Symbiotic Dynamic Memory Balancing for Virtual Machines in Smart TV Systems

Junghoon Kim, Taehun Kim, Changwoo Min, Hyung Kook Jun, Soo Hyung Lee, Won-Tae Kim, and Young Ik Eom

Smart TV is expected to bring cloud services based on virtualization technologies to the home environment with hardware and software support. Although most physical resources can be shared among virtual machines (VMs) using a time sharing approach, allocating the proper amount of memory to VMs is still challenging. In this paper, we propose a novel mechanism to dynamically balance the memory allocation among VMs in virtualized Smart TV systems. In contrast to previous studies, where a virtual machine monitor (VMM) is solely responsible for estimating the working set size, our mechanism is symbiotic. Each VM periodically reports its memory usage pattern to the VMM. The VMM then predicts the future memory demand of each VM and rebalances the memory allocation among the VMs when necessary. Experimental results show that our mechanism improves performance by up to 18.28 times and reduces expensive memory swapping by up to 99.73% with negligible overheads (0.05% on average).

Keywords: Smart TV system, virtual machine, memory balancing, semantic gap, memory swapping.

Manuscript received Jan. 29, 2014; revised May 12, 2014; accepted May 29, 2014.

This work was supported by the IT R&D Program of MKE/KEIT [10035708, The Development of CPS (Cyber-Physical Systems) Core Technologies for High Confidential Autonomic Control Software].

Junghoon Kim (myhuni20@skku.edu), Changwoo Min (multics69@skku.edu), and Young Ik Eom (corresponding author, yieom@skku.edu) are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Rep. of Korea.

Taehun Kim (taehun.kim@navercorp.com) is with the System Operations Center, Naver Business Platform, Seongnam, Rep. of Korea.

Hyung Kook Jun (hkjun@etri.re.kr), Soo Hyung Lee (soohyung@etri.re.kr), and Won-Tae Kim (wtkim@etri.re.kr) are with the SW·Content Research Laboratory, ETRI, Daejeon, Rep. of Korea.

I. Introduction

Google released Smart TV in 2010. Since then, major manufacturers, such as Samsung, LG, and Sony, have invested in the development of Smart TV products. This trend shows that the TV has been transformed from a home appliance handling only broadcast media, to a full technology system that provides other valuable services, such as web browsing, online gaming, video streaming, and other Internet-based content [1]. Due to the paradigm shift toward the TV becoming a software-centric system, we expect that next-generation Smart TV systems will become ubiquitous in home environments by introducing cloud services based on virtualization technologies [2]–[3]. This is supported by the following hardware and software trends. First, the improvement of hardware performance has been accelerated (for example, multicore processors, 3D acceleration, high-speed networks, and various interfaces). Moreover, ARM processors, which are mainly used in embedded devices, now support hardware virtualization technologies [4]. Second, software solutions for supporting virtualized environments in embedded devices have been significantly studied [5]–[8].

In virtualized Smart TV systems, where a virtual machine monitor (VMM) is responsible for allocating physical resources to virtual machines (VMs), efficient resource management is a key success factor [9]. Even though most physical resources, such as a CPU and I/O devices, can be shared by multiplexing among VMs, allocating the proper amount of memory to VMs is still challenging [10]–[11]. Simply increasing the physical memory seems an easy solution. However, in embedded devices, such as a Smart TV, the installation of larger physical memory chips entails higher material cost and power consumption, so it deteriorates the competitiveness of products. To efficiently share the memory resource among VMs, previous studies have proposed dynamic memory balancing techniques while maintaining a reasonable quality of service (QoS). However, these techniques still have limitations. Statistical sampling [12] cannot estimate a memory demand larger than the current memory allocated to the VM. In addition, by increasing the number of sampling pages for the high estimation accuracy, the monitoring overhead linearly increases. Geiger [13] proposed techniques that can predict the page miss ratio and determine appropriate VM memory allocation by intercepting all I/O operations at the VMM level. However, it cannot estimate the working set size of a VM (that is, the amount of machine memory needed without causing significant memory swapping) smaller than its current memory allocation because, as in this case, it does not incur the buffer cache eviction. To cope with such limitation, Lu and Shen [14] proposed a hypervisor exclusive cache technique that can determine the growth and shrinkage of memory allocation. However, this technique cannot be used for an operating system (OS) without a source code, since it requires modification of the guest kernel. Zhao and others [10] and virtual machine memory balancer (VMMB) [11] proposed a least recently used (LRU) histogram–based [15] approach for estimating the working set size of each VM. They divided the VM memory into several sets, whereby a particular set is monitored for estimating the working set size of each VM. However, a workload that heavily accesses the memory causes a large monitoring overhead due to the frequent histogram updates. Furthermore, intercepting memory accesses involves mode switching between the VM and the VMM, which is the most expensive operation in virtualized environments [16]–[19].

In this paper, we propose a novel mechanism to dynamically balance the memory allocation among VMs. In contrast to previous studies, where a VMM is solely responsible for estimating and rebalancing the memory allocation, our approach is symbiotic. Each VM periodically reports its memory usage pattern to the VMM. Then, the VMM simply calculates the memory demands from the collected memory usage patterns and rebalances the memory allocation among the VMs. Our symbiotic approach is particularly useful for resource-limited embedded devices, such as Smart TVs, since it has a low runtime overhead regardless of workloads. We made the following specific contributions:

▪ First, we classify the representative workloads of a Smart TV and characterize the properties of each workload. On the basis of the analysis, we verify that the lack of runtime information of VMs within the VMM, called a semantic gap, incurs expensive memory swapping in the virtualized Smart TV system because the inactive memory of VMs is unable to be efficiently reclaimed. This result shows that dynamic memory balancing is essential for virtualized Smart TV systems.

▪ Second, we propose a novel and simple technique for obtaining memory usage patterns from VMs. Compared to previous studies that attempt to determine VM semantics for estimation of the working set size, our mechanism predicts the memory demands based on the memory usage patterns provided by the VMs. To do this, a monitor driver, which runs in a guest kernel, periodically posts memory usage information of a VM to the VMM. This can be performed without modification of the guest kernel because the monitor driver is a typical loadable driver (for example, a balloon driver [12]). Therefore, our mechanism accurately measures the working set of each VM with negligible overhead regardless of the working set size and memory access pattern of workloads.

▪ Finally, we propose a novel technique to predict the memory demand of each VM dynamically based on exponentially-weighted moving averages [20]. This technique can predict the memory demand, whether or not each VM will require a memory resource larger than the current allocation. Therefore, our mechanism can prevent expensive memory swapping in advance and improve overall performance.

