Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Containers: Challenges with the memory resource controller and its performance

Balbir Singh
IBM
balbir@in.ibm.com

Vaidyanathan Srinivasan
IBM
svaidy@linux.vnet.ibm.com

Abstract

Containers in Linux are under active development and have different uses like security, isolation and resource guarantees. In order to provide a resource guarantee for containers, resource controllers are used as basic building blocks to monitor and control utilization of system resources like CPU time, resident memory and I/O bandwidth, among others. While CPU time and I/O bandwidth are renewable resources, memory is a non-renewable resource in the system. Infrastructure to monitor and control resident memory used by a container adds a new dimension to the existing page allocation and reclaim logic.

In order to assess the impact of any change in memory management implementation, we propose adding parameters to modify VM[1] behavior and instrument code paths and collect data against common workloads like file-server, web-server, database-server and developer desktop. Data of interest would be reclaim rate, page scan density, LRU[2] quantum, page container affinity and page generation.

This paper discusses, in detail, the design and performance issues of RSS controller and pagecache controller within the container framework. Some of the modifications to the current page reclaim logic that could help containers are also evaluated.

1 Background

Server consolidation, virtualization and containers are buzz words in the industry today. As enterprises are consolidating applications and platforms to a single server, either through virtualization or containers, there is a need to differentiate between them. Consider a server that hosts two platforms, one an employee e-mail server and the other a customer call center application. The enterprise would definitely want the customer call center application to have a priority over the employee e-mail server. It would be unacceptable if the employee e-mail server occupied and consumed most of the resources available in the consolidated server thereby affecting performance of critical applications (in this case, the call center application).

Resource management can provide service guarantees by limiting the resource consumption of the employee e-mail server. Resource controllers are part of container framework that would monitor and control certain resource. Controllers generally monitor and limit one resource like memory, CPU time, I/O bandwidth etc. In order to provide isolation between containers from resource perspective, we would primarily need to control memory and CPU time. In this paper we discuss the challenges with the design and implementation of memory controller.

2 Memory controller

A *memory controller* [13] allows us to limit the memory consumption of a group of applications. Several proposals for memory control have been posted to LKML,[3] they are *resource groups* [11], *memory container* [12], *beancounters* [6], and the most recent, *RSS controller* [2] [3]. The design and features supported by each of the proposals is discussed below.

2.1 Resource groups

The *resource groups memory controller* was developed by Chandra Seetharaman, Jiantao Kong, and Valerie Clement [11].

[1] Linux virtual memory manager.
[2] Least recently used page list.
[3] Linux kernel mailing list.

It was built on top of the resource groups, resource management infrastructure and supported both *limits* and *guarantees*. Guarantees and limits were set using the `min_shares` and `max_shares` parameters, respectively. Resource groups control only user-space pages. Various configuration parameters allowed the system administrator to control:

- The percentage of the memory usage limit, at which the controller should start reclaiming pages to make room for new pages;
- The percentage to which the reclaimer should shrink pages, when it starts reclaiming;
- Number of seconds in a shrink interval;
- Number of shrink attempts in a shrink interval.

A page LRU list, broken down from zone LRU, is maintained for every resource group, which helps minimize the number of pages to be scanned during page reclaim.

Task migration is an expensive operation, as it requires the `class` field in each `page` to be updated when a task is migrated.

2.2 Memory container

The salient features of *memory containers* as posted by Rohit Seth [12] are as follows:

- It accounts and limits pagecache and RSS usage of the tasks in the container.
- It scans the mappings and deactivates the pages when either the pagecache or RSS limit is reached.
- When container reclaim is in progress, no new pages are added to it.

The drawbacks of this approach are:

- Task migration is an expensive operation, the container pointer of each `page` requires updating.
- The container that first accesses a file, is charged for all page cache usage of that file.
- There is no support for guarantees.

2.3 Beancounters

The memory controller for Beancounters was developed by Pavel and Kirill [7]. The salient features of this implementation are:

- Initial versions supported only resource limits, whereas later versions supports reclaim of RSS pages as well.
- The system call interface was the only means for setting limits and obtaining resource usage, whereas newer versions have added file system based configuration and control.
- Kernel resources such as page tables, slab usage is accounted for and limited.

The drawbacks of this approach are:

- There is no direct support for guarantees.
- Task migration is supported, however when a task migrates, it does not carry forward the charges of the resources used so far.
- Pagecache control is not present.

2.4 RSS controller

The RSS controller was developed by Balbir Singh [14]. The salient features of this implementation are:

- No change in the size of `page` structure.
- RSS accounting definition is the same as that is presently used in the Linux™ kernel.
- The per-zone LRU list is not altered.
- Shared pages are reclaimed by un-mapping the page from the container when the container is over its limit.

The drawback of this approach is the reclaim algorithm. The reclaimer needs to walk through the per zone LRU of each zone to first find and then reclaim pages belonging to a given container.

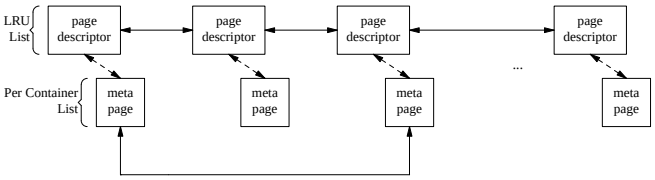


Figure 1: RSS controller with per container list

Pavel enhanced the RSS controller [2] and added several good features to it. The most significant was a per container list of pages.

Figure 1 shows the organization of the RSS controller. Each page has a meta page associated with it. All the meta pages of the pages belonging to a particular container are linked together to form the per container list. When the container is over its limit, the RSS controller scans through the per container list and try to free pages from that list.

The per container list is not the most efficient implementation for memory control, because the list is not in LRU order. Balbir [3] enhanced the code to add per container LRU lists (active and inactive) to the RSS controller.

