Optimizing Memory Swapping Scheme on the Memory Debugging Platform of CE Devices

Youngho Choi[†‡], *Dong Hyun Kang*[†], *Jaeook Kwon*[‡], *and Young Ik Eom*[†]
[†]Sungkyunkwan University, South Korea [‡]Samsung Electronics, South Korea
{yh.choi,kkangsu,yieom}@skku.edu, {yh2005.choi,jook.kwon}@samsung.com

***Abstract*—In this paper, we revisit the traditional memory debugging approach for CE Devices, and propose a novel memory debugging platform by modifying both DUMA library and Zram. Especially, we focus on the memory space wasted while detecting the memory error, because most CE devices have limited memory resource. For evaluation, we implemented a prototype of our debugging platform on Linux kernel version 4.8.17 and conducted various experiments with two different workloads. Our experimental results show that our scheme improves the performance of swapping by up to 58 times.**

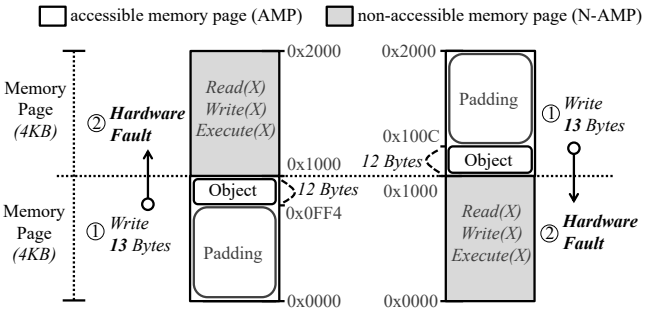
I. INTRODUCTION

In recent years, as the number of application services for CE devices has been increased, software platforms on CE devices are growing in the number of features and size. However, such expansion is very harmful in terms of validation for software reliability because the programming difficulty and complexity become much higher. Especially, in such complex platforms, memory errors [1] (e.g., buffer-overflow, buffer-underflow, and use-after-free) are difficult to diagnose because they manifest themselves only in extremely rare conditions or in very long executions.

Many researchers have focused on automatically detecting the memory errors and proposed various approaches: Valgrind [2], AddressSanitizer [3], SGXBounds [4], Intel MPX [5], and DUMA [6]. Valgrind updates executable code at runtime to detect software bugs based on the dynamic binary instrumentation [7]. AddressSanitizer and SGXBounds insert instrumentation, which helps to check memory error, into the executable file at compile time [8]. Intel MPX employs a feature of the hardware for inspecting bugs on-the-fly. However, even though previous approaches succeed in detecting various kinds of memory errors, they are rarely used in CE devices because of high performance overhead, high memory usage, and requirement on special hardware.

Meanwhile, DUMA tool is widely used for CE devices because it can work on most CPU architectures and provide fast performance [9]. Unfortunately, in order to detect the memory errors, DUMA always allocates memory space at a page granularity even though users request memory allocation at a byte granularity, where we call this as *memory page amplification* (MPA). In this way, DUMA may suffer from the lack of memory. To relieve the lack of memory space caused by DUMA, some engineers compress the data on memory by using a memory swapping scheme (e.g., Zram) [10]. However, this approach is inefficient in small-sized allocation, because its compression result includes padding data of MPA.

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP-2015-0-00284) supervised by the IITP(Institute for Information & communications Technology Promotion)



(a) Buffer-overflow (b) Buffer-underflow

Fig. 1. Examples for buffer-overflow and buffer-underflow with DUMA.

In this paper, we revisit the memory swapping mechanism to further optimize the memory space in case of small-sized allocations. Then, we propose a novel memory debugging platform that is composed of DUMA and memory swapping scheme. Our platform reduces the memory usage efficiently by minimizing the amount of data stored into the memory swap area. Our experiments clearly show that our scheme not only reduces swapped page size by up to 1.8 times but also improves swap response time by up to 58 times under memory-intensive workloads that periodically require memory allocation by using `malloc` system call.

