Compressed Memory Swap for QoS of Virtualized Embedded Systems

Jeaho Hwang, Jinkyu Jeong, Hwanju Kim, Jeonghwan Choi and Joonwon Lee

Abstract — *Virtualization has recently drawn attraction in smart consumer electronics as a way of completely isolating the main applications, which are in charge of the primary functionality of a device, from untrusted third-party applications. In a virtualized embedded device, flexible memory management is required to run multiple VMs efficiently on resource-constrained hardware. This paper presents an in-memory compressed swap device (CSW) for the virtualized consumer electronics environment. It swaps out only the memory of third-party applications in response to memory pressure on the main applications, to ensure its quality of service. To this end, CSW collaborates with memory ballooning, which is a scheme for dynamic memory partitioning between VMs. By compressing the swapped out memory pages, CSW can effectively provide memory to the main applications while preserving the availability of third-party applications. We implemented our scheme in a Xen-based virtualized system that has limited resources similar to conventional consumer electronics devices. The evaluation results show that CSW successfully provides memory to the main VM with a reasonable cost, while the third-party applications are not killed.*[1]

Index Terms — Smartphone, Memory compression, Memory saving, Virtualization

I. INTRODUCTION

A recent trend on consumer electronics is the development of '*smart devices*.' Smart devices, such as smartphones and smart TVs, are differentiated from past products by the adoption of open platforms, which run third-party applications as well as applications provided by device vendors. In this environment, both quality of service (QoS) and the reliability of the main applications, such as phone calling function and SMS function in smartphones, must be always prioritized higher than third-party applications [1]. Virtualization can fortify main applications by running them in a separated virtual machine from the untrusted third-party applications [14]. Due to the advantage of strong isolation, virtualization has drawn attention in consumer electronics recently [14], [20], [21], though it has been mostly used in servers and desktops for a decade.

Although virtualization effectively isolates main applications, it has been challenging to flexibly manage physical memory because it is usually statically partitioned. For example, if a virtual machine (VM) for the main applications (*main VM*) takes a specified maximum amount of memory for the main applications, a VM for the third-party applications (*3rd party VM*) could not be provided enough memory even though a part of the main VM's memory is usually idle. Although it can guarantee QoS of the main VM, it wastes memory and degrades the performance of third-party applications.

Memory ballooning [17], a dynamic memory partitioning scheme, was introduced to utilize memory in that situation. Using the balloon driver, the main VM can lend its idle memory to the 3rd party VM to improve the availability of the third-party applications. The 3rd party VM must return this memory when the main VM requests the memory. It could cause a trouble to the 3rd party VM if the lent memory is still in use.

A naïve solution to provide memory to the main VM in this situation is to kill some third-party applications. The reclaimed memory from the killed applications can be returned to the main VM. However, it sacrifices the whole availability of the third-party applications. By using a memory swap, memory can be provided to the main applications without killing applications in the 3rd party VM.

In this paper, we propose *Compressed SWap (CSW)*, an in-memory compressed swap device supporting virtualized embedded systems, to keep the availability of the third-party applications while QoS of the main applications is prioritized. By compressing memory pages we can make free memory to provide to the main VM. We use the main memory for the swap device because flash memory, which is equipped in various consumer electronics (CE) devices, is not suitable for a memory swap device [2]. Storing in the main memory also shows less latency than storing in flash so that the requested memory can be swiftly provided with less performance degradation of the third-party applications. Our work is differentiated from previous in-memory swap compression techniques by the following characteristics.

First, we target a CE device that adopts virtualization, by which the main and third-party applications are completely

[1] This work was partly supported by the IT R&D program of MKE/KEIT [KI001810041244 , SmartTV 2.0 Software Platform] and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2012-0000148). This work was also partly supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

J. Hwang, J. Jeong and H. Kim are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, South Korea (e-mail: jhhwang@calab.kaist.ac.kr, jinkyu@calab.kaist.ac.kr, hjukim@calab.kaist.ac.kr)

J. Choi and J. Lee are with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, South Korea (e-mail: jechoi@skky.edu, joonwon@skku.edu).

