Moving from Logical Sharing of Guest OS to Physical Sharing of Deduplication on Virtual Machine

Kuniyasu Suzaki† Toshiki Yagi† Kengo Iijima† Nguyen Anh Quynh† Cyrille Artho† Yoshihito Watanebe‡

† National Institute of Advanced Industrial Science and Technology ‡ Alpha Systems Inc.

Abstract

Current OSes include many logical sharing techniques (shared library, symbolic link, etc.) on memory and storage. Unfortunately they cause security and management problems which come from the dynamic management of logical sharing; e.g., search path replacement attack, GOT (Global Offset Table) overwrite attack, Dependency Hell, etc. This paper proposes that self-contained binaries eliminate the problems caused by logical sharing. The memory and storage overheads caused by self-contained binaries are mitigated by physical sharing (memory and disk deduplication). The effect of deduplication was investigated on the KVM virtual machine with KSM (Kernel Samepage Merging) and LBCAS (Loopback Content Addressable Storage).

1. Introduction

Current OSes include many logical sharing techniques that reduce consumption of memory and storage. For example, dynamic-link shared library is a technique to share common functions and reduce memory usage. Symbolic link is a technique to share files and reduce storage usage. These techniques are useful, but they require dynamic management and cause some problems. A dynamic-link shared library of ELF format has a security issue called search path replacement attack[7] and GOT (Global Offset Table) overwrite attack [7,14]. A symbolic link has version mismatch problem called Dependency Hell. Parts of the problems can be solved by a static-link and substantial copy. Unfortunately, this approach requires source code, and many applications deeply depend on logical sharing. Furthermore, this solution increases memory and storage usage.

On the other hand, memory and storage virtualization is now advanced, and physical sharing, called deduplication, has become popular. Deduplication is a technique to share same-content chunks (chunk is a unit of continuous data in a block image) at low level, reducing the total usage. Memory deduplication [1,3,9,12,19] is mainly used to reduce same-content memory pages among virtual machine instances. Storage deduplication [5,13,15,18,20] is mainly used by CAS (Content addressable Storage), which reduces same-content chunks among software versions. The technologies have become popular in cloud computing, because they reduce the consumption of physical resources. The user does not need to care about redundancy of memory and storage.

This paper proposes the possibility of the replacement of logical sharing (dynamic-link shared library and symbolic link) by “self-contained” binaries which include dynamic-link shared libraries. The memory and storage overheads are mitigated by physical sharing (memory and storage deduplication). It shows the feasibility of deduplication on a single OS image to improve security.

This paper is organized as follows. Section 2 reviews deduplication for memory and storage. Section 3 introduces issues of logical sharing. Section4 describes the method to replace logical sharing with self-contained binaries. Section 5 evaluates the current implementation. Section 6 discusses future work, and Section 7 summarizes our conclusions.

2. Deduplication

2.1 Memory Deduplication

Memory deduplication is mainly used on a virtual machine monitor. The memory images of virtual machine instances include many same-content pages, especially when same guest OS runs on several virtual machines. The same-content pages can be merged on physical memory.

There are many recent implementations of memory deduplication. Early memory deduplication on a virtual machine monitor was implemented on Disco [3], which is called Transparent Page Sharing. VMWare ESX revised it as Content-Based Page Sharing [19]. Xen has two major implementations called Differential Engine [9] and Satori [12]. KVM uses KSM (Kernel Samepage Merging) [1], which is a general memory deduplication to merge memory image on a single process or multiple processes. The function of KSM is included from Linux kernel 2.6.32.

2.2 Storage Deduplication

Storage deduplication is mainly used by CAS (Content addressable Storage), which has become a popular method to manage disk images for many OSes. In CAS systems, data is addressed not by its physical location but by a name that is derived from the content of that data (a secure hash is used as a unique name usually). A CAS system can reduce its total volume by deduplication, which aggregates same-content chunks with a unique name.

CAS systems are divided into two categories, fixed- or variable-length chunk. Venti [13], CASPER [18] and LBCAS [15] use fixed-length chunk. DeepStore [20] and NEC-Hydra[5] use variable-length chunk. From the view of management overhead, fixed alignment is easy to treat.

The effects of deduplication on several operating systems were evaluated in [10, 11]. Liguori [11] reported that Linux distributions (Fedora, Ubuntu, and OpenSUSE) had many same-content chunks and 10% of the image was deduplicated among the Linux distributions. Jin [10] reported that the effect of deduplication on a single OS image was not large except for zero-cleared chunks.