The remainder of the paper is organized as follows. Section II analyzes Smart TV workloads and describes the memory management problem in a virtualized Smart TV system. In Section III, we describe the detailed design of our dynamic memory balancing mechanism. In Section IV, we show experimental results of our mechanism and verify that our mechanism is practical in a virtualized Smart TV system by comparing with previous studies. Section V presents the related work. Finally, in Section VI, we conclude the paper and suggest future directions.

II. Analysis of Smart TV Workloads

Smart TV has changed the function of the TV by providing various services such as video playback, web browsing, video streaming, and online gaming. For this reason, an analysis of Smart TV workloads should be conducted first to provide efficient memory management in a virtualized Smart TV system. In this section, we analyze memory usage patterns of the representative workloads in Smart TV. We then show why efficient memory management is essential for a virtualized Smart TV system.

1. Memory Usage Patterns of Smart TV Workloads

We first analyze four representative workloads in Smart TV, including video playback, video streaming, web browsing, and online gaming. Details of each workload are as follows. In the

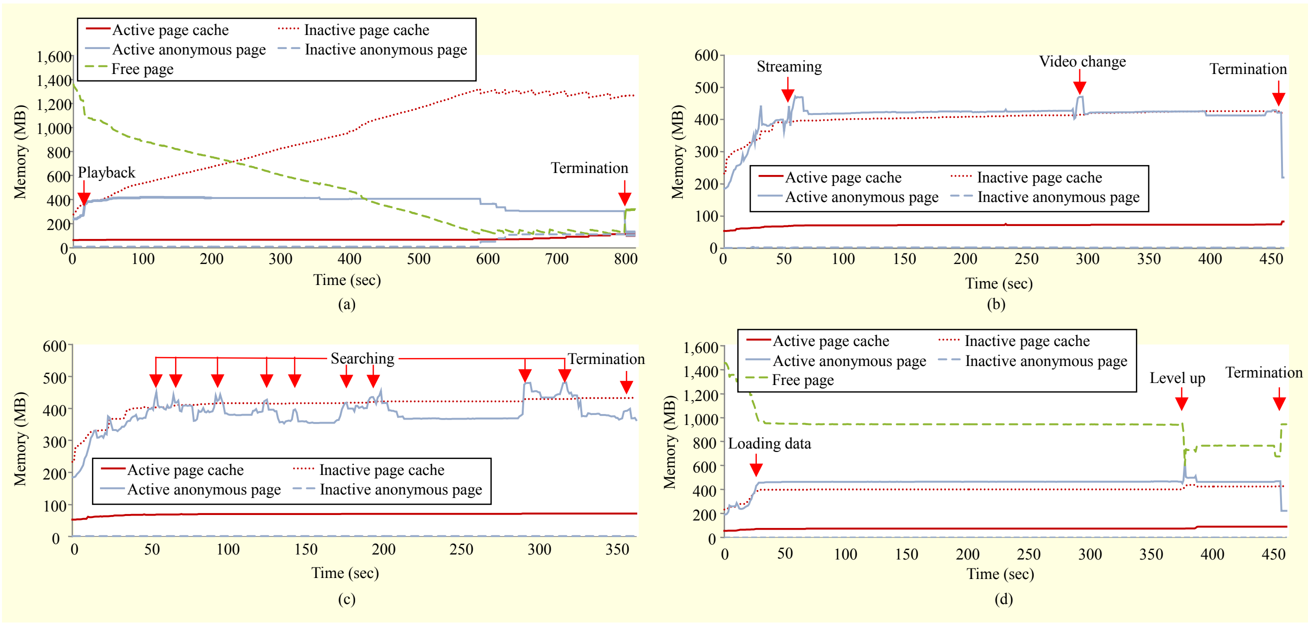


Fig. 1. Memory usage patterns of Smart TV workloads: (a) video playback, (b) video streaming, (c) web browsing, and (d) online game.

case of the video playback workload, we played 1,080p high-definition videos using the XBMC player [21], which is a media center platform of the Xbox game player. In the case of the video streaming workload, we opened a web browser and played arbitrary videos from YouTube [22]. In the case of the web browsing workload, we opened a web browser and searched arbitrary keywords. In the case of the online gaming workload, we played Nexuiz [23], which is a high-quality FPS game. All experiments were performed on a Linux-based system (see Section IV for the detailed description of the environment).

To analyze these workloads accurately, we measured memory usage, including active page cache, inactive page cache, active anonymous page, inactive anonymous page, and free page, from the OS performance statistics each second. Page cache acts as a transparent cache for disk-backed pages, and it is kept in the main memory for fast access. Anonymous page, which does not map a file on the disk, is usually used for the process's stack or heap area. The OS categorizes the pages in use into two lists: the active list and the inactive list. Pages on the active list are considered hot data and are not available for eviction. On the other hand, pages on the inactive list are considered cold data and can be evicted.

Figure 1 shows the memory usage pattern of each workload. As shown in Fig. 1(a), the video playback workload increases the active anonymous page at the beginning of the playback for running the XBMC player. Until the free memory is exhausted, the inactive page cache then continues to increase. This is because the OS caches as much data from the disk as possible to minimize slow disk I/O operations. Figure 1(b) shows the memory usage pattern of the video streaming workload. We found that the active anonymous page increased during streaming or changing video content due to the buffered data. In the case of the web browsing workload, as shown in Fig. 1(c), the active anonymous page increased whenever we tried to search a new keyword. Figure 1(d) shows the memory usage pattern of the online game workload. A large amount of memory is required at the beginning of loading the data. After all data is loaded, extra memory is required only when the stage steps up to the next level.

In summary, the video playback and online game workloads continuously spend the memory resource during the entire playing duration. On the other hand, the video streaming and web browsing workloads require the memory resource in accordance with the user requests.

2. Semantic Gap in Virtualized Smart TV Systems

To verify the problem of the semantic gap in the virtualized Smart TV systems, we played the video and online game simultaneously in both a native and a virtualized Smart TV system. In the experiments, the memory allocation for each VM is configured as 512 MB of 2 GB physical memory. Figure 2(a) shows the memory usage pattern in the native Smart TV system. When we played the video, the OS started to cache as much data as possible. As a result, the inactive page cache, which is regarded as less likely to be reused, rapidly increased (Fig. 1(a)). The reason for the memory usage pattern of the inactive page cache is that the video playback workload reads data sequentially and the data is never accessed again.