Contributed Paper
Manuscript received 07/15/12
Current version published 09/25/12
Electronic version published 09/25/12.

isolated from each other. In this environment, flexible memory management between the VMs is needed for efficient memory utilization. We introduce the necessity of an in-memory swap scheme for a virtualized CE device that employs flexible memory management.

Second, we particularly ensure QoS of the main applications. CSW activates only when the main VM requests the memory return and swaps out only the memory of the 3rd party VM. If the main VM swaps out its own memory pages instead of reclaiming the lent memory, it could degrade the performance of the main applications due to the long latency to swap in its swapped-out pages.

To this end, in CSW swapping-out is synchronized with memory ballooning, which is used to reclaim lent memory to the main VM. CSW handles memory pages to be swapped out only while the balloon driver is expanding its memory allocation for the main VM; this indicates the memory return. CSW does not work for the memory pressure of the 3rd party VM itself to avoid harmful effects on the main applications which can result from malfunctions of third-party applications.

The rest of this paper is organized as follows. Section II describes the background and related work of our scheme. In sections III and IV, we explain the design and the implementation of CSW respectively. The next section shows the evaluation results of our prototype, and we discuss our future direction of swap compression in Section VI. Finally we conclude in Section VII.

II. Background

A. Virtualization on Consumer Electronics

For a few years, virtualization has become mainstream in the server market. The capability of virtualization has enabled server consolidation which can run multiple operating systems in one physical machine simultaneously. Virtualization also improves the security of a system by isolating important data and applications from untrusted ones. Because of these properties, it is also attractive for CE devices [16].

The main advantage of VM consolidation in CE devices is improved system reliability. For example, when multiple mobile platforms, which are encapsulated in separated VMs, are integrated into a single smartphone, a main functionality, such as the phone-calling application is protected from untrusted 3rd party applications by leveraging the isolation capability of VM. In this environment, end users can make use of diverse mobile platforms without losing the reliability of the CE device. To exploit these advantages, many hardware and software platforms for embedded system virtualization are emerging these days; virtualization solutions for embedded systems have been released [20], and CPU manufacturers are introducing full virtualization support on their products for CE devices [16], as well as for servers and desktops.

Fig. 1 describes our target environment in which virtualization is adopted for the isolation of main applications. There are two VMs, main VM and 3rd party VM, consolidated on a thin virtual machine monitor (VMM). To preserve the functionality and availability of the main applications, the main VM is given the highest priority with regard to resource allocation.

To manage memory partitioning dynamically, a memory balloon driver [17] is installed in the 3rd party VM as a kernel module, which has a private channel to the VMM. To provide some memory in the 3rd party VM to the main VM, the balloon driver *inflates* its balloon to allocate free pages and provides them to the main VM. The balloon is *deflated* when the given memory pages are return to the 3rd party VM.

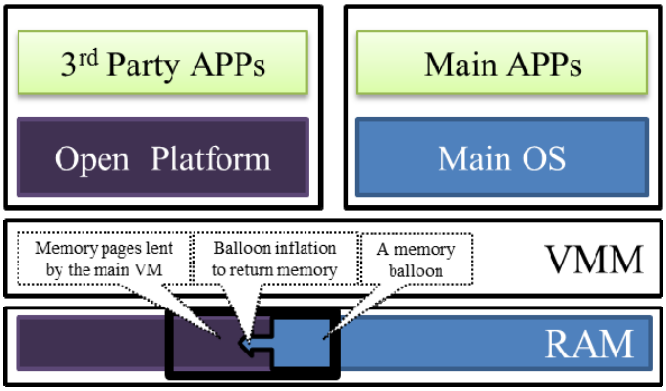


Fig. 1. Target environment

B. Motivation

To achieve two goals, flexible memory partitioning and prioritized memory provisioning for the main applications, the ballooning in our target works as follows. First, the partitioned memory size for the main VM is determined by the maximum memory requirement of the main applications in the main VM. The rest of the physical memory is given to the 3rd VM. This guarantees the QoS of the main applications by guaranteeing their required memory size. When the main application is idle, it does not require all the initially partitioned memory. Accordingly, the idle memory in the main VM is lent to the 3rd VM by deflating the balloon in the 3rd VM. This makes memory partitioning more flexible in the virtualized CE devices. When the main applications become busy and require more memory, the balloon in the 3rd VM inflates and returns the allocated memory (by the balloon) back to the main VM.

