OpenFloo : Exploiting shared memory pages for inter-guest OS communication

Chirayu Shah, Jitesh Shah, Ketaki Kulkarni, Vedang Manerikar, Kedar Sovani

{chill.cps, jitesh.1337, ketakik86, ved.manerikar, kedars}@gmail.com

Abstract— **There has been a class of problems in Operating systems that deals with maintaining efficiency of two or more interacting or data sharing processes. The same set of problems surfaces again, albeit at a different granularity when multiple virtual machines running on the same physical host are interacting or sharing data. Although the problems are similar, the change from processes to virtual machines creates some more problems of its own. This paper explains how we went about optimising inter-virtual machine communication by means of floo-channels. Floo-channels are analogous to pipes. Although pipes have been in use for inter-process communication for a really long period, interesting problems are seen while doing pipes across two OSes.**

Index Terms—virtualisation, inter-vm communication, shared pages

I. Introduction

The developments in the virtualisation domain are making all the news today. This technique which has been around for a long time is seeing significant commercial success. As virtualisation becomes more main stream the focus is shifting towards the performance impacts of virtual machines. A lot of attention is on improving the I/O performance of virtual machines. As virtual machines are deployed to handle most of the tasks, inter virtual machine communication becomes vital. As of writing this paper, Xen has come out with its optimisation of inter-vm communication, “Xen-loop”. VmWare has implented its own VMCI (Virtual Machine Commincation Interface). This paper explains how we went about optimising inter-virtual machine communication by means of floo-channels. Floo-channels are temporary channels set-up between two communicating Vms. At the risk of being very naive, they can be thought of as inter-OS pipes. Although pipes have been in use for inter-process communication for a really long period, interesting problems are seen while doing pipes across two OSes.

Out of the myriad options for virtualisation available for Linux, we chose to implement this on co-Linux. Co-linux works with x86 architectures. Thus the discussion henceforth is is context of the x86 architecure.

An overview of the co-Linux framework is provided in the second section. The design section highlights our design decisions with the pros/cons of the available options. The Implementation section mentions how the design decisions were mapped to colinux specific structures. While the performance section presents some basic performance numbers of our implementation.

II. Overview of co-linux

CoLinux enables linux to run co-operatively on ring 0 along with the host OS. As the guest OSes co-operate with the host OS for proper functioning, the guest OSes are para-virtualised. The co-linux **manager** runs on the host OS and it is responsible for keeping track of all guest Oses running on the host machine. It maintains the manager structure, which keeps track of statistics and metadata for all the guest Oses. It is also reponsible for bringing up or teminating guest OSes. Furthermore, every guest OS is associated with a **monitor**. The monitor maintains metadata about the guest OS and provides this to the manager. The monitor is responsible for booting up the guest, initialising its address space and creation of the passage page. Co-linux installs its own IDTR and handles the page faults and other internal processor exceptions on its own (0x0-0x1f). Other interrupts are forwarded to the host OS. Also, the guests don't have direct access to block and network devices. A passage page is used to context switch between host and the guest OS.

Passage Page

The passage page is a page shared between the guest and the host OS. For switching between the host and the guest OS, the address space needs to be changed. A change is required in both CR3 and IP. Both have to be changed simultaneously, since (a) changing CR3 first would render the current IP meaningless and (b) changing IP first is pointless since, the code to be executed is in another address space altogether. Co-linux solves this problem by sharing a page between the host and the guest. The OS switching code resides on this page. Thus even though the CR3 is updated, both the address space have the same page mapped, and hence the current IP is meaningful.

The passage page is also used to implement a minimum form of communication between the host and the guest.

Pseudo Physical RAM: (PPRAM)

The Linux kernel expects the entire RAM to be allocated to itself. It thus assumes that the physical RAM is contiguous. But, a guest allocates memory from the host OS and this may not be contiguous. In order to give the guest OS a contiguous view of the physical RAM, PPRAM is used. PPRAM is basically another level of abstraction that maps from the pseudo-physical (contiguous) page frame numbers (visible to the guest) to the actual physical page frame numbers (visible to the host). Co-linux also maintains a reversed pseudo-physical mapping.