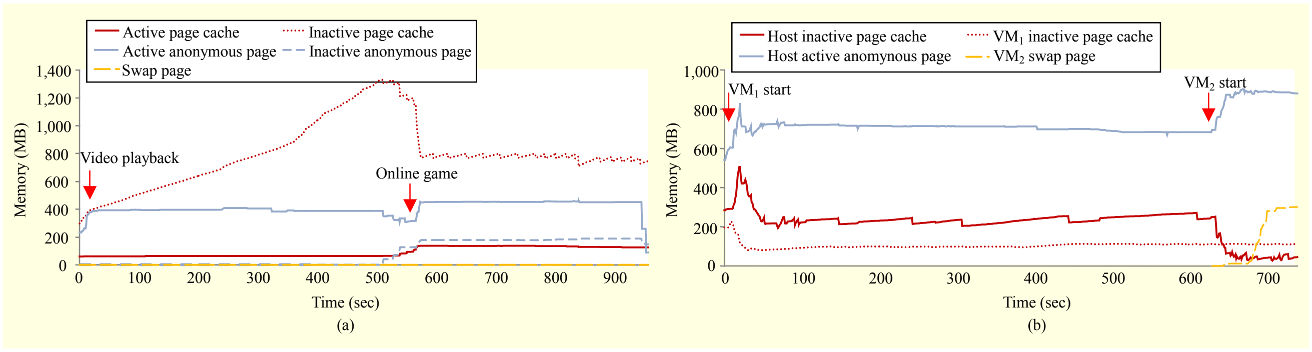


Fig. 2. Problem of memory resource management in virtualized Smart TV system: (a) native system and (b) virtualized system.

Then, when we played the online game, the OS reclaimed the inactive page cache efficiently to allocate the anonymous page for the online game workload. Thus, expensive memory swapping did not occur.

We performed the same experiment in the virtualized Smart TV system, as shown in Fig. 2(b). The video playback workload runs in VM_1, and the online game workload runs in VM_2. In the virtualized Smart TV system, the VMM suffers from lack of runtime information of VMs, called the semantic gap. Thus, when VM_2 started playing the online game, the VMM could not reclaim the inactive page cache of VM_1 and incurred expensive memory swapping by up to 300 MB in VM_2 as a result. Note that if the VMM efficiently reclaimed more than 100 MB of the inactive page cache from VM_1, then we could minimize memory swapping in advance. When there are few VMs that heavily use the inactive page cache, this problem becomes more critical. This result shows that dynamic memory balancing is essential for virtualized Smart TV systems.

III. System Design

In this section, we first describe the overall architecture of our system. We then describe the three design issues of our mechanism: (a) monitoring the memory usage patterns of VMs with negligible overhead regardless of the characteristics of workloads, (b) a technique for predicting the memory demand of each VM, and (c) dynamic memory balancing based on the results of the prediction.

1. System Overview

The proposed mechanism consists of three parts, as shown in Fig. 3. First, a monitor driver collects the memory usage pattern of a VM from the OS performance statistics and periodically updates it to a shared buffer. Second, whenever new information is updated, a predictor calculates δ and R, which represent the amount of required and reclaimable

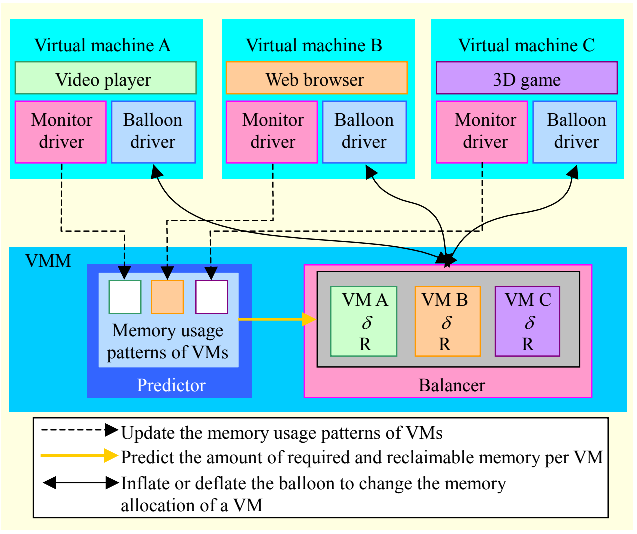


Fig. 3. System architecture.

memory per VM, respectively. Finally, when the VMM decides to rebalance the memory allocation among VMs, the VMM uses the ballooning technique [12]; a balancer inflates or deflates the balloon in a VM to change the memory allocation of the VM. Further detailed design of our mechanism is as follows.

2. Monitoring Memory Usage Patterns of VMs

As shown in the previous section, the memory usage patterns, such as the amount of page cache, anonymous pages, and swap-out pages, differ somewhat from workloads. However, memory management in the VMM is fundamentally limited due to the semantic gap between the VMM and the VM. The VMM considers all the VM memory as anonymous pages. Therefore, the VMM cannot make efficient decisions to adjust memory allocation in VMs and hence incurs expensive memory swapping in virtualized Smart TV systems. In previous studies, attempts have been made to estimate a working set size of each VM within a VMM by intercepting memory access [10]–[12] or tracing the buffer cache eviction

[13]–[14] so as to accurately balance the memory allocation. However, whenever this operation is triggered, the mode switching between the VM and the VMM causes a large overhead. In addition, the estimated working set size is not completely accurate.

We propose an efficient technique for monitoring the memory usage patterns of VMs. Instead of estimating the working set size of VMs, the VMM obtains the memory usage patterns of VMs through a monitor driver interface. The monitor driver runs in a guest kernel for posting this information periodically. It does not require modification of the guest kernel because the monitor driver is a typical loadable driver. The posted information includes the amount of active anonymous pages, active page cache, inactive page cache, free pages, and swap-out pages. The amount of active anonymous pages and active page cache indicates the memory actively in use, such as the working set size. Other information is used for predicting the memory demands and rebalancing the memory allocation among VMs. By using OS performance statistics in the VM, the VMM can obtain accurate information with negligible overhead regardless of the working set size and memory access pattern of workloads.

3. Prediction of Memory Demand per VM

To dynamically balance the memory allocation among VMs, the VMM needs to determine the working set size per VM. The VMM already knows the amount of memory actively in use (as described in Section III-2). However, to prevent expensive memory swapping in advance, the VMM needs to estimate the future memory demand.