3 Pagecache control

Linux VM will read pages from disk files into a main memory region called *pagecache*.[4] This acts as a buffer between the user application's pages and actual data on disk. This approach has the following advantages:

- Disk I/O is very expensive compared to memory access, hence the use of free memory to cache disk data improves performance.
- Even though the application may read only a few bytes from a file, the kernel will have to read multiple disk blocks. This extra data needs to be stored somewhere so that future reads on the same file can be served immediately without going to disk again.
- The application may update few bytes in a file repeatedly, it is prudent for the kernel to cache it in memory and not flush it out to disk every time.
- Application may reopen the same file often, there is a need to cache the file data in memory for future use even after the file descriptor is closed. Moreover the file may be opened by another application for further processing.

The reader might begin to ponder why we need to control the pagecache? The problem mainly arises from backup applications and other streaming data applications that bring in large amounts of disk data to memory that is most likely not to be reused. As long as free memory is available, it is best to use them for pagecache pages since that would potentially improve performance. However, if there is no free memory, then cold pages belonging to other applications would be swapped out to make room for new pagecache data. This behavior would work fine in most circumstances, but not all. Take the case of a database server that does not access records through pagecache as it uses direct I/O. Applications, like the database server, manage their own memory usage and prefer to use their own disk caching algorithms. The OS is unlikely to predict the disk cache behavior of the application as well as the application can. The pagecache might hurt the performance of such applications. Further, if there is a backup program that moves large files on the same machine, it would end up swapping pages belonging to database to make room for pagecache. This helps the backup program to get its job done faster, but after the backup is done, the system memory is filled with pagecache pages while database application pages are swapped-out to disk. When the database application needs them back, it will have to wait for the pages to be swapped-in. Hence the database application pays the price for backup application's inappropriate use of pagecache.[5]

The problem becomes more visible with server consolidation and virtualization. Now there is a need to limit the usage of pagecache for a certain group of applications in order to protect the working set of other critical applications. Limiting the pagecache usage for less important tasks would impact its performance, which is acceptable, because the system performance is judged only based on the performance of the critical application like database or web server.

The RSS controller does control pagecache to some extent. When an application maps files, the pages are accounted in its resident set and would be reclaimed by

[4] Also referred to as disk cache.

[5] The pagecache feature is less important in the example given, since the throughput of the database is more important than the speed of the backup.

RSS controller if they go over the limit. However, it is possible for application to load data into pagecache memory with read/write system calls and not map all the pages in memory. Hence there is a need to control unmapped pagecache memory as well. The pagecache controller is expected to count unmapped pagecache pages and reclaim them if found over the limit. If the pages are mapped by the application, then they are counted as RSS page and the RSS controller will do the needful. If unmapped pagecache pages are not tracked and controlled, then the pages unmapped by the RSS controller will be marked for swap-out operation. The actual swap-out operation will not happen unless there is a memory pressure. The pages reclaimed by the RSS controller will actually go into swapcache which is part of pagecache. The pagecache controller will count these swap-cache pages as well and create a memory pressure to force the reclaimer to actually swap-out the pages and free system memory. In order to maintain memory limits, for containers, pagecache memory should also be controlled apart from RSS limit. Pagecache controller and RSS memory controller are parts of memory controller for containers. Initial prototype patches are independent, however both these controllers share code paths and hopefully they will eventually be integrated as part of container memory controller.

The Linux VM has various knobs like `/proc/sys/vm/{swappiness, dirty_ratio, dirty_background_ratio}` to control pagecache usage and behavior. However they control system-wide behavior and may affect overall system performance. `swappiness` is a percentage ratio that would control choice of pages to reclaim. If the percentage is greater, *anonymous* pages would be chosen and swapped out instead of *pagecache* pages during page reclaim. Reducing the `swappiness` ratio would reduce pagecache usage. The other two knobs, `dirty_ratio` and `dirty_background_ratio`, control write out of pagecache pages. Dirty pagecache pages needs to be written out to disk before the page can be reused. However a clean pagecache page is as good as free memory because it can be freed with almost zero overhead and then reused for other purposes. The kernel periodically scans for dirty pagecache pages and writes them out based on the desired dirty page ratio.

Container framework and memory controller provide infrastructure to account for and monitor system memory used by group of applications. Extending the available infrastructure to account for and control pagecache pages would provide isolation, control and performance enhancements for certain groups of applications. By default, the Linux kernel would try to use the memory resources in the best suitable manner to provide good overall system performance. However, if applications running in the system are assigned different priorities then kernel's decisions needs to be made taking into account the container's limits which indirectly implies priority.

Limiting the amount of pagecache used by a certain group of applications is the main objective of the pagecache controller under the container framework. Couple of methods to control pagecache have been discussed on LKML in the past. Some of these techniques are discussed below.

3.1 Container pagecache controller

Pagecache accounting and control subsystem under container framework [15] works using the same principle as memory controller. Pages brought into the pagecache are accounted for against the application that brought it in. Shared pagecache pages are counted against the application that first brought it into memory. Once the pagecache limit is reached, the reclaimer is invoked that would pick unmapped pages in inactive list and free them.

The code reclaim path for the RSS controller and pagecache controller is common except for few additional condition checks and different scan control fields. All reclaim issues discussed in RSS controller section applies to the pagecache controller as well.

3.2 Allocation-based pagecache control

Roy Huang [4] posted a pagecache control technique where the existing `kswapd()` would be able to reclaim pages when the pagecache goes over limit. A new routine `balance_pagecache()` is called from various file I/O paths that would wake up `kswapd()` in order to reclaim pagecache pages if they are over the limit. `kswapd()` checks to see if pagecache is over the limit and then it uses `shrink_all_memory()` to reclaim all pagecache pages. The pagecache limit is set through a `/proc` interface.

The generic reclaimer routine is used here, which prefers pagecache pages over mapped pages. However, if the pagecache limit is set to a very small percentage, then the reclaimer will be called too often and it will end up unmapping other mapped pages as well. Another drawback in this technique is not distinguishing mapped pagecache pages that might be in use by the application. If a mapped page is freed, then the application will most probably page-fault for it soon.

Aubrey Li [9] took a different approach by adding a new allocation flag, `__GFP_PAGECACHE`, to distinguish pagecache allocations. This new flag is passed during allocation of pagecache pages. Pagecache limit is set through `/proc` as in the previous case. If the utilization is over the limit, then code is added to flag a low zone watermark in the `zone_watermark_ok()` routine. The kernel will take the default action to reclaim memory until sufficient free memory is available and `zone_watermark_ok()` would return true. The reclaim technique has the same drawbacks cited in Roy's implementation.

