# Collaborative Memory Management in Hosted Linux Environments

Martin Schwidefsky
*IBM Deutschland Entwicklung GmbH*
Martin.Schwidefsky@de.ibm.com

Hubertus Franke
*IBM T.J. Watson Research Center*
frankeh@us.ibm.com

Ray Mansell
*IBM T.J. Watson Research Center*
Ray.Mansell@us.ibm.com

Himanshu Raj
*Georgia Tech*
rhim@cc.gatech.edu

Damian Osisek
*IBM Systems and Technology Group*
dlosisek@us.ibm.com

JongHyuk Choi
*IBM T.J. Watson Research Center*
jongchoi@us.ibm.com

## Abstract

In hosted environments, multiple guest operating systems are hosted on top of a host operating system or hypervisor. The problem of overcommitting physical memory is either solved by dynamically adjusting the memory sizes of the guests or through transparent host paging. Both approaches can introduce significant overhead in heavily overcommitted memory scenarios due to frequent resize requests or due to high paging I/O activity. This paper introduces a novel approach to this problem, called collaborative memory management (CMM). In CMM, guests and host operating system exchange page usage and residency information. This information is primarily used by the host to reduce the amount of paging it needs to do for the pages of its guests. The CMM design and the Linux implementation and a prototype for Linux for zSeries and the z/VM hypervisor will be discussed.

## 1 Introduction

With the re-emergence of virtual machines (VMM) as a means for workload and server consolidation, memory pressure again has become an important issue to solve. The problem of memory pressure stems from the fact that guest operating systems, such as Linux, attempt to utilize any available memory given to the guest for its own caching purposes. As a result a static "partitioning" of the system would significantly be limited by the available memory in the system. A static memory partitioning is also contrary to the nature of many system utilizations seen today, often bursty and with time varying resource requirements (whether cpu, memory, or I/O). It is exactly this variability that virtualization tends to exploit.

Memory overcommitment is an attribute of the application mix that runs on a system and as such can not be eliminated. Ultimately, the memory pressure resulting from memory overcommitment has to be dealt with by either pushing it back into the guest OS or by resolving it in the host. Therefore, there is the potential for a high paging I/O rate in either the host, the guest, or in both. High paging rates have nonlinear impact on the application and system response times and thus can limit the number of guests that can effectively be hosted. This nonlinear performance impact makes dealing with memory overcommitment unique as compared to overcommitting other resources. Nevertheless, through proper global memory management, one can hope to reduce the symptoms experienced due to memory overcommitment.

With respect to memory management among multiple guests, two main approaches to overcommitting memory are commonly deployed:

- **Dynamic Partitioning**: individually guest OSes are forced to dynamically change their memory size to accommodate a global memory strategy.
- **Memory Virtualization**: the host swaps/pages guest memory similar to how any operating system overcommits its memory to its applications.

The work in this paper was motivated by IBM's Linux virtualization stack for the zSeries, which virtualizes guest Linux systems over the z/VM host/hypervisor. Besides the guest memory paging, z/VM also deploys dynamic partitioning. We have seen that customers deploy hundreds of virtual machines over a single host, sometimes resulting in unsustainable memory overcommitments that neither dynamic partitioning nor memory virtualization can satisfy to meet quality of service expectations with regard to response times.

Dynamic partitioning is the only possible means when thin, non-paging hypervisors are deployed. An example is XEN [2], which para-virtualizes page table, i.e. the hypervisor merely ensures that the guest creates only valid mappings for the memory assigned to it. The dynamicity of the memory sizing is achieved by a technology known as ballooning [3, 2]. A balloon driver, operating in each guest, communicates with the hypervisor and receives directives for modifying the guest memory size. This is accomplished by allocating pages, thus often forcing the guest's page reclamation daemon to run, and returning those pages to the hypervisor for allocation to different guests. The hypervisor disables access to these pages for the donating guest and enables access to them for the receiving guest. By growing and shrinking memory balloons, OS memory sizes can be adopted to deal with changing memory requirements by individual operating systems and in the system overall.

In stable workloads with infrequently changing guest working set sizes, the ballooning method is quite adequate to "squeeze" the guests into their right size. However, in highly overcommitted memory scenarios with rapidly changing or bursty memory requirements, the ballooning approach to memory overcommitment does pose various shortcomings. A host agent (typically running at the hypervisor level) has to constantly estimate working set sizes (for instance through monitoring the page fault rates and dispatch rate of its guest partitions) and resize the guest OS memory sizes. Though memory size estimation can be done with limited overhead[3], the time to invoke the guest operating system (idle or not) and execute the ballooning module can lead to reduced response times at the other guest(s) requiring more memory. Furthermore, it can pose a scalability problem when hundreds of guest OSes are to be hosted. First, the amount of memory harvested per image per balloon invocation decreases with the number of images. Second, the achievable overcommitment is limited as guest images require a minimum amount of memory to operate and to avoid the dreaded OOM killer.

Memory virtualization as realized through host operating system paging of its guests is the alternative approach to overcommitting memory. In highly overcommitted scenarios, host paging can provide a more responsive approach, because (i) the host swaps guest OS memory based on a global usage view, able to steal pages from other guests and (ii) memory can be overcommitted beyond the sum of the minimal guest sizes. On the other hand host paging has its own set of problems, most of which are related to a very high swap rate. For instance, assume that the host has elected to swap out an older page. If the guest now decides that this page needs to be swapped at the guest level, it needs to be first brought back at the host level. Other examples are unused guest pages that the host blindly pages out. This clearly identifies that there are overheads and unnecessary operations involved at this end of the spectrum of dealing with memory pressure.