To do this, we propose a novel technique that predicts the memory demand of each VM based on the exponentially-weighted moving average [20]. We define δ as the memory demand, whether or not each VM will require more memory. When updating the memory usage pattern of VM_i at the t-th interval, the predictor calculates δ_i^t as follows:

$$\Delta M_i^t = \Delta p_i^t + \Delta a_i^t + \Delta s_i^t, \tag{1}$$

$$\delta_i^t = \begin{cases} \alpha \times \Delta M_i^t + (1-\alpha) \times \delta_i^{t-1} & \text{if } \ t > 1, \\ \Delta M_i^t & \text{if } \ t = 1, \end{cases} \tag{2}$$

where ΔM_i^t is the variation of required memory in VM_i at the t-th interval (compared with the previous interval), α is a constant, and δ_i^{t-1} is the cumulative memory demand of VM_i up to the previous interval. Summing Δp_i^t, Δa_i^t, and Δs_i^t (that is, summing the variation of the active page cache, active anonymous pages, and swap-out pages in VM_i, respectively) gives ΔM_i^t. We show how the predictor behaves under different α in Section IV-1. Also, we define R as the amount of reclaimable memory in the VM. The predictor calculates R_i^t at each interval as follows:

$$R_i^t = l_i^t + f_i^t - 0.1T_i^t, \tag{3}$$

where l_i^t, f_i^t, and T_i^t represent the amount of inactive page cache, free pages, and total memory of VM_i at the t-th interval, respectively. We consider that 10% of total memory in a VM is not immediately reclaimable because most OSs, such as Linux, BSD Unix, and Windows, maintain a minimum amount of free memory to handle sudden memory allocation requests [12].

4. Dynamic Memory Balancing

Based on the results of prediction, the VMM balances the memory allocation among VMs dynamically. A balancer checks the δ and R of each VM at every interval. When the balancer detects VM_k, of which δ_k^t is larger than R_k^t, the VMM decides to balance the memory allocation among VMs because there is a possibility of memory swapping. To prevent swap-out operations, the VMM asks the other VMs to return their reclaimable memory to the VMM. If the balancer detects more than two VMs, of which δ_k^t is larger than R_k^t, then the VMM serves memory resource to each VM in random order. This policy can be easily extended by the QoS requirements of the VMs (for example, the service level agreement).

$$S_i^t = \begin{cases} \dfrac{R_i^t - \delta_i^t}{\sum_{j=1}^{n}(R_j^t - \delta_j^t)} \times \delta_k^t & \text{if } \displaystyle\sum_{j=1}^{n}(R_j^t - \delta_j^t) > \delta_k^t, \\ R_i^t - \delta_i^t & \text{otherwise.} \end{cases} \tag{4}$$

Equation (4) represents a reclaiming formula. The amount of reclaimed memory from VM_i to the VMM at the t-th interval is represented by S_i^t. Let there be n VMs that have reclaimable memory (that is, R_i^t is greater than δ_i^t). In this case, it is possible to reclaim the difference between R_i^t and δ_i^t by the VMM. If the sum of the differences is larger than δ_k^t, then the VMM reclaims the memory proportionally to the difference of

Table 1. Description of variables.

Variable	Description
ΔM_i^t	Variation of required memory in VM_i at the t-th interval
δ_i^t	Predicted memory demand in VM_i at the t-th interval
R_i^t	Amount of reclaimable memory in VM_i at the t-th interval
S_i^t	Amount of reclaimed memory from VM_i to the VMM at the t-th interval

each VM. Otherwise, only the sum of differences is reclaimed by the VMM. Table 1 shows a summary of the variables used in our mechanism.

IV. Evaluation

Our system is implemented using the KVM [24] VMM. Linux kernel 3.7.10 is used for guest and host OSs. In our prototype implementation, the memory usage updating interval is set to one second. All experiments were performed on a system with a 2.8 GHz Intel Core i7 processor and 2 GB of physical memory.

In this section, we show the evaluation results of our mechanism. First, we show experimental results of the proposed technique for predicting the memory demand of each VM. We then evaluate the overhead of our mechanism compared to prior work. Lastly, we evaluate the effectiveness of memory balancing through various workloads.

1. Memory Demand Prediction

Figures 4 and 5 show the experimental results of our prediction technique measured using the web browsing and online game workloads, respectively. We compare the predicted memory demand with an actual memory demand, changing the value of α; a constant used in the prediction. In the prediction technique, the more the value of α increases, the more ΔM_i^t, which is the variation of required memory at the t-th interval, is reflected. On the contrary, the more the value of α decreases, the more δ_i^{t-1}, which is the cumulative memory demand, is closely reflected.

When the value of α is 0.25, as shown in Figs. 4(a) and 5(a), our prediction technique passively reflects the recent variation of required memory; thus, the risk of memory swapping is high. Meanwhile, the more the value of α increases, the more the prediction technique reflects the recent variation of required memory; thus, this decreases the risk of memory swapping. Figures 4(d) and 5(d) present the most aggressive policy for preventing expensive memory swapping. However, the memory resource is wasted in a VM whose actual memory demand changes frequently and whose gap between minimum and maximum memory demand is high. In addition, when few VMs show this memory demand pattern, the VMM balances the memory allocation among VMs unnecessarily. Therefore, we set 0.75 as the value of α in our prototype implementation.

2. System Overhead

For our mechanism to be practical in virtualized Smart TV systems, its overhead should be minimal, regardless of the characteristics of workloads. In the experiments, we use the SPEC CPU 2000 [25] benchmark suite for comparing the performance overhead with previous studies. To evaluate the monitoring overhead of our mechanism, we measure the execution time of SPEC 2000 workloads with only the monitor driver and predictor enabled.

As shown in Table 2, the average overhead of our system is negligible: 0.05% for SPEC 2000, compared with 2.13% in Zhao and others' method [10] and 0.67% in VMMB [11]. More importantly, all measured overheads of SPEC 2000

