	53,223	94,512(14.7)	107,497(16.7)	85	285
	116 (200)	20	100,478	149,819(17.4)	186,158(21.7)	118	336

* Init method: random; Checkout unit: *CU*; Access ratio: 100%; Locality: temporal; Overflow handling method in SWVM: Paging; S_{comp}: 10KB.

In the memory access step, when *checkout unit* is *coarse*, SWVM is slightly faster than the typical because of the same reason that it shows a better performance in the init step. However, when *checkout unit* is *fine*, its speed is about twice slower than that of the typical. This is because SWVM repeatedly checks out and in the specific parts of heap blocks, which incur many split and merge operations. Thus, the coarse-

grained checkout has the better performance if the memory access pattern is random. In the deallocation step, SWVM takes a longer time than the typical because it has to reallocate all the physical chunks.

Second, the execution of SWVM was measured when *total size* is larger than the available physical memory size (58MB). The physical memory size was 64MB and about 2MB was used for the code, bss, data, stack, and I/O buffer. Due to fragmentations, 58MB was the maximum allocable heap memory space. In this case, the typical faces a memory overflow as shown in Table 1 and thus the program was not able to run correctly. In the table, ratio in the access time field means the normalized access time against to that of the typical. On the other hand, the SWVM-based program overcomes the overflow problem with a reasonable degradation in the performance. With the coarse-grained *checkout unit*, its speed is degraded by about six to seventeen times when only ninety to fifty percent of the required memory space is available. In hardware-based VM systems, application performance is also dropped by about ten times if only half of the required memory space is available [15]. Thus, this implies that SWVM has a comparable performance to the VM systems.

Third, we measured the performance of SWVM with various parameter configurations of benchmark program. When we set *init method* to be constant, which initializes each heap block by using a unique value, the initialization time almost eliminated (e.g., less than 0.5 millisecond). This is because *swvm_alloc()* marks that the logically allocated block is initialized by a specific value, and the value is actually assigned to the block when *swvm_checkout()* is called. If checkout operation is requested to a particular part of the block, then only that particular part is initialized. Because of this, memory overflow can be significantly lessened in SWVM with constant *init method* as described in Table 2. For example, when *total size* is 87MB and *checkout unit* is coarse, the normalized access time is 1.0, which means that memory overhead was not occupied. Similarly, when *total size* is 64MB and *access ratio* is 200%, the normalized access time is 2.0 and 3.6 for coarse-grained and fine-grained *checkout unit*, respectively. With the random *init method*, the normalized access time was 5.8 and 13.0 when *total size* is 64MB and *access ratio* is 100%. Furthermore, this constant init method is also effective for compression method because it has the superb compression efficiency. In practice, several applications typically initialize the heap block by a zero value, strongly confessing the effectiveness of SWVM for real applications.

Table 2. Performance characteristics of SWVM.

Default Parameter	Custom Parameter	Value	Total Size MB (%)	Init. Time (ms)	Access Time (ms (%))	
					CU=Coarse	*CU*=Fine
A,B,C,D1,E	-	-	64 (110)	0	4,870 (1.0)	08,005 (1.7)
		-	87 (150)	0	6,610 (1.0)	11,024 (1.7)
		-	116 (200)	0	54,994 (6.4)	25,359 (2.9)
A,C,D1,E	Access Ratio	130%	64 (110)	0	6,291 (1.3)	10,727 (2.3)
		200%	64 (110)	0	9,568 (2.0)	17,259 (3.6)
		300%	64 (110)	0	29,297 (6.2)	49,161 (10.4)
A,B,C,D2,E	-	-	64 (110)	0	4,870 (1.0)	8,006 (1.7)

* Init method: constant(A); Access ratio: 100%(B); Locality: temporal(C); Overflow handling method: Paging(D1), Compression(D2); S_{comp}: 10KB(E); Checkout unit: *CU*.

5 Conclusion

In this paper, we have presented a software reproduction of VM functionalities, compaction, compression, and paging. The experimental results have shown that SWVM with the coarse-grained checkout has a comparable performance to a typical heap allocator when the requested memory size is smaller than the physical memory size. Furthermore if the requested is larger than the physical, SWVM-based applications run correctly with a reasonable performance degradation. Thus, SWVM is the most valuable for systems requiring a large memory space only in a short period of time. For future work, we plan to reproduce other VM features, e.g., protection and sharing.

References

1. J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, "The Mote Revolution: Low Power Wireless Sensor Network Devices," *In Proceedings of Hot Chips 16: A Symposium on High Performance Chips*, 2004.
2. U. Vahalia, *UNIX Internals: The New Frontiers*, Prentice Hall, 1996.
3. H. G. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett Pub., 1996.
4. D. Kleidermacher and M. Griglock, "Safety-Critical Operating Systems," *Embedded Systems Programming*, Vol. 14, No. 10, 2001.
5. B. Middha, M. Simpson, and R. Barua, "MTSS: Multi Task Stack Sharing for Embedded Systems," *In Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pp. 191-201, 2005.
6. S. Biswas, M. Simpson, and R. Barua, "Memory Overflow Protection for Embedded Systems using Run-time Checks, Reuse and Compression," *Proc. CASES*, pp. 280-291, 2004.
7. A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 7th Ed., John Wiley & Sons, 2004.
8. B. Abali, S. Xiaowei, H. Franke, D. E. Poff, and T. B. Smith, "Hardware Compressed Main Memory: Operating System Support and Performance Evaluation," *IEEE Transactions on Computers*, Vol. 50, Issue 11, pp. 1219-1233, 2001.
9. R. S. Castro, A. P. Lago, and D. D. Silva, "Adaptive Compressed Caching: Design and Implementation," *In Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (HPCA)*, 2003.
10. C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min, "Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory," *Proc. EMSOFT*, pp. 114-124, 2004.
11. A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
12. K. S. Yim *et al.* (Samsung Electronics), "Apparatus and Method for Controlling Virtual Memory," *Republic of Korea Patent*, Filed No. P2005-0107146, 2005.
13. D. Lea, "A Memory Allocator," *Germany UNIX/Mail*, Hanser Verlag, 1996.
14. Samsung Electronics, *TFS4 Programmer's Guide*, TR SEC-TFS4-PG0001, 2004.
15. H. Garcia-Molina and L. R. Rogers, "Performance through memory," *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 122-131, 1987.
16. J. Bonwick. "The Slab Allocator: An object-caching kernel memory allocator," *In Proceedings of the USENIX Summer Technical Conference*, pp. 87-98, 1994.
17. Samsung Electronics, *S3C2440A Application Notes (Development Kit for ARM9 SoC)*, Product Manual, 2004. (http://www.samsung.com and http://www.mcukorea.com)
18. J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann Publishers, 2000.