Christoph Lameter [8] refined Aubrey's approach [9] and enhanced the `shrink_zone()` routine to use different scan control fields so that only pagecache pages are freed. He introduced per-zone pagecache limit and turned off `may_swap` in scan control so that mapped pages would not be touched. However, there is a problem with not unmapping mapped pages because pagecache stats count both mapped and unmapped pagecache pages. If the mapped part is above limit, like if an application `mmap()` file causes pagecache to go over the limit, then the reclaimer will be triggered repeatedly, which does not unmap pages and reduce the pagecache utilization. We should account for only unmapped pagecache pages for the limit in order to workaround this issue. Mapped pagecache pages will be accounted by the RSS memory controller. The possibility of user space-based control of pagecache was also discussed.

3.3 Usermode pagecache control

Andrew Morton [10] posted a user-mode pagecache control technique using `fadvise()` calls to hint the application's pagecache usage to kernel. The POSIX `fadvise()` system call can be used to indicate to the kernel how the application intends to use the contents of the open file. There are a couple of options like NORMAL, RANDOM, SEQUENTIAL, WILLNEED, NOREUSE, or DONTNEED that the application can use to alter caching and read-ahead behavior for the file.

Andrew has basically overridden read/write system calls in libc through LD_PRELOAD and inserted `fadvise()` and `sync_file_range()` calls to zero out the pagecache utilization of the application. The application under control is run using a shell script to override its file access calls, and the new user space code will insert hidden fadvise calls to flush or discard pagecache pages. This effectively make the application not use any pagecache and thus does not alter other memory pages used in the system.

This is a very interesting approach to show that pagecache control can be done from user space. However some of the disadvantages are:

- The application under control suffers heavy performance degradation due to almost zero pagecache usage, along with added system call overheads. The intent was to limit pagecache usage and not to avoid using it.
- Group of applications working on the same file data will have to bring in data again from disk which would slow it down further.

More work needs to be done to make `fadvise()` more flexible to optimally limit pagecache usage while still preserving reasonable performance. The containers approach is well suited to target a group of applications and control their pagecache utilization rather than per process control measures.

4 Challenges

Having looked at several memory controller implementations, we now look at the challenges that memory control poses. We classify these challenges into the following categories:

1. Design challenges
2. Implementation challenges
3. Usability challenges

We will look at each challenge and our proposed solution for solving the problem.

4.1 Design challenges

The first major design challenge was to avoid extending the `struct page` data structure. The problem with extending `struct page` is that the impact can be large. Consider an 8 GB machine, which uses a 4 KB page size. Such a system has 2,097,152 pages. Extending the page size by even 4 bytes creates an overhead of 8 MB.

Controlling the addition using a preprocessor macro definition is not sufficient. Linux distributions typically ship with one enterprise kernel and the decision regarding enablement of a feature will have to be made at compile time. If container feature is enabled and end users do not use them, they incur an overhead of memory wastage.

Our Solution. At first, we implemented the RSS controller without any changes to `struct page`. But without a pointer from the page to the meta page, it became impossible to quickly identify all pages belonging to a container. Thus, when a container goes over its limit and tries to reclaim pages, we are required to walk through the per zone LRU list each time. This is a time-consuming operation and the overhead, in terms of CPU time, far outweighs the disadvantage of extending `struct page`.

The second major challenge was to account shared pages correctly. A shared page can be charged:

- To the first container that brings in the page. This approach can lead to unfairness, because one container could end up bearing the charge for all shared pages. If the container being charged for the shared page is not using the page actively, the scenario might be treated as an unfair implementation of memory control.
- To all containers using the shared page. This scenario would lead to duplicate accounting, where the sum of all container usage would not match the total number of pages in memory.

Our Solution. The first RSS container implementation accounted for every shared page to each container. Each container `mm_struct` was charged for every page it touched. In the newer implementations, with their per-container LRU list, each page can belong to only one container at a time. The unfairness issue is dealt with using the following approach: A page in the per container LRU list is aged down to the inactive list if it is not actively used by the container that brought it in. If the page is in active use by other containers, over a period of time this page is freed from the current container and the other container that is actively using this page, will map it in. The disadvantage of this approach is that a page needs to be completely unmapped from all mappings, before it can move from one container to another.

The third major challenge was to decide on whether we should account per thread memory usage or per process memory usage. All the threads belonging to a process share the same address space. It is quite possible that two threads belonging to the same process might belong to two different containers. This might be due to the fact that they may belong to different container groups for some other resource, like CPU. They might have different CPU usage limits. This leads to more accounting problems as:

- By default all pages in a thread group are shared. How do we account for pages in a thread group?
- We now need to account every page to the thread that brought it in, thus requiring more hooks into `task_struct`.

Our Solution. We decided to charge the thread group leader for all memory usage by the thread group. We group tasks virtually for memory control by thread group. Threads can belong to different containers, but their usage is charged to the container that contains the thread group leader.

4.2 Implementation challenges

The first major implementation challenge was low cost task migration. As discussed earlier, one of the disadvantages of the memory controller implementations was the time required to migrate a task from one container to another. It typically involved finding all pages in use by the task and changing their container pointer to the new container. This can be a very expensive operation as it involves walking through the page tables of the page being migrated.

Our Solution. In the first implementation of the RSS controller, `struct page` was not modified, hence

there were no references from the page descriptor to the container. Task migration was handled by adding a memory usage counter for each `mm_struct`. When a process is moved from one container to another, the accumulated memory usage was subtracted from the source container and added to the destination container. If the newly migrated task put the destination container over its memory usage limit, page reclaim is initiated on migration. With the new RSS controller implementation that has a per-container LRU list, a member of `struct page` points to the `meta page` structure. The meta page structure, in turn, points to the container. On task migration, we do not carry forward any accounting/charges, we simply migrate the task and ensure that all new memory used by the task is charged to the new container. When the pages that were charged to the old container are freed, we uncharge the old container.