II. DUMA LIBRARY

In this section, we describe DUMA library and related studies, to understand our memory debugging platform in detail.

DUMA, one of the open source libraries, helps to detect memory errors based on the specialized memory allocator that hooks memory-related system calls (e.g., `malloc`, `new`, `free`, etc.) of C/C++ applications. The specialized memory allocator employs the page protection feature of the CPU, that raises hardware fault, called segmentation fault, when an application accesses the page with wrong access right. However, this scheme has the MPA issue because it must allocate two memory pages; *non-accessible memory page* (N-AMP) that does not have any access permission and *accessible memory page* (AMP) that includes user data with access permission. To detect buffer-overflow, the specialized memory allocator first places the AMP underneath the N-AMP, and then copies user data into the top of the AMP. Finally, the remained space of AMP is filled up with padding data.

Figure 1 shows the examples of memory layout to detect buffer-overflow and buffer-underflow with DUMA. In this

978-1-5386-3025-9/18/$31.00 ©2018 IEEE

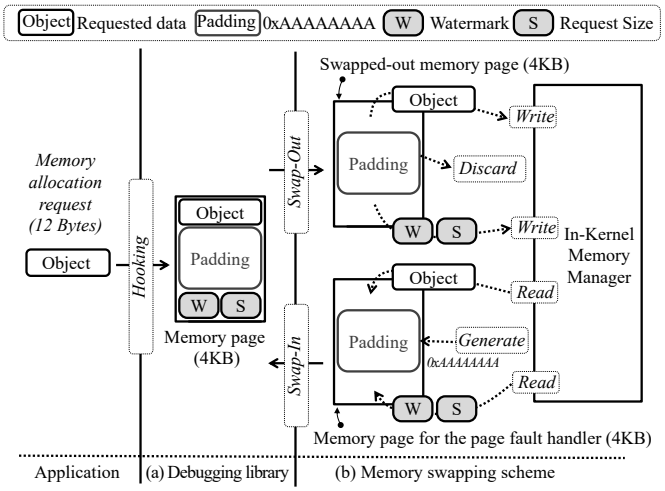


Fig. 2. The memory transformation performed in our scheme.

example, an application requests small-sized memory allocation (e.g., 12 bytes). At that time, DUMA hooks the requests and allocates two memory pages: AMP and N-AMP page. If an application tries to access the memory address that overruns the allocated boundary (e.g., 0x1000), a hardware fault is raised (Figure 1a). The application is ended up, stopping along with the memory dump that gives developers hints for the fault. In the case of the buffer-underflow, the two memory pages are placed in opposite direction. Unfortunately, DUMA wastes page space by using the page-granularity allocation. In other words, 4KB AMP is allocated to inspect 12 bytes user data in this example. Unsurprisingly, some engineers enhanced DUMA with the memory swapping scheme, called Zram. This is because most memory space is allocated as anonymous pages and the pages are handled by swapping mechanism. As a result, they could significantly reduce the amount of memory consumed by DUMA. However, their approach is inefficient in case of small-sized memory allocations because it compresses not only user data but also padding data. In addition, the compressed padding data would be larger than the user data in the case of small-sized memory allocations. This observation motivates us to design an enhanced memory swapping scheme.

III. Design and Implementation

In this paper, our goal is to save memory space by managing swapped-out memory pages efficiently. To reach this, we designed and implemented a novel memory debugging platform that consists of DUMA and memory swapping scheme. We enhanced not only DUMA but also memory swapping scheme for them to share semantic information. Figure 2 shows the operations of our debugging platform with an example.