Note that size of the pre-inflated balloon is easily determined at the system development phase by analysis of the memory requirement of the main applications. We suppose that the main applications are trusted not to require more memory than the pre-inflated size.

If balloon inflation occurs when the 3rd party VM does not have enough reclaim able memory such as page caches, some 3rd party applications could be killed to make free memory, or some memory pages should be swapped out. Most CE devices, however, have no swap device. Therefore, we suggest an in-memory swap device that stores swapped-out pages in a compressed form. By preserving the virtual memory pages of the 3rd party VM in the compressed swap device, we can avoid killing 3rd party applications. We describe the detailed design and implementation in the following sections.

C. Related Work

Previous work on memory compression is categorized into two classes: hardware and software based memory compression [6]. Hardware-based memory compression techniques insert a hardware compression and decompression unit between cache and RAM. A memory page is uncompressed when it is transferred to a cache, and it is compressed on the fly when it is stored in memory [5], [10]. Despite the simple design and efficiency, this approach comes at a cost of additional hardware. In our work, we aim at a software-based approach without any additional hardware cost.

Many proposals for software-based memory compression schemes are classified into two categories: compressed caching and swap compression [6]. Compressed caching [7], [13], [15] uses a part of the memory to store compressed data as a cache between RAM and disk. The common goal is to improve system performance by decreasing the number of page faults that must be serviced by hard disks, which have much longer access latency than RAM.

Swap compression compresses swapped-out pages and stores them in a memory region [6], [8], [9]. Swap compression only caches swapped-out pages whereas compressed caching caches the whole system memory. Our work is also a kind of swap compression, and we target a virtualized environment on a CE device.

Yang et al. proposed CRAMES, which is a swap compression scheme for embedded systems [6]. They suggested a new compression algorithm and an adaptive memory management scheme. *Compcache* is another implementation of swap compression which is included in the Linux kernel. However, they still do not consider the virtualized environment and the QoS of the main applications comparing with our work.

III. DESIGN

In this section, we present the design of our compressed swap scheme, CSW, for a virtualized CE device. We first describe the mechanism of providing memory to the main VM using CSW. We then introduce a garbage collector, which is a supporting tool to manage memory efficiently.

A. The CSW mechanism

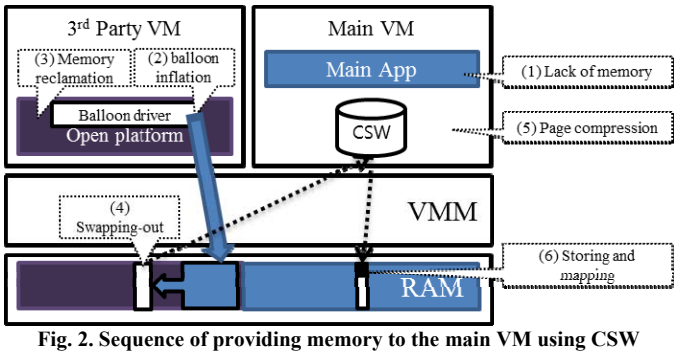


Fig. 2. Sequence of providing memory to the main VM using CSW

CSW works in response to a memory return operation requested by the main VM. When the free memory size of the main VM becomes less than a predefined threshold, the main VM starts to obtain memory from 3rd party VM as Fig. 2 shows. At first, the main VM sends an inflation request to the balloon driver in the 3rd party VM. Then, the balloon driver in the 3rd party VM starts to allocate memory for the main VM. If the 3rd party VM does not have enough free memory, it starts to reclaim its own memory pages, which should be swapped out to CSW when no more reclaimable pages are left. CSW sends the data of the page to the main VM and compresses it. Finally CSW stores the compressed data in the swap region in the main VM. After the swapping-out is finished, the balloon driver provides a vacant page, which results from the swapping-out, to the main VM.