3. Drawback of Logical Sharing

3.1 Problems of Memory Sharing

Dynamic-link shared library is a popular technique to reduce memory usage, but it has security and maintenance problems.

Dynamic-link has a problem called “search path replacement attack”. Dynamic-link searches a shared library at run time using a search path. Search path is defined by an environment variable, for example, “LD_LIBRARY_PATH” environment variable on Linux. It is convenient because shared library is replaced for each process. However, if a search path is replaced for malicious library, a malware is loaded easily, because caller program has no methods to certify libraries.

ELF format for dynamic-link has a problem called “GOT (Global Offset Table) overwrite attack” [7,14]. The GOT redirects position-independent address calculations to an absolute location and is located in the .got section of an ELF executable or shared object. It stores the final (absolute) location of a function call (symbol), used in dynamically linked code. GOT is mapped to the data segment and easily overwritten by malware.

A dynamic-link shared library has a management problem called Dependency Hell (DLL Hell in Windows). A partial change of a library makes it incompatible with programs that were built against an earlier version. Windows has been particularly vulnerable to this, because of its emphasis on dynamic linking of C++ libraries and OLE (Object Linking and Embedding) objects. The same problem has occurred on Linux and MacOS.

A dynamic-link shared library also incurs performance problems. The dynamic binding of the indirect table takes much time. It was reported that the boot time of KDE was dominated by its dynamic binding and that half of the time of booting KDE was wasted. To solve the problem, some techniques have been proposed (i.e., prelink on Linux, prebinding on MacOS, etc.) but they are not widely used.

3.2 Problems of Storage Sharing

Symbolic links are a popular technique to reduce storage usage, but they also have problems.

Symbolic links can cause Dependency Hell easily, because most libraries are symbolic-linked in order to control minor updates. Although a package manager maintains the versions, symbolic links are easily replaced by hand. The adverse impact of this practice is hidden, because the caller program has no methods to certify a symbolic link.

4. From Logical Sharing to Self-Contained Binary

In order to mitigate the problems of logical sharing, we thought the replacement of dynamic-links with static-links would solve the problems. Unfortunately, current applications deeply depend on dynamic-link shared libraries and they are not easy to replace with static-link. This is due to flexibility of dynamic-link shared libraries, and due to avoiding license contamination problems. For example, the libraries of GNOME assume to be dynamic-link shared libraries. Gentoo, which builds all binaries from source code, ignores the configuration for the static-link compile option. Furthermore, this approach requires source code and does not apply to commercial applications.

Instead of a replacement with static-link shared libraries, we customized ELF executables with a “pseudo-static” converter. A pseudo-static converter includes dynamic-link shared libraries into an ELF executable. It aims to bring a binary onto another machine without the need to drag all its libraries. Some tools are developed for Linux; e.g., statifier[17], ermine[6], autopackage[2], etc.

In this paper we use statifier, which is open source software. Statifier takes a “memory snapshot” of a process, created by the loader when the loader has ALREADY finished relocation of the dynamic-link shared library (_ld_start() of ld-linux.so) and BEFORE the loader invokes any INT functions (_ld_start_user() of ld-linux.so). The relocation information and dynamic-link shared libraries are included into the executable ELF file as data. Statifier includes all shared libraries which include linux-gate.so, the special library for Linux system calls, and ld-linux.so, the ELF interpreter and loader.

The included relocation information and shared libraries are loaded by the starter of statifier, which is also embedded in the ELF file. The Linux kernel recognizes the "statifier"ed ELF binary as a static-link, because statifier includes the ELF interpreter (ld-linux.so) and there is no INTERP segment to call it. The ldd command shows no dynamic-link shared libraries in a "statifier"ed file.

The resulting self-contained ELF binary prevents search path replacement attack and Dependency Hell, because the shared libraries are included. Although GOT still exists in a "statifier"ed ELF file, but statifier mitigates GOT overwrite attacks, because the address prefixed and detects falsification with the relocation information in the ELF file.

The memory and storage overhead caused by statifier are mitigated by memory and storage deduplication, respectively.

5. Performance evaluation

This section describes the performance of self-contained binaries (statifier Linux) on deduplication, compared with normal Linux. We estimated the effect on two Linux distributions (Debian and Gentoo) and confirmed the same results. This section shows the result on Gentoo.