The second major implementation challenge is the implementation of the reclaim algorithm. The reclaim algorithm in Linux has been enhanced, debugged and maintained in the last few years. It works well with a variety of workloads. Changing the reclaim algorithm for containers is not a feasible solution. Any major changes might end up impacting performance negatively or introduce new regressions or corner cases.

Our Solution. We kept the reclaim algorithm for the RSS controller very simple. Most of the existing code for the reclaim algorithm has been reused. Other functions that mimic global reclaim methodology for containers have been added. The core logic is implemented in the following routines:
`container_try_to_free_pages`
`container_shrink_active_list`
`container_shrink_inactive_list`, and
`container_isolate_lru_pages`.
These are similar to their per-zone reclaim counterparts:
`try_to_free_pages`
`shrink_active_list`
`shrink_inactive_list`, and
`isolate_lru_pages`, respectively.

We've defined parameters for measuring reclaim performance. These are described in Section 5.

4.3 Usability challenges

Some of the challenges faced by the end-users of containers and resource controllers are described below:

Container configuration:

Containers bring in more knobs for end-user and overall system performance and ability of the system to meet its expected behavior is entirely dependent on the correct configuration of container. Misconfigured containers in the system would degrade the system performance to an unacceptable level. The primary challenge with memory controller is choice of memory size or limit for each container. The amount of memory that is allocated for each container should closely match the workload and its resident memory requirements. This involves more understanding of the workloads or user applications.

There are enough statistics like delay accounting and container fail counts to measure the extent to which container is unable to meet the workload's memory requirement. Outside of containers, the kernel would try to do the best possible job, given the fixed amount of system RAM. If performance is unacceptable, the user would have to cut down the applications (workload) or buy more memory. However, with containers, we are dicing the system into smaller pieces and it becomes the system administrator's job to match the right sized piece to the right sized job. Any mismatch will produce less than the desired result.

There is a need for good user space and system-management tools to automatically analyze the system behavior and suggest the right container configuration.

Impact on other resource dimensions:

There is an interesting side effect with container resource management. Resources like CPU time and memory can be considered independent while configuring the containers. However, practical case studies indicate that there is a relationship between different resources, even though they are accounted for and controlled independently. For example, reducing the working set of an application using a memory controller would indirectly reduce its CPU utilization because the application is now made to wait for page I/O to happen. Restricting working set or pagecache of a workload increases its I/O traffic and makes it progressively I/O bound even though the application was originally CPU bound when running unrestricted.

Similarly, reducing the CPU resource to a workload may reduce its I/O requirement because the application is not able to generate new data at the same rate. These kinds

of interactions suggest that configuring containers may be more complex than we may have considered.

5 Reclaim parameters

The reclaim algorithm is a very critical implementation challenge. To visualize and gain insight into the reclaim algorithm of the container, a set of parameters have been defined. These parameters are discussed in the following sections.

5.1 Page reclaim rate

Page reclaim rate measures the rate at which pages are being reclaimed from the container. The number of pages reclaimed and the duration of the reclaim cycle are taken into account.

$$Page\ reclaim\ rate = \frac{nr\_reclaimed}{(t_{start} - t_{end})}$$

Where t_{start} and t_{end} are the time stamp at the beginning and end of one reclaim cycle (`container_shrink_pages`) and *nr_reclaimed* is the number of pages freed during this time. From a memory controller point of view, freeing a page is as good as unmapping them from the process address space. The page can still be in memory and may additionally be dirty, pending a write-out or swap operation.

A very low reclaim rate value indicates we are taking more time to free pages:

- All pages are in active list and it takes more reclaim cycles to move them to inactive list and then ultimately reclaim them.
- We have been searching the wrong set of pages and it took time to find the right page.
- Most candidate pages are dirty and we are blocked on write-out or swap I/O operation.

5.2 Page container affinity

The page container affinity measures the affinity of physical page to a particular container. When system is running multiple containers, each of the containers is expected to free pages and consume it again. If containers grab each others page, that means that too much concurrent reclaim and allocations are happening, whereby a page just freed by container A is immediately allocated by container B. This could also happen if Container B was under the limit and A was over the limit and we are purposely taking memory pages away from A and giving it to B.

5.3 Page generation

Page generation is the number of times a page was freed by a container. A very high value for page generation indicates that:

- The container size is very low; this implies that we are actively freeing our working set, which keeps coming back in.
- The reclaim algorithm is freeing the wrong set of pages from the container.

5.4 LRU quantum

The reclaimer mainly works on the active list and inactive list of pages belonging to the container. New allocations or recently referenced allocations would go to the head of the active list. The `container_shrink_active_list` routine picks appropriate pages from the active list and moves them to inactive list. While `container_shrink_inactive_list` calls `shrink_page_list` to free aged pages at the tail of the inactive list.

Newest pages are supposed to be at the head of the active list while the oldest page would be at the tail of the inactive list. LRU quantum is the time difference between these two pages. This is an important parameter because this gives an indication of how fast the active and inactive lists are churned.

A greater value of LRU quantum indicates a stable container, where the working set fits the available memory. The reclaimer is run less often and pages take a while before they falls off the end of inactive list.

A smaller value of LRU quantum indicates churning of the list. Combined with page generation, this means there is too little memory for the container. If page generation is low while LRU quantum is high then it could indicate a problem in the LRU aging algorithm used.