What is needed is a balanced approach that allows us to reap the benefits of both approaches, while avoiding their shortcomings. Hence, in this paper we introduce the *Collaborative Memory Management* (CMM) framework. CMM deploys "infrequent" ballooning (we refer to it as CMM1) to apply sufficient long-term pressure on the various guests. There is limited focus on this part in this paper, as this is a well known approach in the literature [3]. Instead, in this paper we focus on the second component of CMM, namely CMM2, a novel information-sharing between host and guests (we refer to it as CMM2). CMM2 enables us to identify and avoid unnecessary host operations, thus reducing the host paging rate and improving response and throughput of memory allocation requests for a guest OS when overall system memory is overcommitted.

We have implemented the CMM framework for IBM's z/Architecture System z9 and in particular using the z/VM host and the Linux guest. The remainder of the paper is organized as follows. The general framework of CMM is described in Section 2. The prototype implementation of this framework utilizing Linux guests and the z/VM hypervisor is discussed in Section 3. The changes we made to the z/Architecture and the z/VM host are described in Section 4. The current state of our analysis is presented in Section 5. Conclusions, ongoing work, and future directions are discussed in Section 6.

## 2 Collaborative Memory Management Framework

The introduction already identified ballooning and host paging as the two fundamental approaches to memory overcommitment. It also identified their individual drawbacks, namely overhead and increased latency by inducing pressure on the guests to release memory in the case of ballooning, and increased host paging activity in the case of host paging.

Ultimately, we believe a combined approach that (i) deploys ballooning to deal with the longer range shaping of guest memory sizes and (ii) utilizes host swapping for the short term oscillations in memory requirements, (iii) utilizes host paging for the case where ballooning can no further reduce the guest memory sizes due to minimum operating memory requirements by the guests, promises the best results.

The host deploys its own global host page eviction algorithm (LRU) and can identify pages that have system-wide aged the most, where as an individual guest only has a limited view of its own pages. On the other hand the host does not have any knowledge about the utilization of a guest page and as a result it must save the content of a guest page to the host swap area. The basic idea of CMM2 within collaborative memory management is to allow a highly efficient mutual sharing of page status information between the guest and the host operating system in order to optimize for overall system performance and in order to reduce unnecessary swap operations in the host.

In CMM2, the guest defines and maintains the page usage state for each of its absolute pages. By doing so, it indicates the content preservation requirements for each page expected by the host page management. Equipped with this information, the host/hypervisor knows at all times and precisely whether a guest page that has been selected for eviction/swapping, needs to be preserved or not. Furthermore, the host can make more informed decisions concerning which guest memory pages to steal, thus minimizing the conflicts that otherwise inevitably arise when two systems both believe they are solely responsible for managing storage. In the same manner knowing residency information of its absolute memory can be utilized during a guest's paging operation. For instance, a guest trying to swap a page that has already been swapped by the host, creates a scenario known as dual-swapping, where the guest needs to have the page resident to swap it to its own swap device, thus creating two additional I/O operations. This can be avoided based on having access to the residency information.

The *page state* for each guest absolute memory page (or its associated host virtual page), defined by the cross product of the *page usage state* and the *page residency state*, is maintained by and shared through a page hypervisor assist facility (**HVA**).

The **usage states** of a page are as follows:

- Stable (**S**): This is the default state for guest memory pages. The content remains what the guest sets it to; the host is responsible for preserving the page content.
- Unused (**U**): The page content is meaningless to the guest; the host may discard the page content at will.
- Volatile (**V**): The guest has indicated that it can tolerate the loss of the page content; however, the page still contains data that may be useful in the future.
- Potential Volatile (**P**): This state is similar to the volatile state. The guest can tolerate the loss of the page content as long as the page has not been modified. Page modification is indicated by the page dirty bit. This bit needs to be accessible by the host. If the host can not access the dirty bit then the state machine can be simplified by removing this state.

While the usage state of a page is primarily modified by the guest, the page residency state is only to be modified by the host, though the guest can query it. The **residency states** of a page are as follows:

- Resident (**r**): The page is assigned to a backing frame in host memory and may be referenced at machine speed.
- Preserved (**p**): No frame is associated; the host has written the page contents to auxiliary swap storage.
- Logically zero (**z**): There is no associated frame or backed content. The content of the page is considered to be zero.

Thus the page state can be represented in 4-bits per page. The details and maintenance of the HVA, such as the location of the page state bits and the means to issue the host service call, are obviously highly architecture dependent. Nevertheless, various constraints must be satisfied. Only the host must be allowed to modify the residency state; modifications by the guest would pose a serious functionality and security problem. Certain guest operations must be conditional based on the host state. For instance, an operation frequently used is `SetStableIfNotDiscarded` which only makes a page stable if it has not been discarded. Thus the most obvious implementation, namely allocating or mapping a page status vector in the guest with r/w permission is erroneous. However, separating the guest and host state poses the problem of atomic access, as states need to be modified and accessed atomically to ensure proper synchronization between guest and host. Allowing a lock to span across guest and host is a serious design flaw, as it would allow a faulty guest to lock up the host. Instead, for atomic accesses a `compare_and_swap` needs to be utilized.

Therefore, the most sensible implementation is to maintain the page state only in the host. The host can utilize load/store access to the page state. The guest uses a host service call to modify the page state. The primitives the host has to provide to the guest are the following: `SetStable`, `SetUnused`, `SetVolatile`, and `SetPotentialVolatile` which set the page usage state to the requested target state (**S**, **U**, **V**, or **P**), and the already mentioned `SetStableIfNotDiscarded`.

The state transition diagram is shown in Figure 1. There are several noteworthy comments to be made. The states **Vz** and **Pz** mark the special "discarded" condition of a page entered through a previous host discard operation. If a guest accesses a **Vz** or **Pz** page, the host will present a special discard fault to notify the guest that the page has been removed and that it needs to be recreated by the guest.