Gentoo (kernel 2.6.31) was installed on a 32GB virtual disk (31GB ext3, 1GB swap) as a guest OS on the KVM virtual machine. The ELF binaries under /bin (82 files), /sbin (74), /usr/bin (912), /usr/sbin (94) were customized by statifier.

The two virtual disks (original and statifier) were translated to LBCAS (LoopBack Content Addressable Storage) [15] which offered fixed-size chunk deduplication. KSM of Linux was used to investigate the effect of memory deduplication. KVM ran on Ubuntu 9.10 (kernel: vanilla-2.6.32.1) with 768MB memory for a virtual machine. KVM allowed using snapshot mode because LBCAS was configured as a read-only virtual device.

5.1 Effect of Dynamic-link Shared Library

At the beginning, we estimated the logical sharing effect of a dynamic-link shared library on a normal Gentoo installation. Memory usage was measured by exmap [8].

At the end of the boot stage, 42 processes were running. All of them used linux-gate.so, ld-linux.so, and ibc.so. Exmap showed the usage of virtual memory and real memory. The total virtual memory used by all processes was 127,928KB. The memory used for unique data (namely, except shared libraries) was 41,744KB. The real used memory was 54,760KB. From the results we confirmed that shared libraries used 13,016 (54,760 – 41,744) KB of real memory. The shared libraries were expanded on virtual memory and used 86,184 (127,928 – 41,744) KB. The effect of dynamic-link shared library was 6.62 (86,184/13,016). The result means that code of shared libraries overlapped 6.62 times in average. We confirmed that dynamic-link shared library reduced memory usage. The effect has to be transformed to deduplication.

5.2 Expansion by Statifier

In this section we show the impact of statifier on Gentoo. Statifier includes shared libraries into each binary and increases the volume of storage. Fortunately, the increase is the view of the guest OS, and it is mitigated by deduplication on a real storage.

In our experiments statifier transformed 1,162 ELF binary files in /bin, /sbin, /usr/bin, and /usr/sbin to self-contained binary files. Table 1 shows the increased caused by statifier on ELF files. The total volume of the original ELF binaries was 87.87MB. It was increased to 3572.9MB (40.66 times) by statifier, because ELF binaries included the all necessary dynamic-link shared libraries. The average of a binary file was changed from 75.6KB to 3,075KB (40.66 times). The biggest change was /usr/bin/gnome-open which included 22 dynamic-link shared libraries and increased from 5,400B to 8,732,672B (1617.16 times). The smallest change was /usr/bin/qmake which included 6 libraries (linux-gate.so, ld-linux.so, libc.so, libm.so, libgcc_s.so, and libstdc++.so) and increased from 3,426,340B to 6,094,848B (1.78 times).

Normal Gentoo used 3,754MB storage. Statifier increased it to 7,075 MB, which was 1.88 times bigger than the original.

Table 1. Increase caused by Statifier on ELF files.

	Original	Statifier	Increase
Total	87,865,480	3,572,936,704	40.66
Average	75,615	3,074,816	40.66
Max	5,400	8,732,672	1617.16
Min	3,426,340	6,094,848	1.78

5.3 Statifier versus static linking

A subset of 1,162 ELF files were re-compiled with static linking (57 in /bin, 22 in /sbin, 76 in /usr/bin, 9 in /usr/sbin). We compared the 185 static-link ELF files with statifiered files in table 2. It shows the total of statifiered ELF files are 2.63 times bigger than static-link. The biggest difference was with `bzip2recover` which increased 5.99 times with 3 shared libraries. The smallest difference was with `busybox` which was increased 1.56 times with 7 shared libraries. In all cases, we found that statifier produced larger files than static linking.

This subset was too small to impact the security and management problem due to library sharing.

The comparison was available on Gentoo because all ELF files are created from source code on a client machine. It would require more effort with distributions that are managed with binary packages.

Table 2. Size comparison for 164 ELF files created with statifier and with static linking. () Indicates the difference with dynamic linking. [] Indicates the difference between static linking and statifier.

	Original	Static link	Statifier
Total	8,065,092	99,734,044 (12.37)	262,627,328 (32.56) [2.63]
Average	49,177	608,134 (12.37)	1,601,386 (32.56) [2.63]
Max (bzip2recover)	9,512	527,252 (55.43)	3,160,672 (332.28) [5.99]
Min (busybox)	896,076	1,682,984 (1.88)	2,629,632 (2.93) [1.56]

5.4 Result of Memory Deduplication

We investigated the memory usage on normal and statifier Gentoo with or without KSM. Figure 1 shows the consumed 4KB memory pages.