5.5 Page scan density

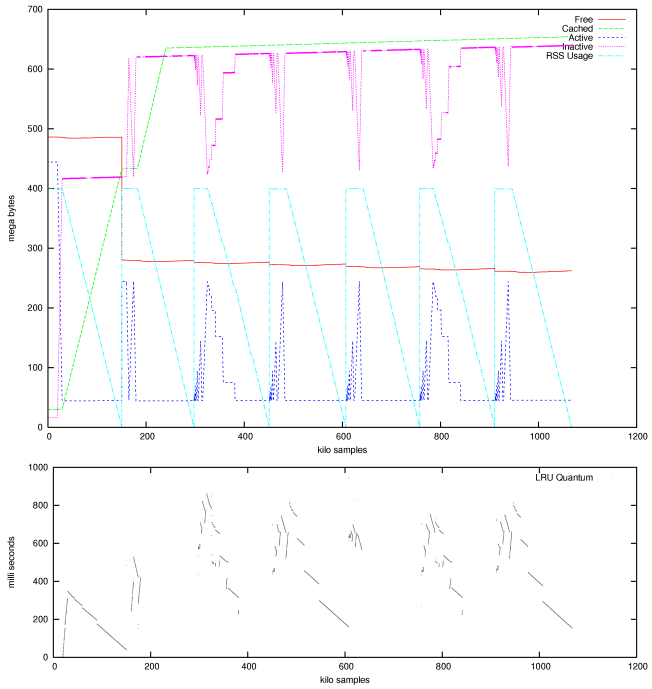
Page scan density is the number of times a page was scanned before it was actually freed. A lower the value indicates that the reclaimer has been choosing the right pages to free and it is quite smart. Higher values of page scan density for a wider range of pages means the reclaimer is going through pages and is unable to free them, or perhaps the reclaimer is looking at the wrong end of the LRU list.

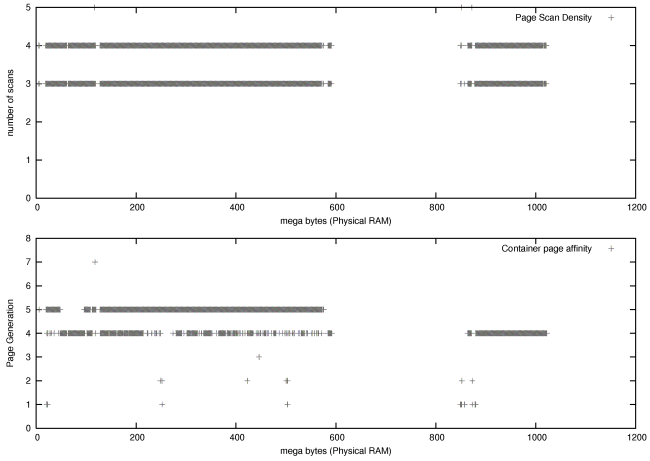
6 Case studies

A few typical workload examples have been studied in order to understand various parameters and its variations, depending upon workload and container configuration. The following section describes the parameters traced during execution of simple workloads and test programs.

6.1 Sequential memory access workload

Pagetest is a simple test program that allocates memory and touches each page sequentially for *n* number of times. In the following experiment, the pagetest program was run with an RSS limit of 400 MB, while the program would sequentially touches 600 MB of memory five times.





Observations:

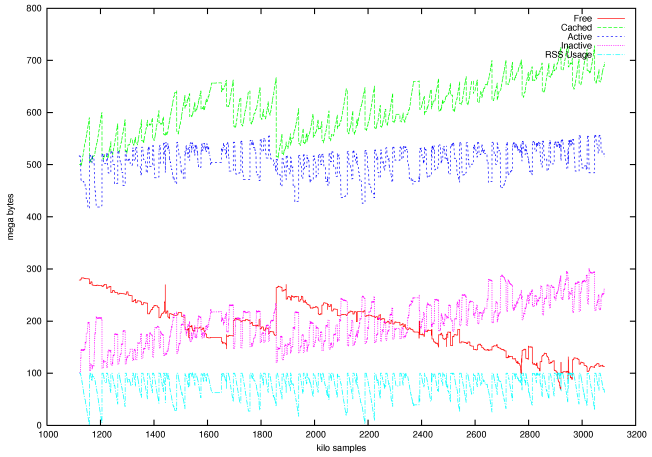
- Memory reclaim pattern shows that once the RSS usage limit is hit, then all the pages are reclaimed and the RSS usage drops to zero
- The usage immediately shoots to 400 MB because the plots is approximately by time and we did not have samples during the interval when the application was under limit and slowly filled its RSS up to the limit.
- Active list and inactive list size are mirror image of each other since the sum of active and inactive size is constant. The variations in list size corresponds to the page reclaim process.
- Free memory size dropped initially and it remains constant while cached memory size initially increased and then remained constant. Free memory size is not affected by the reclaim process since pages reclaimed by RSS controller was pushed to swapcache and stays there until touched again or there is enough memory pressure to swap-out to disk. Since we had enough free memory in this experiment, the swapcache grew and no swap to disk happened.
- LRU quantum was less than one second in this case. The time difference between pages at the head of active list and tail of inactive list was high just before the reclaim started and then quickly dropped down as pages are reclaimed.
- Page scan density shows that we scanned pages three to four times before reclaiming them. This shows that the reclaim algorithm has maintained the active and inactive list optimally and has been

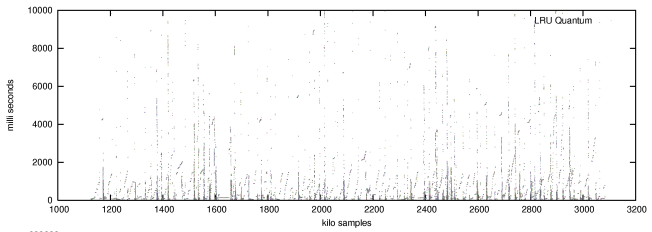
choosing the right pages. We would not see a uniform distribution if the list aging algorithm was incorrect.

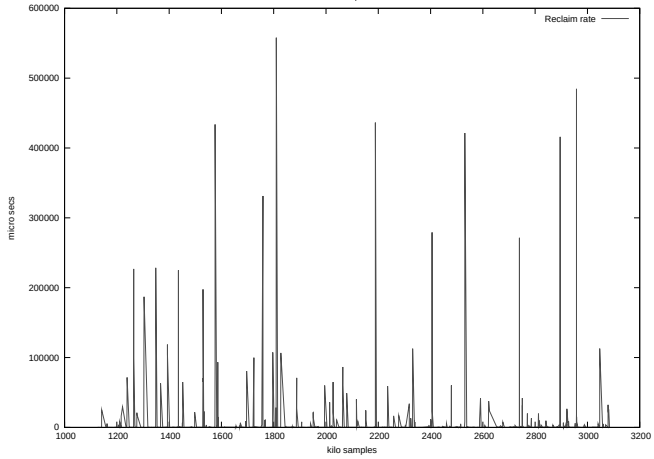
- Page generation shows that most part of physical RAM was reused 5 times during the test which corresponds to the loop iteration of 5.