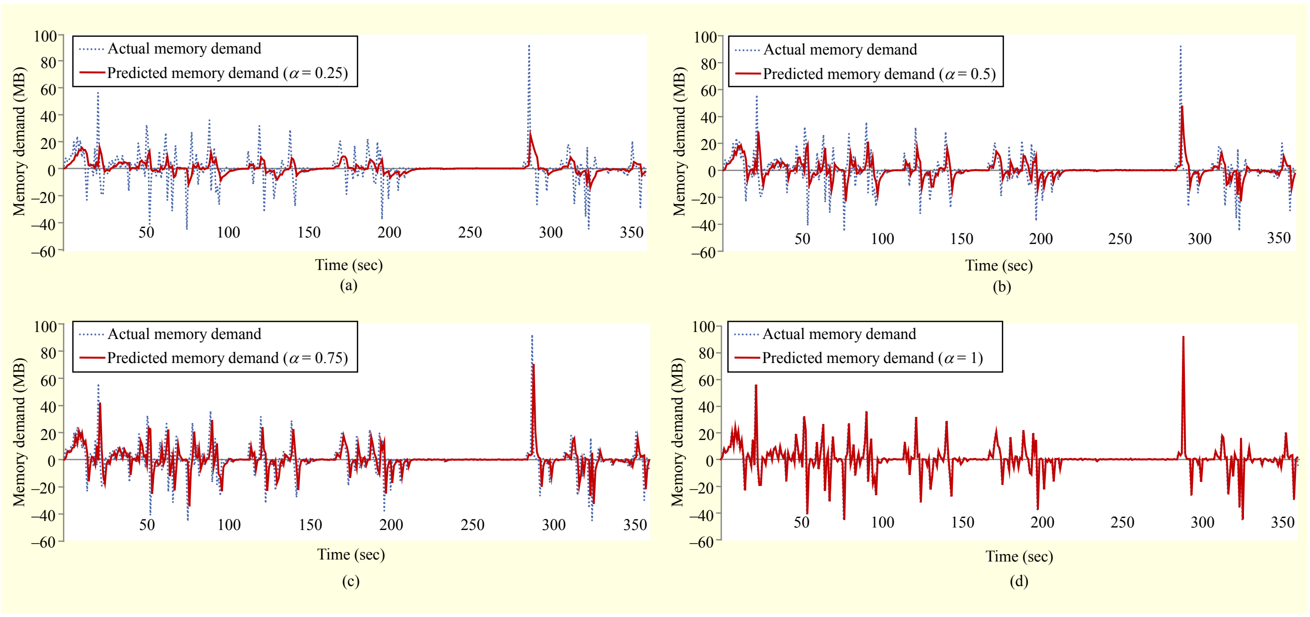


Fig. 4. Memory demand prediction of web browsing workload: (a) $\alpha = 0.25$, (b) $\alpha = 0.5$, (c) $\alpha = 0.75$, and (d) $\alpha = 1$.

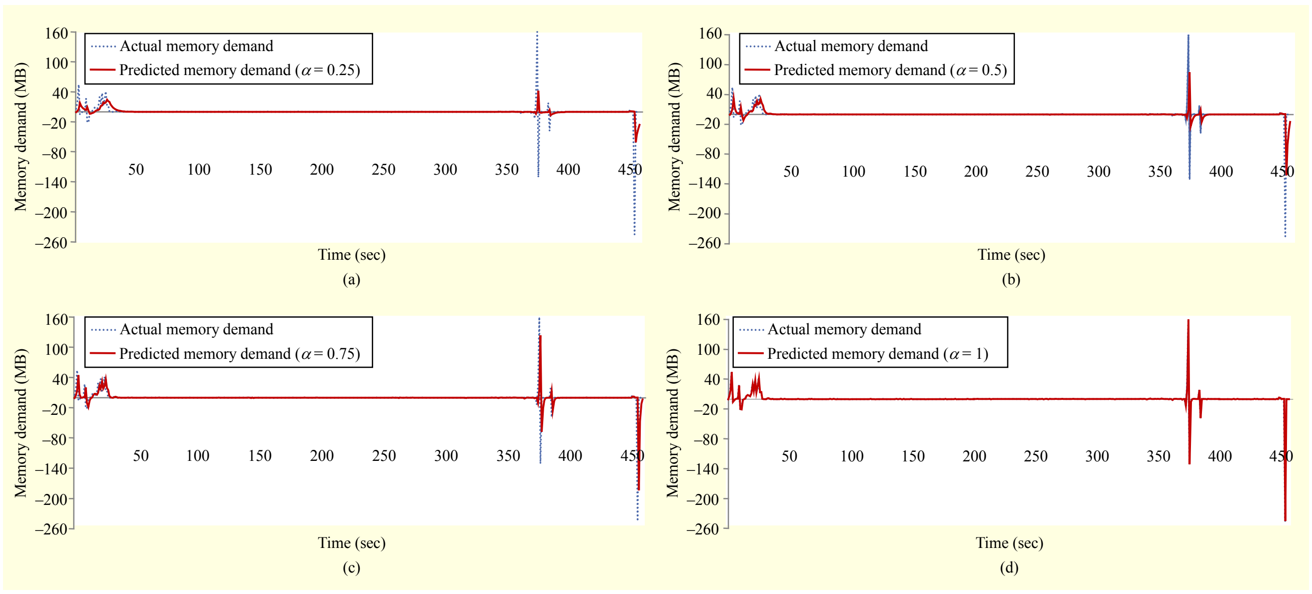


Fig. 5. Memory demand prediction of online game workload: (a) $\alpha = 0.25$, (b) $\alpha = 0.5$, (c) $\alpha = 0.75$, and (d) $\alpha = 1$.

Table 2. Monitoring overhead of SPEC 2000 workloads.

Workload	Monitoring overhead (%)
164.gzip	0.05
175.vpr	0.06
176.gcc	0.05
181.mcf	0.04
186.crafty	0.07
197.parser	0.05
252.eon	0.06
253.perlbmk	0.05
254.gap	0.05
255.vortex	0.05
256.bzip2	0.06
300.twolf	0.06
Average	0.05

workloads are equal within the margin of error because our mechanism is not affected by the working set size and memory access pattern of the workloads. For example, the overhead of a memory intensive workload such as 181.mcf is 0.04% in our mechanism. On the contrary, the runtime overhead of Zhao and others's method [10] and VMMB [11] are not negligible, and are as much as 24% and 1.65%, respectively. When there are few VMs running a workload with a large working set, the monitoring overhead becomes more critical. This result shows that our mechanism effectively obtains memory usage information of various workloads.

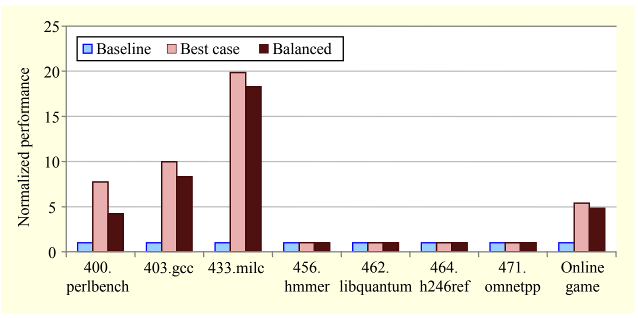


Fig. 6. Performance of workloads in VM_1.

3. Effectiveness of Memory Balancing

We evaluate how efficiently our mechanism balances the memory allocation among VMs and how this impacts the overall performance of a virtualized Smart TV system. We measure the performance of VM_1, which runs selected workloads in the SPEC CPU 2006 [26] benchmark suite, and the online game for realistic workload, while VM_2 and VM_3 run the video playback and the web browsing workloads, respectively, in the background.