A. Memory debugging library

To detect the memory error caused by an application, we follow the process of DUMA library. However, we enhance it with two additional parameters: watermark and request size. The watermark indicates whether a page is to be handled by DUMA or not. The request size means the object size requested by an application. Like DUMA, our debugging library first

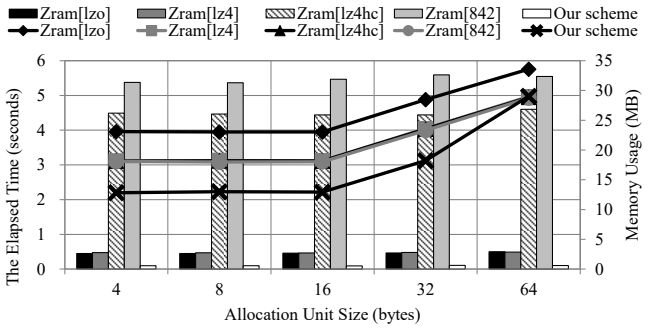


Fig. 3. Swap response time and memory usage comparison with various synthetic workloads.

hooks all memory-related system calls and allocates memory space (i.e., page) according to the request size. Then, it reorganizes the internal structure of the allocated page to monitor the memory error (Figure 2a).

B. Memory swapping scheme

We also enhance the traditional memory swapping scheme, called Zram, because of two reasons. First, Zram can accelerate the performance of our debugging library by using the fixed-size memory area as the swap partition. Second, Zram significantly saves space of the swap partition by compressing the data to be swapped out.

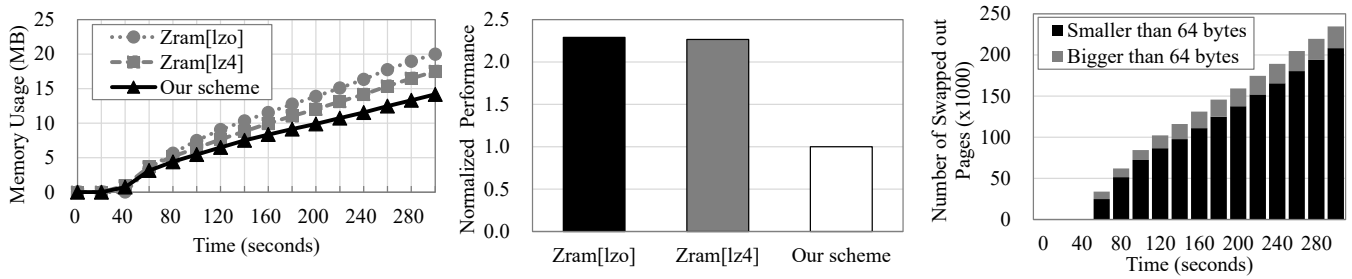
If the number of free pages runs low during detecting the memory error, *kswapd*, which is a kernel demon, starts to trigger swap-out operations (Figure 2b). At this time, our memory swapping scheme first separates the contents of the page that is swapped out into four sections: object, padding, watermark, and request size. Then, it stores the contents of the page into the in-kernel memory area (i.e., swap partition), excluding the padding data. This is because we can easily recover the contents belonging to the padding data in swap-in process. As a result, we can prevent the space waste of the swap partition incurred by padding. In other words, our memory swapping only stores necessary contents into the swap partition with the metadata, which is used to map between an object swapped out and the corresponding information, such as watermark and request size.

Meanwhile, if the application tries to read the data on the swap partition, our memory swapping scheme triggers a swap-in operation by going through the following steps. First, our scheme allocates a page for swap-in operation and then migrates the object from in-kernel memory area to the page. Second, it fills out padding data by generating dummy data (i.e., 0xAAAAAAAA). Finally, our scheme attaches the watermark and request size underneath the padding area.

IV. Performance Evaluation

In this paper, we briefly explain our implementation and show the evaluation setup and results with two types of workloads: synthetic and real-world workloads.

We implemented our prototype of memory debugging platform on Linux kernel version 4.8.17 by modifying both DUMA library and Zram. In addition, to provide application-



(a) Memory usage (b) Swap response time (c) Statistics for pages swapped out

Fig. 4. Memory usage, swap response time and internal analysis under real-world workload.

level transparency, we only changed the internal layout of DUMA and the internal compression and decompression functions on Zram. To clearly show the number of pages swapped out, we conducted all experiments on a virtual machine that is configured with 4 GB RAM, one CPU core, and runs the modified Linux kernel version 4.8.17. The host machine has an 8-core 3.4 GHz Intel Core i7 CPU with 16GB RAM and runs the unmodified Ubuntu-16.04 (Linux kernel version 4.4.59).