For reasons of symmetry and architectural completeness, the $\{\mathbf{S},\mathbf{V},\mathbf{P}\}\mathbf{p} \rightarrow \mathbf{Up}$ transition is included in the state diagram. In principle, a **Up** state makes little sense, as the backing

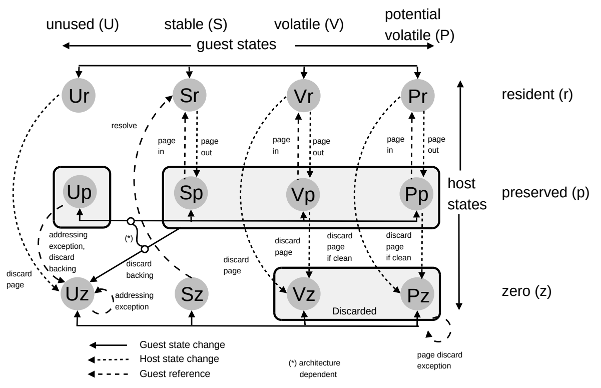


Figure 1: State Transition Diagram

storage for this page would have to be maintained despite the fact that the page is unused. However, in an implementation where the guest state can be manipulated without the involvement of a host service, this is the only valid path. Subsequently moving the page into **Sp** and accessing it would force a reload of the page from the host swapping area, in which case the opportunity for the elimination of a host swap operation is lost. In contrast, in implementations where the guest accomplishes all guest transitions through a host service, the $\{\mathbf{S},\mathbf{V},\mathbf{P}\}\mathbf{p} \rightarrow \mathbf{Uz}$ transition can be immediately made and at the same time the backing storage can be freed. In more general terms, if the guest page state transitions are implemented through a host service call, we can always tag an implicit host state transition onto that guest page transition in order to optimize operations like in the **Up** vs. **Uz** case. The other case is also true, namely that host page transitions can cause implicit guest state transitions. The state machine can be simplified in several ways if the implementation for a particular architecture requires and/or allows it:

- Collapse the two discarded states **Vz** and **Pz** to a single discarded state **Vz**.
- Remove the **Vp** state. If the host can not profit from preserving volatile pages, it

can always choose to discard pages that would enter the **Vp** state.

- Remove the **Pp** state. Without this state it depends on the dirty bit what happens with the page. If the page dirty bit is set, the host needs to preserve the page and will set the combined target state to **Sp**. If the page is clean the host can discard the page.
- Remove the potentially volatile **P** page usage state. This simplification is necessary for architectures that do not have hardware per page dirty bits and no reasonably fast alternative way to access the page dirty information from the host system.

Equipped with this framework, CMM2 now requires the guest operating systems to identify its *discardable pages*. It is reasonable to expect that both guest and host deploy some form of LRU algorithm and that the aging order of pages established in the guest is also roughly established by the host. The benefit of CMM2 hence comes from the fact that when the host discards a page based on its LRU information, it conceptually does what the balloon driver would have done (namely identifying an old page and evicting it). However, it does so with reduced latency since the guest and its balloon driver do not have to be scheduled. The fact that a page was discarded will be recognized the next time the page is accessed by the guest (at which point a discard fault is obtained) or during the guest's own reclamation process (at which time no extra cost is incurred).

## 3 Linux Guest Implementation

The goal of the guest implementation for the collaborative memory management optimization is to mark free pages as unused and to get as many pages into volatile or potential volatile state as possible. Since the host can choose to remove unused and volatile pages anytime and potential volatile pages if they are not dirty, there needs to be special cleanup code to deal with discarded pages if the guest tries to access them.

For free pages only two state transitions are needed: the free operation of the buddy allocator sets all pages of the freed block to unused, and the allocation operation makes the pages stable again. A guest access to an unused page is a programming error, the host implementation can either return an arbitrary value to the guest instruction—preferably zero for security reasons—or present some kind of exception. No additional code is required to deal with accesses to unused pages.

The host ensures that all pages of a Linux guest have an initial page usage state of stable (**S**). In case of z/VM as the host, the initial page state is stable, logically zero (**Sz**). When the pages are added to the buddy allocator their page usage state changes to unused for the first time. All other pages that are not entered into the buddy system will always have a page usage state of stable.

For each class of non-free pages that are considered for one of the volatile states, additional code is required to clean up after a discard fault. For the majority of page allocators in the kernel the amount of code necessary to deal with the discard faults makes it hard if not impossible to make the pages volatile.

### 3.1 Volatile page and swap cache

The two classes of pages with the biggest potential are the page cache and the swap cache. The amount of code that is needed for the state transitions and to deal with the discard faults is acceptable and usually there are many pages in the page or swap cache that can be made volatile. All clean page and swap cache pages that do have a backing on secondary storage are candidates for one of the volatile states.

In an ideal situation all clean, read-only pages in the page and swap cache which do have a backing would be volatile, and all read-write pages with a backing would be potentially volatile. There are several conditions that either preclude or make it hard to keep the pages in a volatile state. For each user of a cached page the page either needs to be made stable or there is code in the discard fault handler that is able to remove the reference to the page from that user. For example, each reference to a page in a page table represents a user of the page. The discard fault handler is able to remove these entries for discarded pages. On the other hand each page address involved in an I/O operation represents a user as well but the discard fault handler is not able to remove these entries.