Figure 6 shows the performance of workloads in VM_1. *Baseline* is the performance through static partitioning that allocates 512 MB of memory for each. *Best case* is the performance when we run a VM with 1,536 MB of memory. This represents the ideal performance, because no CPU and cache contentions occur, and sufficient memory is provided for preventing memory swapping. *Balanced* is the performance through our balancing mechanism. In the experimental results, the performance of workloads such as 400.perlbench, 403.gcc, 433.milc, and the online game are boosted by up to 18.28 times

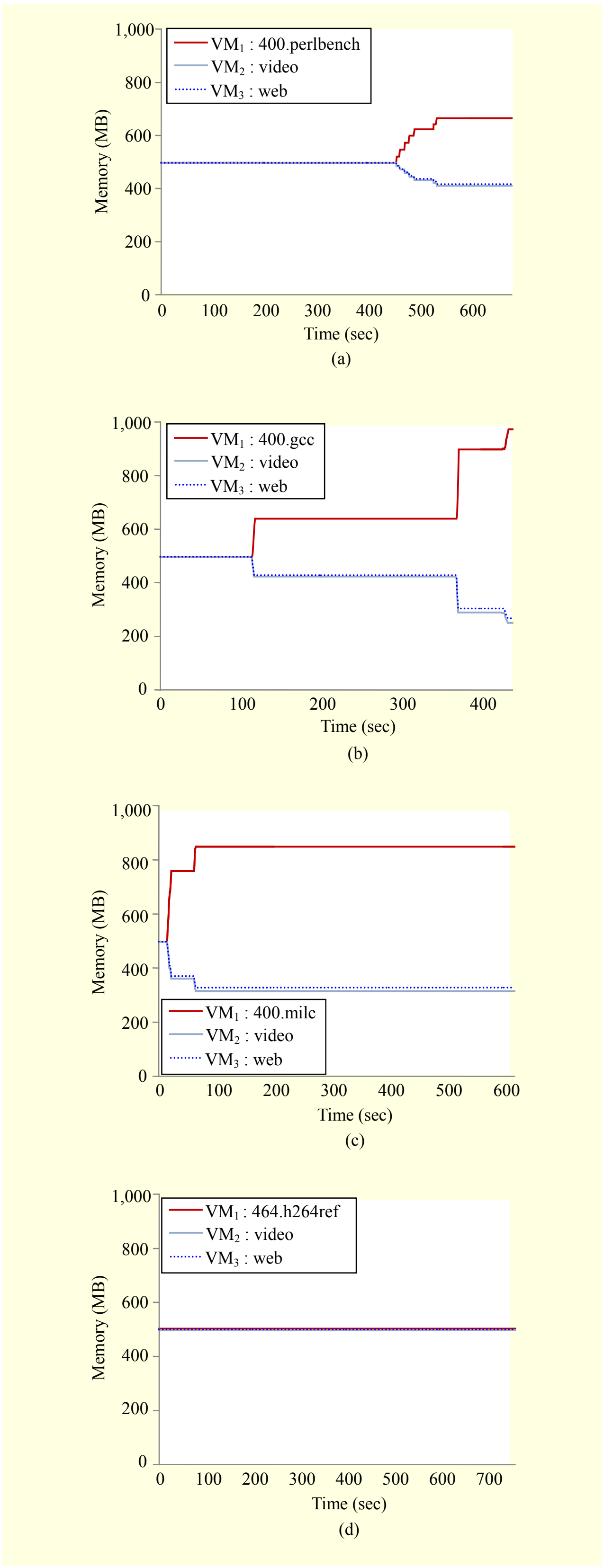


Fig. 7. Representative results of variation of allocated memory among VMs: (a) 400.perlbench, (b) 403.gcc, (c) 433.milc, and (d) 464.h264ref.

Table 3. Reduced amount of memory swapping.

Category	Workload	Reduced amount (%)
VM_1	400.perlbench	83.89
	403.gcc	97.03
	433.milc	99.73
	456.hmmer	-
	462.libquantum	-
	464.h264ref	-
	471.omnetpp	-
	Online game	88.93
VM_2	Video playback	-
VM_3	Web browsing	-

compared with *Baseline*. Among these workloads, the performance gains of 403.gcc, 433.milc, and the online game are close to *Best case*. On the other hand, 400.perlbench shows relatively less performance improvement. In the case of workloads such as 456.hmmer, 462.libquantum, 464.h264ref, and 471.omnetpp, no significant difference is observed in the performance among *Baseline*, *Best case*, and *Balanced*.

To analyze the experimental results, we measure the allocated memory among VMs over time. Figure 7 shows the representative results of the variation of allocated memory among VMs. As shown in Fig. 7(a), Fig. 7(b), and Fig. 7(c), the VMM gives more memory to VM_1, which runs the memory intensive workloads with a large working set. This is because our mechanism predicts the memory demand of each VM accurately and balances the memory among VMs dynamically. On the contrary, no memory rebalancing is observed in Fig. 7(d). This is because our mechanism only balances the memory allocation on demand when there is a possibility of memory swapping. This helps to reduce memory thrashing, whereby unnecessary balancing arises from a workload in which the actual memory demand increases and decreases frequently.

To further analyze the experimental results, we measure the amount of memory swapping in each VM. Table 3 shows the reduced amount of memory swapping, compared with *Baseline*. We found that our mechanism prevents expensive memory swapping by up to 99.73% by giving more memory to VM_1. Also, we found that no memory swapping occurs in VM_2 and VM_3. This shows that our mechanism efficiently predicts the required and reclaimable memory of VMs and that the VMM reclaims the memory from VM_2 and VM_3 without performance degradation. In the case of 400.perlbench, the reduced amount of memory swapping is relatively small. Thus, relatively less performance improvement is shown in Fig. 6. This is because little memory resource is required with a repeated pattern, as shown in Fig. 7(a). If we increase the value of α by more than 0.75 in our prototype implementation, then the prediction technique reflects the recent variation of the required memory more aggressively and the performance improvement of 400.perlbench will be better. However, this may lead to performance degradation in other VMs due to overestimation. Thus, when determining the value of α, the administrator should consider the QoS requirements of individual VMs that have different importance factors.

V. Related Work

Our mechanism can greatly reduce the amount of memory swapping by predicting the memory demand of each VM efficiently and rebalancing the memory allocation among VMs dynamically. In this regard, this section briefly presents the existing technologies that are most closely related to our work and compares them to our mechanism.