Statifier Gentoo required 2.64 (86,506/32,706) times more memory than normal Gentoo. The result was caused by redundant loading of the same shared libraries.

Fortunately, redundant loading was deduplicated by KSM. The required physical memory was reduced to 34.4% (29,732 pages from 86,506), which was almost same to the normal Gentoo (30,410 pages with KSM). The small decrease might be caused by the memory management of shared library, because a self-contained shared ELF binary releases its memory when it terminates. This result means memory deduplication brings the same benefit as dynamic-link shared libraries. Interestingly, the memory of normal Gentoo was also deduplicated by KSM and reduced to 93.0% (30,410 pages from 32,706). It means that there is redundant memory on normal Gentoo.

Figure 1 also shows the number of unique pages and deduplicated pages on KSM. Normal Gentoo had many unique pages (29,928), because dynamic-link shared libraries are counted as unique pages. On the other hand, statifier Gentoo has fewer unique pages (25,219) than normal. It was caused by shared libraries which weere treated as deduplicated pages.

Figure 2 shows the trace of memory deduplication at boot time on normal and statifier Gentoo, using either the Loopback device or LBCAS. The results show that physical memory consumption is almost the same on normal and statifier but the boot time is delayed by the overhead of deduplication. The overhead at boot time is mentioned in Section 5.6.

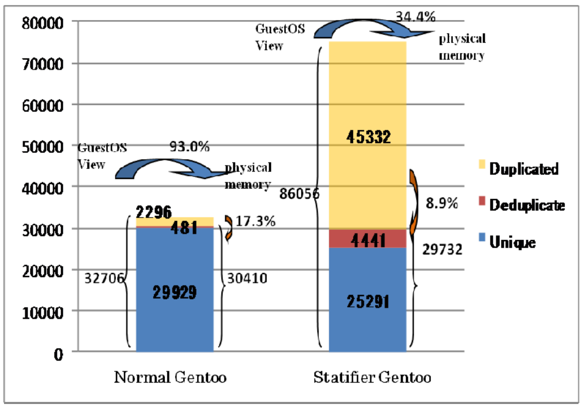

Figure 1. Physical memory pages (4KB) used by Normal Gentoo and Statifier Gentoo on KVM with/without KSM.

5.5 Result of Storage Deduplication

We investigated the effect of deduplication on Gentoo with LBCAS (fixed chunk size was varied by 16KB, 64KB and 256KB). Table 3 shows the results. The left side shows the total storage size ("static"), and the right side shows the volume of required chunks at boot time ("dynamic").

The static image of statifier Gentoo was 1.88 times bigger than the normal on the loopback device. However, the ratios were reduced by LBCAS, because the same chunks were deduplicated. The smaller LBCAS chunk size showed the lower ratio of increase (1.04 at 16KB, 1.12 at 64KB, 1.18 at 256KB LBCAS) because the smaller LBCAS chunk size allowed more chunks to be deduplicated than the larger LBCAS chunk size.

The dynamic image of statifier Gentoo was 2.25 times bigger than normal on the loopback device. The ratio was higher than the static image, because executable ELF binaries were expanded by statifier. In the case of 16KB, auto login was timed out and

Table 3. Number of chunks on LBCAS (fixed chunk size was varied by 16KB, 64KB, and 256KB.). The upper row shows the volumes required by the guest OS. Left columns show the numbers of the static storage image. Right columns show the numbers of required chunks at boot time. The parenthesis indicates the ratio of statifier compared to normal.

	Static		**Dynamic (boot)**	
	normal	**statifier**	**Normal**	**statifier**
Volume GuestOS	**3,754MB**	→ **7,075MB (1.88)**	**151.7MB**	→ **341.0MB (2.25)**
16KB	268,454	→ 278,499 (1.04)	---	→ ----
64KB	74,679	→ 83,863 (1.12)	3,481	→ 4,866 (1.40)
256KB	22,806	→ 26,892 (1.18)	1,560	→ 2,019 (1.29)

booting stopped at the Gnome graphical login GDK. It was caused by the overhead of 16KB LBCAS. The ratio on 64KB and 256KB was also reduced by LBACS but the effect was different from the static image. It depended on the ratio of deduplication and the number of accesses to a page.

In any case the results show that the storage expansion caused by statifier can be reduced by deduplication.

5.6 Boot Time

We investigated the boot time with and without KSM and LBCAS. The boot time was logged until the end of auto login. Table 4 shows the results, which indicate the overhead of KSM and LBCAS.