To avoid having to keep track of each individual user of a page, a simple strategy is used. Whenever the Linux memory management does something with a page that the discard fault handler can not undo, the page is made stable. After the memory management removed a condition that made it necessary to keep the page in stable state, it is *attempted* to make the page volatile again. This attempt can fail due to the following reasons:

1. The page is reserved. Reserved pages are special and may never be removed from memory by the Linux guest, nor discarded from memory by the host.
2. The page is marked dirty in the Linux internal page structure. The page content is more recent than the data on the backing device. The page content needs to get written to the backing device first before the page can be removed or discarded.
3. The page is in writeback. The page content is still needed until the I/O operation has finished.
4. The page is locked. As long as the page is locked the code that acquired the lock has exclusive access to the page.
5. The page is anonymous. The page does not have a backing, the only copy of the page content is in memory.
6. The page has no mapping. Again the page has no backing, the guest can not recreate the page.
7. The page is not up to date. An I/O operation to get the page content into memory has not yet completed. It does not make sense to discard the page before it has been up to date once, particularly since the I/O was likely started due to an access.
8. The page is private. There is additional information associated with the page via the `page->private` pointer, e.g. journaling data. To keep things simple, pages with private information are kept stable.
9. The page is already discarded.
10. The page map count is not equal to the page reference count minus one. There is one reference for the cache itself and one for each mapping of the page to a user space process. The discard fault handler can remove the cache entry and the user space mappings but not the references of any other user of the page.
11. The page has writable mappings, but the platform lacks the potentially volatile state.
12. The page is mlocked. The semantics of memory locked pages it that they are available without doing guest I/O, therefore the page has to be stable.

If any of these conditions is true, the page can not be made volatile. These are the rules for the state transition to a volatile state, however, the page state does not necessarily have to be adjusted if one of the conditions changes. It depends on the operation that is done with the page if the page state needs to change. As a rule of thumb, transitions to stable state are non-negotiable. Transitions to less stringent states (volatile or unused) can be done at a more convenient time and with the idea in mind to keep the hot code paths lean.

## 3.2 Page and swap cache state transitions

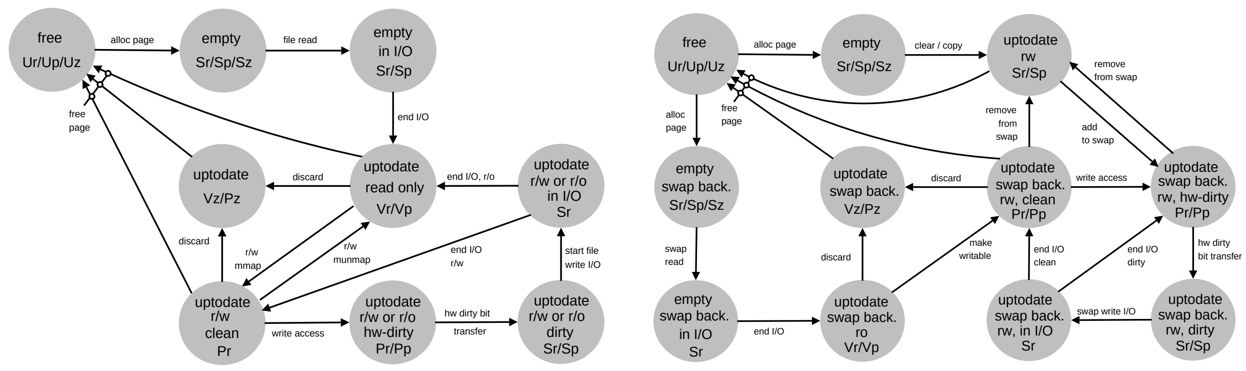
The page usage state transitions can be divided into transitions to stable state and the attempts to do a transition to a volatile state. For the transition to stable state there is always a user of a page who requires the stable state. The prevalent method to get a new reference to a page is to use `find_get_page` or one of its variants. To give back a reference `page_cache_release` is used. There are only three more relevant code paths in regard to the transition to stable state, namely the `get_user_pages` function, and the copy on write breaks in `do_wp_page` and `do_no_page`. The state transitions are conditional through the `SetStableIfNotDiscarded` call, which only moves a page into stable if the page has not been discarded. If the page was discarded, it is removed from the page cache and functions return `notfound`. In case of the copy on write breaks, the operations fails with `VM_FAULT_DISCARD` and the instruction that triggered the copy-on-write is repeated. This will cause a standard page fault for a non-existent page and the page will get loaded again.

The question when to try to move a page into volatile state is not defined as sharply as the question when a page needs to be stable. In principle the attempt to make a page volatile can be done anytime. To get the maximum number of pages into volatile state, a check of all twelve conditions would be required whenever one of the conditions becomes false. Due to concurrent operations in the memory management this would be difficult to implement and the resulting code would be slow. We can afford to be less stringent for the state transitions to volatile, there is no harm done if a small percentage of the suitable pages are not made volatile. By experimentation we found that it is enough to do the checking for the volatile transition when a page gets unlocked, when it has finished writeback, when the page reference counter is decreased, and when the page map counter is increased.

To get an idea how the state of a page changes during the lifetime of the page, see the two diagrams in Figure 2, which represent the state transitions based on various events in the kernel for two common types of pages, shared filemapped pages and anonymous pages. The diagrams only show the state changes due to read / write access via memory mapped pages. There are other triggers for I/O operations that are not covered in the diagrams, as they would get too complex.

## 3.3 Concurrent page state updates

In a multiprocessor system the usage state of a page can get updated concurrently on different processors. To ensure that the page has the correct state, a make stable operation may not "overtake" the attempt to make it volatile. If the make volatile has already done all the necessary checking, it will proceed with a `SetVolatile` operation. If at the same time another user of the page does a `SetStableIfNotDiscarded`, it depends on the timing if the page state is volatile or stable after the two operations complete. The check of the twelve conditions and the `SetVolatile`



(a) shared filemapped page. (b) anonymous page.