III. Design

Overview of issues

The main idea behind *floo* channels is to do to two virtual machines what pipes do to two processes. If one or more pages can be shared across multiple virtual machines and treated as queues, the machines could directly read/write data through the shared pages, thus making up a much faster and quicker communication channel. Much of the complexity in the implementation arises because of two factors. How to go about sharing pages dynamically between two guest Oses and how to implement synchronization (since, there is no single controlling entity like the kernel in the case of inter-process pipes).

Inter-OS sleep-wakeup present an interesting problem in this scheme. If data is available on a shared page, any possible process that are waiting for this data should be woken up. Since this wake-up is to be delivered to a process on another host OS, mechanisms should be developed to achieve this.

One of our important objectives was to make sure that the current applications work without any modifications to the application binaries. The aim was to map network socket calls to our openfloo calls at run-time in the kernel. Thus, when two guests running on the same host are talking to each other, automatically an openfloo channel is used, instead of a network channel. The rest of the calls proceed as normal through the socket layers. Thus, we maintain the client-server architecture commonly used in network socket programming.

Design

Every on-going communication channel setup between two guest OSes is associated with a handle which is a unique identifier for that communication channel. A handle uniquely identifies the communication end-points i.e. the tuple {source ip, source port, destination ip, destinaton port, protocol}. The manager structure keeps a track of all the handles. Every handle also keeps track of the shared pages allocated to that channel which can range from anywhere between 0 to OPENFLOO_MAX_SHARED_MEM (Set to 256 pages by default) depending on the volume of the traffic on that channel. A per-OS handle list is also maintained in every Guest OS. It additionally stores the pid associated with that handle. With this architecture in place, hooking the network stack amounts to mapping a socket to a handle.

Sharing a page

coLinux takes memory from the host dynamically. The "mem=<amount>" boot-time parameter simply specifies the maximum limit. For every page taken from the host, a mapping is setup in the PPRAM. So, we defined two flags __GFP_OPENFLOO_SERVER and __GFP_OPENFLOO_CLIENT to inform the host OS what type of memory is to be allocated. When the __GFP_OPENFLOO_SERVER flag is used with alloc_page(), a new page is taken from the host and its information is stored in the corresponding handle structure in the manager. When invoked with __GFP_OPENFLOO_CLIENT, a page stored above is retrieved and its reference count is incremented. Some error handling is also provided to avoid memory leakages due to server/client crash. Fig 1 below shows a shared page with its mapping in the PPRAM

Network Hooks:

The socket *accept* call is mapped to *openfloo_accept* call. This call makes sure that a *listening handle* is created in the host OS. The handle data structure keeps a note of the guest OSes that are communicating through this handle. A list of all the handles, currently in use, is maintained by the host OS. The guest OSes also maintain a list of mappings from the process in that guest to the handle that the process is using. The *openfloo_accept* call blocks waiting for a client to connect to it.

The client's *connect* call is mapped to the *openfloo_connect* call. This call identifies the handle that it should connect to and associates itself with that handle. It also wakes up the server process that was waiting for a client to connect to this handle. This required a mechanism of handling inter-OS sleeps and wakeup. More information about how we attained this is available in the "Inter-OS sleep-wake up" section.

Actual page allocations and deallocations are done on-demand i.e. during the *send* and *receive* calls (which are mapped to *openfloo_send* and *openfloo_receive*, respectively), thus, a channel is allocated only the amount of memory it actually requires. The details of how it is done follows in the "How it all gels together" section "Inter-OS sleep-wakeup".

The handle is cleared after the processes associated with the handle are destroyed.