CSW works only for providing memory to the main VM but not for the lack of memory in the 3rd party VM by itself. If CSW accepts swapped-out pages to reclaim memory for the third-party applications, it is against the policy of our goal, which aims at QoS of the main applications, since the swapped-out pages occupy the memory of the main VM. It could cause a harmful effect to the availability of the main applications because untrusted third-party applications could have bugs such as a memory leak. If an excessive number of pages produced from a memory leak of a third-party application are swapped out, it could cause a crash of the main applications, or even a crash of the main VM. Therefore, when a 3rd party application requests memory, the 3rd party VM must handle it by itself or kill its applications. The main applications, on the other hand, do not cause such excessive swapping, since they are verified not to exceed the specified maximum memory size.

To invoke swap-out operations in response to the memory return of the main VM, CSW works in synchronization with the balloon driver. When the balloon driver starts to inflate its balloon, which indicates that the main VM requires return of memory, CSW also becomes activated and handles swapping-out operations. It is deactivated after the inflation is finished, and refuses any swapping-out operation during other balloon states. In this way it can avoid swapping-out requests from the memory pressure of the 3rd party VM. It always handles swapping-in operations regardless of the state of the balloon.

The swap compression sequence in CSW is as follows. When a swapped-out page is sent to CSW, it first compresses the page into a buffer whose size is the same as the original page size, because we cannot predict the compressed size. Then, it allocates a swap region with the compressed size and copies the compressed page to the swap region. Finally it maps the logical offset of the swap device and the actual address of the swap region for the compressed page. When swap-in occurs, CSW locates a corresponding compressed page by looking up the mapping table, then decompresses it and sends it to the 3rd party VM.

CSW uses the LZO compression scheme [18], which is widely used for memory compression [3], [4], [12], to compress swapped-out pages. The LZO algorithm is a Lempel-Ziv implementation designed to be fast, especially for decompression. Using this algorithm, CSW can promptly serve compressed pages in response to swap-in requests [4].

Because compressed pages are stored in dynamically allocated swap regions, a mapping from the logical offset of the swap device to the actually stored address should be maintained. If compressed pages are stored in fixed-partitioned swap regions, memory space could be wasted by fragmentation, since the sizes of compressed pages are different. We use a hash table for this mapping; the offset is used as a key, and a value consists of the stored address and the size of the compressed page.

B. Swap Garbage Collection

When a swapped-out page becomes no longer referenced, the OS deletes the metadata for the page. However, the stale data stored in the swap device is not immediately deleted and remains as *swap garbage* until a new swapped-out page is overwritten in that region. It is not a problem in a general disk-based swap device since the device (or a partition) is dedicated to memory swap only. CSW, on the other hand, should clear the swap garbage since CSW resides in the memory of the main VM. If swap garbage in CSW is neglected, the memory of the main VM could be seriously wasted.

For this problem, we proposed a *swap garbage collector* in the OS for the 3rd party VM. It notifies the CSW when a swap-out page becomes unused. To reduce the cost of communication with the CSW, the offsets of swap garbage are buffered in a *garbage table*, and the table is sent when it is full. The CSW clears the swap garbage in the table.

IV. Implementation

In this section, we describe our implementation of the CSW. We implemented the prototype of CSW driver on Xen 3.4.0 and the prototype of a swap garbage collector on Linux kernel 2.6.18 for Xen. We configured domain 0 as the main VM and a domain U as the 3rd party VM.