Figure 2: LifeCycle of two common Types of Pages in Linux

need to be done atomically in regard to the `SetStableIfNotDiscarded` and one of the conditions need to evaluate to true before doing the `SetStableIfNotDiscarded`. To provide the atomicity, a new page flag `PG_state_change` is used. The function that makes a page stable will wait until it can acquire the new page flag to give it exclusive access to the page state.

The make volatile operation does not have to wait, it can just return instead. The current implementation does this to avoid a potential dead lock on the `PG_state_change` bit. The worst thing that can happen is another suitable page not in a volatile state. The end of I/O interrupt usually releases the page lock which results in a try to make a page volatile. If a cpu is interrupted while holding the `PG_state_change` bit for a page this would be a dead lock if the make volatile function waits for the bit as well. The alternative solution would be to disable the interrupts while holding the `PG_state_change` bit. Disabling interrupts is expensive, therefore the preferable solution is to let the make volatile function return immediately if the `PG_state_change` bit is unavailable.

### 3.4 Memory locked pages

The `mlock()` system call needs special attention in regard to discardable pages. A memory locked page may not be removed from the page or swap cache. This means that memory locked pages need to be stable. The function that tries to make a page volatile needs a way to check if a page has been locked. This information is kept in the flags field of the virtual memory areas that refer to the page. To avoid traversing vma lists, which could significantly impact performance, a field is added in the struct address space. This flag field is set in the `mlock()` code when a vma of the address space gets locked. The flag is never removed; once the address space of a file had an mlocked vma, all future pages added to it will stay stable. The already present pages are made stable with a call to `get_user_pages`.

### 3.5 Writable page table entries

For writable pages there is code required that allows the pages to be put into the correct state. For platforms without the ability to access the guest page dirty bit information from the host, the correct state is the stable state, for platforms with the ability, it is the potentially volatile state. In both cases, whenever a writable page table entry is created, a call to a function is required that checks if the page state needs to be corrected. The state change has to be done before the first writable mapping is established. To avoid unnecessary state transitions or the need for a counter, a new page flag PG_writable is added, that is set with the creation of the first writable mapping. Subsequent writable mappings just check the bit and skip the state transition if it is set. To avoid a search over all mappers of a page for writable page table entries, every time a writable page table gets removed the bit PG_writable stays set until all read-only mappers of the page have been unmapped as well. Only then is the PG_writable bit reset again.

## 3.6 Minor fault optimization

An important optimization is the avoidance of page state changes for minor faults. All processes start with empty page tables. Each page accessed by the process gets mapped in reaction to a page fault. In the straightforward implementation, even if the pages required by a process are already present in the page cache, each minor fault will cause two page state changes. find_get_page will force the page into the stable state for a short period of time until the page map counter is increased. Using a special variant of find_get_page that does not change the page state, it is possible to handle minor faults without doing a single state change. If the page has been discarded by the host the first access of the guest will generate a discard fault which causes the page cache page to get removed from memory, including all page table entries referring to the page.

That removes the state transitions on the minor fault path. A page that has been mapped will eventually be unmapped again. On the unmap path each page that has been removed from the page table is freed with a call to page_cache_release. In general that causes an unnecessary page state transition from volatile to volatile. To avoid this unnecessary state transition special variants of put_page_testzero and page_cache_release are introduced that do not try to make the page volatile. page_cache_release_nohv is then used in free_page_and_swap_cache and release_pages. This makes the unmapping of pages state transition free.

## 3.7 Removing discarded pages

Before a discarded page can be freed all references to the page have to be released. The direct removal of the references is not possible in all cases. The discard fault handler can remove the references of the page cache and all entries of the page in the page tables. It can not remove references that are not stored in known places. Consider a process that wants to access a page that is cached in the page cache. After the page has been found in the page cache with a call to find_get_page, the new reference to that page is not stored somewhere in memory but in a dynamic variable of some function. Most likely it will even be cached in some cpu register. If the page gets discarded before the new reference is stored in a memory location where the discard fault handler can find it, the reference will remain valid. That means that after the discard fault handler completed, the page might still exist. To prevent that a page gets removed from the page cache more than once, the discard fault handler marks the page with the PG_discarded page flag. Any subsequent discard fault will only remove page table entries. The discard fault handler will remove a page from the page cache without clearing the page->mapping field. Due to races in the memory management, a page can get mapped to a process after the discard fault has removed the page cache entry for the page. Any discard fault for a page that occurs after the page has been removed still needs the mapping information to be able to remove the remaining page table entries.

Further the `PG_discarded` bit is used to postpone the freeing of discarded pages. Pages that have been discarded are added to the discarded page list. The pages on this list are freed only if the guest is under memory pressure. There are two reasons why this is desirable:

1. Before a discarded page can be reused, a host action is required to provide a new backing frame for the guest page. It is faster to use only non-discarded pages which do not require a host action as long as the working set of the guest allows it.
2. It depends on the platform which information is delivered by a discard fault. If the discard fault handler gets absolute page addresses instead of a virtual addresses—which is the case for z/VM as the host system—the discard fault handler needs to make sure to get a valid page reference. This is only possible if there are no pending discard faults for a page before the page is freed. To ensure this, a synchronization is done before the discarded page list is freed.

## 4 System z Host Implementation

In this section, we briefly describe what changes were required in the System z machine architecture and in the host operating system to support CMM2.

The prototype implementation on System z with Linux guests and a z/VM host uses a simplified state machine that collapses the two discarded states **Vz** and **Pz**, and does not have the preserved volatile states **Up**, **Vp** and **Pp**. The simplified state diagram is shown in Figure 3.