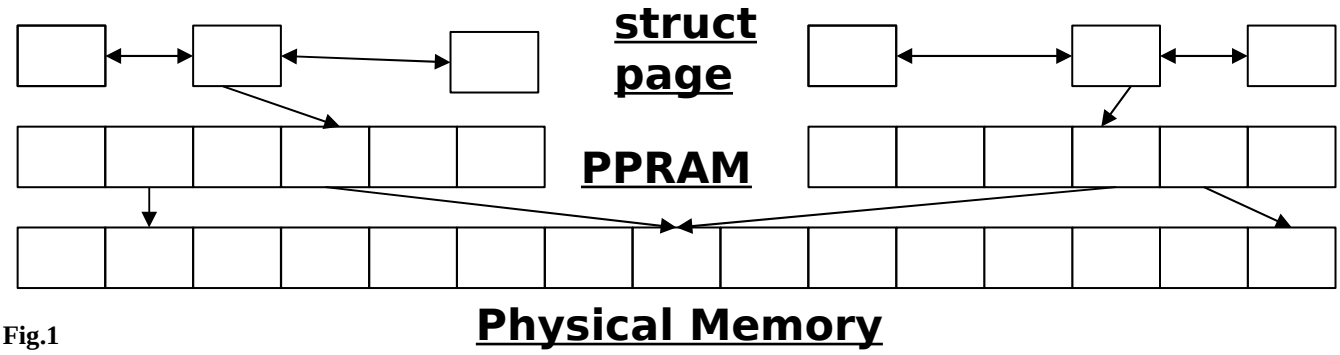

Fig.1

Inter-OS sleep-wakeup:
As mentioned earlier, a server waits for the client to connect to it. The server should be woken up when a client connects to it. Blocking is also observed in read and write calls for availability of data or resources respectively. These sleeping processes should be woken up as is appropriate. The problem with this is that the process to be woken up is in another OS altogether. We explored two alternative solutions for this problem.
The first one is that we plant an interrupt in the target OS from the host and change the process state of the pid associated with that handle to Ready in the ISR. A guest OS can request such a wake-up in another OS through the passage page.
Another less intrusive solution is to modify the Linux scheduler to check the wake-up bit in the passage page and change the corresponding process state to Ready. Obviously, the performance implications of this need to be measured since we will be using up a few clock cycles everytime the scheduler runs (whenever the guest kernel is compiled with OpenFloo support). We implemented this method.

Irrespective of whatever method of the above is used, here is what the mechanism of inter-OS sleep-wakeup looks like. Whenever, a guest wants to wake-up a process of another guest OS, it adds such a request alongwith the handle to its passage page and requests an OS switch to the host. The host looks up the target guest OS using the global handle structures and sets the wake-up bit and writes the handle to the target guest OSs' passage page (This can be done because the passage page is shared among the guest and the host and so, the host has access to all passage pages). When the host runs the targets guest next, the sleeping process is woken up using one of the methods described above.

Other hypervisors have their own methods to let the host and the guest communicate. The above strategy can be easily ported to any such method.

When to use the shared memory drivers?
Our objective is that the application binaries need not change i.e. they should continue using the usual network calls. This way the applications are truly unaware that they are working under virtualised environments. So, a run-time switch is desired which determines whether to use the OpenFloo shared memory architecture or the usual network stack. If the target machine is another guest OS on the same physical machine, then using the OpenFloo driver can speed up the communication. Otherwise, the network stack should be used.

This implementation requires a range or list of IP addresses that are of the guest OS running on this machine hardcoded in a configuration file. If both the source and then destination IP addresses are in this range/list, only then will the OpenFloo drivers hook the network stack for that socket.

In theory, whether the target machine is a VM on the same machine or not can also be determined at run-time. Specifically, the guest kernel can be patched to report every ip address assignment to the host OS via the passage page. The host can then maintain this data as a {host-id, ip-address} pair in its manager structure. This will obviate the need to hardcore the IP address range in a configuration file. However, we don't have this method implemented.

How it all gels together

Server-side
Whenever any network socket is created, the *accept* call is the socket's fops (functions pointers) is replaced with *openfloo_accept. openfloo_accept* blocks for a connection request. When one is received, a check is made whether the source of the connection is another VM or not. If it is, the *send* and *receive* function pointers are also modified to point to *openfloo_send* and *openfloo_receive.* If not, then *openfloo_accept* passes the structures and *accept* and the usual network stack in used.