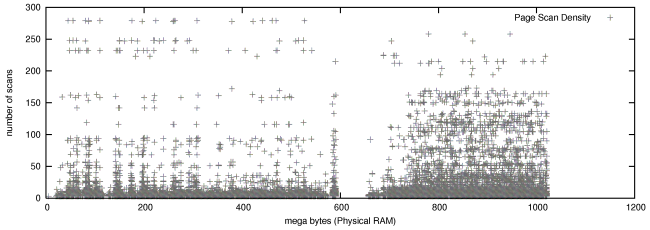
6.2 kernbench test

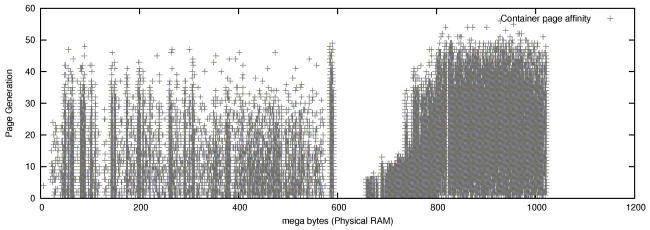
Kernbench compiles a Linux kernel using multiple threads that would consume both anonymous pages and pagecache pages. In the following experiment, the kernbench test was run with 100 threads with RSS controller and memory limit set to 300 MB. The pagecache controller and pagecache limit was not enabled during this experiment.











Observations:

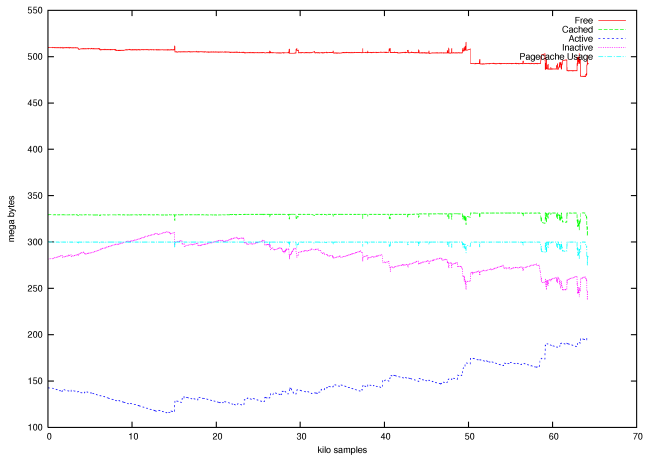
- kernbench has run for long time and the reclaim pattern is compressed. There are many cycles of reclaim. Hence it is difficult to deduce the slope pattern in the memory size and LRU quantum plots

- Page reclaim rate plot has very wide distribution of values and it has been difficult to make sense of the plot. The time taken to reclaim a page in each reclaim cycles is mostly under few milli seconds. However, many times the number of pages reclaimed during a reclaim cycles goes too low making the time shoot up.

- Page scan density and page generation shows that certain region of memory had more pages recycles and their wide distribution corresponds to the complexity of the workload and their memory access pattern.

6.3 dbench test with pagecache limit

The dbench file system benchmark was used to stress the pagecache controller. In the following experiment, dbench was run for 60 seconds with 20 clients. When dbench was run unrestricted, it used around 460 MB pagecache. In this experiment pagecache controller limit was set to 300 MB which would force reclaim of pagecache pages during the run. RSS controller was not enabled during this experiment.



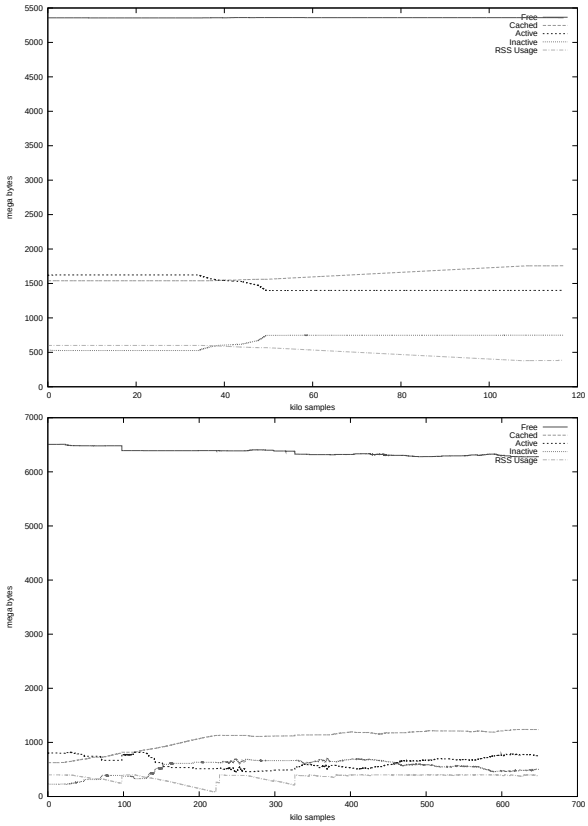
Observations:

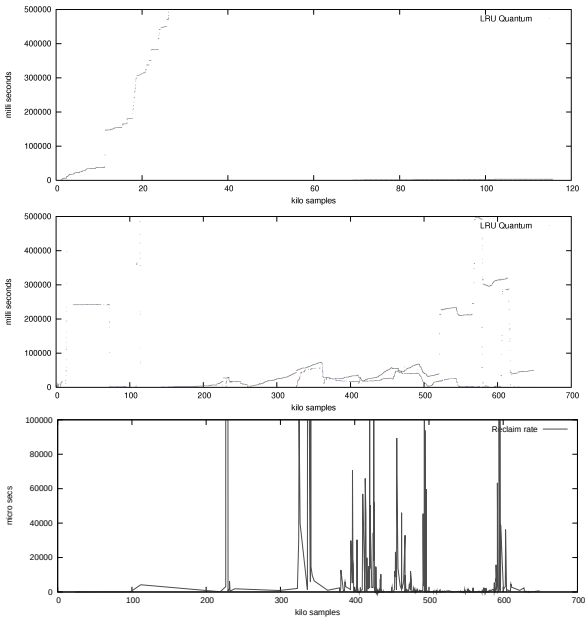
- Pagecache controller would not reclaim all pagecache pages when the limit is hit. The reclaimer would reclaim pages as much as possible to push the container below limit. Hence the pagecache usage and cached memory size is almost a straight line.
- As expected the active and inactive list variations are like mirror image of each other.
- Other parameter like LRU quantum, page generation and pagescan density was similar to pagetest program and not as widely distributed as kernbench. Pagecache usage pattern of dbench is much simpler compared to memory access pattern of kernbench.