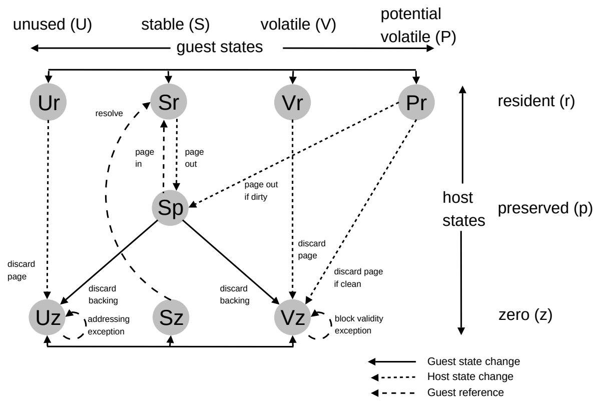


Figure 3: Simplified State Transition Diagram

In order to keep the overhead for the page state transitions low, the prototype uses a special page state transition instruction called Extract and Set Storage Attributes, in short **`ESSA`**. `ESSA` has been introduced with IBM's newest z/Architecture mainframe System z9 and at this point is implemented in millicode. Since the z/Architecture provides separate guest and host managed page tables, which both are concatenated to establish a guest virtual to host absolute mapping, the page states are maintained within the host translation tables associated with each respective guest. The `ESSA` instruction enables atomic page state changes both from the guest and the host with a particular protection domain. This allows the guest transitions to be issued atomically and without entering into the host domain. Yet for guest transitions that require/desire an implicit host transition, the instruction traps into the host and the entire transition and associated host actions are performed.

With the introduction of the ESSA facility to the z Architecture, z/VM [1] was modified to recognize and handle both the extended storage attributes and new storage access exceptions associated with these page state attributes. It also virtualizes the entire ESSA assist in case the ESSA facility is not available on the system, e.g. on previous System z machines. Since one of the primary objectives of the new architecture is to increase the efficiency with which z/VM utilizes memory, its paging algorithms were extended to recognize the new memory attributes. For example, when preparing the list of frames which are candidates for being stolen, frames for pages in the unused state are reclaimed immediately. The demand scan routines—those called to select candidate pages when z/VM needs to free up frames—are executed in several passes, the selection criteria being relaxed on each pass. These routines were changed so that unused pages are unconditionally selected on the first pass, and volatile pages are unconditionally selected on the second pass, regardless of the current selection criteria. The net effect of these changes is to leave in memory, for as long as possible, those pages the guest has identified as being of primary importance, thereby significantly reducing the risk of stealing a page at random and then finding that it needs to be paged back in again almost immediately.

## 5 Evaluation

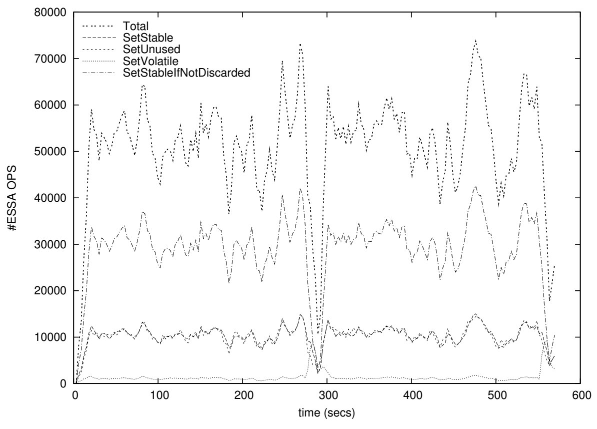
In this section we present results from a set of experiments to establish the overhead and scalability of CMM. In the first set of experiments, we execute a set of particular workloads on a single Linux guest without any z/VM host memory constraints in order to study the frequency with which state transitions are issued and the amount of discardable pages observed during the runs. These discardable pages can be exploited in overcommitted memory scenarios. For that we have chosen a kernel compile and a SPECWeb2005 run. We then discuss the overhead of ESSA instructions and the overhead of their emulation.

Since specific state transition accounting is not performed in the millicode instruction, we are executing this on a previous version of the z/Architecture that does not have the millicode enabled. The z/VM host provides an emulation for the missing instruction, which allows us to run CMM2 enabled guests on older machines and to instrument the emulation code to collect the frequency information.

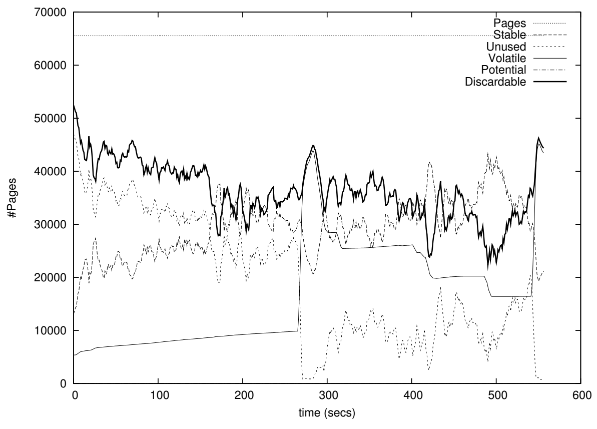
### 5.1 States and Transition Frequency

As the first workload we have chosen a linux kernel build, which is commonly considered as a quick, yet important benchmark. It rapidly executes many processes, utilizes the filecache and issues I/O operations and thus provides a good exercise of many important kernel subsystems. The guest was configured as a 2-way 256MB linux system. The benchmark is comprised of two consecutive identical phases started from a fresh Linux reboot. Each phase consists of a kernel compile, a kernel clean, followed by a search of the entire linux source for a particular string.