First, Waldspurger [12] proposed a statistical sampling approach to estimate the working set size of a VM. During a configurable sample period, a small number of randomly selected pages are monitored, whether or not they are accessed. At the end of the sample period, the fraction of accessed pages over selected pages is considered as the VM working set. However, a working set size larger than that currently allocated memory to a VM cannot be estimated. If the VM begins to thrash, the sampling technique simply reports the working set size, which is 100% of the currently allocated memory to the VM. Thus, as in this case, a trial and error approach is required for VMM to determine the working set size. To do this, the VMM repeatedly gives more memory to the VM until it drops below 100%. However, if physical memory is not available, even this trial and error approach will fail. Thus, a brute-force approach may be needed in this scenario. In addition, as the number of sampling pages increases for the high accuracy of estimation, the monitoring overhead linearly increases. The sampling rate is shown in the tradeoff between overhead and accuracy. In contrast, we exploit a symbiotic approach to predict the future memory demand of each VM. Thus, our mechanism can estimate the working set size larger than the currently allocated memory to the VM with negligible monitoring overhead.

Second, the ghost buffer technique [13]–[14], [27]–[28], has been proposed to predict a page miss ratio under a different memory resource provisioning. Geiger [13] proposed a technique that monitors the buffer cache in virtualized environments. This technique monitors the buffer cache eviction and promotion by intercepting all I/O operations at the VMM level. Through this, the technique can predict the page miss ratio and determine appropriate VM memory allocation. However, it cannot estimate the working set of a VM smaller than its current memory allocation because, as in this case, it does not incur the buffer cache eviction. To cope with such a limitation, Lu and Shen [14] proposed a technique whereby the VMM takes over the management for part of the VM memory; thus, all accesses can be transparently traced by the VMM. Through this, the growth and the shrinkage of memory allocation can be determined. However, this technique cannot be used for an OS without source code, since it requires modification of the guest kernel. In contrast, our mechanism can estimate the working set of a VM smaller than its current memory allocation and efficiently reclaims the VM memory when necessary. Furthermore, our mechanism does not require modification of the guest kernel because the monitor driver is a typical loadable driver.

Third, Zhao and others [10] proposed an LRU histogram–based [15] approach for estimating the working set size of each VM. This approach divides the VM memory into two sets: a hot page set and a cold page set. Then, only the cold page set is monitored for reducing runtime overhead. However, a workload that heavily accesses the memory causes a large monitoring overhead due to frequent histogram updates. To address this problem, VMMB [11] proposed a weighted red–black tree and an adaptive resizing technique. However, it can still run the risk of increasing the monitoring overhead in accordance with the memory access pattern of workloads. Also, this technique incurs expensive mode switching between the VM and the VMM whenever the VM memory accesses. In contrast, we fully exploit OS performance statistics to obtain the memory usage patterns of VMs. Thus, our mechanism has negligible runtime overhead regardless of the characteristics of workloads.

Lastly, Salomie and others [29] proposed a technique, called application-level ballooning (ALB), for rebalancing the memory allocation among a collection of applications that manage their own memory, such as databases and language runtimes. ALB coordinates application, guest OS, and VMM for optimizing the use of the memory pool to maximize performance. This approach differs from ours in that ALB can only be used for specific applications that manage the memory resource themselves. Tesseract [30] and VSWAPPER [31] explored the behavior of uncooperative memory swapping, which is considered a necessary evil in the virtualized environments, and proposed their solutions. Tesseract [30] directly addresses the double-paging problem, which occurs when the VM attempts to page out memory that has previously been swapped out by the VMM, and VSWAPPER [31] describes some more cases. We view their solutions as complementary to our mechanism.

VI. Conclusion

In the near future, Smart TV systems will become ubiquitous in home environments through the introduction of cloud services based on virtualization technologies. To achieve this, efficient memory resource management is a key success factor in virtualized Smart TV systems. In this paper, we proposed a novel mechanism to dynamically balance the memory allocation among VMs in a virtualized Smart TV system by reflecting the characteristics of Smart TV workloads. The VMM obtains the memory usage patterns from VMs with negligible overhead and predicts the memory demand of each VM for preventing memory swapping in advance. Then, based on the predicted memory demands, the VMM rebalances the memory allocation among the VMs. Experimental results show that our mechanism is a practical solution for dynamic memory balancing in next-generation Smart TV systems. As future work, we plan to extend our mechanism for integration with QoS requirements of individual VMs that have different importance factors.

References

[1] M.A. Brahmia, A. Abouaissa, and P. Lorenz, “Improving IPTV Forwarding Mechanism in IEEE 802.16j MMR Networks Based on Aggregation,” *ETRI J.*, vol. 35, no. 2, Apr. 2013, pp. 234–244.

[2] M.R. Cabrer et al., “Controlling the Smart Home from TV,” *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, May 2006, pp. 421–429.

[3] K.S. Cho, H.W. Lee, and W. Ryu, “Service Trends and Prospect on Smart TV,” *Electron. Telecommun. Trends*, vol. 26, no. 4, Aug. 2011, pp. 1–13.

[4] ARM Architecture Group, *ARM Cortex-A15 MPCore Processor*, ARM, 2011. Accessed Jan. 25, 2014. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/CHDCHAED.html

[5] J.-Y. Hwnag et al., “Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones,” *IEEE Consum. Commun. Netw. Conf.*, Las Vegas, NV, USA, Jan. 10–12, 2008, pp. 257–261.

[6] G. Heiser, “Hypervisor for Consumer Electronics,” *IEEE Consum. Commun. Netw. Conf.*, Las Vegas, NV, USA, Jan. 10–13, 2009, pp. 1–5.

[7] K. Barr et al., “The VMware Mobile Virtualization Platform: Is that a Hypervisor in Your Pocket?,” *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 4, Dec. 2010, pp. 124–135.

[8] P. Varanasi and G. Heiser, “Hardware-Supported Virtualization on ARM,” presented at the Proc. Asia-Pacific Workshop Syst., Shanghai, China, July 11–12, 2011.

[9] J. Perello, P. Pavon-Marino, and S. Spadaro, “Cost-Efficient Virtual Optical Network Embedding for Manageable Inter-Data-Center Connectivity,” *ETRI J.*, vol. 35, no. 1, Feb. 2013, pp. 142–145.

[10] W. Zhao, Z. Wang, and Y. Luo, “Dynamic Memory Balancing for Virtual Machines,” *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 3, July 2009, pp. 37–47.

[11] C. Min et al., “VMMB: Virtual Machine Memory Balancing for Unmodified Operating Systems,” *J. Grid Comput.*, vol. 10, no. 1, Mar. 2012, pp. 69–84.

[12] C.A. Waldspurger, “Memory Resource Management in VMware ESX Server,” *ACM SIGOPS Operating Syst. Rev.*, vol. 36, no. SI, 2002, pp. 181–194.

[13] S.T. Jones, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, “Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment,” *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 5, Dec. 2006, pp. 14–24.

[14] P. Lu and K. Shen, “Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache,” *USENIX Annual Techn. Conf.*, Santa Clara, CA, USA, June 17–22, 2007, pp. 29–43.