In any case the overhead with KSM was larger than without KSM. It was caused by memory deduplication. Especially statifier cases showed a larger overhead than normal, because there were more occurrences of memory deduplication. The volume of deduplication was confirmed in Figure 2.

The overhead of LBCAS was also large on statifier Gentoo. It was caused by more I/O requests than normal, because statifier expanded each binary and increased the volume of I/O. In Table 3 shows that statifier required 2.25 times more volume than normal at boot time. However the overhead was not proportional to the LBCAS chunk size, because many chunks were deduplicated and statifier required 1.29 times more than normal.

In the case of using loopback without KSM, statifier was faster than normal. It was caused by the elimination of relocation time, symbol resolution time, and binary loading time by statifier, which has the same effect as prelink.

In any case KSM and LBCAS caused time overhead because it was a trade-off between CPU and storage. Considering the improved security, we regard the overhead as acceptable.

Table 4. Boot time without/with KSM on Loopback and LBCAS.

	Without KSM		With KSM	
	Normal	Statifier	Normal	Statifier
Loopback	95s	84s	95s	105s
LBCAS (256KB)	107s	108s	115s	130s

6. Discussion

SLINKY[4] uses the same approach, but it requires a special kernel, because SLINKY does not use a virtual machine, and memory and storage deduplication are not offered. Our approach is practical on virtual machine environments, especially cloud computing. The method can be applied to other operating systems easily.

Satisfier is a substitution of static-link shared libraries. We could use sta.li [16] which has all static-link shared libraries and no /lib directory. However, it has many restrictions because current Linux applications deeply depend on dynamic-link shared libraries. The replacement by pseudo-static is practical and applicable to existing Linux distributions. On the other hand, there are some projects that increase the number of self-contained binaries for application migration, e.g, Google NaCl (Native Client) and VMWare ThinApps (former Thinstall) on Windows. They require more memory but deduplication will mitigate it.

7. Conclusions

This paper describes the possibility of replacing logical sharing by self-contained binaries. The overhead of memory and storage is mitigated by physical sharing (memory and disk deduplication). A self-contained binary mitigates the problems which come from the dynamic management of logical sharing, such as search path replacement attack, GOT overwrite attack, and Dependency Hell.

Experiments demonstrated the effect of self-contained binaries, memory deduplication (KSM), and storage deduplication (LBCAS). Self-contained binaries increased the files to 40.66 times bigger than normal, but deduplication mitigated the overhead to less than 1.4 times on memory, storage, and time.

Current deduplication has been used for multiple virtual machine instances or multiple versions of storage images. However, this paper shows that deduplication is useful on a single OS image. This direction indicates a new use for deduplication.

Reference

[1] A.Arcangeli, I.Eidus, and C.Wright, Increasing memory density by using KSM, Linux Symposium, 2009.

[2] Autopackage, http://autopackage.org/

[3] E.Bugnion, S.Devine, and M.Rosenblum, Disco: Running Commodity Operating Systems on Scalable Multiprocessors, Symposium on Operating Systems Principles, 1997.

[4] C.Collberg, J.H.Hartman, S.Babu, and S.K.Udupa, SLINKY: Static Linking Reloaded, USENIX Annual Tech Conf 2005.

[5] C.Dubnicki, L.Gryz, L.Heldt, M.Kaczmarczyk, W.Kilian, P.Strzelczak, J.Szczepkowski, C.Ungureanu, and M.Welnicki. HYDRAstor: A Scalable Secondary Storage, USENIX Conference on File and Storage Technologies (FAST) 2009.

[6] Ermine, http://www.magicermine.com/

[7] J.Erickson, Hacking: The art of exploitation, O'Reilly, 2003.

[8] exmap http://www.berthels.co.uk/exmap/

[9] D.Gupta, S.Lee, M.Vrable , S.Savage, A.C.Snoeren, G.Varghese, G.M.Voelker, and A.Vahdat, Difference Engine: Harnessing Memory Redundancy in Virtual Machines, USENIX Symposium on Operating Systems Design and Implementation, 2008.

[10] K.Jin and E.L.Miler, The Effectiveness of Deduplication on Virtual Machine Disk Images, The Israeli Experimental Systems Conference (SYSTOR), 2009.

[11] A.Liguori, and E.V.Hensbergen, Experiences with Content Addressable Storage and Virtual Disks, First Workshop on I/O Virtualization (WIOV), 2008.