In case of *openfloo* drivers, an entry is made into the global handle list in the manager structure of the host and also, the per-OS handle list. No page is allocated yet.

Client-side
Check similar to above are done when a client calls the *connect.* In case of *openfloo* drivers, the *send* and *receive* calls are also hooked (So, is *close).* When *openfloo_connect* completes , a unique handle has been defined with valid source and destination addresses. The channel has been established.

Send
openfloo_send checks whether there is enough space to append the application data (Network calls are hooked above the transport layer itself. So its the application data that we get). If yes, the *send* call simply reduces to a memory write. If not, more pages need to be allocated.

The driver sends a request to the host asking for an allocation (__GFP_OPENFLOO_SERVER flag is set). Depending on the last allocation request and the allocation strategy, the host allocates a certain number of pages and the *send* call then continues.

Receive
openfloo_receive checks the metadata to see whether data is available in the FIFO. If it is, the bytes are simply read and the corresponding FIFO pointers repositioned. If the read crosses a page boundary during this time, an asynchronous request to free the page is sent just before *openfloo_receive* exits.

If the end pointer points to a page that isn't yet allocated, the client calls *alloc_page* with __GFP_OPENFLOO_CLIENT

flag which merely retrieves the pages already allocated for that handle and the reference count in incremented. These are the shared pages. The *openfloo_receive* call then continues to read the FIFO.

Close

An *openfloo_close* call frees all the pages pertaining to that handle. In the host, the pages aren't freed until the reference count in zero, so that we don't worry about the other OS writing to or reading from a non-existent page.

An OS triggers a page reclaiming code in the host OS that automatically reduced the reference count of the shared page. The other OS is notified of this crash via the passage page and the connection is terminated.

Page allocation strategy

During the discussion of the *openfloo_send* call, we mentioned that the number of pages allocated depends on the page allocation strategy. Here we describe the strategy we used.

The objective is that for low traffic TCP connections, less number of pages should be allocated. Also, the high traffic connections shouldn't have to suffer asking for memory to the host often and incurring costly OS switches.

So, for the first allocation, only one page is allocated. For next allocations, we keep increasing the size of allocation by a power of two i.e. 2 pages, 4 pages, etc. The maximum is 16 pages. After that for any allocation request, 16 pages will be allocated.

Synchronization

Synchronization is at the heart of any shared memory communication scheme. For this purpose, a lock is defined alongwith the start and end pointers of the FIFO. This lock is manipulated by architecture specific atomic routines. Example the test_and_set instruction.

The co-Linux page pool

The memory allocated is dynamic only limited by the OPENFLOO_MAX_SHARED_MEM parameter per connection.

IV. Performance Analysis

We did some very basic performance analysis of the implementation on a single core 2.00 GHz x86 machine with 1 GB RAM. We wrote an application to transfer files from 1MB to 100MB broken down to 1k packets. We used tbench for out benchmarks. We setup inter-guest communication using the TUN/TAP drivers and port forwarding.

The average speed WITHOUT shared memory: 40-50 MB/s
The average speed WITH shared memory : 450 MB/s

Most of the savings can be owed to the fact that *send* is reduced to memory writes. So, we avoid costly OS switches and redundant buffer copies from guest-to-host and from host-back-to-guest in the network stack, using Floo channels

V. Future Work

Currently, the synchronization mechanism provided could run into race conditions on an SMP.

VI. Conclusions

We have seen how eliminating redundant buffer copies and costly OS switches boosts the network transfer speed among guest OSes in a virtualised environment.

VII. Applications

Application areas are where fast inter-VM communication is necessary. Web services, high-performance grid applications, consolidated servers, transaction processing, graphics rendering are some relevant fields.

VIII. References

[1] Paper by Dan Aloni on co-Linux at the Linux Symposium in 2004.

[2] http://www.colinux.wikia.com : How-Tos and achitecture