[15] R.L. Mattson et al., “Evaluation Techniques for Storage Hierarchies,” *IBM Syst. J.*, vol. 9, no. 2, June 1970, pp. 78–117.

[16] K. Adams and O. Agesen, “A Comparison of Software and Hardware Techniques for x86 Virtualization,” *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 5, Dec. 2006, pp. 2–13.

[17] A. Landau, M. Ben-Yehuda, and A. Gordon, “SplitX: Split Guest/Hypervisor Execution on Multicore,” *USENIX Workshop I/O Virtualization*, Portland, OR, USA, June 14–17, 2011, pp. 1–7.

[18] O. Agesen et al., “Software Techniques for Avoiding Hardware Virtualization Exits,” *USENIX Annual Techn. Conf.*, Boston, MA, USA, June 13–15, 2012, pp. 373–386.

[19] N. Har’El et al., “Efficient and Scalable Paravirtual I/O System,” *USENIX Annual Techn. Conf.*, San Jose, CA, USA, June 26–28, 2013, pp. 231–242.

[20] J.S. Hunter, “The Exponentially Weighted Moving Average,” *J. Quality Technol.*, vol. 18, no. 4, Oct. 1986, pp. 203–207.

[21] XBMC Media Center. Accessed Jan. 25, 2014. http://xbmc.org

[22] YouTube. Accessed Jan. 25, 2014. http://www.youtube.com

[23] Nexuiz. Accessed Jan. 25, 2014. http://www.nexuiz.com

[24] A. Kivity et al., “KVM: The Linux Virtual Machine Monitor,” *Linux Symp.*, Ottawa, Canada, June 27–30, 2007, pp. 225–230.

[25] *SPEC CPU2000.* Accessed Jane 25, 2014. http://www.spec.org/cpu2000/

[26] *SPEC CPU2006.* Accessed Jan. 25, 2014. http://www.spec.org/cpu2006/

[27] T.M. Wong and J. Wilkes, “My Cache or Yours? Making Storage More Exclusive,” *USENIX Annual Techn. Conf.*, Monterey, CA, USA, June 10–15, 2002, pp. 161–175.

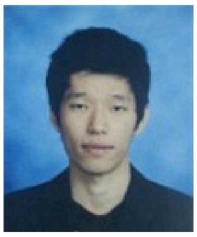
[28] Z. Chen, Y. Zhou, and K. Li, “Eviction-Based Cache Placement for Storage Caches,” *USENIX Annual Techn. Conf.*, San Antonio, TX, USA, June 9–14, 2003, pp. 269–282.

[29] T.-I. Salomie et al., “Application-Level Ballooning for Efficient Server Consolidation,” *European Conf. Comput. Syst.*, Prague, Czech Republic, Apr. 14–17, 2013, pp. 337–350.

[30] K. Arya, Y. Baskakov, and A. Garthwaite, “Tesseract: Reconciling Guest I/O and Hypervisor Swapping in a VM,” *ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments*, Salt Lake City, UT, USA, Mar. 1–2, 2014, pp. 15–28.

[31] N. Amit, D. Tsafrir, and A. Schuster, “VSWAPPER: A Memory Swapper for Virtualized Environments,” *ACM Conf. Archit. Support Programming Languages Operating Syst.*, Salt Lake City, UT, USA, Mar. 1–5, 2014, pp. 349–366.

Junghoon Kim received his BS degree in computer engineering and his MS degree in mobile systems engineering from Sungkyunkwan University, Suwon, Rep. of Korea, in 2010 and 2012, respectively. He is currently a PhD candidate in the Department of IT Convergence at Sungkyunkwan University. His research interests include storage systems, embedded systems, virtualization, and operating systems.



Taehun Kim received his BS degree in computer engineering from Korea Polytechnic University, Siheung, Rep. of Korea, in 2012 and his MS degree in electrical and computer engineering from Sungkyunkwan University, Suwon, Rep. of Korea, in 2014. Since 2014, he has been a system engineer at Naver Business Platform, Seongnam, Rep. of Korea. His research interests include virtualization, operating systems, and storage systems.

Changwoo Min received his BS and MS degrees in computer science from Soongsil University, Seoul, Rep. of Korea, in 1996 and 1998, respectively and his PhD degree in mobile systems engineering from Sungkyunkwan University, Suwon, Rep. of Korea, in 2014. From 1998 to 2005, he was a research engineer in the Ubiquitous Computing Laboratory of IBM, Seoul, Rep. of Korea. From 2005 to 2014, he was a research engineer at Samsung Electronics, Suwon, Rep. of Korea. Currently, he is a postdoctoral researcher at Sungkyunkwan University. His research interests include operating systems, storage systems, virtualization, and runtime systems.

Hyung Kook Jun received his BS degree in computer science and engineering and his MS degree in electrical and computer engineering from Sungkyunkwan University, Suwon, Rep. of Korea, in 1999 and 2001, respectively. Since 2001, he has been a senior researcher in the Cyber-Physical Systems (CPS) research team, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea. His research interests include CPS, embedded systems, communication middleware, and multimedia systems.

Soo Hyung Lee received his BS and MS degrees in electronic engineering from Hanyang University, Seoul, Rep. of Korea, in 1991 and 1993, respectively and his PhD degree in computer engineering from Chungnam National University, Daejeon, Rep. of Korea, in 2012. In August 1993, he joined the network design laboratory of DACOM corporation. Since October 2000, he has been a principal member of the engineering staff in the Cyber-Physical Systems (CPS) research team, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea. His research interests include IT converging systems, CPS, distributed communication, and network security.

Won-Tae Kim received his BS, MS, and PhD degrees in electronic engineering from Hanyang University, Seoul, Rep. of Korea, in 1994, 1996, and 2000, respectively. He established a venture company named Rostic Technologies Inc. in January 2001 and worked as CTO until February 2005. He joined the Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea, in March 2005, and now he is the manager of the Cyber-Physical Systems (CPS) research section of SW Contents Technology Research Lab. He has been the chairman of the CPS project group of the Telecommunication Technology Association since 2011. In addition, he has been a vice president of the Military SW Research Association at the Korean Institute of Information Scientists and Engineers since 2011.

Young Ik Eom received his BS, MS, and PhD degrees in computer science from Seoul National University, Seoul, Rep. of Korea, in 1983, 1985, and 1991, respectively. He was a visiting scholar at the Department of Information and Computer Science, University of California, Irvine, USA, from September 2000 to August 2001. Since 1993, he has been a professor at Sungkyunkwan University, Suwon, Rep. of Korea. His research interests include system software, operating systems, virtualization, cloud systems, and system securities.