[12] G.Miło´s, D.Murray, S.Hand, and M.A.Fetterman, Satori: Enlightened page sharing, USENIX Annual Tech Conf 2009.

[13] S.Quinlan and S.Dorward, Venti: A New Approach to Archival Storage, USENIX Conference on File and Storage Technologies, 2002.

[14] R.C.Seacord, Secure Coding in C and C++, Addison-Wesley, 2006.

[15] K.Suzaki, T.Yagi, K.Iijima, N.A.Quynh, and Y.Watanabe, Effect of readahead and file system block reallocation for LBCAS (LoopBack Content Addressable Storage), Linux Symposium 2009.

[16] stali, http://sta.li/

[17] Statifier http://statifier.sourceforge.net/

[18] N.Tolia, M.Kozuch, M.Satyanarayanan, B.Karp, A.Perrig, and T.Bressoud, Opportunistic use of content addressable storage for distributed file systems, USENIX Annual Tech Conf 2003.

[19] C.A.Waldspurger, Memory Resource Management in VMware ESX Server, Symposium on Operating Systems Principles, 2002.

[20] L.L.You, K.T.Pollack, and D.D.E. Long, Deepstore: An archival storage system architecture. Proceedings 21st International Conference on Data Engineering (ICDE) 2005.

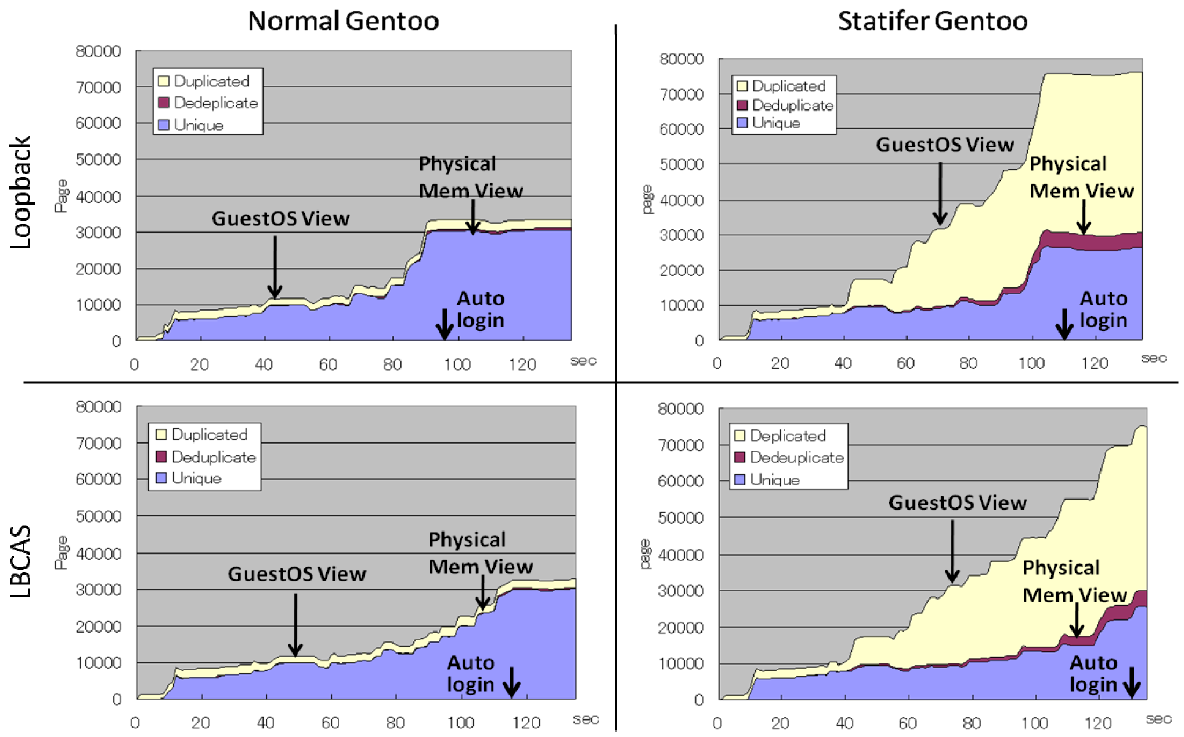

Figure 2. Trace of memory usage on the deduplication of Linux-KSM at boot time. The upper and lower rows show the results on a normal loopback device and on LBCAS. The left and right columns show the results on Normal Gentoo and on Statifier Gentoo.