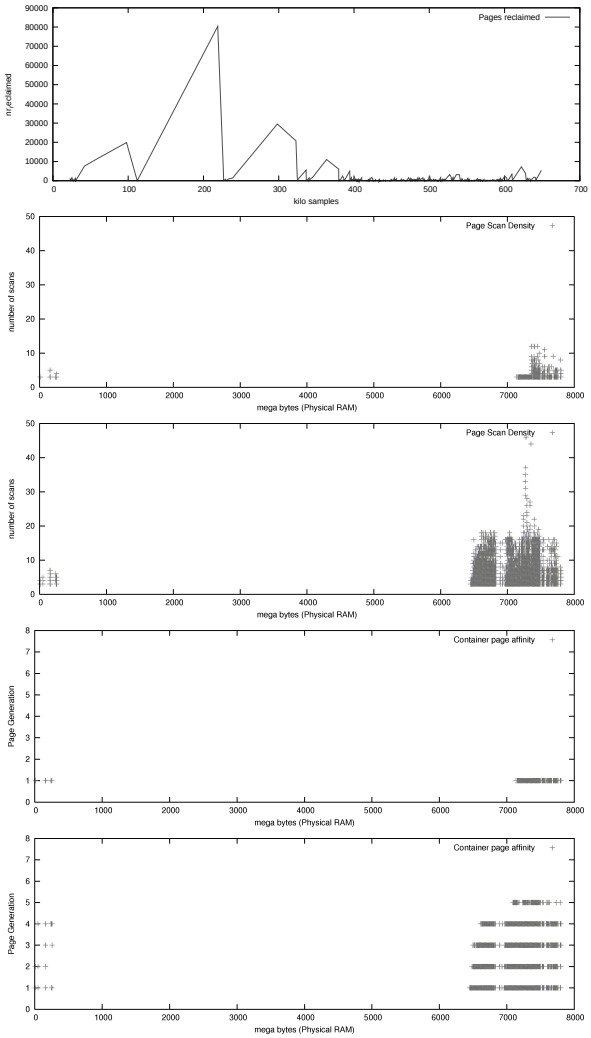
6.4 Web server workload

The daytrader benchmark application (with stock size of 2000 and 800 concurrent users) was run with IBM® Websphere™ community edition. Only the RSS control was enabled. The figures show reclaim parameter variation for container sizes of 600 MB and 400 MB respectively. The Web server workload involved an in-built database server called `derby`, which stores all daytrader data. The daytrader database results were obtained using the following steps:

1. Reset the daytrader data.
2. The configuration parameters are selected (direct transaction, no EJB, synchronous commit).
3. The database is populated with data.
4. We use a load balancer (web stress) tool to access the `/daytrader/scenario` URL of the Web server. We've used the Apache HTTP server benchmarking tool `ab` [1] in our testing.







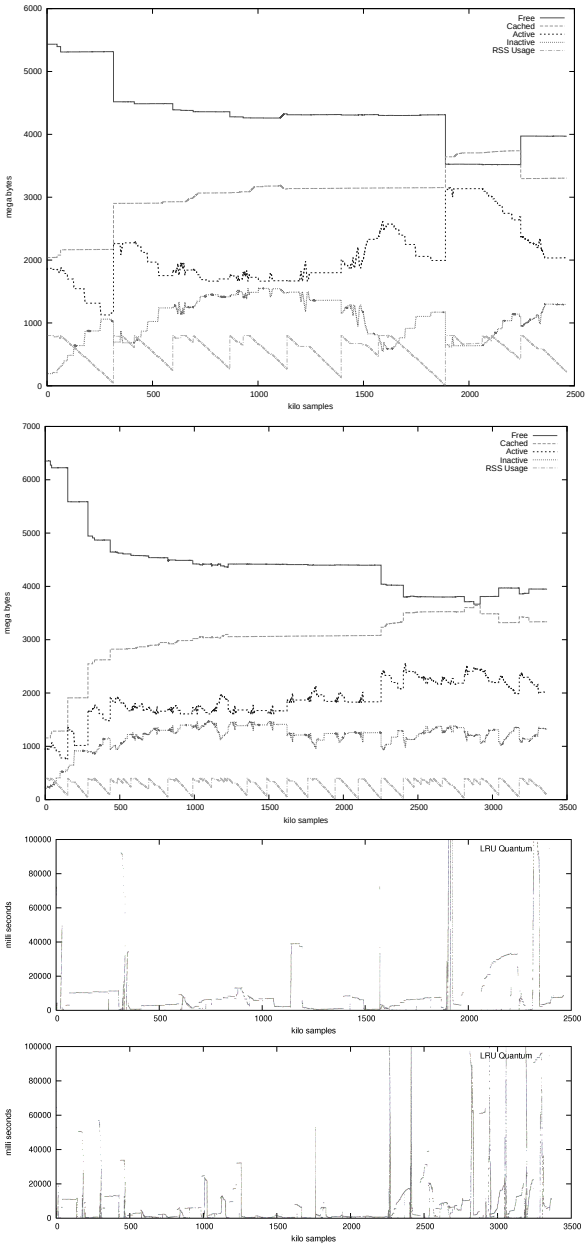
Observations:

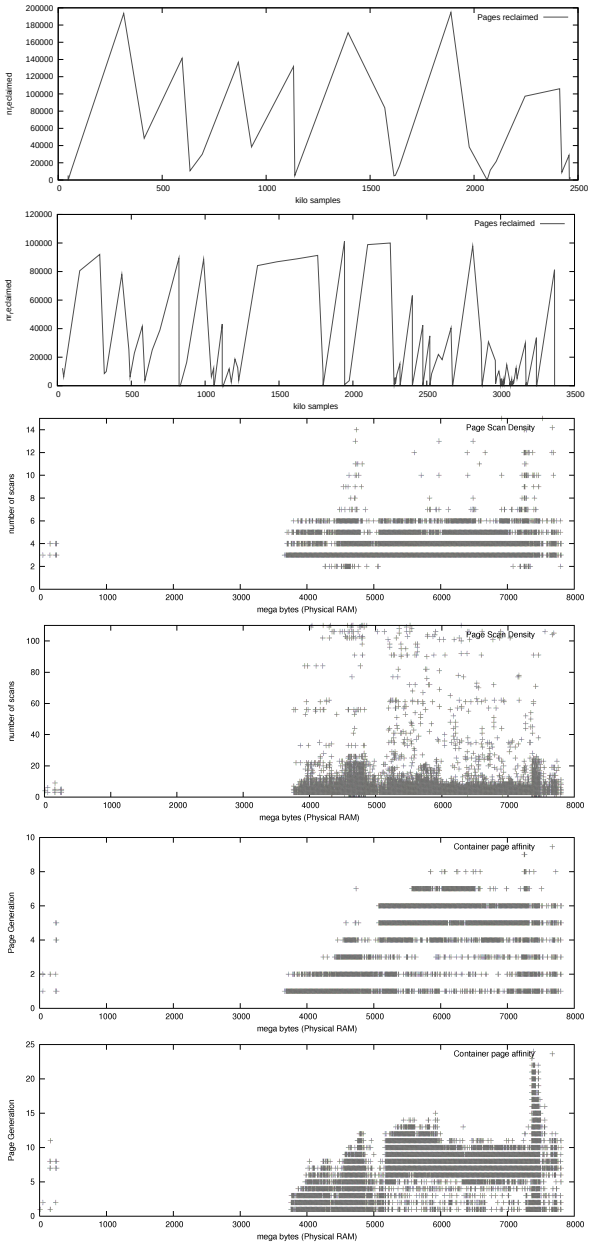
- Decreasing the size of the container reduced the LRU quantum value.
- Reclaim performance was poor when the number of pages reclaimed were low which resulted in high reclaim time.
- Decreasing the size of the container increased the page scan density. Each page was scanned more often before it could be freed.
- The range of physical memory used was independent of the size of the container used.
- The page generation went up as the size of the container was decreased.

6.5 Database workload

The `pgbench` [5] benchmark was run with only RSS control enabled. The figures show the reclaim parameter variation for container sizes of 800 MB and 400 MB respectively. The results were obtained using the following steps:

1. The database was initialized, with a scale factor of 100.
2. The benchmark `pgbench` was run with a scale factor of 100, simulating ten clients, each doing 1000 transactions.





Observations:

- Decreasing the size of the container increased the rate of change of LRU quantum.
- High LRU quantum values resulted in more pages being reclaimed.
- Decreasing the size of the container increased the page scan density Each page was scanned more often before it could be freed.
- The range of physical memory used was independent of the size of the container used.
- The range of physical memory used is bigger than the maximum RSS of the database server.
- The page generation went up as the size of the container was decreased.
- The range of physical memory used decreased as the page generation increased.

7 Future work

We plan to extend the basic RSS controller and the page-cache controller by adding an `mlock(2)` controller and support for accounting kernel memory, such as slab usage, page table usage, VMAs, and so on.

8 Conclusion

Memory control comes with the overhead of increased CPU time and lower throughput. This overhead is expected as each time the container goes over its assigned limit, page reclaim is initiated, which might further initiate I/O. A group of processes in the container are unlikely to do useful work if they hit their limits frequently, thus it is important for the page reclaim algorithm to ensure that when a container goes over its limit, it selects the right set of pages to reclaim. In this paper, we've looked at several parameters, that help us assess the performance of the workload in the container. We've also looked at the challenges in designing and implementing a memory controller.

The performance of a workload under a container is deteriorated as expected. Performance data shows that the impact of changing the container size might not be linear. This aspect requires further investigation along with the study of performance of pages shared across containers.

9 Open issues

The memory controller currently supports only limits. Guarantees support can be built on top of the current framework using limits. One desirable feature for controllers is excess resource distribution. Resource groups use *soft limits* to redistribute unutilized resources. Each container would get a percentage of unutilized resources in proportion to its soft limit. We have to analyze the impact of implementing such a feature.

10 Legal Statement

References

[1] Apache http benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] Pavel Emelianov. Memory controller with per-container page list. http://lkml.org/lkml/2007/3/6/198.

[3] Pavel Emelianov and Balbir Singh. Memory controller with per-container lru page list. http://lkml.org/lkml/2007/3/9/361.

[4] Roy Huang. Pagecache control through page allocation. http://lkml.org/lkml/2007/1/15/26.

[5] Tatsuo Ishii. Pgbench postgresql benchmark. http://archives.postgresql.org/pgsql-hackers/1999-09/msg00910.php.

[6] Kirill Korotaev. Beancounters v2. http://lkml.org/lkml/2006/8/23/117.

[7] Kirill Korotaev. Beancounters v6. http://lkml.org/lkml/2006/11/9/135.

[8] Christoph Lameter. Pagecache control through page allocation. http://lkml.org/lkml/2007/1/23/263.

[9] Aubrey Li. Pagecache control through page allocation. http://lkml.org/lkml/2007/1/17/202.

[10] Andrew Morton. Usermode pagecache control: fadvise(). http://lkml.org/lkml/2007/3/3/110.

[11] Chandra Seetharaman. Resource groups. http://lkml.org/lkml/2006/4/27/378.

[12] Rohit Seth. Containers. http://lkml.org/lkml/2006/9/14/370.

[13] Balbir Singh. Memory controller rfc. http://lkml.org/lkml/2006/10/30/51.

[14] Balbir Singh. Memory controller v2. http://lkml.org/lkml/2007/2/26/8.

[15] Vaidyanathan Srinivasan. Container pagecache controller. http://lkml.org/lkml/2007/3/06/51.