A. A CSW driver

A CSW driver was implemented as a new *blktap* disk format by using the blktap userspace toolkit, which provides a user-level disk I/O interface for Xen environment [19], [20]. We implemented it by modifying the RAM disk format which is included in the blktap userspace toolkit. Fig. 3 describes the sequence of swapping-out to CSW which we implemented. When a swap I/O operation is requested by the 3rd party VM, the *blkfront*, which is the frontend driver of blktab in the 3rd party VM, sends the request and the data to the *blkback*, which is the backend driver of blktab in the main VM. After the blkback receives them, it calls an appropriate function in the CSW driver for the request, that is, write function for swapping-out and read function for swapping-in. After these functions are called, they carry out the sequence as described in the previous section: compressing the page, allocating the swap region and mapping it.

In addition, the first 4KB of a swap device is reserved for the swap device header. Therefore, when reading offset zero is requested, the CSW driver returns its header data instead of uncompressing the data for the offset zero. We use the *minilzo* library, which is also used in [3], for LZO compression and the *glibc* library for hash table implementation.

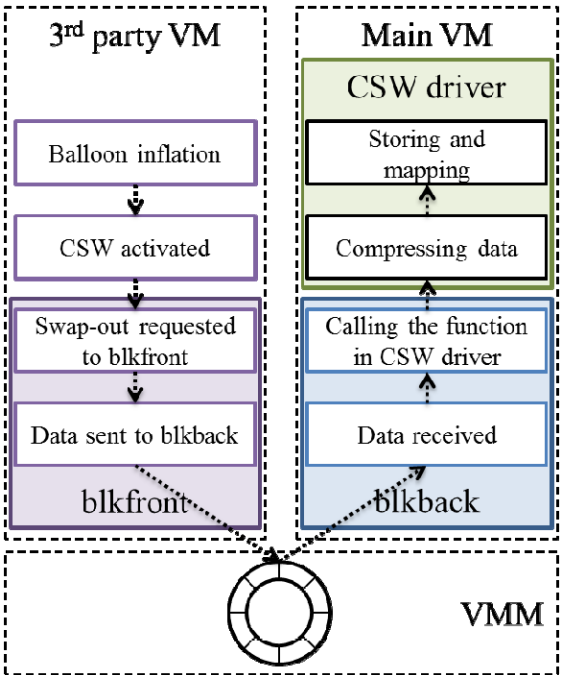


Fig. 3. Implementation of the CSW swap-out

B. Swap Garbage Collection

We implemented a new scheme for our prototype of a swap garbage collector, although a higher version of Linux kernel provides the callback function to notify the swap device that a swapped-out page is no longer used.

When a swap cache entry, a data structure for swap information in the Linux kernel, is no longer used, the offset of the swap garbage in that swap cache entry is inserted into a garbage table. If the garbage table becomes full, the table is written to offset zero of the compressed swap device. Offset zero is never written in a normal case because it is reserved for the swap device header. Therefore CSW regards a write request with offset zero as a garbage collection request, and then it clears data and mappings. The garbage table is cleared after swap garbage collection is finished. When a new swap cache entry is allocated to the offset which is in the garbage table, the offset is deleted from the table before it is transferred. The garbage table is also transferred periodically even though it is not full.

V. Evaluations

We evaluated our prototype on a system equipped with a 1.6GHz CPU for low-power devices, 512MB main memory and 640GB HDD. Although it is organized as a tiny PC, it can be used to mimic the consumer electronics environment because the processor is developed for a small mobile device, and CE products with similar specification have been released on the market recently.

We measured the performance of the system using CSW and compared it to the performance when using a hard disk as a swap device. We first ran a memory-intensive workload that reduced a 3.6MB jpeg image to 200KB in the 3rd party VM 256MB memory, and measured the compression rate of CSW, execution time of the workload and overhead of CSW. We

then ran the same workload repeatedly with reducing memory to 128MB by 32MB, to simulate balloon inflation and providing memory to the main VM.