Synthetic workload: To clearly understand the effectiveness of our debugging platform, we developed a micro-benchmark that periodically repeats fixed-size memory allocation and write operation until *kswapd* triggers enough swap-out operations (e.g., 400,000 times). Figure 3 shows the evaluation results that are measured by varying the memory allocation size: 4, 8, 16, 32, 64 bytes. As we expected, the *lzo*, which is default compression algorithm of Zram, incurs much memory space waste. As shown in Figure 3, other algorithms based on Zram also wastes memory space significantly. The reason behind these results is that the Zram needs more memory space by compressing not only user data but also padding. On the other hand, our memory swapping scheme consumes the least memory usage in all cases. Especially, our scheme reduces the memory usage of swapped-out pages by up to 1.8 times in case of 4 bytes memory allocation compared with the *lzo* algorithm (Figure 3). In addition, our scheme improves the swapping time by up to 58 times compared with the *842* algorithm. Figure 3 shows that the memory usage of our scheme increases as the allocation unit size becomes large, in which case another optimization scheme may be necessary. We have a plan to study this issue in our future work.

Real-world workload: For realistic performance evaluation, we conducted additional experiments with real-world workload. Figure 4a and Figure 4b show the evaluation results that are measured by compressing Linux kernel source along with *tar* and *gzip*. Figure 4a clearly confirms that our scheme reduces the memory usage even in the real-world workload: by up to 1.41 and 1.24 times for compared with *lzo* and *lz4*, respectively. In addition, Figure 4b shows that our scheme improves the swapping performance by up to 2.3 times compared with the two algorithms. These results have very similar pattern to those of Figure 3. Therefore, we analyzed the real-world workload in terms of the amount of memory allocation and found out that most pages swapped out have smaller data set than 64 bytes (Figure 4c)

In summary, in this paper, we only showed the performance on memory swapping. This is because we believe that the performance improvement on memory swapping directly affects the performance of the memory error detection.

V. Conclusion

Nowadays, detecting the memory error is very crucial in CE devices because the number of applications that frequently require memory allocation is increasing. Therefore, in this paper, we introduced a novel memory debugging platform that is composed of enhanced DUMA and memory swapping scheme (i.e., Zram). Especially, our memory swapping scheme efficiently avoids the memory space waste by saving only user data into the swap area. Our experimental results show that our swapping scheme significantly reduces the memory usage by up to 1.8 times in the best case.

References

[1] V. Van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future," in *Proc. of the Research in Attacks, Intrusions, and Defenses*, pp. 86-106, 2012.

[2] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 89-100, 2007.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. of the USENIX Annual Technical Conference*, pp. 309-318, 2012.

[4] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: Memory safety for shielded execution," in *Proc. of the 12th European Conference on Computer Systems*, pp. 205-221, 2017.

[5] Intel memory protection extensions enabling guide (rev.1.01), https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf, 2016.

[6] H. Aygün and M. Eddington, "Duma-detect unintended memory access," http://duma.sourceforge.net/, 2013.

[7] N. Nethercote, "Dynamic binary analysis and instrumentation," PhD thesis, University of Cambridge, 2004.

[8] O. Hernandez, H. Jin, and B. Chapman, "Compiler support for efficient instrumentation," in *Proc. of the Advances in Parallel Computing*, pp. 661-668, 2008.

[9] Y. Gribov, M. Guseva, A. Ryabinin, J. Kwon, S. Lee, H. Lee, and C. Woo, "Fast memory debugger for large software projects," *International Journal of Open Information Technologies*, Vol. 3, No. 9, pp. 26-30, 2015.

[10] N. Gupta, "Compcache: In-memory compressed swapping. now renamed as zram," http://lwn.net/Articles/334649/, 2014.