The number of various page state transitions per second experienced during the run and represented by their associated ESSA instructions is shown in Figure 4(a). To dampen the high frequency oscillations, we have applied a simple moving average SMA(3) filter. We can see that state transitions are approximately at 50K/sec. `SetStable`(**S**) and `SetUnused`(**U**) track each other very closely. This is due to the fact that individual compile processes have short life times and page allocations (SetStable) are shortly followed by their respective frees (SetUnused). The higher number of `SetStableIfNotDiscarded` transitions is due to the fact that I/O is performed



(a) State Transition Frequency (ops/sec)



(b) State Distribution

Figure 4: Anatomy of a Kernel Compile on a 2-way 256M Guest

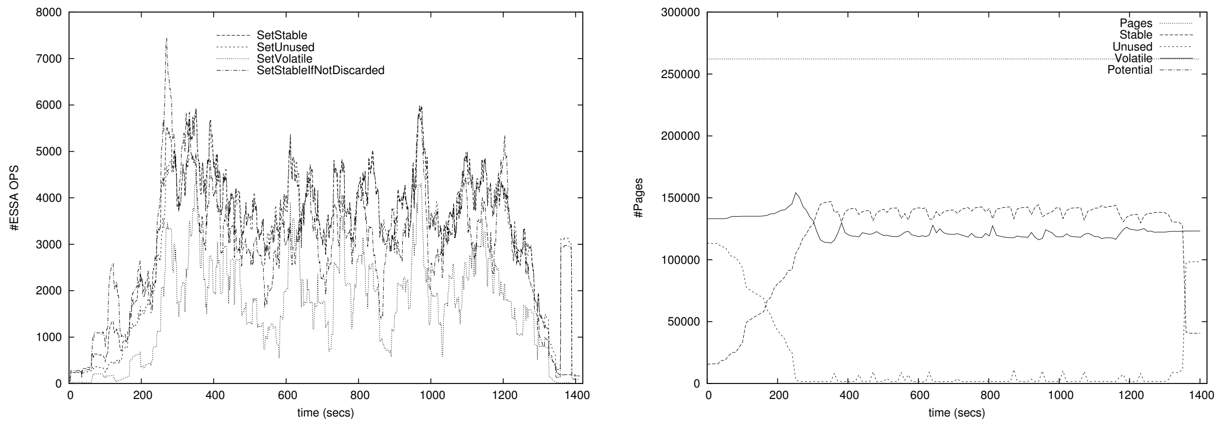
using `read/write` operations, which have to go through the filecache and use the `find_get_page` variants.

A utility program executing concurrently on the guest "scans" the all guest pages every second using the ESSA extract instruction to establish the number of pages in each usage state. The result (not including the extract ESSA) is shown in Figure 4(b). Again a SMA(3) filter is applied for smoothing effects. The thick solid line defines the number of discardable pages ($U+V$), which on average is about half of all the guest memory. The first kernel compile slowly increases the number of volatile pages, which essentially is due to the increased number of files that have been read from the linux source and remain in the file cache. At t=267s the deep source search is initiated which essentially brings the entire source tree into the file cache depleting the free page pool. In the second phase, the kernel source residency is slowly reduced again as the source gets pushed out of memory. This causes the decrease of the number of volatile pages.

To demonstrate the effect of the collaborative memory management in a realistic enterprise computing workload, we show the state transition characteristics of the SPECweb2005 benchmark, which is modeled after typical enterprise IT service scenarios in e-commerce, banking, and corporate customer support Web servers. In SPECweb2005, workload generators send HTTP requests to the Web servers under evaluation at a given concurrency and observe whether they are capable of handling them without violating the service level guarantee in terms of response time and goodness of responses.

We configured a Linux guest on z/VM as a self-contained SPECweb2005 testbed. The single guest Linux hosted Apache 2.2 as the front end Web server, a SPECweb2005 backend simulator, and SPECweb2005 workload generating client along with JVM. The guest was configured to have a single CPU and 1GB of real memory. The support scenario of SPECweb2005 was used in the experiment.

Figure 5(a) shows the different state transition frequencies (as expressed by their associated ESSA ops and with a SMA(30) filter) over a 20 minute run when the JVM is configured with 256MB heap. The run is comprised of 3 phases, (i) initialization [0:50] secs, (ii) rampup [50:230] secs, and (iii) steady state run [230–]. One can see from the figure, that the state change rate is rapidly changing in re-



(a) State Transition Frequency (ops/sec) (b) State Distribution

Figure 5: Anatomy of a SPECweb2005 run on 1-way 1G Guest

sponse to varying request conditions. The average ESSA rate in steady state is about 15K/sec.

Figure 5(b) shows the dissection of memory pages seen from the guest for the same run. At steady state about half of the pages are Stable while the other half remains Volatile. A large portion of the stable pages is attributable to the 130M page JVM heap. Volatile pages are comprised of the page cache pages for the html files and the download files of the SPECweb2005 Support benchmark.

The reasonable large amount of volatile pages observed both in the Kernel Compile as well as in SPECweb2005 confirms that the collaborative memory management of the host VM will be able to find discardable pages for fast memory provisioning in a typical enterprise workload. By utilizing this dynamic state transition information, the host VM is able to reallocate pages to those guests which need more pages in order to meet their service level by harvesting the discardable pages from other guest systems and without invoking the victim guests. Nevertheless, the high rate of state transitions, concerned us from the start and let us to explore the architectural support through the ESSA instruction.

## 5.2 Guest State Transition Overhead

We have timed the common non-trapping ESSA instructions representing the guest transitions ($*r \to *r$) on a 1.65GHz z9 processor and obtained the following results: (i) Extract: 97.9nsecs; (ii) SetStable: 100.8nsecs; (iii) SetVolatile: 103.7nsecs; (iv) SetUnused: 102.5nsecs; and (v) SetStableIfNotDiscarded: 106.5nsecs. For the kernel compile, which poses a very high transition rate of 25K/sec per cpu, the overhead amounts to ~0.25% and for the SPECweb2005 run it amounts to ~0.15%.