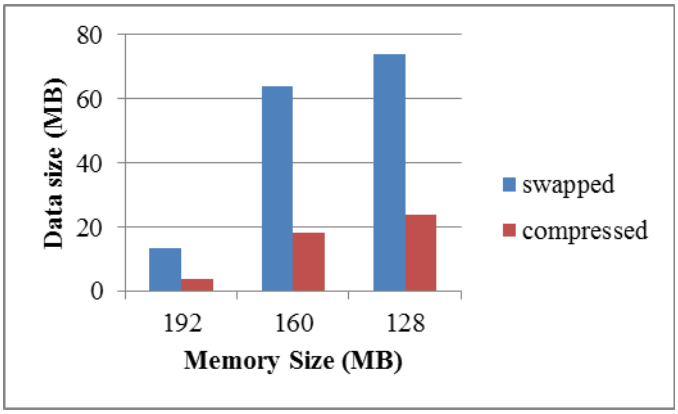


Fig. 4. Compressed size of swap pages

Fig. 4 describes the total size of swapped-out pages and their compressed size. Swapping-out occurs when the memory is less than 224MB. The results show that CSW reduces the size of swapped-out pages to 31% of their original size. We can expect that 72.5MB of the memory of the 3rd party VM should be swapped out to reclaim 50MB memory to return. After the workload is finished and the swap garbage from the workload is cleared, the size of swap data left on CSW is about 15% of the original size. We speculate that this reduced rate is because of the rarely used data which is mostly filled with zero.

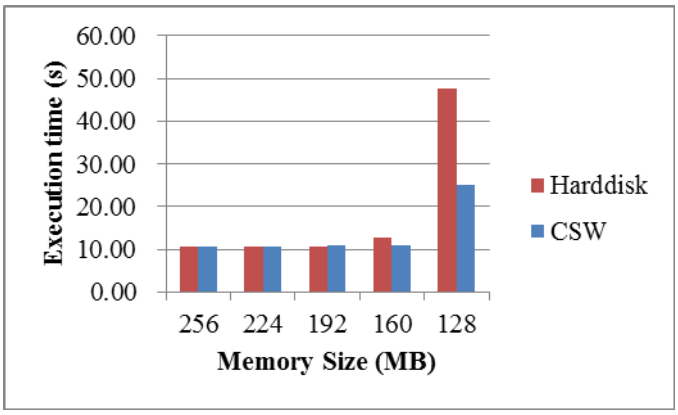


Fig. 5. Execution time to reduce image

Fig. 5 shows the execution time results of the experiment. It shows the same execution time with 256MB and 224MB system memory because they caused no swapping-out. Using CSW, the execution times running with 192MB and 160MB are similar to the result with no swapping out. This indicates that third-party application can maintain its execution while the main application is prioritized. The workload process is not even killed in 128MB, which shows memory thrashing, though the execution time is 2.5 times slower than in 256MB.

Furthermore, using CSW is twice as fast as using a hard disk. We demonstrated that faster swap I/O speed saves execution time. This indicates that the main VM can be served memory promptly as it requests the return of lent memory.

TABLE I
MAXIMUM CPU USAGE FOR EACH SWAP DEVICES

Swap Device	Maximum CPU usage
hard disk	15%
CSW	25%

CSW uses additional CPU for compression and decompression. We measured the additional CPU usage of the main VM, which carries out compression and decompression. As seen in TABLE I, it shows up to 25% CPU usage for CSW whereas 15% CPU usage is shown for hard disk. This 10% additional overhead is acceptable for achieving both main VM performance and 3rd party VM availability. We expect that it would be reduced with the advance of hardware performance in the future.

From the two evaluation results, we can determine that when the main application requires about 50MB of memory, the balloon in the 3rd VM inflates to about 72.5MB. In addition, this balloon inflation does not cause the 3rd party application to be killed, though it slows down the 3rd party application by 2.5 times (comparing the execution times of 224MB case and 128MB case) with 10% CPU overhead. We believe that this is reasonable because the evaluation results meet our goal: guaranteeing the required memory for the main VM and avoiding killing the 3rd party applications.

VI. DISCUSSION

A. CSW Implementation on Blktap

We implemented CSW on Xen blktap, which was running as a user-mode driver, as we explained in the previous section. Running the driver in the user mode has some advantages compared to kernel mode [19], [20]. At first, it is easy to change or improve its implementation. For example, we can simply change the compression library to another scheme such as *bzip2*. Since it shows a better compression ratio and requires more system resources than LZO, this change could be applied to a device with rich system resources. Implementing in kernel mode, however, the compression library must be compiled for the kernel on which CSW is running. In addition, the CSW driver can also be simply killed in an emergency situation since it is running as a user program.

B. Optimization of CSW

Our design and implementation of CSW would work better with several optimization techniques. First, the memory management of CSW could be improved. In our implementation, a swap region to store a compressed page is allocated using the *malloc()* function which could cause a fragmentation problem. By applying advanced management schemes such as extent, in which a fixed size unit consists of several memory pages [11], and the *xvmalloc* scheme, which is implemented for Compcache, CSW could save more memory. By recognizing zero pages before compression, we could also save more memory because we would not need to keep zero pages.

Selecting which VM to do computation for compression/decompression is an important decision for the performance of the main applications. In the implementation of

our current prototype, the computation is entirely offloaded to the main VM. In the case of a multi-core environment, the required computation is distributed to multiple cores. However, if the CPU for the main VM is busy, offloading could worsen the performance of the main VM. CSW could compress pages in the 3rd party VM in this case. When both VMs are too busy to perform compression, the 3rd party VM should take the compression job on its own in favor of the priority of the main application.

When the shortage of memory in the 3rd party VM is caused by a third-party application, the 3rd party VM must handle it by itself. In this situation, the 3rd party VM should kill some third-party applications. To prevent the killing of applications, the 3rd party VM could adopt another swap compression technique such as Compcache, which stores compressed data in the 3rd party VM's own memory. We do not consider it in this paper since we care primarily about the QoS of the main applications.

IX. CONCLUSION

In this paper, we proposed CSW, an in-memory compressed swap scheme supporting main applications in virtualized CE devices. CSW stores the compressed swapped-out pages of the 3rd party VM when its memory is being reclaimed to be provided to the main VM, so that the 3rd party VM can keep the running third-party applications from out-of-memory. Swap garbage collector sweeps swap garbage in the CSW efficiently using the memory space of swap regions. Based on the evaluation results, the CSW can reduce the total size of swapped-out memory to 31% of its original size, with 10% additional CPU overhead. It can also swap out memory twice as fast as a hard disk. These results imply that CSW successfully achieves our goal, keeping availability of third-party applications with providing its memory to the main applications swiftly.

Despite the acceptable CPU overhead, as discussed, it could pose a problem when both 3rd party VM and main VM are busy. Furthermore additional CPU overhead could also pose a power-consumption issue if CSW is applied to a battery-powered CE device. Adaptively deciding between compressed swapping and OOM-killer invocation while considering battery consumption would be an interesting issue. We regard these issues as our future work.

REFERENCES

[1] J. Hwang et al., "AppWatch: detecting kernel bug for protecting consumer electronics applications,", *IEEE Trans. Consumer Electron.*, vol.56, no.2, pp.687-694, May 2010

[2] O. Kwon and K. Koh, "Swap space management technique for portable consumer electronics with NAND flash memory," *IEEE Trans. Consumer Electron.*, vol.56, no.3, pp.1524-1531, Aug. 2010

[3] D. Gupta et al, "Difference engine: harnessing memory redundancy in virtual machines," in *Proc. 8th USENIX conf. Operating systems design and implementation*, San Diego, California, December 08-10, 2008, pp.309-322

[4] P. Wilson et al., "The case for compressed caching in virtual memory systems," in *Proc. USENIX Annu. Tech. Conf.*, Monterey, California, June 06-11, 1999, pp.8-8

[5] B. Abali et al., "Performance of Hardware Compressed Main Memory," in *proc. 7th Int. Symp. High-Performance Computer Architecture*, Nuevo Leone, Mexico, January 2001, pp.0073.

[6] L. Yang et al., "High-performance operating system controlled online memory compression," *ACM Trans. Embed. Comput. Syst.* 9, 4, Article 30, April 2010.

[7] I. Tuduce and T. Gross, "Adaptive main memory compression," In *Proc. USENIX Annu. Tech. Conf.,* Berkeley, California, USA, June 2005.

[8] T. Cortes et al., "Swap compression: resurrecting old ideas," *Softw. Pract. Exper.* 30, 5, pp.567-587, April 2000.

[9] S. Roy et al., "Improving System Performance with Compressed Memory," in *Proc. 15th Int. Parallel and Distributed Processing Symp.*, San Francisco, California, 2001.

[10] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," *SIGARCH Comput. Archit. News* 33, 2, pp.74-85, May 2005.

[11] T. Makatos et al., "Using transparent compression to improve SSD-based I/O caches," in *Proc. 5th European conf. Computer systems*, ACM, New York, NY, USA, April 2010, pp1-14.

[12] S. Park et al., "Compressed swapping or NAND flash memory based embedded systems," in *Proc. IEEE Workshop on Signal Processing Systems*, August 2003.

[13] R. de Castro et al., "Adaptive Compressed Caching: Design and Implementation," in *Proc. 15th Symp. Computer Architecture and High Performance Computing*, São Paulo/SP – Brazil, November 2003, pp. 10.

[14] G. Heiser, "Hypervisors for Consumer Electronics," *6th Annu. IEEE Consumer Communications and Networking Conf.*, Las Vegas, Nevada, 10-13 Jan. 2009, pp.1-5.

[15] F. Douglis, "The compression cache: Using on-line compression to extend physical memory," in *Proc. 1993 Winter USENIX Conf.*, San Diego, California, January 1993, pp. 519-529.

[16] R. Mijat and A. Nightingale, "Virtualization is Coming to a Platform Near You", *White paper*, ARM.

[17] C. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.* 36, SI, 181-194m, December 2002.

[18] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory,* vol.23, no.3, pp. 337- 343, May 1977.

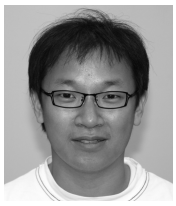
[19] D. Chisnall, *The definite guide to the Xen hypervisor*. New York: Prentice Hall, 2007.

[20] Ken Barr et al., "The VMware mobile virtualization platform: is that a hypervisor in your pocket?" *SIGOPS Oper. Syst. Rev,* vol.44, no.4, pp.124-135, December 2010.

[21] A. Suzuki, S. Oikawa, "Implementation of Virtual Machine Monitor for ARM Architecture," *IEEE 10th Int. Conf. Computer and Information Technology*, Bradford, UK, June 29-July 1 2010, pp.2244-2249.

BIOGRAPHIES

Jeaho Hwang received his BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2007, and MS degree in computer science from KAIST in 2009. He is currently a PhD candidate in the Computer Science Department, KAIST. His current research interests include operating systems, system reliability, virtualization and embedded systems.



Jinkyu Jeong received his BS degree from the Computer Science Department, Yonsei University, and the MS degree in computer science from the Korea Institute of Science and Technology (KAIST). He is currently a PhD candidate in the Computer Science Department, KAIST. His current research interests include real-time system, operating systems, virtualization and embedded systems.

Hwanju Kim received his BS degree in information and computer engineering from Ajou University, Korea, in 2006, and MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2008. He is a PhD in the Computer Science Department, KAIST. His research interests include virtual machine, embedded system, and storage system.

Jeonghwan Choi received the BS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1990 and 2007, respectively. He received MS degree in information and communication engineering from KAIST in 1997. He has been a research professor at Sungkyunkwan University since 2009. His research interests span the power management, system virtualization, and thermal management.

Joonwon Lee received his BS degree in computer science from Seoul National University in 1983 and the MS and PhD degrees from the Georgia Institute of Technology in 1990 and 1991, respectively. He is currently a professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was a professor at the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea from 1992 to 2008. His current research interests include low power embedded systems, system software, and virtual machines.