For systems which do not have the ESSA millicode enabled, each ESSA instruction must trap into z/VM and is emulated there. The execution times of emulating these instructions are roughly 10 fold. This should give some bounds on what to expect if this service is implemented on other architectures through hypervisor traps.

## 5.3 Scalability

We now present our preliminary data on a scalability and comparison analysis of various memory management technologies. To do so, a z/VM partition with 34 Linux guests was set up. 32 guests each ran an Apache Webserver that serves a 1200 1MB files. Two guests function as web clients to continuously request random files from random servers. During the runs on a 4-way host partition, the two clients consumed ~50% of cpu cycles while each server consumed ~6% of cpu cycles. The transaction rate was measured under varying host physical memory size *PM* of the z/VM partition. The relative degradation as we shrank the host memory size from $PM = 64GB$ to $PM = 256MB$ is shown in Figure 6. We observed the following memory management strategies:

- **Partitioned**: physical memory is partitioned equally among all guests. As a result all memory pressure is local to the guests.
- **HostPaging**: the guests remain at a constant guest memory size and overcommitment is handled in the host.
- **CMM1/Balooning**: guests are dynamically sized, yet host paging is also allowed.
- **CMM2**: host and guest coordinate through the HVA facility.

At $PM = 64G$ all methods exhibit the same performance as no effective difference exists. First, in the static partitioned scenario, the guest memory size $RM_i = PM/32$ is varied down to 64MB and there is no effective performance drop. This suggests that the working set size of this workload is extremely small and the file-cache is ineffective. The *PM* can not be reduced any further, as the guests are not able to boot or run with less then $RM_i = 64MB$. Next, in the host paging case where clients are setup with $RM_i = 1.5GB$, the system relies completely on host paging to deal with the overcommitment. The system became unresponsive beyond an overcommitment ratio of

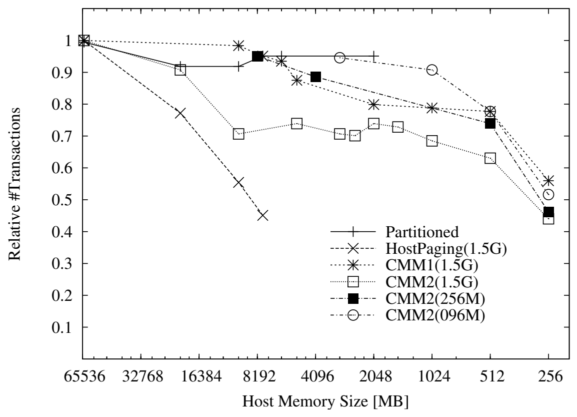


Figure 6: Relative degradation in transaction rate for a 32 server Apache benchmark for various global memory management methods and under tightening memory constraints

$(1.5 * 32)/8$ ($PM = 8GB$). With the existing z/VM-Linux ballooning method, CMM1, a reconfiguration sample every 30 seconds and a setting to allow guests to shrink down to $RM_i = 64MB$, we were able to continue to reduce the *PM* to 256M at about 50% performance loss for guests that were configured with the initial guest maximum guest size of $RM_i^{max} = 1.5GB$. This is due to the fact that the workload is stable and exhibits a small working set size and ballooning can shrink the $RM_i$ towards their working set size. With CMM2 for $RM_i = 1.5GB$, we can see that performance lacks the CMM1 ballooning curve, which is due to the fact that the guests have to manage a larger amount of memory (1.5GB) as compared to the ballooning scenario which effectively reduces the memory that needs to be managed. CMM2 for $RM_i = 256MB$ guests very closely tracks the ballooning method. To continue on that path, we ran CMM2 for $RM_i = 96MB$ guests, we see that in the range of 512M-3GB, CMM2 outperforms CMM1 ballooning. This underscores that there is potential in having CMM2 and CMM1 deployed together, namely utilize CMM1 to size the guests reasonably and then utilize CMM2 for short term overcommitments.

## 6 Conclusions and Future Work

In this paper we introduced a novel approach to collaborative memory management in hosted operating system environments. We described the problems that are associated with pure dynamic memory partitioning and pure host paging. We defined an architecture that will allow us to reap the benefits of both approaches, while avoiding the drawbacks. Our approach relies on an information sharing of guest page usage and host residency information to facilitate and coordinate both the host and the guest page reclamation process. The framework has been implemented on IBM's z/Architecture running Linux on zSeries guests and the z/VM host operating system. The information sharing was implemented as a millicode instruction.

In the current state of our work, we have shown for various scenarios that we can successfully identify discardable guest pages in the host and that the overhead can be kept within 0.25% for maintaining the state information. We have also presented our first scalability analysis that has shown that CMM2 can outperform host paging and CMM1 ballooning even for a very stable non-bursty workload, as long as we can rely on a mechanism to approximately size the guest images.

The current ongoing work is a comprehensive scalability analysis of the kernel compile, the SPECWeb2005 and bursty workloads. In particular, the latter two we expect to exhibit better performance with CMM2 as compared to the other methods. We are also working on an extension that eliminates double paging faults.

## References

[1] D.L.Osisek, K.M.Jackson, and P.H.Gum, *Esa/390 interpretive-execution architecture - foundation for vm/esa*, IBM Systems Journal **30** (1991), no. 1, 34–51.

[2] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, *Xen 3.0 and the Art of Virtualization*, Proceedings of the Ottawa Linux Symposium, 2005.

[3] Carl A. Waldspurger, *Memory resource management in vmware esx server*, SIGOPS Oper. Syst. Rev. **36** (2002), no. SI, 181–194.

# Proceedings of the Linux Symposium

## Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada