Proceedings of the Linux Symposium

Volume Two

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

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

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.*

Unifying Virtual Drivers

Jon Mason

3Leaf Systems

`jon.mason@3leafsystems.com`

Dwayne Shows

3Leaf Systems

`dwayne.shows@3leafsystems.com`

Dave Olien

3Leaf Systems

`dave.olien@3leafsystems.com`

Abstract

Para-virtualization presents a wide variety of issues to Operating Systems. One of these is presenting virtual devices to the para-virtualized operating system, as well as the device drivers which handle these devices. With the increase of virtualization in the Linux kernel, there has been an influx of unique drivers to handle all of these new virtual devices, and there are more devices on the way. The current state of Linux has four in-tree versions of a virtual network device (IBM pSeries Virtual Ethernet, IBM iSeries Virtual Ethernet, UML Virtual Network Device, and TUN/TAP) and numerous out-of-tree versions (one for Xen, VMware, 3leaf, and many others). Also, there are a similar number of block device drivers.

This paper will go into why there are so many, and the differences and commonalities between them. It will go into the benefits and drawbacks of combining them, their requirements, and any design issues. It will discuss the changes to the Linux kernel to combine the virtual network and virtual block devices into two common devices. We will discuss how to adapt the existing virtual devices and write drivers to take advantage of this new interface.

1 Introduction to I/O Virtualization

Virtualization is the ability of a system through hardware and/or software to run multiple instances of Operating Systems simultaneously. This is done through abstracting the physical hardware layer through a software layer, known as a hypervisor. Virtualzation is primarily implemented through two distinct ways: Full Virtualization and Para-virtualization.

Full Virtualization is a method that fully simulates hardware devices through emulating physical hardware components in software. This simulation of hardware allows the OS and other components to run their software unmodified. In this technique, all instructions are translated through the hypervisor into hardware system calls. The hypervisor controls the devices and I/O, and simulated hardware devices are exported to the virtual machine. This hardware simulation does have a significant performance penalty when compared to running the same software on native hardware, but allows the user to run multiple instances of a VM (virtual machine) simultaneously (thus allowing a higher utilization of the hardware and I/O). Examples of this type of virtualization are QEMU and VMware.

Para-virtualization is a method that modifies the OS running inside the VM to run under the hypervisor. It is modified to support the hypervisor and avoid unnecessary use of privileged instructions. These modifications allow the performance of the system to be near native. However, this type of virtualization requires that virtual devices be exposed for access to the underlying I/O hardware. UML and Xen are examples of such virtualization implimentations. The scope of this paper is in considering only para-virtualized virtual machine abstractions; this is where virtual device implementations have proliferated.

To have better I/O performance in a virtual machine, the VMs have virtual devices exported to them and have the native device located in another VM or in the hypervisor. These virtual devices have virtual device drivers for

them to appear in the VM as if they were native, physical devices. These virtual device drivers allow the data to be passed from the VM to the physical hardware. This hardware then does the required work, and may or may not return information to the virtual device driver. This level of abstraction allows for higher performance I/O, but the underlying procedure to perform the requested operation differs on each para-virtualzation implementation.

2 Linux Virtualization support

There are many Linux-centered virtualization implementations, both in the kernel and outside the kernel. The implementations that are currently in the Linux kernel have been there for some time and have matured over that time to be robust. Those implementations are UML, IBM's POWER virtualization, and the tun/tap device driver. Those implementations that are currently living outside the kernel tree are gaining popularity, and may one day be included in the mainline kernel. The implementations of note are Xen and 3leaf Systems virtualization (though many others exist).

2.1 In-tree Linux Virtualization support

2.1.1 User Mode Linux

User Mode Linux (UML) provides virtual block and network interfaces with a couple of predominant drivers.

Briefly, the network drivers recommended for UML are TUN/TAP. These are described in a later section. The virtual devices created can be used with other virtual devices on other VMs or with systems on the external network. To configure for forwarding to other VMs, UML comes with a switch daemon that allows user-level forwarding of packets. UML supports hot plug devices: new virtual devices can be configured dynamically while the system is up. As long as a multicast-capable NIC is available on the host, UML supports multicast devices and it is possible to multicast to all the VMs.

UML's virtual storage solution consists of exporting a file from a filesystem on storage known by the host to the guest. That storage can be used directly through reads and writes or with a feature called IO memory emulation that allows a file to be mapped as an IO region.

From the kernel address space the driver can use for user processes to mmap to their own address spaces.

All the guest reads and writes go through an API to the host and are translated to host reads and writes. As such, these operations are using the buffer cache on the host. This causes a disproportionate amount of memory to be consumed by the UMLs without limits as to how much they can utilize. As is characteristic to Linux, flushing modified buffers back to disk will be done when appropriate, based upon memory pressure in the host.

Another feature of UML virtual block device is supports "copy on write" (COW) partitions. COW partitions allow a common image of a root device to be shared among many instances of UML. Writes to blocks on the COW partition are kept in a unique device for that UML instance. This is more convenient than the Xen recommendation for using the LVM to provide COW functionality.

UML provides a number of APIs for file IO. These APIs hook into native versions largely unmodified Linux code. These APIs provides the virtual interface to the physical device on the host. Examples of these interfaces are `os_open_file`, `os_read_file`, and `os_write_file`.

2.1.2 IBM POWER virtualization

IBM's PowerPC-based iSeries and pSeries hardware provides a native hypervisor in system firmware, allowing the system to be easily virtualized. The user-level tools allow the underlying hardware to be assigned to specific virtual machines, and in certain cases no I/O hardware at all. In the latter case, one virtual machine is assigned the physical adapters and manages all I/O to and from those adapters. The hypervisor then exposes virtual devices to the other virtual machines. The virtual devices are presented to the OS via the system's Open Firmware device tree, and from the OS's perspective appear to be on a system bus [4]. Currently, the hypervisor supports virtual SCSI, ethernet, and TTY devices.

For the virtual ethernet device, the hypervisor implements a system-wide, VLAN-capable switch [4]. This enables a virtual ethernet device in one VM to have a connection to another VM via its virtual ethernet device. This virtual switch can be connected to the external network only by connecting a virtual ethernet device to a

physical device in one of the VMs. In Linux, this would be done via the kernel level ethernet bridging code.

For the virtual SCSI device, the connection is based on a client-server model [4]. The SCSI virtual client device works like most SCSI host controllers. It handles the SCSI commands via the SCSI mid-layer and issues SCSI commands to those devices. The SCSI virtual server device receives all SCSI commands and is responsible for handling them. The virtual SCSI inter-partition communication protocol is the SCSI RDMA Protocol.

2.1.3   TUN/TAP driver

The tun/tap driver has two basic funtions, a network tap and a network tunnel. The network tap simulates a ethernet device, and encapsulates incoming data inside an ethernet frame. The network tunnel simulates an IP layer device, and encapsulates the incoming data in an IP packet. It can be viewed as a simple Point-to-Point or Ethernet device, which instead of sending and receiving packets from a physical media, sends and receives them from user-space programs [5].

This enables virtualization programs running in user space (for example UML) access to the network without needing exclusive access to a network device, or any additional network configuration or OS kernel changes.

2.2   Out-of-tree Linux Virtualization support

2.2.1   Xen

Xen provides an architecture for device driver isolation using a split driver architecture. The virtual device functionality is provided by front-end and back-end device drivers. The front-end driver runs in the unprivileged "guest" domain, and the back-end runs in a "privileged" domain with access to the real device hardware. For block devices, these drivers are called *blkfront* and *blkback*; and for the network devices, *netfront* and *netback*. On the front end, the virtual device appears as a physical device and receives IO requests from the guest kernel.

The front-end driver must issue requests to the back-end driver since it doesn't have access to physical hardware. The back-end verifies that the request is safe and issues it to the real device. The back-end appears to the hypervisor as a normal user of in-kernel IO functionality. When the IO completes, the back-end notifies the front-end that its data is ready. The front-end driver reports IO completions to its own kernel. The back-end is responsible for translating device addresses and verifying that requests are correct and do not violate isolation guarantees.

Xen accomplishes device virtualization through a set of clean and simple device abstractions. IO data is transferred to and from each domain via grant tables using shared-memory, asynchronous buffer-descriptor rings. These are said to provide a high-performance communication mechanism for passing buffer information vertically through the system, while allowing Xen to efficiently perform validation checks—for example, of a domain's credits. This shared-memory interface is the fundamental mechanism supporting the split device drivers for network and block IO.

Each domain has its own grant table. This data structure is shared with the hypervisor, allowing the domain to tell Xen what kinds of permissions other domains have on its pages. Entries in the grant table are identified by a grant reference, an integer, which indexes into the grant table. It acts as a capability which the grantee can use to perform operations on the granter's memory. This mechanism allows shared-memory communications between unprivileged domains. A grant reference also encapsulates the details of a shared page, removing the need for a domain to know the real machine address of a page it is sharing. This makes it possible to share memory correctly among domains running in fully virtualized memory.

Grant table manipulation, the creation and destruction of grant references, is done by direct access to the grant table. This removes the need to involve the hypervisor when creating grant references, changing access permissions, etc. The grantee domain invokes hypercalls to use the grant reference.

Xen uses event-delivery mechanism for sending asynchronous notifications to a domain, similar to a hardware interrupt. These notifications are made by updating a bitmap of pending event types, and optionally calling an event handler specified by the guest. The events can be held off at the discretion of the guest.

Xenstore is the mechanism by which these event channels are set up, along with the shared memory frame.

It is used for setting up shared memory regions and event channels for use with the split device drivers. The store is arranged as a hierarchical collection of key-value pairs. Each domain has a directory structure containing data related to its configuration.

The setup protocol for a device channel should consist of entering the configuration data into the Xenstore area. The store allows device discovery without requiring the relevant device structure to be loaded. The probing code in the guest should see the Xen "bus."

Once communications is established between a pair of front- and back-end drivers, the two can communicate by directly placing requests/responses into shared memory and then on the event channel. This separation allows for message batching, making for efficient device access.

Xen Network IO

As mentioned, the shared memory communication area is shared between front-end and back-end domains. From the point of view of other domains, the back-end is viewed as a virtual ethernet switch, with each domain having one or more virtual network interfaces connected to it.

From the reference point of the back-end domain, the network driver on the back end consists of a number of ethernet devices. Each of these has a connection to a virtual network device in another domain. This allows the back-end domain to route, bridge, firewall, etc. all traffic from and to the other domains using the usual Linux mechanisms.

The back end is responsible for:

- Validation of data. The back end ensures the front ends do not attempt to generate invalid traffic. The back end may look at headers to validate MAC or IP addresses, making sure they match the interface they have been sent from.

- Scheduling. Since a number of domains can share the same physical NIC, the back end must schedule between domains that can have packets queued for transmission, or that may have ingress traffic. The back end is capable of traffic-shaping or rate-limiting schemes. Logging/Accounting on the back end can be configured to track/record events.

Ingress packets from the network are received by the back end. The back end simply acts as a demultiplexer, forwarding incoming packets to the correct front end via the appropriate virtual interface.

The asynchronous shared buffer rings described earlier are used for the network interface to implement transmit and receive rings. Each descriptor ring identifies a block of contiguous machine memory allocated to the domain. The transmit ring carries packets to transmit from the guest to the back-end domain. The return path of this ring carries messages indicating contents have been transmitted. This signals that the back-end driver does not need the pages of memory associated with that request.

To receive packets, the guest puts descriptors for unused pages of memory on the receive ring. The back end exchanges these pages in the domain's memory with new pages containing the received packet and passing back descriptors regarding the new packets in the ring. This is a zero-copy approach, allowing the back end to maintain a pool of free pages to receive packets into, delivering them to the associated domains after reading their headers. This is known as *page flipping*.

A domain that doesn't keep its receive ring filled with empty buffers will have dropped packets. This is seen as an advantage by Xen because it limits live-lock problems because the overloaded domain will stop receiving further data. Similarly, on the transmit path, it provides the application the feedback on the rate at which the packets can leave the system.

Flow control on the rings is managed by an independent mechanism from the flow of data on the transmit/receive rings. In this way the ring is divided into two message queues, one in each direction.

Xen Block IO

All disk access uses the virtual block device interface. It allows domains access to block storage devices visible to the block back-end device. The virtual block device is a split driver, like the network interface. A single shared memory ring is used between the front and back end for each virtual device. This memory ring handles all IO requests and responses for that virtual device.

Many storage types can be exported by the back-end domain for use by the front end—various network-based

block devices such as iSCSI, or NBD, as well as loopback and multipath devices. These devices get mapped to a device node on the front end in a static way defined by the guest's startup configuration.

The ring used by block IO supports two message types, read and write. For a read, the front end identifies the device and location to read from and attaches pages for data to be copied into. The back end acknowledges completed reads after the data is transferred from the device into the buffer, typically the underlying physical device's DMA engine. Writes are analogous to reads, except data moves from the front end to the back end.

Xen IO Configuration

Domains with physical device access (i.e., driver domains) receive access to certain PCI devices on a limited basis, acquiring access to interrupts and bus address space. Many guests attempt to determine the PCI configuration by accessing the PCI BIOS. Xen forbids such access and provides a hypercall, `physdev_op` to set/query configuration details.

2.2.2 3leaf Virtualization

3leaf Systems provides virtual storage and network interfaces built on top of typical Linux APIs. Their devices span physical machine boundaries so that front-end drivers can be on one system and the back-end drivers hosted where the physical devices reside.

Their front-end drivers communicate to the back-end drivers through a transport-agnostic interface that lends itself to running on any number of transports that meet some minimal set of requirements.

Front-end network devices look like a normal ethernet interface for the purposes of the front-end application/user. Such devices have their own MAC addresses randomly generated. Egress packets get wrapped with header information before being passed to the lower layers of the transport. This header is used for demultiplexing. The corresponding virtual NIC on the back end passes the packet to the bridge, where the packet gets forwarded to its recipient.

The storage front end looks like a SCSI device, usually having a SCSI device backing the back-end driver. The front-end registers with the block device layer so the

SCSI mid-layer can pass off requests with scatterlists. These get wrapped with header information before being passed to the lower layers of transport. This header is used for demultiplexing on the back end where the request is forwarded to the SCSI device. Completions come back to the back end where they are wrapped with the header information for the return trip through the transport to the front end. The front-end driver forwards the completion to the SCSI midlayer.

The 3leaf stack can manage multiple devices, and hotplug events, but can entail software queuing at different levels, especially the networking stack. Interactions between front and back ends are mitigated through thoughtful use of scatter-gather lists to chain requests. 3leaf services do not rely on an operating system being fully or para-virtualized; it is more of a distributed IO services mechanism. In some ways the hardware where the back-end runs is analogous to a hypervisor, whereas the front-end systems can be many and serve as the guests in the model of the virtual IO services surveyed here.

3Leaf virtual storage virtualization supports a number of useful features for providing diskless front-end clients with controlled, high-availability, high-performance access to storage. It also includes tools for assisting centralized provisioning to these distributed clients.

Following is a list of capabilities supported by this storage virtualization implementation. Each capability will be followed by a brief description of its implementation.

Features supported include:

- Multiple redundant access paths to storage. A typical configuration has two or more back-end servers, each with redundant paths to storage on a fibre channel SAN. The fail-over and fail-back between redundant fibre channel paths on the back-end redundant paths is managed by the Linux device mapper multipathing. A front end then has paths to storage through two or more back-end servers. The front-end and back-end storage software manages fail over and fail back between back-end servers.

- Block or SCSI storage devices. The virtual storage devices can appear on the front-end clients as either block or SCSI devices.

- Name persistence. The Linux hotplug software also includes mechanisms that can be used give

storage devices persistent names. For example, these names can be based on the UUID of the physical storage device. This mechanism requires some intervention on the front-end client to establish these mappings. The hotplug mechanism is still available for use on the 3leaf front-end clients. But the 3leaf virtual storage software also maintains its own stable naming mechanism. This can be administered centrally from the back-end storage server.

- Boot support for diskless front-end clients. The front-end client is able to load its operating system from the back-end server and then use a virtual disk as its root device. Access to this root device will also be highly available using the redundant paths provided by the storage virtualization.

- COW virtual devices. As with UML, the 3leaf COW devices are especially useful for root disks. It allows several client front ends to share a common root file system, allocating additional storage only for each client's modifications to that shared image.

- NPIV. N-port interface virtualization is another mechanism which allows the owner of the SAN to regulate access to devices on that SAN by individual front-end clients. This is based upon a virtual host bus adapter model, where the SAN administrator can associate sets of storage devices with virtual HBAs and then associate individual VHBAs with different client machines.

- Centralized provisioning. The back-end servers interact with a distributed set of tools to specify the virtual storage environment for the storage client. Storage devices can be added or removed from the client's environment, generating hotplug events to update the client's storage name space.

The 3leaf virtual storage implementation is conceptually similar to the Xen storage device implementation. Both are based on an efficient and reliable means for passing messages and DMA data transfers between the "guest" or front-end clients, and the "host" or back-end servers.

The messaging mechanism is used to implement an rpc-like communication between the back-end servers and the front-end clients. This rpc mechanism is used to simulate SCSI/Fibre channel behavior when needed, and to support the creation/deletion of virtual disks, the construction of COW devices, and VHBAs.

When the underlying transport supports it (e.g., infiniband RDMA), the transfer of disk data is transferred using RDMA read and RDMA write operations. RDMA read and write operations are initiated on the client. The RDMA operations use "opaque" handles to identify memory on the back-end servers that are the source or target of RDMA reads and writes, respectively. The client memory to be used is identified by a scatter/gather list.

These opqaue rdma handles are generated on the back-end servers as part of registering portions of the server's physical memory to be used for RDMA operations. In the case of infiniband, these handles are encoded in so that they cannot be forged. This provides some isolation between client front ends, making it difficult for clients to maliciously generate RDMA handles to memory they should not have access to. The opaque memory handles are transmitted from the back end server nodes to the front end's servers using the RPC mechanism. Each client is given a set of RDMA handles for segments of server memory. These memory segments sets are not shared between clients. Thus there is no chance of misdirected RDMA operations. Each client has ownership and control of its set of RDMA handles until it releases them. To perform a disk IO transfer, a client allocates an RDMA handle from the set given to it by the destination server. In the case of a write, it first transfers data using RDMA into the server's client memory. Then issues an rpc call to the server instructing it to write that memory to a disk device. In the case of a disk read, the client first sends an rpc to the server instructing it to read data from disk into the server's memory. It then uses its RDMA handle for that memory to transfer that data from the server's memory to the client's.

The rest of the virtual disk implementation is built upon these messaging and RDMA primitives. On the client, the virtual disk implementation appears to the Linux operating system's block layer and SCSI mid-layer as just another disk driver. The driver accepts either SCSI requests or block request structures, and translates them into rpc messages and RDMA operations targeted towards one of the servers providing access to disk storage. If the targeted server fails, these operations will be re-directed to one of the redundant servers for that storage.

On the server side, the disk virtualization is logically at the same level in the Linux disk software stack as the Linux page cache. The client and server's virtualization software manages its own pool of memory for RDMA operations. RPC requests from the client cause block requests to be submitted to the server's block layer to cause disk read and write operations. In this way, disk data transfers are performed without any copying of disk data by the CPUs on either the clients or the servers.

2.2.3 Non-local memory transport of data via IB

OpenFabrics.org is another player in the fabric of virtual IO solutions. Their solutions use infiniband as the transport to provide network and storage solutions to low-cost, front-end machines that run with their drivers and commodity HCA cards.

In this paradigm, virtual network interfaces are provided over the HCA ports with a protocol built on the verbs/access layer called the IPoIB module. This uses the two physical interfaces for each HCA; however, the MAC used is the GUID, which means it becomes difficult to put these packets on ipv4 networks. The ethernet header then needs to be massaged before transmitted packets can go out. When the virtual device's MAC is tied to the hardware, it becomes difficult to migrate virtual devices to other ports. The packets have to be bridged from the IB network to the ethernet network, much as packets from other solutions.

The storage solution provided is a module based on SCSI Request Protocol (SRP) initiator. It provides access to IB-based SRP storage targets.

3 Virtualized I/O

3.1 Virtualized Networking

The need for unified virtual ethernet drivers are many. First, the multitude of pre-existing code, and the potential for more in the future. The partial addition of features in an uncoordinated effort when many of the rest could also benefit from these features. UML provides switching from a user-level daemon which will have less performance than a kernel module. Not all implementations support hot-plug additions of virtual devices to the VM. Self-virtualizing devices are soon to be on the market and a common architecture should be adapted to take advantage of that functionality in the NICs.

3.2 Virtualized Storage

The need for unification of virtualized storage drivers is several fold. First, file-based partitions are slow. While convenient for desktop users, they do not meet the needs of an enterprise-scaled organization. It is well documented that read/write performance in the interfaces UML uses are slow. Additionally, file-based partitions also suffer from unchecked buffer cache growth on the host system. That tends to help performance, keeping much of the disk/partition resident in memory; however, as the number of disks group, it can cause inequities as to which VM has the lion's share of memory tied up. Attempted solutions to resolve that have been controversial and seem too specialized for the particular hypervisor running. `mmap` performance is not much better as an alternative, as some experiences indicate.

Inconsistent CoW implemenatations. The market has shown the need to have a single master root drive that is read-only, backed by a writable device to manage VM-based configuration differences which CoW provides a per-block mechanism for satisfying. Using the Linux Volume Manager as a solution goes beyond its original design; likewise there are limitations in using a bitmap implementation exclusively.

As far as disk-based partitions vs. file-based partitions, a disk gives a better unit of granularity that SAN storage providers have deployed with. It makes concepts like zoning and n-port virtualization much more achieveable.

SCSI-based devices seem a familiar and dependable mechanism to build a framework under. This mechanism is used by many other drivers and is itself a dependable framework supporting multiple device types. SCSI request blocks are already a well defined way to format requests and completions. A flexible back end should support all device types; however, for the scope of this document the discussion will be focused on SCSI disks. Such disks could be SAS (serial attached scsi) or storage that is allocted from a SAN fabric through an HBA (host bus adapter).

Lastly, a single flexible implementation will be better supported by the Linux community.

4 Design requirements and open issues

Abstracting the existing virtual drivers into a generic implementation raises interesting design requirements and

issues. Regardless of how the underlying virtualization actually works, there are a few basic interfaces for the network transport layer to interlock with the virtual ethernet driver, and the storage transport layer to interlock with the virtual SCSI driver.

4.1 Net

All the virtual ethernet device drivers described above contain certain basic features which are common, though the underlying methods of transfer the data from one point to another differs. All virtual network device drivers have functions which create and delete the device, start and stop the network stack, and transmit and receive data. These basic functions can be created in a generic virtual ethernet driver, which would then be supplemented by a specific module that handles any transport-specific calculations and transports the data from one point to another.

By providing a `veth_ops` data structure, with the pointers to the transport layer "helper" functions, this division of labor can greatly increase the ability to separate the existing virtual ethernet driver into a generic and transport layer.

For example in Xen, the packet to be transmitted has a few transport-specific calculations that need to be done (like insertion into the grant table and calculation for the number of pages the skb fits into). Those can be pushed down into a transmit transport layer, to which the skb is handed for transmission. In contrast, the tun/tap driver simply queues the packet onto a skb queue.

```
veth_ops->tx(struct sk_buff *skb,
            struct net_device *netdev);
```

This enables generic error checking and setup that all data transmission routes need in the generic tx stub, and the transport-specific work can be done in the function pointed to by the `veth_ops`.

For receiving, things can be significantly simplier using NAPI. Registering a generic poll routine on device open, and providing a skb queue to pull from, makes this very generic. The transport layer populates the queue as packets are pulled off its transport layer (via interrupt, etc.).

Open and close will need generic and transport-layer constructs to set up the virtual device to transmit and

receive data, as well as destroy any resources allocated. While some of these functions are specific to the transport layer (like creating buffer pools), most of the drivers require the very basic setup of zeroing statistics and starting the transmit queue.

```
veth_ops->open(struct net_device
            *netdev);
veth_ops->close(struct net_device
            *netdev);
```

Unfortunately, driver probe and create are very dependent on how the attributes are passed to the driver. Since some are passed via `hcall` and others are provided by the user-space tools, there really is no abstract way to do this.

There are other generic functions that can easily be made generic. Specificly, functions to change the MTU, transmit timeout, and get statistics are all generic and really do not need any hooks in them to work for all virtualization implementations.

There are currently some offloading technologies which have been implemented in some of the drivers via software, and which have not been proliferated into all of the existing drivers. For example, GSO and checksum offloads. Integrating these functions into existing implementations might be easy; they are very implementation-dependent and should only be abstracted after further investigation.

Our current implementation uses the API defined above to communicate between the generic virtual ethernet driver and a few generic transport layers. Specificly, we generalized the Xen network to behave in the above way, as well as UML and 3leaf.

4.2 Disk

We propose virtualization-aware devices using a unified generic stub. This allows for a transport/virtualization-specific layer. If a transport were to be connected through the generic stubs, the front end and back end would not have to be co-located on the same piece of hardware.

By providing a `vstore_ops` structure, with the structure elements being pointers to transport layer "helper"

functions, one can separate existing virtual storage drivers into a generic and transport layer.

In Xen the IO request results in pages inserted into the grant table for the blkback driver for reads and writes. They can be pushed down to the transport layer in a way that shields the upper layers from the transport. In contrast, UML with its UBD driver does reads and writes to the host operating system, largely driven through the `os_*` interfaces. These are block device interfaces, but the pertinent information is available from the `ioreq` struct. The 3leaf front-end driver can accomodate both. Writes and reads have the same prototypes:

```
vstore_ops->dma_write(struct scatterlist
                      local_buf_list[],
                      int nbuf, int size);
vstore_ops->dma_read(struct scatterlist
                     local_buf_list[],
                     int nbuf, int size);
```

Open and release of block devices are generic enough that the transport-specific portion is easily isolated. The open causes caches of buffers to be allocated and data structure initialization. The release causes those resources to be freed.

```
vstore_ops->open(struct inode *inode,
                 struct file *filp);

vstore_ops->release(struct inode *inode,
                    struct file *filp);
```

The auto-provisioning in the 3leaf model is preferable for many reasons. There needs to be a way for a VM to probe for devices; this callback fills that requirement, not unlike Xen—when it reads its configuration, it's technically accessing the Xen bus.

5 Roadmap/Future work

We plan on extending this work into all virtualization implementations mentioned in this paper. However, due to certain logistical limitations, we have not been able to do this. For example, there was no access to IBM POWER-enabled hardware.

Self-virtualizing devices which can be offloaded with certain virtualization functionality are becoming more prevalent. These devices require that one physical device appear as multiple devices, each VM having a semi-programmable device exclusively for its own. Exporting

the generic driver to these devices could be quite beneficial.

A performance analysis of the merged drivers, with a contrast to the existing drivers, should be done prior to any merging into mainline.

The driver should be expanded to accomodate block devices as well as scsi, using the 3leaf model for supporting both types.

Features in the 3leaf solution should be addressed, including multiple redundant paths to storage, name persistence, and centralized provisioning.

6 Conclusion

We implemented a generic virtual device driver with implementation-specific transportation layer for UML, Xen, and 3leaf Systems. We have shown a breadth of virtualization technologies that would benefit from using this, and the generic API which can be used to do this.

We will continue to clean and extend the usage of this implementation in Linux. Hopefully, by the time this is read, it will be submitted for inclusion into the Linux kernel.

7 Terms

front end, back end
One class of distributed computer system in which the computers are divided into two types: back-end computers and front-end computers. Front-end computers typically have minimal peripheral hardware (e.g., storage and ethernet) and interact with users and their applications. Back-end computers provide the front ends with access to expensive peripheral devices or services (e.g., a database), so as to share the cost of those peripherals across the front ends.

Also: client, server.

bridge
A mechanism to forward network packets between ports or interfaces.

VM (virtual machine)
Software that provides a virtual environment on top of

the hardware platform. This virtual environment provides services for creating and managing virtual IO devices such as disks and network interfaces.

CoW (Copy on Write)
A technique for efficiently sharing the un-modified contents of a disk or file system whose contents are "read mostly"—for example, a root file system. Writes to this shared content are re-directed to a write-able device that is unique to each user.

DMA (Direct Memory Access)
A hardware mechanism used by peripheral devices to transfer data between the pre-determined locations in computer memory and the peripheral device, without using the computer's cpu to copy that data.

RDMA (Remote Direct Memory Access)
An extension of DMA where data is transferred between pre-determined memory locations on two computer systems over a network connection, without utilizing cpu cycles to copy data.

MAC (Media Acess Control) address
A unique identifier associated with a network adapter.

GUID (Globally Unique Identifier)
A 128-bit number, unique identifier that is associated with an infiniband HCA.

IPoIB (Internet Protocol over InfiniBand)
An implementation of the internet protocol on the Infiniband network fabric.

SRP (SCSI RDMA Protocol)
A combination of the SCSI protocol with Infiniband RDMA, providing SAN storage.

UML (User Mode Linux)
A virtual machine implementation where one or more guest Linux operating systems run in user-mode Linux processes.

hypvervisor (also, virtual machine monitor)
The software that provides the virtual machine mechanisms to support guest operating systems.

infiniband
A point-to-point communications link used to provide high performance data and message transfer between computer nodes.

NIC (Network Interface Controller)
A hardware device that allows computers to communicate over a network.

page cache, buffer cache
A cache of disk-backed memory pages. The Linux operating system uses a page cache for holding process memory pages as well as file data.

scatterlist
A list (typically an array) of physical memory addresses and lengths used to specify the source or destination for a DMA transfer.

RPC (Remote Procedure Call)
A protocol where software on one computer can invoke a function on a remote computer.

SCSI (Small Computer System Interconnect)
A standard for physically connecting and transferring data between computer systems and peripheral storage devices.

API (Application Programming Interface)
A software interface definition for providing services.

HCA (Host Channel Adapter)
A hardware device for connecting a computer system to an infiniband communications link.

HBA (Host Bus Adapter)
A hardware device that connecting a computer system to a SCSI or Fibre Channel link.

hotplug (hot plugging)
A method for adding or removing devices from a computer system while that computer system is operating.

N-Port (Node Port)
a Fibre Channel node connection.

Fibre Channel
A network implementation that is used mostly for accessing storage.

SAN (Storage Area Network)
A network architecture for attaching remote storage to a server computer.

Volume Manager
Software for managing and allocating storage space.

8 Legal Statement

All statements regarding future direction and intent are sub-ject to change or withdrawal without notice, and represent goals and objectives only. Information is provided "AS IS" without warranty of any kind. This article could include tech-nical inaccuracies and typographical errors. Improvements and/or changes in the product(s) and/or the program(s) de-scribed in this publication may be made at any time without notice. This paper represents the view of its authors and not necessarily the view of 3leaf Systems. Other company, prod-uct, or service names may be the trademarks of others. Void where prohibited.

References

[1] M.D. Day, R. Harper, M. Hohnbaum, A. Liguori, & A. Theurer. "Using the Xen Hypervisor to Supercharge OS Deployment," Proceedings of the 2005 Linux Symposium, Ottawa, Vol. 1, pp. 97–108.

[2] K. Fraser, S. Hand, C. Limpach, and I. Pratt. "Xen and the Art of Open Source Virtualization," Proceedings of the 2004 Linux Symposium, Ottawa, Vol. 2, p. 329.

[3] Xen 3.0 Virtualization Interface Guide
`http://www.linuxtopia.org/online_`
`books/linux_virtualization/xen_3.0_`
`interface_guide/linux_virualization_`
`xen_interface_25.html`

[4] Dave Boutcher, Dave Engebretsen. "Linux Virtualization on IBM POWER5 Systems," Proceedings of the 2004 Linux Symposium, Ottawa, Vol. 1, p. 113.

[5] `http://www.kernel.org/pub/linux/`
`kernel/people/marcelo/linux-2.4/`
`Documentation/networking/tuntap.txt`

[6] Raj, Ganev, Schwan, and Xenidis. Scalable IO Virtualization via Self-Virtualizing Devices,
`http://www-static.cc.gatech.edu/`
`~rhim/Self-VirtTR-v1.pdf`

[7] Jeff Dike, *User Mode Linux*, Prentice Hall, 2006.

[8] `http://wiki.openvz.org`

The new ext4 filesystem: current status and future plans

Avantika Mathur, Mingming Cao, Suparna Bhattacharya
IBM Linux Technology Center
`mathur@us.ibm.com, cmm@us.ibm.com, suparna@in.ibm.com`
Andreas Dilger, Alex Tomas
Cluster Filesystem Inc.
`adilger@clusterfs.com, alex@clusterfs.com`
Laurent Vivier
Bull S.A.S.
`laurent.vivier@bull.net`

Abstract

Ext3 has been the most widely used general Linux® filesystem for many years. In keeping with increasing disk capacities and state-of-the-art feature requirements, the next generation of the ext3 filesystem, ext4, was created last year. This new filesystem incorporates scalability and performance enhancements for supporting large filesystems, while maintaining reliability and stability. Ext4 will be suitable for a larger variety of workloads and is expected to replace ext3 as the "Linux filesystem."

In this paper we will first discuss the reasons for starting the ext4 filesystem, then explore the enhanced capabilities currently available and planned for ext4, discuss methods for migrating between ext3 and ext4, and finally compare ext4 and other filesystem performance on three classic filesystem benchmarks.

1 Introduction

Ext3 has been a very popular Linux filesystem due to its reliability, rich feature set, relatively good performance, and strong compatibility between versions. The conservative design of ext3 has given it the reputation of being stable and robust, but has also limited its ability to scale and perform well on large configurations.

With the pressure of increasing capabilities of new hardware and online resizing support in ext3, the requirement to address ext3 scalability and performance is more urgent than ever. One of the outstanding limits faced by ext3 today is the 16 TB maximum filesystem size. Enterprise workloads are already approaching this limit, and with disk capacities doubling every year and 1 TB hard disks easily available in stores, it will soon be hit by desktop users as well.

To address this limit, in August 2006, we posted a series of patches introducing two key features to ext3: larger filesystem capacity and extents mapping. The patches unavoidably change the on-disk format and break forwards compatibility. In order to maintain the stability of ext3 for its massive user base, we decided to branch to ext4 from ext3.

The primary goal of this new filesystem is to address scalability, performance, and reliability issues faced by ext3. A common question is why not use XFS or start an entirely new filesystem from scratch? We want to give the large number of ext3 users the opportunity to easily upgrade their filesystem, as was done from ext2 to ext3. Also, there has been considerable investment in the capabilities, robustness, and reliability of ext3 and e2fsck. Ext4 developers can take advantage of this previous work, and focus on adding advanced features and delivering a new scalable enterprise-ready filesystem in a short time frame.

Thus, ext4 was born. The new filesystem has been in mainline Linux since version 2.6.19. As of the writing of this paper, the filesystem is marked as developmental, titled ext4dev, explicitly warning users that it is not ready for production use. Currently, extents and 48-bit block numbers are included in ext4, but there are many new filesystem features in the roadmap that will be discussed throughout this paper. The current ext4 development git tree is hosted at `git://git.kernel.org/`

`pub/scm/linux/kernel/git/tytso/ext4`. Up-to-date ext4 patches and feature discussions can be found at the ext4 wiki page, `http://ext4.wiki.kernel.org`.

Some of the features in progress could possibly continue to change the on-disk layout. Ext4 will be converted from development mode to stable mode once the layout has been finalized. At that time, ext4 will be available for general use by all users in need of a more scalable and modern version of ext3. In the following three sections we will discuss new capabilities currently included in or planned for ext4 in the areas of scalability, fragmentation, and reliability.

2 Scalability enhancements

The first goal of ext4 was to become a more scalable filesystem. In this section we will discuss the scalability features that will be available in ext4.

2.1 Large filesystem

The current 16 TB filesystem size limit is caused by the 32-bit block number in ext3. To enlarge the filesystem limit, the straightforward method is to increase the number of bits used to represent block numbers and then fix all references to data and metadata blocks.

Previously, there was an extents[3] patch for ext3 with the capacity to support 48-bit physical block numbers. In ext4, instead of just extending the block numbers to 64-bits based on the current ext3 indirect block mapping, the ext4 developers decided to use extents mapping with 48-bit block numbers. This both increases filesystem capacity and improves large file efficiency. With 48-bit block numbers, ext4 can support a maximum filesystem size up to $2^{(48+12)} = 2^{60}$ bytes (1 EB) with 4 KB block size.

After changing the data block numbers to 48-bit, the next step was to correct the references to metadata blocks correspondingly. Metadata is present in the superblock, the group descriptors, and the journal. New fields have been added at the end of the superblock structure to store the most significant 32 bits for block-counter variables, s_free_blocks_count, s_blocks_count, and s_r_blocks_count, extending them to 64 bits. Similarly, we introduced new 32-bit fields at the end of the block group descriptor structure to store the most significant bits of 64-bit values for bitmaps and inode table pointers.

Since the addresses of modified blocks in the filesystem are logged in the journal, the journaling block layer (JBD) is also required to support at least 48-bit block addresses. Therefore, JBD was branched to JBD2 to support more than 32-bit block numbers at the same time ext4 was forked. Although currently only ext4 is using JBD2, it can provide general journaling support for both 32-bit and 64-bit filesystems.

One may question why we chose 48-bit rather than full 64-bit support. The 1 EB limit will be sufficient for many years. Long before this limit is hit there will be reliability issues that need to be addressed. At current speeds, a 1 EB filesystem would take 119 years to finish one full e2fsck, and 65536 times that for a 2^{64} blocks (64 ZB) filesystem. Overcoming these kind of reliability issues is the priority of ext4 developers before addressing full 64-bit support and is discussed later in the paper.

2.1.1 Future work

After extending the limit created by 32-bit block numbers, the filesystem capacity is still restricted by the number of block groups in the filesystem. In ext3, for safety concerns all block group descriptors copies are kept in the first block group. With the new uninitialized block group feature discussed in section 4.1 the new block group descriptor size is 64 bytes. Given the default 128 MB(2^{27} bytes) block group size, ext4 can have at most $2^{27}/64 = 2^{21}$ block groups. This limits the entire filesystem size to $2^{21} * 2^{27} = 2^{48}$ bytes or 256TB.

The solution to this problem is to use the *metablock group* feature (META_BG), which is already in ext3 for all 2.6 releases. With the META_BG feature, ext4 filesystems are partitioned into many metablock groups. Each metablock group is a cluster of block groups whose group descriptor structures can be stored in a single disk block. For ext4 filesystems with 4 KB block size, a single metablock group partition includes 64 block groups, or 8 GB of disk space. The metablock group feature moves the location of the group descriptors from the congested first block group of the whole filesystem into the first group of each metablock group itself. The backups are in the second and last group of each metablock group. This increases the 2^{21} maximum
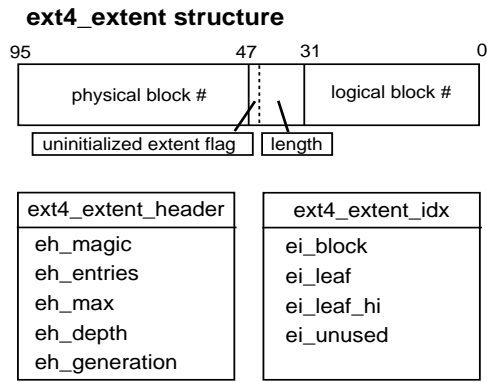
Figure 1: Ext4 extents, header and index structures

block groups limit to the hard limit 2^{32}, allowing support for the full 1 EB filesystem.

2.2 Extents

The ext3 filesystem uses an indirect block mapping scheme providing one-to-one mapping from logical blocks to disk blocks. This scheme is very efficient for sparse or small files, but has high overhead for larger files, performing poorly especially on large file delete and truncate operations [3].

As mentioned earlier, extents mapping is included in ext4. This approach efficiently maps logical to physical blocks for large contiguous files. An *extent* is a single descriptor which represents a range of contiguous physical blocks. Figure 1 shows the extents structure. As we discussed in previously, the physical block field in an extents structure takes 48 bits. A single extent can represent 2^{15} contiguous blocks, or 128 MB, with 4 KB block size. The MSB of the extent length is used to flag uninitialized extents, used for the preallocation feature discussed in Section 3.1.

Four extents can be stored in the ext4 inode structure directly. This is generally sufficient to represent small or contiguous files. For very large, highly fragmented, or sparse files, more extents are needed. In this case a constant depth extent tree is used to store the extents map of a file. Figure 2 shows the layout of the extents tree. The root of this tree is stored in the ext4 inode structure and extents are stored in the leaf nodes of the tree.

Each node in the tree starts with an extent header (Figure 1), which contains the number of valid entries in
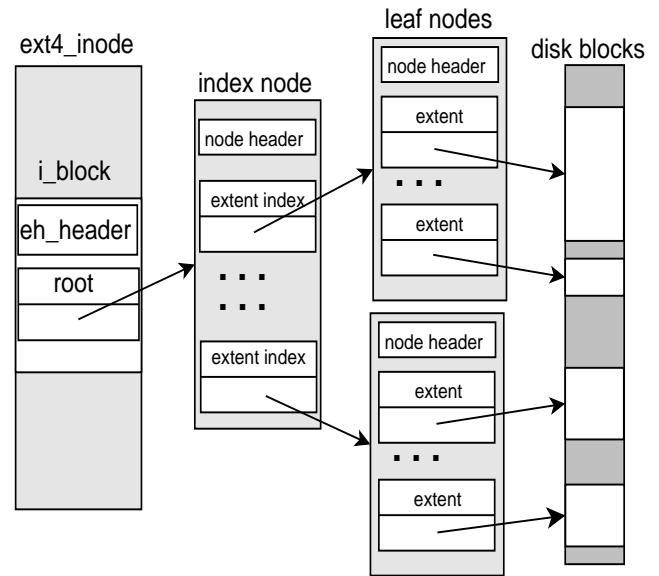
Figure 2: Ext4 extent tree layout

the node, the capacity of entries the node can store, the depth of the tree, and a magic number. The magic number can be used to differentiate between different versions of extents, as new enhancements are made to the feature, such as increasing to 64-bit block numbers.

The extent header and magic number also add much-needed robustness to the on-disk structure of the data files. For very small filesystems, the block-mapped files implicitly depended on the fact that random corruption of an indirect block would be easily detectable, because the number of valid filesystem blocks is a small subset of a random 32-bit integer. With growing filesystem sizes, random corruption in an indirect block is by itself indistinguishable from valid block numbers.

In addition to the simple magic number stored in the extent header, the tree structure of the extent tree can be verified at runtime or by e2fsck in several ways. The ext4_extent_header has some internal consistency (eh_entries and eh_max) that also depends on the filesystem block size. eh_depth decreases from the root toward the leaves. The ext4_extent entries in a leaf block must have increasing ee_block numbers, and must not overlap their neighbors with ee_len. Similarly, the ext4_extent_idx also needs increasing ei_block values, and the range of blocks that an index covers can be verified against the actual range of blocks in the extent leaf.

Currently, extents mapping is enabled in ext4 with the *extents* mount option. After the filesystem is mounted,

any new files will be created with extent mapping. The benefits of extent maps are reflected in the performance evaluation Section 7.

2.2.1 Future work

Extents are not very efficient for representing sparse or highly fragmented files. For highly fragmented files, we could introduce a new type of extent, a block-mapped extent. A different magic number, stored in the extent header, distinguishes the new type of leaf block, which contains a list of allocated block numbers similar to an ext3 indirect block. This would give us the increased robustness of the extent format, with the block allocation flexibility of the block-mapped format.

In order to improve the robustness of the on-disk data, there is a proposal to create an "extent tail" in the extent blocks, in addition to the extent header. The extent tail would contain the inode number and generation of the inode that has allocated the block, and a checksum of the extent block itself (though not the data). The checksum would detect internal corruption, and could also detect misplaced writes if the block number is included therein. The inode number could be used to detect corruption that causes the tree to reference the wrong block (whether by higher-level corruption, or misplaced writes). The inode number could also be used to reconstruct the data of a corrupted inode or assemble a deleted file, and also help in doing reverse-mapping of blocks for defragmentation among other things.

2.3 Large files

In Linux, file size is calculated based on the i_blocks counter value. However, the unit is in sectors (512 bytes), rather than in the filesystem block size (4096 bytes by default). Since ext4's i_blocks is a 32-bit variable in the inode structure, this limits the maximum file size in ext4 to $2^{32} * 512$ bytes $= 2^{41}$ bytes $= 2$ TB. This is a scalability limit that ext3 has planned to break for a while.

The solution for ext4 is quite straightforward. The first part is simply changing the i_blocks units in the ext4 inode to filesystem blocks. An ROCOMPAT feature flag HUGE_FILE is added in ext4 to signify that the i_blocks field in some inodes is in units of filesystem block size. Those inodes are marked with a flag EXT4_HUGE_FILE_FL, to allow existing inodes to keep i_blocks in 512-byte units without requiring a full filesystem conversion. In addition, the i_blocks variable is extended to 48 bits by using some of the reserved inode fields. We still have the limitation of 32 bit logical block numbers with the current extent format, which limits the file size to 16TB. With the flexible extents format in the future (see Section 2.2.1), we may remove that limit and fully use the 48-bit i_blocks to enlarge the file size even more.

2.4 Large number of files

Some applications already create billions of files today, and even ask for support for trillions of files. In theory, the ext4 filesystem can support billions of files with 32-bit inode numbers. However, in practice, it cannot scale to this limit. This is because ext4, following ext3, still allocates inode tables statically. Thus, the maximum number of inodes has to be fixed at filesystem creation time. To avoid running out of inodes later, users often choose a very large number of inodes up-front. The consequence is unnecessary disk space has to be allocated to store unused inode structures. The wasted space becomes more of an issue in ext4 with the larger default inode. This also makes the management and repair of large filesystems more difficult than it should be. The uninitialized group feature (Section 4.1) addresses this issue to some extent, but the problem still exists with aged filesystems in which the used and unused inodes can be mixed and spread across the whole filesystem.

Ext3 and ext4 developers have been thinking about supporting dynamic inode allocation for a while [9, 3]. There are three general considerations about the dynamic inode table allocation:

- Performance: We need an efficient way to translate inode number to the block where the inode structure is stored.

- Robustness: e2fsck should be able to locate inode table blocks scattered across the filesystem, in the case the filesystem is corrupted.

- Compatibility: We need to handle the possible inode number collision issue with 64-bit inode numbers on 32-bit systems, due to overflow.

These three requirements make the design challenging.

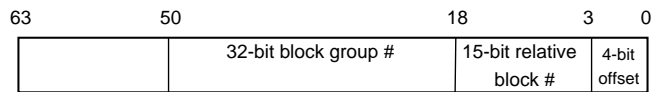| 63	50		18	3	0
	32-bit block group #		15-bit relative block #	4-bit offset	

Figure 3: 64-bit inode layout

With dynamic inode tables, the blocks storing the inode structure are no longer at a fixed location. One way to efficiently map the inode number to the block storing the corresponding inode structure, is encoding the block number into the inode number directly, similar to what is done in XFS. This implies the use of 64-bit inode numbers. The low four to five bits of the inode number store the offset bits within the inode table block. The rest store the 32-bit block group number as well as 15-bit relative block number within the group, shown in Figure 3. Then, a cluster of contiguous inode table blocks (ITBC) can be allocated on demand. A bitmap at the head of the ITBC would be used to keep track of the free and used inodes, allowing fast inode allocation and deallocation.

In the case where the filesystem is corrupted, the majority of inode tables could be located by checking the directory entries. To further address the reliability concern, a magic number could be stored at the head of the ITBC, to help e2fsck to recognize this metadata block.

Relocating inodes becomes tricky with this block-number-in-inode-number proposal. If the filesystem is resized or defragmented, we may have to change the location of the inode blocks, which would require changing all references to that inode number. The proposal to address this concern is to have a per-group "inode exception map" that translates an old block/inode number into a new block number where the relocated inode structure is actually stored. The map will usually be empty, unless the inode was moved.

One concern with the 64-bit inode number is the possible inode number collision with 32-bit applications, as applications might still be using 32-bit stat() to access inode numbers and could break. Investigation is underway to see how common this case is, and whether most applications are currently fixed to use the 64-bit stat64(). One way to address this concern is to generate 32-bit inode numbers on 32-bit platforms. Seventeen bits is enough to represent block group numbers on 32-bit architectures, and we could limit the inode table blocks to the first 2^{10} blocks of a block group to construct the

32-bit inode number. This way user applications will be ensured of getting unique inode numbers on 32-bit platforms. For 32-bit applications running on 64-bit platforms, we hope they are fixed by the time ext4 is in production, and this only starts to be an issue for filesystems over 1TB in size.

In summary, dynamic inode allocation and 64-bit inode numbers are needed to support large numbers of files in ext4. The benefits are obvious, but the changes to the on-disk format may be intrusive. The design details are still under discussion.

2.5 Directory scalability

The maximum number of subdirectories contained in a single directory in ext3 is 32,000. To address directory scalability, this limit will be eliminated in ext4 providing unlimited sub-directory support.

In order to better support large directories with many entries, the directory indexing feature[6] will be turned on by default in ext4. By default in ext3, directory entries are still stored in a linked list, which is very inefficient for directories with large numbers of entries. The directory indexing feature addresses this scalability issue by storing directory entries in a constant depth HTree data structure, which is a specialized BTree-like structure using 32-bit hashes. The fast lookup time of the HTree significantly improves performance on large directories. For directories with more than 10,000 files, improvements were often by a factor of 50 to 100 [3].

2.5.1 Future work

While the HTree implementation allowed the ext2 directory format to be improved from linear to a tree search compatibly, there are also limitations to this approach. The HTree implementation has a limit of 510 * 511 4 KB directory leaf blocks (approximately 25M 24-byte filenames) that can be indexed with a 2-level tree. It would be possible to change the code to allow a 3-level HTree. There is also currently a 2 GB file size limit on directories, because the code for using the high 32-bits for i_size on directories was not implemented when the 2 GB limit was fixed for regular files.

Because the hashing used to find filenames in indexed directories is essentially random compared to the linear order in which inodes are allocated, we end up doing random seeks around the disk when accessing many

inodes in a large directory. We need to have readdir in hash-index order because directory entries might be moved during the split of a directory leaf block, so to satisfy POSIX requirements we can only safely walk the directory in hash order.

To address this problem, there is a proposal to put the whole inode into the directory instead of just a directory entry that references a separate inode. This avoids the need to seek to the inode when doing a readdir, because the whole inode has been read into memory already in the readdir step. If the blocks that make up the directory are efficiently allocated, then reading the directory also does not require any seeking.

This would also allow dynamic inode allocation, with the directory as the "container" of the inode table. The inode numbers would be generated in a similar manner as previously discussed (Section 2.4), so that the block that an inode resides in can be located directly from the inode number itself. Hard linked files imply that the same block is allocated to multiple directories at the same time, but this can be reconciled by the link count in the inode itself.

We also need to store one or more file names in the inode itself, and this can be done by means of an extended attribute that uses the directory inode number as the EA name. We can then return the name(s) associated with that inode for a single directory immediately when doing readdir, and skip any other name(s) for the inode that belong to hard links in another directory. For efficient name-to-inode lookup in the directory, we would still use a secondary tree similar to the current ext3 HTree (though it would need an entry per name instead of per directory block). But because the directory entries (the inodes themselves) do not get moved as the directory grows, we can just use disk block or directory offset order for readdir.

2.6 Large inode and fast extended attributes

Ext3 supports different inode sizes. The inode size can be set to any power-of-two larger than 128 bytes size up to the filesystem block size by using the *mke2fs -I [inode size]* option at format time. The default inode structure size is 128 bytes, which is already crowded with data and has little space for new fields. In ext4, the default inode structure size will be 256 bytes.
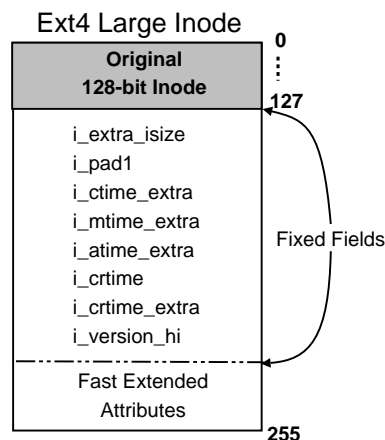
Figure 4: Layout of the large inode

In order to avoid duplicating a lot of code in the kernel and e2fsck, the large inodes keep the same fixed layout for the first 128-bytes, as shown in Figure 4. The rest of the inode is split into two parts: a fixed-field section that allows addition of fields common to all inodes, such as nanosecond timestamps (Section 5), and a section for fast extended attributes (EAs) that consumes the rest of the inode.

The fixed-field part of the inode is dynamically sized, based on what fields the current kernel knows about. The size of this area is stored in each inode in the i_extra_isize field, which is the first field beyond the original 128-byte inode. The superblock also contains two fields, s_min_extra_isize and i_want_extra_isize, which allow down-level kernels to allocate a larger i_extra_isize than it would otherwise do.

The s_min_extra_isize is the guaranteed minimum amount of fixed-field space in each inode. s_want_extra_isize is the desired amount of fixed-field space for new inode, but there is no guarantee that this much space will be available in every inode. A ROCOMPAT feature flag EXTRA_ISIZE indicates whether these superblock fields are valid. The ext4 code will soon also be able to expand i_extra_isize dynamically as needed to cover the fixed fields, so long as there is space available to store the fast EAs or migrate them to an external EA block.

The remaining large inode space may be used for storing EA data inside the inode. Since the EAs are already in memory after the inode is read from disk, this avoids a costly seek to external EA block. This can greatly

improve the performance of applications that are using EAs, sometimes by a factor of 3–7 [4]. An external EA block is still available in addition to the fast EA space, which allows storing up to 4 KB of EAs for each file.

The support for fast EAs in large inodes has been available in Linux kernels since 2.6.12, though it is rarely used because many people do not know of this capability at mke2fs time. Since ext4 will have larger inodes, this feature will be enabled by default.

There have also been discussions about breaking the 4 KB EA limit, in order to store larger or more EAs. It is likely that larger single EAs will be stored in their own inode (to allow arbitrary-sized EAs) and it may also be that for many EAs they will be stored in a directory-like structure, possibly leveraging the same code as regular ext4 directories and storing small values inline.

3   Block allocation enhancements

Increased filesystem throughput is the premier goal for all modern filesystems. In order to meet this goal, developers are constantly attempting to reduce filesystem fragmentation. High fragmentation rates cause greater disk access time affecting overall throughput, and increased metadata overhead causing less efficient mapping.

There is an array of new features in line for ext4, which take advantage of the existing extents mapping and are aimed at reducing filesystem fragmentation by improving block allocation techniques.

3.1   Persistent preallocation

Some applications, like databases and streaming media servers, benefit from the ability to preallocate blocks for a file up-front (typically extending the size of the file in the process), without having to initialize those blocks with valid data or zeros. Preallocation helps ensure contiguous allocation as far as possible for a file (irrespective of when and in what order data actually gets written) and guaranteed space allocation for writes within the preallocated size. It is useful when an application has some foreknowledge of how much space the file will require. The filesystem internally interprets the preallocated but not yet initialized portions of the file as zero-filled blocks. This avoids exposing stale data for each

block until it is explicitly initialized through a subsequent write. Preallocation must be persistent across reboots, unlike ext3 and ext4 block reservations [3].

For applications involving purely sequential writes, it is possible to distinguish between initialized and uninitialized portions of the file. This can be done by maintaining a single high water mark value representing the size of the initialized portion. However, for databases and other applications where random writes into the preallocated blocks can occur in any order, this is not sufficient. The filesystem needs to be able to identify ranges of uninitialized blocks in the middle of the file. Therefore, some extent based filesystems, like XFS, and now ext4, provide support for marking allocated but uninitialized extents associated with a given file.

Ext4 implements this by using the MSB of the extent length field to indicate whether a given extent contains uninitialized data, as shown in Figure 1. During reads, an uninitialized extent is treated just like a hole, so that the VFS returns zero-filled blocks. Upon writes, the extent must be split into initialized and uninitialized extents, merging the initialized portion with an adjacent initialized extent if contiguous.

Until now, XFS, the other Linux filesystem that implements preallocation, provided an ioctl interface to applications. With more filesystems, including ext4, now providing this feature, a common system-call interface for fallocate and an associated inode operation have been introduced. This allows filesystem-specific implementations of preallocation to be exploited by applications using the posix_fallocate API.

3.2   Delayed and multiple block allocation

The block allocator in ext3 allocates one block at a time during the write operation, which is inefficient for larger I/O. Since block allocation requests are passed through the VFS layer one at a time, the underlying ext3 filesystem cannot foresee and cluster future requests. This also increases the possibility of file fragmentation.

Delayed allocation is a well-known technique in which block allocations are postponed to page flush time, rather than during the write() operation [3]. This method provides the opportunity to combine many block allocation requests into a single request, reducing possible

fragmentation and saving CPU cycles. Delayed allocation also avoids unnecessary block allocation for short-lived files.

Ext4 delayed allocation patches have been implemented, but there is work underway to move this support to the VFS layer, so multiple filesystems can benefit from the feature.

With delayed allocation support, multiple block allocation for buffered I/O is now possible. An entire extent, containing multiple contiguous blocks, is allocated at once rather than one block at a time. This eliminates multiple calls to ext4_get_blocks and ext4_new_blocks and reduces CPU utilization.

Ext4 multiple block allocation builds per-block group free extents information based on the on-disk block bitmap. It uses this information to guide the search for free extents to satisfy an allocation request. This free extent information is generated at filesystem mount time and stored in memory using a buddy structure.

The performance benefits of delayed allocation alone are very obvious, and can be seen in Section 7. In a previous study [3], we have seen about 30% improved throughput and 50% reduction in CPU usage with the combined two features. Overall, delayed and multiple block allocation can significantly improve filesystem performance on large I/O.

There are two other features in progress that are built on top of delayed and multiple block allocation, trying to further reduce fragmentation:

- In-core Preallocation: Using the in-core free extents information, a more powerful in-core block preallocation/reservation can be built. This further improves block placement and reduces fragmentation with concurrent write workloads. An inode can have a number of preallocated chunks, indexed by the logical blocks. This improvement can help HPC applications when a number of nodes write to one huge file at very different offsets.

- Locality Groups: Currently, allocation policy decisions for individual file are made independently. If the allocator had knowledge of file relationship, it could intelligently place related files close together, greatly benefiting read performance. The locality groups feature clusters related files together by a given attribute, such as SID or a combination of SID and parent directory. At the deferred page-flush time, dirty pages are written out by groups, instead of by individual files. The number of non-allocated blocks are tracked at the group-level, and upon flush time, the allocator can try to preallocate enough space for the entire group. This space is shared by the files in the group for their individual block allocation. In this way, related files are place tightly together.

In summary, ext4 will have a powerful block allocation scheme that can efficiently handle large block I/O and reduce filesystem fragmentation with small files under multi-threaded workloads.

3.3 Online defragmentation

Though the features discussed in this section improve block allocation to avoid fragmentation in the first place, with age, the filesystem can still become quite fragmented. The ext4 online defragmentation tool, e4defrag, has been developed to address this. This tool can defragment individual files or the entire filesystem. For each file, the tool creates a temporary inode and allocates contiguous extents to the temporary inode using multiple block allocation. It then copies the original file data to the page cache and flushes the dirty pages to the temporary inode's blocks. Finally, it migrates the block pointers from the temporary inode to the original inode.

4 Reliability enhancements

Reliability is very important to ext3 and is one of the reasons for its vast popularity. In keeping with this reputation, ext4 developers are putting much effort into maintaining the reliability of the filesystem. While it is relatively easy for any filesystem designer to make their fields 64-bits in size, it is much more difficult to make such large amounts of space actually usable in the real world.

Despite the use of journaling and RAID, there are invariably corruptions to the disk filesystem. The first line of defense is detecting and avoiding problems proactively by a combination of robust metadata design, internal redundancy at various levels, and built-in integrity checking using checksums. The fallback will always be doing

integrity checking (fsck) to both detect and correct problems that will happen anyway.

One of the primary concerns with all filesystems is the speed at which a filesystem can be validated and recovered after corruption. With reasonably high-end RAID storage, a full fsck of a 2TB ext3 filesystem can take between 2 to 4 hours for a relatively "clean" filesystem. This process can degrade sharply to many days if there are large numbers of shared filesystem blocks that need expensive extra passes to correct.

Some features, like extents, have already added to the robustness of the ext4 metadata as previously described. Many more related changes are either complete, in progress, or being designed in order to ensure that ext4 will be usable at scales that will become practical in the future.

4.1 Unused inode count and fast e2fsck

In e2fsck, the checking of inodes in pass 1 is by far the most time consuming part of the operation. This requires reading all of the large inode tables from disk, scanning them for valid, invalid, or unused inodes, and then verifying and updating the block and inode allocation bitmaps. The uninitialized groups and inode table high watermark feature allows much of the lengthy pass 1 scanning to be safely skipped. This can dramatically reduce the total time taken by e2fsck by 2 to 20 times, depending on how full the filesystem is. This feature can be enabled at mke2fs time or using tune2fs via the *-O uninit_groups* option.

With this feature, the kernel stores the number of unused inodes at the end of each block group's inode table. As a result, e2fsck can skip both reading these blocks from disk, and scanning them for in-use inodes. In order to ensure that the unused inode count is safe to use by e2fsck, the group descriptor has a CRC16 checksum added to it that allows validation of all fields therein.

Since typical ext3 filesystems use only in the neighborhood of 1% to 10% of their inodes, and the inode allocation policy keeps a majority of those inodes at the start of the inode table, this can avoid processing a large majority of the inodes and speed up the pass 1 processing. The kernel does not currently increase the unused inodes count, when files are deleted. This counter is updated on every e2fsck run, so in the case where a block group had
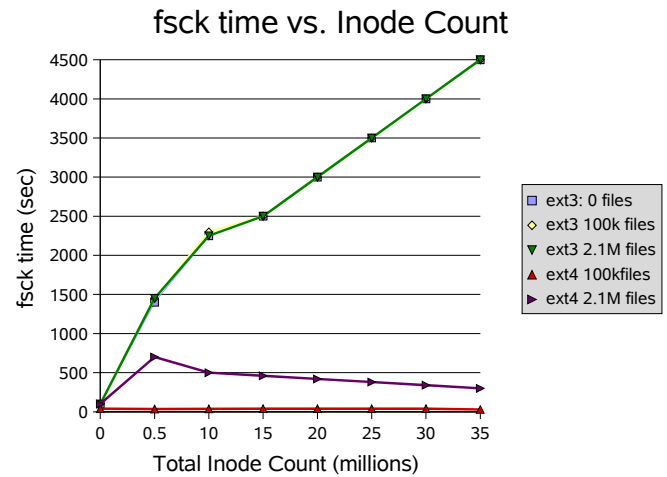
Figure 5: e2fsck performance improvement with uninitialized block groups

many inodes deleted, e2fsck will be more efficient in the next run.

Figure 5 shows that e2fsck time on ext3 grows linearly with the total number of inodes in filesystem, regardless of how many are used. On ext3, e2fsck takes the same amount of time with zero used files as with 2.1 million used files. In ext4, with the unused inode high watermark feature, the e2fsck time is only dependent on the number of used inodes. As we can see, fsck of an ext4 filesystem with 100 000 used files takes a fraction of the time ext3 takes.

In addition to the unused inodes count, it is possible for mke2fs and e2fsck to mark a group's block or inode bitmap as uninitialized, so that the kernel does not need to read them from disk when first allocating from the group. Similarly, e2fsck does not need to read these bitmaps from disk, though this does not play a major role in performance improvements. What is more significant is that mke2fs will not write out the bitmaps or inode tables at format time if the *mke2fs -O lazy_bg* feature is given. Writing out the inode tables can take a significant amount of time, and has been known to cause problems for large filesystems due to the amount of dirty pages this generates in a short time.

4.2 Checksumming

Adding metadata checksumming into ext4 will allow it to more easily detect corruption, and behave appropriately instead of blindly trusting the data it gets from

disk. The group descriptors already have a checksum added, per the previous section. The next immediate target for checksumming is the journal, because it has such a high density of important metadata and is constantly being written to, so has a higher chance of wearing out the platters or seeing other random corruption.

Adding checksumming to the ext4 journal is nearly complete [7]. In ext3 and ext4, each journal transaction has a header block and commit block. During normal journal operation the commit block is not sent to the disk until the transaction header and all metadata blocks which make up that transaction have been written to disk [8]. The next transaction needs to wait for the previous commit block to hit to disk before it can start to modify the filesystem.

With this two-phase commit, if the commit block has the same transaction number as the header block, it should indicate that the transaction can be replayed at recovery time. If they don't match, the journal recovery is ended. However, there are several scenarios where this can go wrong and lead to filesystem corruption.

With journal checksumming, the journal code computes a CRC32 over all of the blocks in the transaction (including the header), and the checksum is written to the commit block of the transaction. If the checksum does not match at journal recovery time, it indicates that one or more metadata blocks in the transaction are corrupted or were not written to disk. Then the transaction (along with later ones) is discarded as if the computer had crashed slightly earlier and not written a commit block at all.

Since the journal checksum in the commit block allows detection of blocks that were not written into the journal, as an added bonus there is no longer a need for having a two-phase commit for each transaction. The commit block can be written at the same time as the rest of the blocks in the transaction. This can actually speed up the filesystem operation noticeably (as much as 20% [7]), instead of the journal checksum being an overhead.

There are also some long-term plans to add checksumming to the extent tail, the allocation bitmaps, the inodes, and possibly also directories. This can be done efficiently once we have journal checksumming in place. Rather than computing the checksum of filesystem metadata each time it is changed (which has high overhead for often-modified structures), we can write

the metadata to the checksummed journal and still be confident that it is valid and correct at recovery time. The blocks can have metadata-specific checksums computed a single time when they are written into the filesystem.

5 Other new features

New features are continuously being added to ext4. Two features expected to be seen in ext4 are nanosecond timestamps and inode versioning. These two features provide precision when dealing with file access times and tracking changes to files.

Ext3 has second resolution timestamps, but with today's high-speed processors, this is not sufficient to record multiple changes to a file within a second. In ext4, since we use a larger inode, there is room to support nanosecond resolution timestamps. High 32-bit fields for the atime, mtime and ctime timestamps, and also a new crtime timestamp documenting file creation time, will be added to the ext4 inode (Figure 4). 30 bits are sufficient to represent the nanosecond field, and the remaining 2 bits are used to extend the epoch by 272 years.

The NFSv4 clients need the ability to detect updates to a file made at the server end, in order to keep the client side cache up to date. Even with nanosecond support for ctime, the timestamp is not necessarily updated at the nanosecond level. The ext4 inode versioning feature addresses this issue by providing a global 64-bit counter in each inode. This counter is incremented whenever the file is changed. By comparing values of the counter, one can see whether the file has been updated. The counter is reset on file creation, and overflows are unimportant, because only equality is being tested. The i_version field already present in the 128-bit inode is used for the low 32 bits, and a high 32-bit field is added to the large ext4 inode.

6 Migration tool

Ext3 developers worked to maintain backwards compatibility between ext2 and ext3, a characteristic users appreciate and depend on. While ext4 attempts to retain compatibility with ext3 as much as possible, some of the incompatible on-disk layout changes are unavoidable. Even with these changes, users can still easily upgrade their ext3 filesystem to ext4, like it is possible

from ext2 to ex3. There are methods available for users to try new ext4 features immediately, or migrate their entire filesystem to ext4 without requiring back-up and restore.

6.1 Upgrading from ext3 to ext4

There is a simple upgrade solution for ext3 users to start using extents and some ext4 features without requiring a full backup or migration. By mounting an existing ext3 filesystem as ext4 (with extents enabled), any new files are created using extents, while old files are still indirect block mapped and interpreted as such. A flag in the inode differentiates between the two formats, allowing both to coexist in one ext4 filesystem. All new ext4 features based on extents, such as preallocation and multiple block allocation, are available to the new extents files immediately.

A tool will also be available to perform a system-wide filesystem migration from ext3 to ext4. This migration tool performs two functions: migrating from indirect to extents mapping, and enlarging the inode to 256 bytes.

- Extents migration: The first step can be performed online and uses the defragmentation tool. During the defragmentation process, files are changed to extents mapping. In this way, the files are being converted to extents and defragmented at the same time.

- Inode migration: Enlarging the inode structure size must be done offline. In this case, data is backed up, and the entire filesystem is scanned and converted to extents mapping and large inodes.

For users who are not yet ready to move to ext4, but may want to in the future, it is possible to prepare their ext3 filesystem to avoid offline migration later. If an ext3 filesystem is formatted with a larger inode structure, 256 bytes or more, the fast extended attribute feature (Section 2.6) which is the default in ext4, can be used instantly. When the user later wants to upgrade to ext4, then other ext4 features using the larger inode size, such as nanosecond timestamps, can also be used without requiring any offline migration.

6.2 Downgrading from ext4 to ext3

Though not as straightforward as ext3 to ext4, there is a path for any user who may want to downgrade from ext4 back to ext3. In this case the user would remount the filesystem with the *noextents* mount option, copy all files to temporary files and rename those files over the original file. After all files have been converted back to indirect block mapping format, the INCOMPAT_EXTENTS flag must be cleared using tune2fs, and the filesystem can be re-mounted as ext3.

7 Performance evaluation

We have conducted a performance evaluation of ext4, as compared to ext3 and XFS, on three well-known filesystem benchmarks. Ext4 was tested with extents and delayed allocation enabled. The benchmarks in this analysis were chosen to show the impact of new changes in ext4. The three benchmarks chosen were: Flexible Filesystem Benchmark (FFSB) [1], Postmark [5], and IOzone [2]. FFSB, configured with a large file workload, was used to test the extents feature in ext4. Postmark was chosen to see performance of ext4 on small file workloads. Finally, we used IOzone to evaluate overall ext4 filesystem performance.

The tests were all run on the 2.6.21-rc4 kernel with delayed allocation patches. For ext3 and ext4 tests, the filesystem was mounted in writeback mode, and appropriate extents and delayed allocation mount options were set for ext4. Default mount options were used for XFS testing.

FFSB and IOzone benchmarks were run on the same 4-CPU 2.8 Ghz Intel(R) Xeon(tm) System with 2 GB of RAM, on a 68GB ultra320 SCSI disk (10000 rpm). Postmark was run on a 4-CPU 700 MHz Pentium(R) III system with 4 GB of RAM on a 9 GB SCSI disk (7200 rpm). Full test results including raw data are available at the ext4 wiki page, `http://ext4.wiki.kernel.org`.

7.1 FFSB comparison

FFSB is a powerful filesystem benchmarking tool, that can be tuned to simulate very specific workloads. We have tested multithreaded creation of large files. The test
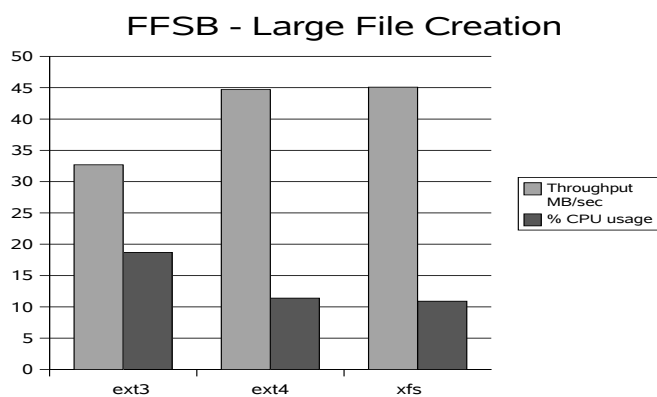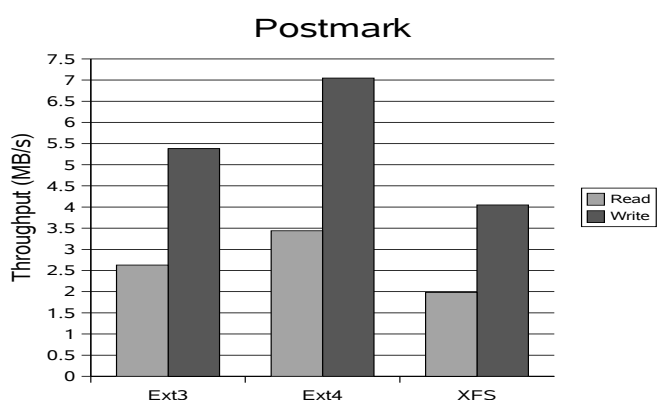
Figure 6: FFSB sequential write comparison

Figure 7: Postmark read write comparison

runs 4 threads, which combined create 24 1-GB files, and stress the sequential write operation.

The results, shown in Figure 6, indicate about 35% improvement in throughput and 40% decrease in CPU utilization in ext4 as compared to ext3. This performance improvement shows a diminishing gap between ext4 and XFS on sequential writes. As expected, the results verify extents and delayed allocation improve performance on large contiguous file creation.

7.2 Postmark comparison

Postmark is a well-known benchmark simulating a mail server performing many single-threaded transactions on small to medium files. The graph in Figure 7 shows about 30% throughput gain with with ext4. Similar percent improvements in CPU utilization are seen, because metadata is much more compact with extents. The write throughput is higher than read throughput because everything is being written to memory.
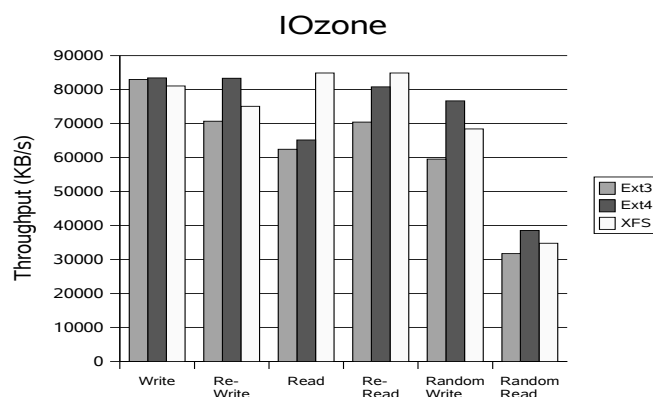
Figure 8: IOzone results: throughput of transactions on 512 MB files

These results show that, aside from the obvious performance gain on large contiguous files, ext4 is also a good choice on smaller file workloads.

7.3 IOzone comparison

For the IOzone benchmark testing, the system was booted with only 64 M of memory to really stress disk I/O. The tests were performed with 8 MB record sizes on various file sizes. Write, rewrite, read, reread, random write, and random read operations were tested. Figure 8 shows throughput results for 512 MB sized files. Overall, there is great improvement between ext3 and ext4, especially on rewrite, random-write and reread operations. In this test, XFS still has better read performance, while ext4 has shown higher throughput on write operations.

8 Conclusion

As we have discussed, the new ext4 filesystem brings many new features and enhancements to ext3, making it a good choice for a variety of workloads. A tremendous amount of work has gone into bringing ext4 to Linux, with a busy roadmap ahead to finalize ext4 for production use. What was once essentially a simple filesystem has become an enterprise-ready solution, with a good balance of scalability, reliability, performance and stability. Soon, the ext3 user community will have the option to upgrade their filesystem and take advantage of the newest generation of the ext family.

Acknowledgements

Legal Statement

References

[1] Ffsb project on sourceforge. Technical report. http://sourceforge.net/projects/ffsb.

[2] Iozone. Technical report. http://www.iozone.org.

[3] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Ottawa Linux Symposium*, 2005.

[4] Jonathan Corbet. Which filesystem for samba4? Technical report. http://lwn.net/Articles/112566/.

[5] Jeffrey Katcher. Postmark a new filesystem benchmark. Technical report, Network Appliances, 2002.

[6] Daniel Phillips. A directory index for ext2. In *5th Annual Linux Showcase and Conference*, pages 173–182, 2001.

[7] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci Dusseau. Iron file systems. In *SOSP'05*, pages 206–220, 2005.

[8] Stephen Tweedie. Ext3 journalling filesystem. In *Ottawa Linux Symposium*, 2000.

[9] Stephen Tweedie and Theodore Y Ts'o. Planned extensions to the linux ext2/3 filesystem. In *USENIX Annual Technical Conference*, pages 235–244, 2002.

The 7 dwarves: debugging information beyond gdb

Arnaldo Carvalho de Melo
Red Hat, Inc.
acme@redhat.com
acme@ghostprotocols.net

Abstract

The DWARF debugging information format has been so far used in debuggers such as gdb, and more recently in tools such as systemtap and frysk.

In this paper the author will show additional scenarios where such information can be useful, such as:

- Showing the layout of data structures;

- Reorganizing such data structures to remove alignment holes;

- Improving CPU cache utilization;

- Displaying statistics about inlining of functions;

- Re-creating structs and functions from the debugging information;

- Showing binary diffs to help understand the effects of any code change.

And much more.

1 Introduction

This paper talks about new ways to use the DWARF debugging information inserted into binaries by compilers such as gcc.

The author developed several tools that allow:

- Extracting useful information about data structures layout;

- Finding holes and padding inserted by the compiler to follow alignment constraints in processor architectures;

- To find out possibilities for reduction of such data structures;

- Use of information about function parameters and return types to generate Linux kernel modules for obtaining data needed for generation of callgraphs and values set to fields at runtime;

- A tool that given two object files shows a binary diff to help understanding the effects of source code changes on size of functions and data structures.

Some use cases will be presented, showing how the tools can be used to solve real world problems.

Ideas discussed with some fellow developers but not yet tried will also be presented, with the intent of *hopefully* having them finally tested in practice by interested readers.

2 DWARF Debugging Format

DWARF [3] is a debugging file format used by many compilers to store information about data structures, variables, functions and other language aspects needed by high level debuggers.

It has had three major revisions, with the second being incompatible with the first, the third is an expansion of the second, adding support for more C++ concepts, providing ways to eliminate data duplication, support for debugging data in shared libraries and in files larger than 4 GB.

The DWARF debugging information is organized in several ELF sections in object files, some of which will be mentioned here. Please refer to the DWARF [3] specification for a complete list. Recent developments in tools such as elfutils [2] allow for separate files with the

debugging information, but the common case is for the information to be packaged together with the respective object file.

Debugging data is organized in tags with attributes. Tags can be nested to represent, for instance, variables inside lexical blocks, parameters for a function and other hierarchical concepts.

As an example let us look at how the ubiquitous hello world example is represented in the ".debug_info" section, the one section that is most relevant to the subject of this paper:

```
$ cat hello.c
int main(void)
{
        printf("hello, world!\n");
}
```

Using gcc with the -g flag to insert the debugging information:

```
$ gcc −g hello.c −o hello
```

Now let us see the output, slightly edited for brevity, from eu-readelf, a tool present in the elfutils package:

```
$ eu−readelf −winfo hello
DWARF section ,.debug_info, at offset
0x6b3:
[Offset]
Compilation unit at offset 0:
Version: 2, Abbrev section offset: 0,
Addr size: 4, Offset size: 4
[b] compile_unit
     stmt_list   0
     high_pc     0x0804837a
     low_pc      0x08048354
     producer    "GNU C 4.1.1"
     language    ISO C89 (1)
     name        "hello.c"
     comp_dir    "~/examples"
[68] subprogram
      external
      name        "main"
      decl_file   1
      decl_line   2
      prototyped
      type        [82]
      low_pc      0x08048354
      high_pc     0x0804837a
      frame_base  location list [0]
```

```
[82] base_type
     name        "int"
     byte_size   4
     encoding    signed (5)
```

Entries starting with [*number*] are the DWARF tags that are represented in the tool's source code as `DW_TAG_tag_name`. In the above output we can see some: `DW_TAG_compile_unit`, with information about the object file being analyzed, `DW_TAG_subprogram`, emitted for each function, such as "main" and `DW_TAG_base_type`, emitted for the language basic types, such as `int`.

Each tag has a number of attributes, represented in source code as `DW_AT_attribute_name`. In the `DW_TAG_subprogram` for the "main" function we have some: `DW_AT_name` ("main"), `DW_AT_decl_file`, that is an index into another DWARF section with the names for the source code files, `DW_AT_decl_line`, the line in the source code where this function was defined, `DW_AT_type`, the return type for the "main" routine. Its value is a tag index, that in this case refers to the [82] tag, which is the `DW_TAG_base_type` for "int," and also the address of this function, `DW_AT_low_pc`.

The following example exposes some additional DWARF tags and attributes used in the seven dwarves. The following struct:

```
struct swiss_cheese {
  char a;
  int  b;
};
```

is represented as:

```
[68] structure_type
     name           "swiss_cheese"
     byte_size      8
[7d]  member
      name          "a"
      type          [96]
      data_member_location 0
[89]  member
      name          "b"
      type          [9e]
      data_member_location 4
[96] base_type
      name          "char"
      byte_size     1
```

```
[9e] base_type
      name          "int"
      byte_size     4
```

In addition to the tags already described we have now `DW_TAG_structure_type` to start the representation of a struct, that has the `DW_AT_byte_size` attribute stating how many bytes the struct takes (8 bytes in this case). There is also another tag, `DW_TAG_member`, that represents each struct member. It has the `DW_AT_byte_size` and the `DW_AT_data_member_location` attribute, the offset of this member in the struct. There are more attributes, but for brevity and for the purposes of this paper, the above are enough to describe.

3 The 7 dwarves

The seven dwarves are tools that use the DWARF debugging information to examine data struct layout (`pahole`), examine executable code characteristics (`pfunct`), compare executables (`codiff`), trace execution of functions associated with a struct (`ctracer`), pretty-print DWARF information (`pdwtags`), list global symbols (`pglobal`), and count the number of times each set of tags is used (`prefcnt`).

Some are very simple and still require work, while others, such as `pahole` and `pfunct`, are already being helpful in open source projects such as the Linux kernel, xine-lib and perfmon2. One possible use is to pretty-print DWARF information accidentally left in binary-only kernel modules released publicly.

All of these tools use a library called `libdwarves`, that is packaged with the tools and uses the DWARF libraries found in elfutils [2]. By using elfutils, many of its features such as relocation, reading of object files for many architectures, use of separate files with debugging information, etc, were leveraged, allowing the author to concentrate on the features that will be presented in the next sections.

Unless otherwise stated, the examples in the following sections use a Linux kernel image built for the x86-64 architecture from recent source code.[1]

The Linux kernel configuration option `CONFIG_DEBUG_INFO` has to be selected to instruct the compiler

---

[1]circa 2.6.21-rc5.

to insert the DWARF information. This will make the image much bigger, but poses no performance impact on the resulting binary, just like when building user space programs with debug information.

3.1 pahole

Poke-a-hole, the first dwarf, is used to find alignment holes in structs. It is the most advanced of all the tools in this project so far.

Architectures have alignment constraints, requiring data types to be aligned in memory in multiples of their word-size. While compilers do automatically align data structures, careful planning by the developer is essential to minimize the paddings ("holes") required for correct alignment of the data structures members.

An example of a bad struct layout is in demand to better illustrate this situation:

```
struct cheese {
  char  name[17];
  short age;
  char  type;
  int   calories;
  short price;
  int   barcode[4];
};
```

Adding up the sizes of the members one could expect that the size of struct cheese to be 17 + 2 + 1 + 4 + 2 + 16 = 42 bytes. But due to alignment constraints the real size ends up being 48 bytes.

Using `pahole` to pretty-print the DWARF tags will show where the 6 extra bytes are:

```
/* <11b> ~/examples/swiss_cheese.c:3 */
struct cheese {
  char  name[17];   /*  0 17 */

  /* XXX 1 byte hole, try to pack */

  short age;        /* 18  2 */
  char  type;       /* 20  1 */

  /* XXX 3 bytes hole, try to pack */

  int   calories;   /* 24  4 */
  short price;      /* 28  2 */
```

```
    /* XXX 2 bytes hole, try to pack */

  int   barcode[4]; /* 32 16 */
}; /* size: 48, cachelines: 1 */
    /* sum members: 42, holes: 3 */
    /* sum holes: 6 */
    /* last cacheline: 48 bytes */
```

This shows that in this architecture the alignment rule state that `short` has to be aligned at a multiple of 2 offset from the start of the struct, and `int` has to be aligned at a multiple of 4, the size of the word-size on the example architecture.

Another alignment rule aspect is that a perfectly arranged struct on a 32-bit architecture such as:

```
$ pahole long
/* <67> ~/examples/long.c:1 */
struct foo {
  int   a;     /*  0 4 */
  void *b;     /*  4 4 */
  char c[4];   /*  8 4 */
  long g;      /* 12 4 */
}; /* size: 16, cachelines: 1 */
    /* last cacheline: 16 bytes */
```

has holes when built on an architecture with a different word-size:

```
$ pahole long
/* <6f> ~/examples/long.c:1 */
struct foo {
  int    a;    /*  0 4 */

    /* XXX 4 bytes hole, try to pack */

  void *b;     /*  8 8 */
  char c[4];   /* 16 4 */

    /* XXX 4 bytes hole, try to pack */

  long g;      /* 24 8 */
}; /* size: 32, cachelines: 1 */
    /* sum members: 24, holes: 2 */
    /* sum holes: 8 */
    /* last cacheline: 32 bytes */
```

This is because on x86-64 the size of pointers and long integers is 8 bytes, with the alignment rules requiring these basic types to be aligned at multiples of 8 bytes from the start of the struct.

To help in these cases, `pahole` provides the `--reorganize` option, where it will reorganize the struct trying to achieve optimum placement regarding memory consumption, while following the alignment rules.

Running it on the x86-64 platform we get:

```
$ pahole --reorganize -C foo long
struct foo {
  int  a;      /*  0 4 */
  char c[4];   /*  4 4 */
  void *b;     /*  8 8 */
  long g;      /* 16 8 */
}; /* size: 24, cachelines: 1 */
    /* last cacheline: 24 bytes */
    /* saved 8 bytes! */
```

There is another option, `--show_reorg_steps` that sheds light on what was done:

```
$ pahole --show_reorg_steps
--reorganize -C foo long
/* Moving 'c' from after 'b' to after 'a' */
struct foo {
  int  a;      /*  0 4 */
  char c[4];   /*  4 4 */
  void *b;     /*  8 8 */
  long g;      /* 16 8 */
}; /* size: 24, cachelines: 1 */
    /* last cacheline: 24 bytes */
```

While in this case there was just one step done, using this option in more complex structs can involve many steps, that would have been shown to help understanding the changes performed. Other steps in the `--reorganize` algorithm includes:

- Combining separate bit fields

- Demoting bit fields to a smaller basic type when the type being used has more bits than required by the members in the bit field (e.g. `int a:1, b:2;` being demoted to `char a:1, b:2;`)

- Moving members from the end of the struct to fill holes

- Combining the padding at the end of a struct with a hole

Several modes to summarize information about all the structs in object files were also implemented. They will be presented in the following examples.

The top ten structs by size are:

```
$ pahole −−sizes vmlinux | sort −k2 −nr
| head
hid_parser: 65784 0
hid_local: 65552 0
kernel_stat: 33728 0
module: 16960 8
proto: 16640 2
pglist_data: 14272 2
avc_cache: 10256 0
inflate_state: 9544 0
ext2_sb_info: 8448 2
tss_struct: 8320 0
```

The second number represents the number of alignment holes in the structs.

Yes, some are quite big and even the author got impressed with the size of the first few ones, which is one of the common ways of using this tool to find areas that could get some help in reducing data structure sizes. So the next step would be to pretty-print this specific struct, `hid_local`:

```
$ pahole −C hid_local vmlinux
/* <175c261>
       ~/net-2.6.22/include/linux/hid.h:300 */
struct hid_local {
  uint usage[8192];        // 0 32768
  // cacheline 512 boundary (32768 bytes)
  uint cindex[8192];       // 32768 32768
  // cacheline 1024 boundary (65536 bytes)
  uint usage_index;        // 65536 4
  uint usage_minimum;      // 65540 4
  uint delimiter_depth;    // 65544 4
  uint delimiter_branch;   // 65548 4
}; /* size: 65552, cachelines: 1025 */
   /* last cacheline: 16 bytes */
```

So, this is indeed something to be investigated, not a bug in `pahole`.

As mentioned, the second column is the number of alignment holes. Sorting by this column provides another picture of the project being analyzed that could help finding areas for further work:

```
$ pahole −−sizes vmlinux | sort −k3 −nr
| head
net_device: 1664 14
vc_data: 432 11
tty_struct: 1312 10
task_struct: 1856 10
request_queue: 1496 8
module: 16960 8
mddev_s: 672 8
usbhid_device: 6400 6
device: 680 6
zone: 2752 5
```

There are lots of opportunities to use `−−reorganize` results, but in some cases this is not true because the holes are due to member alignment constraints specified by the programmers.

Alignment hints are needed, for example, when a set of fields in a structure are "read mostly," while others are regularly written to. So, to make it more likely that the "read mostly" cachelines are not invalidated by writes in SMP machines, attributes are used on the struct members instructing the compiler to align some members at cacheline boundaries.

Here is one example, in the Linux kernel, of an alignment hint on the struct `net_device`, that appeared on the above output:

```
/*
 * Cache line mostly used on receive
 * path (including eth_type_trans())
 */
  struct list_head poll_list
____cacheline_aligned_in_smp;
```

If we look at the excerpt in the `pahole` output for this struct where `poll_list` is located we will see one of the holes:

```
  /* cacheline 4 boundary (256 bytes) */
  void *dn_ptr;       /* 256   8 */
  void *ip6_ptr;      /* 264   8 */
  void *ec_ptr;       /* 272   8 */
  void *ax25_ptr;     /* 280   8 */

  /* XXX 32 bytes hole, try to pack */

  /* cacheline 5 boundary (320 bytes) */
  struct list_head poll_list;
      /* 320 16 */
```

These kinds of annotations are not represented in the DWARF information, so the current `--reorganize` algorithm can not be precise. One idea is to use the DWARF tags with the file and line location of each member to parse the source code looking for alignment annotation patterns, but this has not been tried.

Having stated the previous possible inaccuracies in the `--reorganize` algorithm, it is still interesting to use it in all the structs in an object file to present a list of structs where the algorithm was successful in finding a new layout that saves bytes.

Using the above kernel image the author found 165 structs where holes can be combined to save some bytes. The biggest savings found are:

```
$ pahole --packable vmlinux | sort -k4
-nr | head
vc_data          432    176 256
net_device      1664   1448 216
module         16960  16848 112
hh_cache         192     80 112
zone            2752   2672  80
softnet_data    1792   1728  64
rcu_ctrlblk      128     64  64
inet_hashinfo    384    320  64
entropy_store    128     64  64
task_struct     1856   1800  56
```

The columns are: struct name, current size, reorganized size and bytes saved. In the above list of structs only a few clearly, from source code inspection, do not have any explicit alignment constraint. Further analysis is required to verify if the explicit constraints are still needed after the evolution of the subsystems that use such structs, if the holes are really needed to isolate groups of members or could be reused.

The `--expand` option is useful in analyzing crash dumps, where the available clue was an offset from a complex struct, requiring tedious manual calculation to find out exactly what was the field involved. It works by "unfolding" structs, as will be shown in the following example.

In a program with the following structs:

```
struct spinlock {
  int magic;
  int counter;
};
```

```
struct sock {
  int protocol;
  struct spinlock lock;
};
```

```
struct inet_sock {
  struct sock sk;
  long        daddr;
};
```

```
struct tcp_sock {
  struct inet_sock inet;
  long cwnd;
  long ssthresh;
};
```

the `--expand` option, applied to the `tcp_sock` struct, produces:

```
struct tcp_sock {
  struct inet_sock {
    struct sock {
      int   protocol; /*  0 4 */
      struct spinlock {
        int magic;    /*  4 4 */
        int counter;  /*  8 4 */
      } lock;         /*  4 8 */
    } sk;             /*  0 12 */
    long    daddr;    /* 12 4 */
  } inet;             /*  0 16 */
  long    cwnd;       /* 16 4 */
  long    ssthresh;   /* 20 4 */
}; /* size: 24 */
```

The offsets are relative to the start of the top level struct (`tcp_sock` in the above example).

3.2 pfunct

While `pahole` specializes on data structures, `pfunct` concentrates on aspects of functions, such as:

- number of goto labels

- function name length

- number of parameters

- size of functions

- number of variables

- size of inline expansions

It also has filters to show functions that meet several criteria, including:

- functions that have as parameters pointers to a struct

- external functions

- declared inline, un-inlined by compiler

- not declared inline, inlined by compiler

Also, a set of statistics is available, such as the number of times an inline function was expanded and the sum of these expansions, to help finding candidates for un-inlining, thus reducing the size of the binary.

The top ten functions by size:

```
$ pfunct --sizes vmlinux | sort -k2 -nr
| head
hidinput_connect: 9910
load_elf32_binary: 6793
load_elf_binary: 6489
tcp_ack: 6081
sys_init_module: 6075
do_con_write: 5972
zlib_inflate: 5852
vt_ioctl: 5587
copy_process: 5169
usbdev_ioctl: 4934
```

One of the attributes of the `DW_AT_subprogram` DWARF tag, that represents functions, is `DW_AT_inline`, which can have one of the following values:

- `DW_INL_not_inlined` – **Neither declared inline nor inlined by the compiler**

- `DW_INL_inlined` – **Not declared inline but inlined by the compiler**

- `DW_INL_declared_not_inlined` – **Declared inline but not inlined by the compiler**

- `DW_INL_declared_inlined` – **Declared inline and inlined by the compiler**

The `--cc_inlined` and `--cc_uninlined` options in `pfunct` use this information. Here are some examples of functions that were not explicitly marked as inline by the programmers but were inlined by gcc:

```
$ pfunct --cc_inlined vmlinux | tail
do_initcalls
do_basic_setup
smp_init
do_pre_smp_initcalls
check_bugs
setup_command_line
boot_cpu_init
obsolete_checksetup
copy_bootdata
clear_bss
```

For completeness, the number of inlined functions was 2526.

3.3 codiff

An object file diff tool, `codiff`, takes two versions of a binary, loads from both files the debugging information, compares them and shows the differences in structs and functions, producing output similar to the well known `diff` tool.

Consider a program that has a `print_tag` function, handling the following struct:

```
struct tag {
  int  type;
  int  decl_file;
  char *decl_line;
};
```

and in a newer version the struct was changed to this new layout, while the `print_tag` function remained unchanged:

```
struct tag {
  char type;
  int  decl_file;
  char *decl_line;
  int  refcnt;
};
```

The output produced by `codiff` would be:

```
$ codiff tag-v1 tag-v2
tag.c:
```

```
  struct tag |   +4
 1 struct changed
  print_tag |   +4
 1 function changed, 4 bytes added
```

It is similar to the `diff` tool, showing how many bytes were added to the modified struct and the effect of this change in a routine that handles instances of this struct.

The `--verbose` option tells us the details:

```
$ codiff -V tag-v1 tag-v2
tag.c:
  struct tag |   +4
   nr_members: +1
    +int refcnt   /* 12 4 */
    type
     from: int    /*  0 4 */
     to:   char   /*  0 1 */
 1 struct changed
  print_tag |   +4 # 29 → 33
 1 function changed, 4 bytes added
```

The extra information on modified structs includes:

- Number of members added and/or removed

- List of new and/or removed members

- Offset of members from start of struct

- Type and size of members

- Members that had their type changed

And for functions:

- Size difference

- Previous size -> New size

- Names of new and/or removed functions

3.4 ctracer

A class tracer, `ctracer` is an experiment in creating valid source code from the DWARF information.

For `ctracer` a method is any function that receives as one of its parameters a pointer to a specified struct. It looks for all such methods and generates kprobes entry and exit functions. At these probe points it collects information about the data structure internal state, saving the values in its members in that point in time, and records it in a relay buffer. The data is later collected in userspace and post-processed, generating html + CSS callgraphs.

One of the techniques used in `ctracer` involves creating subsets of data structures based on some criteria, such as member name or type. This tool so far just filters out any non-integer type members and applies the `--reorganize` code[2] on the resulting mini struct to possibly reduce the memory space needed for relaying this information to userspace.

One idea that probably will be pursued is to generate SystemTap [1] scripts instead of C language source files using kprobes, taking advantage of the infrastructure and safety guards in place in SystemTap.

3.5 pdwtags

A simple tool, `pdwtags` is used to pretty-print DWARF tags for data structures (struct, union), enumerations and functions, in a object file. It is useful as an example of how to use libdwarves.

Here is an example on the hello world program:

```
$ pdwtags hello
/* <68> /home/acme/examples/hello.c:2 */
int main(void)
{
}
```

This shows the "main" `DW_TAG_subprogram` tag, with its return type.

In the previous sections other examples of DWARF tag formatting were presented, and also tags for variables, function parameters, goto labels, would also appear if `pdwtags` was used on the same object file.

---

[2]Discussed in the `pahole` section.

3.6 pglobal

`pglobal` is an experimentation to print global variables and functions, written by a contributor, Davi Arnault.

This example:

```
$ cat global.c
int variable = 2;

int main(void)
{
  printf("variable=%d\n",
    variable);
}
```

would present this output with `pglobal`:

```
$ pglobal —ve hello
/* <89> /home/acme/examples/global.c:1 */
int variable;
```

which shows a list of global variables with their types, source code file, and line where they were defined.

3.7 prefcnt

`prefcnt` is an attempt to do reference counting on tags, trying to find some that are not referenced anywhere and could be removed from the source files.

4 Availability

The tools are maintained in a git repository that can be browsed at `http://git.kernel.org/?p=linux/kernel/git/acme/pahole.git`, and rpm packages for several architectures are available at `http://oops.ghostprotocols.net:81/acme/dwarves/rpm/`.

Acknowledgments

The author would like to thank Davi Arnault for pglobal, proving that libdwarves was not so horrible for a tool writer; all the people who contributed patches, suggestions, and encouragement on writing these tools; and Ademar Reis, Aristeu Rozanski, Claudio Matsuoka, Glauber Costa, Eduardo Habkost, Eugene Teo, Leonardo Chiquitto, Randy Dunlap, Thiago Santos, and William Cohen for reviewing and suggesting improvements for several drafts of this paper.

References

[1] Systemtap.
    `http://sourceware.org/systemtap`.

[2] Ulrich Drepper. elfutils home page.
    `http://people.redhat.com/drepper`.

[3] DWARF Debugging Information Format Workgroup. Dwarf debugging information format, December 2005. `http://dwarfstd.org`.

Adding Generic Process Containers to the Linux Kernel

Paul B. Menage[*]
Google, Inc.
menage@google.com

Abstract

While Linux provides copious monitoring and control options for individual processes, it has less support for applying the same operations efficiently to related groups of processes. This has led to multiple proposals for subtly different mechanisms for process aggregation for resource control and isolation. Even though some of these efforts could conceptually operate well together, merging each of them in their current states would lead to duplication in core kernel data structures/routines.

The Containers framework, based on the existing cpusets mechanism, provides the generic process grouping features required by the various different resource controllers and other process-affecting subsystems. The result is to reduce the code (and kernel impact) required for such subsystems, and provide a common interface with greater scope for co-operation.

This paper looks at the challenges in meeting the needs of all the stakeholders, which include low overhead, feature richness, completeness and flexible groupings. We demonstrate how to extend containers by writing resource control and monitoring components, we also look at how to implement namespaces and cpusets on top of the framework.

1 Introduction

Over the course of Linux history, there have been and continue to be multiple efforts to provide forms of modified behaviour across sets of processes. These efforts have tended to fall into two main camps, resource control/monitoring, and namespace isolation (although some projects contain elements of both).

Technically resource control and isolation are related; both prevent a process from having unrestricted access to even the standard abstraction of resources provided by the Unix kernel. For the purposes of this paper, we use the following defintions:

Resource control is any mechanism which can do either or both of:

- tracking *how much* of a resource is being consumed by a set of processes

- imposing quantative limits on that consumption, either absolutely, or just in times of contention.

Typically resource control is visible to the processes being controlled.

Namespace isolation[1] is a mechanism which adds an additional indirection or translation layer to the naming/visibility of some unix resource space (such as process ids, or network interfaces) for a specific set of processes. Typically the existence of isolation itself is invisible to the processes being isolated.

Resource Control and Isolation are both subsets of the general model of a subsystem which can apply behaviour differently depending on a process' membership of some specific group. Other examples could include:

- basic job control. A job scheduling system needs to be able to keep track of which processes are part of a given running "job," in the presence of `fork()` and `exit()` calls from processes in that job. This is the simplest example of the kind of

---

With additional contributions by Balbir Singh and Srivatsa Vaddagiri, IBM Linux Technology Center, {balbir,vatsa}@in.ibm.com

[1]We avoid using the alternative term *virtualization* in this paper to make clear the distinction between lightweight in-kernel virtualization/isolation, and the much more heavyweight virtual machine hypervisors such as Xen which run between the kernel and the hardware.

process tracking system proposed in this paper, as the kernel need do nothing more than maintain the membership list of processes in the job.

- tracking memory pressure for a group of processes, and being able to receive notifications when memory pressure reaches some particular level (e.g. as measured by the scanning level reached in `try_to_free_pages()`), or reaches the OOM stage.

Different projects have proposed various basic mechanisms for implementing such process tracking. The drawbacks of having multiple such mechanisms include:

- different and mutually incompatible user-space and kernel APIs and feature sets.

- The merits of the underlying process grouping mechanisms and the merits of the actual resource control/isolation system become intertwined.

- systems that could work together to provide synergistic control of different resources to the same sets of processes are unable to do so easily since they're based on different frameworks.

- kernel structures and code get bloated with additional framework pointers and hooks.

- writers of such systems have to duplicate large amounts of functionally-similar code.

The aim of the work described in this paper is to provide a generalized process grouping mechanism suitable for use as a base for current and future Linux process-control mechanisms such as resource controllers; the intention is that writers of such controllers need not be concerned with the details of how processes are being tracked and partitioned, and can concentrate on the particular resource controller at hand, and the resource abstraction presented to the user. (Specifically, this framework *does not* attempt to prescribe any particular resource abstraction.) The requirements for a process-tracking framework to meet the needs of existing and future process-management mechanisms are enumerated, and a proposal is made that aims to satisfy these requirements.

For the purposes of this paper, we refer to a tracked group of processes as a *container*. Whether this is a suitable final name for the concept is currently the subject of debate on various Linux mailing lists, due to its use by other development groups for a similar concept in userspace. Alternative suggestions have included *partition* and *process set*.

2 Requirements

In this section we attempt to enumerate the properties required of a generic container framework. Not all mechanisms that depend on containers will require all of these properties.

2.1 Multiple Independent Subsystems

Clients of the container framework will typically be resource accounting/control systems and isolation systems. In this paper we refer to the generic client as a *subsystem*. The relationship between the container framework and a subsystem is similar to that between the Linux VFS and a specific filesystem—the framework handles many of the common operations, and passes notifications/requests to the subsystem.

Different users are likely to want to make use of different subsystems; therefore it should be possible to selectively enable different subsystems both at compile time and at runtime. The main function of the container framework is to allow a subsystem to associate some kind of policy/stats (referred to as the *subsystem state*) with a group of processes, without the subsystem having to worry in too much detail about how this is actually accomplished.

2.2 Mobility

It should be possible for an appropriately-privileged user to move a process between containers. Some subsystems may require that processes can only move into a container at the point when it is created (e.g. a virtual server system where the process becomes the new *init* process for the container); therefore it should be possible for mobility to be configurable on a per-subsystem basis.

2.3 Inescapability

Once a process has been assigned to a container, it shouldn't be possible for the process (or any of its children) to move to a different container without action by a privileged (i.e. root) user, or by a user to whom that capability has been delegated, e.g. via filesystem permissions.

2.4 Extensible User Interface

Different subsystems will need to present different configuration and reporting interfaces to userspace:

- a memory resource controller might want to allow the user to specify guarantees and limits on the number of pages that processes in a container can use, and report how many pages are actually in use.

- a memory-pressure tracker might want to allow the user to specify the particular level of memory pressure which should cause user notifications.

- the *cpusets* system needs to allow users to specify various parameters such as the bitmasks of CPUs and memory nodes to which processes in the container have access.

From the user's point of view it is simplest if there is at least some level of commonality between the configuration of different subsystems. Therefore the container framework should provide an interface that captures the common aspects of different subsystems but which still allows subsystems sufficient flexibility. Possible candidates include:

- A filesystem interface, where each subsystem can register files that the user can write (for configuration) and/or read (for reporting) has the advantages that it can be manipulated by many standard unix tools and library routines, has built-in support for permission delegation, and can provide arbitrary input/output formats and behaviour.

- An API (possibly a new system call?) that allows the user to read/write named properties would have the advantages that it would tend to present a more uniform interface (although possibly too restrictive for some subsystems) and potentially have slightly better performance than a filesystem-based interface.

2.5 Nesting

For some subsystems it is desirable for there to be multiple nested levels of containers:

- The existing cpusets system inherently divides and subdivides sets of memory nodes and CPUs between different groups of processes on the system.

- Nesting allows some fraction of the resources available to one set of processes to be delegated to a subset of those processes.

- Nested virtual servers may also find hierarchical support useful.

Some other subsystems will either be oblivious to the concept of nesting, or will actively want to avoid it; therefore the container system should allow subsystems to selectively control whether they allow nesting.

2.6 Multiple Partitions

The container framework will allow the user to partition the set of processes. Consider a system that is configured with the cpusets and beancounters subsystems. At any one time, a process will be in one and exactly one cpuset (a cpuset A may also be a child of some other cpuset B, in which case in a sense all processes in A are indirectly members of cpuset B as well, but a process is only a direct member of one cpuset). Similarly a process is only a direct member of one beancounter. So cpusets and beancounters are each a partition function on the set of processes.

Some initial work on this project produced a system that simply used the same partition function for all subsystems, i.e. for every cpuset created there would also be a beancounter created, and all processes moved into the new cpuset would also become part of the new beancounter. However, very plausible scenarios were presented to demonstrate that this was too limiting.

A generic container framework should support some way of allowing different partitions for different subsystems. Since the requirements suggest that supporting hierarchical partitions is useful, even if not required/desired for all subsystems, we refer to these partitions, without loss of generality, as *hierarchies*.

For illustration, we consider the following examples of dividing processes on a system.
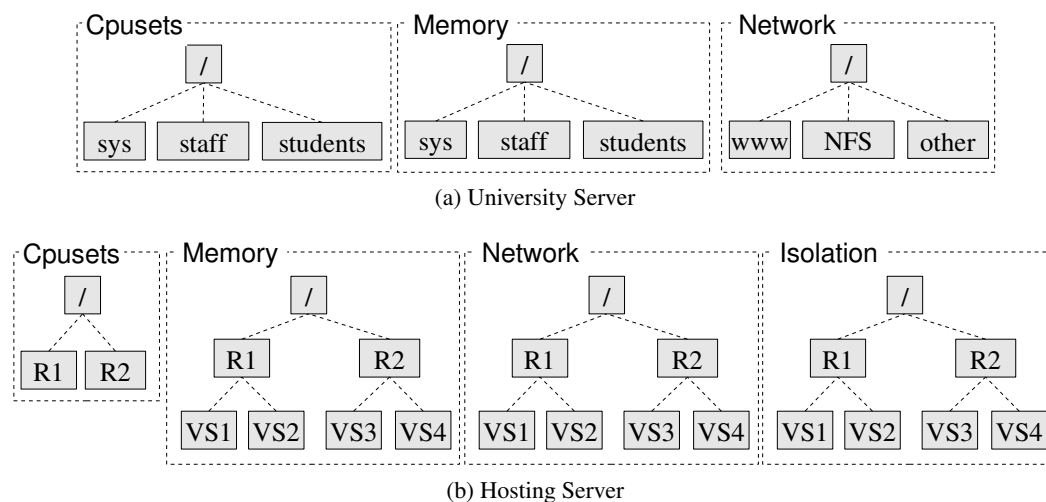
(a) University Server

(b) Hosting Server

Figure 1: Example container divisions with independent (single-subsystem) hierarchies

A University Timesharing System[2]

A university server has various users—students, professors, and system tasks. For CPU and memory, it is desired to partition the system according to the process owner's user ID, whereas for network traffic, it is desired to partition the system according to the traffic content (e.g. WWW browser related traffic across all users shares the same limit of 20%). Any single way of partitioning the system will make this kind of resource planning hard, potentially requiring creating a container for each tuple of the cross-product of the various configured resource subsystems.. Allowing the system to be partitioned in multiple ways, depending on resource type, is therefore a desirable feature.

A Virtual Server System

A large commercial hosting server has many NUMA nodes. Blocks of nodes are sold to resellers (R1, R2) to give guaranteed CPU/memory resources. The resellers then sell virtual servers (VS1–VS4) to end users, potentially overcommitting their resources but not affecting other resellers on the system.

The server owner would use cpusets to restrict each reseller to a set of NUMA memory nodes/CPUs; the reseller would then (via root-delegated capabilities) use a server isolation subsystem and a resource control subsystem to create virtual servers with various levels of resource guarantees/limits, within their own cpuset resources.

---

[2]Example contributed by Srivatsa Vaddagiri.

Two possible approaches to supporting these examples are given below; the illustrative figures represent a kernel with cpusets, memory, network, and isolation subsystems.

2.6.1 Independent hierarchies

In the simplest approach, each hierarchy provides the partition for exactly one subsystem. So to use e.g. cpusets and beancounters on the same system, you would need to create a cpuset hierarchy and a beancounters hierarchy, and assign processes to containers separately for each subsystem.

This solution can be used to implement any of the other approaches presented below. The drawback is that it imposes a good deal of extra management work to userspace in the (we believe likely) common case when the partitioning is in fact the same across multiple or even all subsystems. In particular, moving processes between containers can have races—if userspace has to move a process as two separate actions on two different hierarchies, a child process forked during this operation might end up in inconsistent containers in the two hierarchies.

Figure 1 shows how containers on the university and hosting servers might be configured when each subsystem is an independent hierarchy. The cpusets and memory subsystems have duplicate configurations for the university server (which doesn't use the isolation subsystem), and the network, memory and isolation subsys-

(a) University Server

(b) Hosting Server

Figure 2: Example container divisions with multi-subsystem hierarchies

tems have duplicate configurations for the hosting server since they're the same for each virtual server.

2.6.2 Multi-subsystem hierarchies

An extension to the above approach allows multiple subsystems to be bound on to the same hierarchy; so e.g. if you were using cpusets and beancounters, and you wanted the cpuset and beancounter assignments to be isomorphic for all processes, you could bind cpusets and beancounters to the same hierarchy of containers, and only have to operate on a single hierarchy when creating/destroying containers, or moving processes between containers.

Figure 2 shows how the support for multiple subsystems per hierarchy can simplify the configuration for the two example servers. The university server can merge the configurations for the cpusets and memory subsystems (although not for the network subsystem, since that's using an orthogonal division of processes). The hosting server can merge all four subsystems into a single hierarchy.

2.7 Non-process references to containers

Although the process is typically the object most universally associated with a container, it should also be possible to associate other objects—such as pages, file

handles or network sockets—with a container, for accounting purposes or in order to affect the kernel's behaviour with respect to those objects.

Since such associations are likely to be subsystem-specific, the container framework needs primarily to be able to provide an efficient reference-counting mechanism, which will allow references to subsystem state objects to be made in such a way that they prevent the destruction of the associated container until the reference has been released.

2.8 Low overhead

The container framework should provide minimal additional runtime overhead over a system where individual subsystems are hard-coded into the source at all appropriate points (pointers in `task_struct`, additional `fork()` and `exit()` handlers, etc).

3 Existing/Related Work

In this section we consider the existing mechanisms for process tracking and control in Linux, looking at both those already included in the Linux source tree, and those proposed as bases for other efforts.

3.1 Unix process grouping mechanisms

Linux has inherited several concepts from classical Unix that can be used to provide some form of association be-

tween different processes. These include, in order of increasing specificity: group id (gid), user id (uid), session id (sid) and process group (pgrp).

Theoretically the gid and/or uid could be used as the identification portion of the container framework, and a mechanism for configuring per-uid/gid state could be added. However, these have the serious drawbacks that:

- Job-control systems may well want to run multiple jobs as the same user/group on the same machine.

- Virtual server systems will want to allow processes within a virtual server to have different uids/gids.

The sid and/or the pgrp might be suitable as a tracking base for containers, except for the fact that traditional Unix semantics allow processes to change their sid/pgrp (by becoming a session/group leader); removing this ability would be possible, but could resulting in unexpected breakage in applications. (In particular, requiring all processes in a virtual server to have the same sid or pgrp would probably be unmanageable).

3.2   Cpusets

The cpusets system is the only major container-like system in the mainline Linux kernel. Cpusets presents a pseudo-filesystem API to userspace with semantics including:

- Creating a directory creates a new empty cpuset (an analog of a container).

- Control files in a cpuset directory allow you to control the set of memory nodes and/or CPUs that tasks in that cpuset can use.

- A special control file, `tasks` can be read to list the set of processes in the cpuset; writing a pid to the `tasks` file moves a process into that cpuset.

3.3   Linux/Unix container systems

The Eclipse [3] and *Resource Containers* [4] projects both sought to add quality of service to Unix. Both supported hierarchical systems, allowing free migration of processes and threads between containers; Eclipse

used the independent hierarchies model described in Section 2.6.1; Resource Containers bound all schedulers (subsystems) into a single hierarchy.

PAGG [5] is part of the SGI Comprehensive System Accounting project, adapted for Linux. It provides a generic container mechanism that tracks process membership and allows subsystems to be notified when processes are created or exit. A crucial difference between PAGG and the design presented in this paper is that PAGG allows a free-form association between processes and arbitrary containers. This results in more expensive access to container subsystem state, and more expensive `fork()`/`exit()` processing.

Resource Groups [2] (originally named CKRM – *Class-based Kernel Resource Management*) and BeanCounters [1] are resource control frameworks for Linux. Both provide multiple resource controllers and support additional controllers. ResGroups' support for additional controllers is more generic than the design proposed in this paper—we feel that the additional overheads that it introduces are unnecessary; BeanCounters is less generic, in that additional controllers have to be hard-coded into the existing BeanCounters source. Both frameworks also enforce a particular resource model on their resource controllers, which may be inappropriate for resource controllers with different requirements from those envisaged—for example, implementing cpusets with its current interface (or an equivalent natural interface) on top of either the BeanCounters or ResGroups abstractions would not be possible.

3.4   Linux virtual server systems

There have been a variety of virtual server systems developed for Linux; early commercial systems included Ensim's Virtual Private Server [6] and SWSoft's Virtuozzo [7]; these both provided various resource controllers along with namespace isolation, but no support for generic extension with user-provided subsystems.

More recent open-sourced virtualization systems have included VServer [8] and OpenVZ [9], a GPL'd subset of the functionality of Virtuozzo.

3.5   NSProxy-based approaches

Recent work on Linux virtual-server systems [10] has involved providing multiple copies of the various

namespaces (such as for IPC, process ids, etc) within the kernel, and having the namespace choice be made on a per-process basis. The fact that different processes can now have different IPC namespaces results in the requirement that these namespaces be reference-counted on `fork()` and released on `exit()`. To reduce the overhead of such reference counting, and to reduce the number of per-process pointers required for all these virtualizable namespaces, the `struct nsproxy` was introduced. This is a reference-counted structure holding reference-counted pointers to (theoretically) all the namespaces required for a task; therefore at `fork()`/`exit()` time only the reference count on the `nsproxy` object must be adjusted. Since in the common case a large number of processes are expected to share the same set of namespaces, this results in a reduction in the space required for namespace pointers and in the time required for reference counting, at the cost of an additional indirection each time one of the namespaces is accessed.

4 Proposed Design

This section presents a proposed design for a container framework based on the requirements presented in Section 2 and the existing work surveyed above. A prototype implementation of this design is available at the project website [11], and has been posted to the `linux-kernel@vger.kernel.org` mailing list and other relevant lists.

4.1 Overview

The proposed container approach is an extension of the process-tracking design used for cpusets. The current design favours the multiple-hierarchy approach described in Section 2.6.2.

4.2 New structure types

The container framework adds several new structures to the kernel. Figure 3 gives an overview of the relationship between these new structures and the existing `task_struct` and `dentry`.

4.2.1 `container`

The `container` structure represents a container object as described in Section 1. It holds parent/child/sibling information, per-container state such as flags, and a set of subsystem state pointers, one for the state for each subsystem configured in the kernel. It holds no resource-specific state. It currently[3] holds no reference to a list of tasks in the container; the overhead of maintaining such a list would be paid whenever tasks `fork()` or `exit()`, and the relevant information can be reconstructed via a simple walk of the tasklist.

4.2.2 `container_subsys`

The `container_subsys` structure represents a single resource controller or isolation component, e.g. a memory controller or a CPU scheduler.

The most important fields in a `container_subsys` are the callbacks provided to the container framework; these are called at the appropriate times to allow the subsystem to learn about or influence process events and container events in the hierarchy to which this subsystem is bound, and include:

create is called when a new container is created

destroy is called when a container is destroyed

can_attach is called to determine whether the subsystem wants to allow a process to be moved into a given container

attach is called when a process moves from one container to another

fork is called when a process forks a child

exit is called when a process exits

populate is called to populate the contents of a container directory with subsystem-specific control files

bind is called when a subsystem is moved between hierarchies.

---

[3]The possibility of maintaining such a task list just for those subsystems that really need it is being considered.

Apart from `create` and `destroy`, implemention of these callbacks is optional.

Other fields in `container_subsys` handle housekeeping state, and allow a subsystem to find out to which hierarchy it is attached.

Subsystem registration is done at compile time— subsystems add an entry in the header file `include/linux/container_subsys.h`. This is used in conjunction with pre-processor macros to statically allocate an identifier for each subsystem, and to let the container system locate the various `container_subsys` objects.

The compile-time registration of subsystems means that it is not possible to build a container subsystem purely as a module. Real-world subsystems are expected to require subsystem-specific hooks built into other locations in the kernel anyway; if necessary, space could be left in the relevant arrays for a compile-time configurable number of "extra" subsystems.

4.2.3 `container_subsys_state`

A `container_subsys_state` represents the base type from which subsystem state objects are derived, and would typically be embedded as the first field in the subsystem-specific state object. It holds housekeeping information that needs to be shared between the generic container system and the subsystem. In the current design this state consists of:

container – a reference to the `container` object with which this state is associated. This is primarily useful for subsystems which want to be able to examine the tree of containers (e.g. a hierarchical resource manager may propagate resource information between subsystem state objects up or down the hierarchy of containers).

refcnt – the reference count of external non-process objects on this subsystem state object, as described in Section 2.7. The container framework will refuse to destroy a container whose subsystems have non-zero states, even if there are no processes left in the container.

To access its state for a given task, a subsystem can call `task_subsys_state(task, <subsys_id>)`.

This function simply dereferences the given subsystem pointer in the task's `css_group` (see next Section).

4.2.4 `css_group`

For the same reasons as described in Section 3.5, maintaining large numbers of pointers (per-hierarchy or per-subsystem) within the `task_struct` object would result in space wastage and reference-counting overheads, particularly in the case when container systems compiled into the kernel weren't actually used.

Therefore, this design includes the `css_group` (container subsystem group) object, which holds one `container_subsys_state` pointer for each registered subsystem. A reference-counted pointer field (called `containers`) to a `css_group` is added to `task_struct`, so the space overhead is one pointer per task, and the time overhead is one reference count operation per `fork()`/`exit()`. All tasks with the same set of container memberships across all hierarchies will share the same `css_group`.

A subsystem can access the per-subsystem state for a task by looking at the slot in the task's `css_group` indexed by its (statically defined) subsystem id. Thus the additional indirection is the only subsystem-state access overhead introduced by the `css_group`; there's no overhead due to the generic nature of the container framework. The space/time tradeoff is similar to that associated with `nsproxy`.

It has been proposed (by Srivatsa Vaddagiri and others) that the `css_group` should be merged with the `nsproxy` to form a single per-task object containing both namespace and container information. This would be a relatively straightforward change, but the current design keeps these as separate objects until more experience has been gained with the system.

4.3 Code changes in the core kernel

The bulk of the new code required for the container framework is that implementing the container filesystem and the various tracking operations. These are driven entirely by the user-space API as described in Section 4.5.

Changes in the generic kernel code are minimal and consist of:
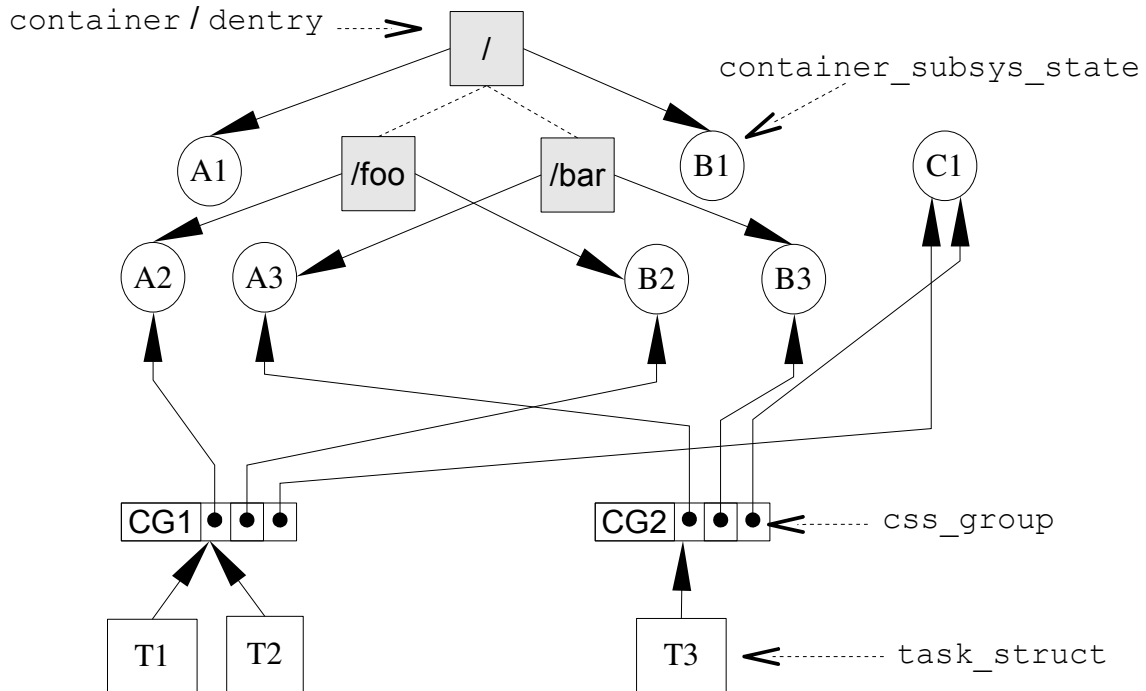
Figure 3: Three subsystems (A, B, C) have been compiled into the kernel; a single hierarchy has been mounted with A and B bound to it. Container directories `foo` and `bar` have been created, associated with subsystem states A2/B2 and A3/B3 respectively. Tasks T1 and T2 are in container `foo`; task T3 is in container `bar`. Subsystem C is unbound so all container groups (and hence tasks) share subsystem state C1.

- A hook early in the `fork()` path to take an additional reference count on the task's `css_group` object.

- A hook late in the `fork()` path to invoke subsystem-specific `fork` callbacks, if any are required.

- A hook in the `exit()` path to invoke any subsystem-specfic `exit` callbacks, if any, and to release the reference count on the task's `css_group` object (which will also free the `css_group` if this releases the last reference.)

4.4 Locking Model

The current container framework locking model revolves around a global mutex (`container_mutex`), each task's `alloc_lock` (accessed via `task_lock()` and `task_unlock()`) and RCU critical sections.

The `container_mutex` is used to synchronize per-container operations driven by the userspace API. These include creating/destroying containers, moving processes between containers, and reading/writing subsystem control files. Performance-sensitive operations should not require this lock.

When modifying the `containers` pointer in a task, the container framework surrounds this operation with `task_lock()`/`task_unlock()`, and follows with a `synchronize_rcu()` operation before releasing the `container_mutex`.

Therefore, in order for a subsystem to be sure that it is accessing a valid `containers` pointer, it suffices for *at least one* of the following three conditions to be true of the current task:

- it holds `container_mutex`

- it holds its own `alloc_lock`.

- it is in an RCU critical section.

The final condition, that of being in an RCU critical section, doesn't prevent the current task being concurrently moved to a different container in some hierarchy—it

simply tells you that the current task was in the specified set of containers at some point in the very recent past, and that any of the subsystem state pointers in that `css_group` object won't be candidates for deletion until after the end of the current RCU critical section.

When an object (such as a file or a page) is accounted to the value that has been read as the current subsystem state for a task, it may actually end up being accounted to a container that the task has just been moved from; provided that a reference to the charged subsystem state is stored somewhere in the object, so that at release time the correct subsystem state can be credited, this is typically sufficient. A subsystem that wants to reliably migrate resources between containers when the process that allocated those resources moves (e.g. cpusets, when the `memory_migrate` mode is enabled) may need additional subsystem-specific locking, or else use one of the two other locking methods listed above, in order to ensure that resources are always accounted to the correct containers.

The subsystem state pointers in a given `css_group` are immutable once the object has been created; therefore as long as you have a pointer to a valid `css_group`, it is safe to access the subsystem fields without additional locking (beyond that mandated by subsystem-specific rules).

4.5 Userspace API

The userspace API is very similar to that of the existing cpusets system.

A new hierarchy is created by mounting an instance of the `container` pseudo-filesystem. Options passed to the `mount()` system call indicate which of the available subsystems should be bound to the new hierarchy. Since each subsystem can only be bound to a single hierarchy at once, this will fail with `EBUSY` if the subsystem is already bound to a different hierarchy.

Initially, all processes are in the root container of this new hierarchy (independently of whatever containers they might be members of in other hierarchies).

If a mount request is made for a set of subsystems that exactly match an existing active hierarchy, the same superblock is reused.

At the time when a container hierarchy is unmounted, if the hierarchy had no child containers then the hierarchy is released and all subsystems are available for reuse. If the hierarchy still has child containers, the hierarchy (and superblock) remain active even though not actively attached to a mounted filesystem.[4]

Creating a directory in a container mount creates a new child container; containers may be arbitrarily nested, within any restrictions imposed by the subsystems bound to the hierarchy being manipulated.

Each container directory has a special control file, `tasks`. Reading from this file returns a list of processes in the container; writing a pid to this file moves the given process into the container (subject to successful `can_attach()` callbacks on the subsystems bound to the hierarchy).

Other control files in the container directory may be created by subsystems bound to that hierarchy; reads and writes on these files are passed through to the relevant subsystems for processing.

Removing a directory from a container mount destroys the container represented by the directory. If tasks remain within the container, or if any subsystem has a nonzero reference count on that container, the `rmdir()` operation will fail with `EBUSY`. (The existence of subsystem control files within a directory does not keep it busy; these are cleared up automatically.)

The file `/proc/PID/container` lists, for each active hierarchy, the path from the root container to the container of which process `PID` is a member.[5]

The file `/proc/containers` gives information about the current set of hierarchies and subsystems in use. This is primarily useful for debugging.

4.6 Overhead

The container code is optimized for fast access by subsystems to the state associated with a given task, and a fast `fork()`/`exit()` path.

Depending on the synchronization requirements of a particular subsystem, the first of these can be as simple as:

---

[4]They remain visible via a `/proc` reporting interface.

[5]An alternative proposal is for this to be a directory holding container path files, one for each subsystem.

```
struct task_struct *p = current;
rcu_read_lock()
struct state *st =
  task_subsys_state(p,
                    my_subsys_id);
...
<Do stuff with st>
...
rcu_read_unlock();
```

On most architectures, the RCU calls expand to no-ops, and the use of inline functions and compile-time defined subsystem ids results in the code being equivalent to `struct state *st = p->containers->subsys[my_subsys_id]`, or two constant-offset pointer dereferences. This involves one additional pointer dereference (on a presumably hot cacheline) compared to having the subsystem pointer embedded directly in the task structure, but has a reduced space overhead and reduced refcounting overhead at `fork()` / `exit()` time.

Assuming none of the registered subsystems have registered `fork()` / `exit()` callbacks, the overhead at `fork()` (or `exit()`) is simply a `kref_get()` (or `kref_put()`) on `current->containers->refcnt`.

5   Example Subsystems

The containers patches include various examples of subsystems written over the generic containers framework. These are primarily meant as demonstrations of the way that the framework can be used, rather than as fully-fledged resource controllers in their own right.

5.1   CPU Accounting Subsystem

The `cpuacct` subsystem is a simple demonstration of a useful container subsystem. It allows the user to easily read the total amount of CPU time (in milliseconds) used by processes in a given container, along with an estimate of the recent CPU load for that container.

The 250-line patch consists of:

- callback hooks added to the `account_*_time()` functions in `kernel/sched.c` to inform the subsystem when a particular process is being charged for a tick.

- declaration of a subsystem in `include/linux/container_subsys.h`

- Kconfig/Makefile additions

- the code in `kernel/cpuacct.c` to implement the subsystem. This can focus on the actual details of tracking the CPU for a container, since all the common operations are handled by the container framework.

Internally, the `cpuacct` subsystem uses a per-container spinlock to synchronize access to the usage/load counters.

5.2   Cpusets

Cpusets is already part of the mainline kernel; as part of the container patches it is adapted to use the generic container framework (the primary change involved removal of about 30% of the code, that was previously required for process-tracking).

Cpusets is an example of a fairly complex subsystem with hooks into substantial other parts of the kernel (particularly memory management and scheduler parameters). Some of its control files represent flags, some represent bitmasks of memory nodes and cpus, and others report usage values. The interface provided by the generic container framework is sufficiently flexible to accomodate the cpusets API.

Internally, cpusets uses an additional global mutex—`callback_mutex`—to synchronize container-driven operations (moving a task between containers, or updating the memory nodes for a container) with callbacks from the memory allocator or OOM killer.

For backwards compatibility the existing `cpuset` filesystem type remains; any attempt to mount it gets redirected to a mount of the `container` filesystem, with a subsystem option of `cpuset`.

5.3   ResGroups

ResGroups (formerly CKRM [2]) is a hierarchical resource control framework that specifies the resource limits for each child in terms of a fraction of the resources available to its parent. Additionally resources

may be borrowed from a parent if a child has reached its resource limits.

The abstraction provided by the generic containers framework is low-level, with free-form control files. As an example of how to provide multiple subsystems sharing a common higher-level resource abstraction, ResGroups is implemented as a container subsystem library by stripping out the group management aspects of the code and adding container subsystem callbacks. A resource controller can use the ResGroups abstraction simply by declaring a container subsystem, with the subsystem's `private` field pointing to a ResGroups `res_controller` structure with the relevant resource-related callbacks.

The ResGroups library registers control files for that subsystem, and translates the free-form read/write interface into a structured and typed set of callbacks. This has two advantages:

- it reduces the amount of parsing code required in a subsystem

- it allows multiple subsystems with similar (resource or other) abstractions to easily present the same interface to userspace, simplifying userspace code.

One of the ResGroups resource controllers (`numtasks`, for tracking and limiting the number of tasks created in a container) is included in the patch.

5.4 BeanCounters

BeanCounters [1] is a single-level resource accounting framework that aims to account and control consumption of kernel resources used by groups of processes. Its resource model allows the user to specify soft and hard limits on resource usage, and tracks high and low watermarks of resource usage, along with resource allocations that failed due to limits being hit.

The port to use the generic containers framework converts between the raw read/write interface and the structured get/store in a similar way to the ResGroups port, although presenting a different abstraction.

Additionally, BeanCounters allows the accounting context (known as a *beancounter*) to be overridden in particular situations, such as when in an interrupt handler or when performing work in the kernel on behalf of another process. This aspect of BeanCounters is maintained unchanged in the containers port—if a process has an override context set then that is used for accounting, else the context reached via the `css_group` pointer is used.

5.5 NSProxy / Container integration

A final patch in the series integrates the NSProxy system with the container system by making it possible to track processes that are sharing a given namespace, and (potentially in the future) create custom namespace sets for processes. This patch is a (somewhat speculative) example of a subsystem that provides an isolation system rather than a resource control system.

The namespace creation paths (via `fork()` or `unshare()`) are hooked to call `container_clone()`. This is a function provided by the container framework that creates a new sub-container of a task's container (in the hierarchy to which a specified subsystem is bound) and moves the task into the new container. The `ns` container subsystem also makes use of the `can_attach` container callback to prevent arbitrary manipulation of the process/container mappings.

When a task changes its namespaces via either of these two methods, it ends up in a fresh container; all of its children (that share the same set of namespaces) are in the same container. Thus the generic container framework provides a simple way to export to userspace the sets of namespaces in use, their hierarchical relationships, and the processes using each namespace set.

6 Conclusion and Future Work

In this paper we have examined some of the existing work on process partitioning/tracking in Linux and other operating systems, and enumerated the requirements for a generic framework for such tracking; a proposed design was presented, along with examples of its use.

As of early May 2007, several other groups have proposed resource controllers based on the containers framework; it is hoped that the containers patches can be trialled in Andrew Morton's `-mm` tree, with the aim of reaching the mainline kernel tree in time for 2.6.23.

7 Acknowledgements

References

[1] Kir Kolyshkin, *Resource management: the Beancounters*, Proceedings of the Linux Symposium, June 2007

[2] *Class-based Kernel Resource Management*, `http://ckrm.sourceforge.net`

[3] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz, *Resource management for QoS in Eclipse/BSD*. In Proceedings of FreeBSD'99 Conf., Berkeley, CA, USA, Oct. 1999

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. *Resource containers: A new facility for resource management in server systems*. In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pages 45–58, 1999

[5] SGI. *Linux Process Aggregates (PAGG)*, `http://oss.sgi.com/projects/pagg/`

[6] Ensim Virtual Private Server, `http://www.ensim.com`

[7] SWSoft Virtuozzo, `http://www.swsoft.com/en/virtuozzo/`

[8] Linux VServer, `http://www.linux-vserver.org/`

[9] OpenVZ, `http://www.openvz.org/`

[10] Eric W. Biederman, *Multiple Instances of the Global Linux Namespaces*, Proceedings of the Linux Symposium, July 2006.

[11] Linux Generic Process Containers, `http://code.google.com/p/linuxcontainers`

KvmFS: Virtual Machine Partitioning For Clusters and Grids

Andrey Mirtchovski
Los Alamos National Laboratory
andrey@lanl.gov

Latchesar Ionkov
Los Alamos National Laboratory
lionkov@lanl.gov

Abstract

This paper describes KvmFS, a synthetic file system that can be used to control one or more KVM virtual machines running on a computer. KvmFS is designed to provide its functionality via an interface that can be exported to other machines for remote configuration and control. The goal of KvmFS is to allow a multi-CPU, multi-core computer to be partitioned externally in a fashion similar to today's computational nodes on a cluster. KvmFS is implemented as a file server using the 9P protocol and its main daemon can be mounted locally via the v9fs kernel module. Communication with the KvmFS occurs through standard TCP sockets. Virtual machines are controlled via commands written to KvmFS' files. Status information about KVM virtual machines is obtained by reading KvmFS. KvmFS allows us to build clusters in which more than one application can share the same SMP/Multi-core node with minimalistic full system images tailored specifically for the application.

1   Introduction

The tendency in high-performance computing is towards building processors with many computational units, or cores, with the goal of parallelizing computation so that many units are performing work at the same time. Dual and quad-core processors are already on the market, and manufacturers are hinting at 8, 32, or even 80 cores for a single CPU, with a single computational node composed of two, four, or more CPUs. This will result in applications running and contending for resources on large symmetric multiprocessor systems (SMPs) composed of hundreds of computational units.

There is a problem with this configuration, however: since clusters are currently the dominant form of node organization in the HPC world (as seen in the latest breakdown by machine type on the Top 500 list of supercomputers), most applications are designed to either run on a single 2- or 4-core machine, or so that separate parts of the program will run on separate 2- or 4-core nodes, and will communicate via some message-passing framework such as MPI. This has resulted in most of the applications running here, at Los Alamos National Laboratory, scaling to at most 8 CPUs on a single node. Furthermore, many applications assume that they are the only ones running on a single node and will not have to contend for resources. To satisfy the requirements of such applications, large SMP computers will have to be partitioned so that applications are ensured dedicated resources without contention. Fail-over and resilience, two very hot topics in High Performance Computing, also require the means to transfer an application from one machine to another in the case of hardware or software component failures on the original computer.

One solution for partitioning hardware and providing resilience has gained widespread adoption and is considered feasible for the HPC world: virtualization using hypervisors. Borrowing from the mainframe, it allows separate instances of an operating system (or indeed separate operating systems) to be run on the same hardware or parts thereof. The two major CPU manufacturers have added support for virtualization to their newest offerings, which provides even greater performance gains than previously thought.

KVM has recently emerged as a fast and reliable (with the hardware support on modern processors) subsystem for virtualizing the hardware on a computer. Our goal with KvmFS is to enable KVM to be remotely controlled by either system operators or schedulers and to allow it to be used for partitioning on clusters composed of large SMP machines, such as the ones already being proposed here at LANL.

Virtualization benefits the system administrator, as well as programmers and scientists running high-performance code on large clusters. One benefit is the

full control over the operating system installation that an application requires. For example, it is not necessary to have all support libraries and software installed on all machines of a cluster, instead, the application is run in an OS instance that already contains all that is required. This greatly simplifies installations in the case where conflicting libraries and support software may be required by different applications. Another possibility is to run a completely different operating system under virtualization, something impossible in current monolithic cluster environments.

With the fast and reliable means of running applications on their own slice of a SMP, it is convenient to be able to extend the control of partitioning and virtualization across the cluster to a control or a head node. This is the niche that KvmFS fills: it provides the fast and secure means to control VMs across a cluster, or indeed a grid environment.

1.1 KVM

KVM [14] is a hypervisor support module in the Linux kernel which utilizes hardware-assisted x86 virtualization on modern Intel processors with Intel Virtualization technology or AMD's Secure Virtual Machine. By adding virtualization capabilities to a standard Linux kernel, KVM provides the benefits of the optimizations that exist in a standard kernel to virtualized programs, greatly increasing performance over "full hypervisors" such as Xen [1] or VMWare [9]. Under the KVM model, every virtual machine is a regular Linux process scheduled by the standard Linux scheduler. Its memory is allocated by the Linux memory allocator.

KVM works in conjunction with QEMU to deliver the processor's virtualization capabilities to the end user.

1.2 QEMU

QEMU [2] is a machine emulator which can run an unmodified target operating system (such as Windows or Linux) and all its applications in a virtual machine. QEMU runs on several host operating systems such as Linux, Windows, and Mac OSX.

The primary usage of QEMU is to run one operating system on another, such as Windows on Linux or Linux on Windows. Another usage is debugging, because the virtual machine can be easily stopped, and its state can

be inspected, saved, and restored. Moreover, specific embedded devices can be simulated by adding new machine descriptions and new emulated devices.

Although the host and target operating systems can be different, our software will focus on Linux as the host system since Linux is the primary OS on all of our recent clusters at LANL and is widely adopted for HPC environments. Also, KVM currently exists only for the Linux kernel.

2 Design

KvmFS was created allow its users to run and control virtual machines in a heterogeneous networked environment. As such, KvmFS was designed to fulfill the following tasks:

functionality provide an interface that allows management of VMs on a cluster

scalability provide the ability for fast creation of multiple identical VMs on different nodes connected via a network

checkpoint and restart provide the ability to suspend virtual machines and resume their execution, potentially on a different node

The design of KvmFS follows the well established model of providing functionality in the form of synthetic file systems which clients operate on using standard I/O commands such as `read` and `write`. This method has proven successful in various operating systems descendant from UNIX. The `/proc` [7] file system is a very well established example. The "Plan 9" operating system further extends this concept. It presents the network communication subsystem as mountable files [13] or even the graphics subsystem and the window manager written on top of it, as a file system.

Implementations such as the above suggest that the concept is feasible and that implementing interfaces to resources in the form of a file system and exporting them to other machines is a very good way to quickly allow access to them from remote machines, especially since files are the single most exported resource in a networked environment such as a cluster.

KvmFS is structured as a two-tiered file server to which clients connect either from the local machine or across

the network. The file server allows them to copy image files and boot virtual machines using those image files. The file server also allows controlling running virtual machines (start, stop, freeze), as well as migrating them from one computer to another.

The top-level directory KvmFS serves contains two files providing information about the architecture of the machine as well as starting a new session for a new VM. Each session already started is presented as a numbered subdirectory. The subdirectory itself presents files which can be used to control the execution of the VM, as well as a subdirectory which allows arbitrary image files to be copied to it and used by the VM. The KvmFS filesystem is presented in detail in section 3.

3   The KVM File System

KvmFS presents a synthetic file system to its clients. The file system can be used for starting and controlling all aspects of the runtime of the virtual machines running on the machine on which kvmfs is running.

```
clone
arch
vm#/
        ctl
        info
        id
        fs/
```

3.1   Top-level files

Arch is a read-only file; reading from it returns the architecture of the compute node in a format *operating-system/processor-type*.

Clone is a read-only file. When it is opened, KvmFS creates a new session and the corresponding session directory in the filesystem.

Reading from the file returns the name of the session directory.

Vm# is a directory corresponding to a session created by a KvmFS client. Even though a session may not be running, Vm# will exist as long as that client keeps the clone file open. If the virtual machine corresponging to a session is running the clone file may be closed without causing the Vm# file to disappear.

3.2   Session-level files

These files are contained in the session directory which is created when a client opens the clone file of a KvmFS server.

Ctl is used to execute and control a session's main process. Reading from the file returns the main process pid if the process is running, and –1 otherwise. The operations on the session are performed by writing to it.

Reading from info returns the current memory and device configuration of the virtual machine. The format of the information is identical to the commands written to ctl file.

Id is used to set and get the user-specified VM identifier.

The fs directory points to the temporary storage created for the virtual machine. The user can copy disk images and saved VM state files that can be used in the VM configuration.

4   KvmFS Commands

The following section describes the set of commands available for controlling KvmFS instances:

dev *name image*   Specifies the device image for a specific device. *Name* is one of *hda, hdb, hdc, hdd*. If *image* is not an absolute path, it should point to a file that is copied in the fs directory. An optional *boot* parameter can be provided to specify that the device should be used to boot from.

net *id mac*   Creates a network device with ID *id* and MAC *mac*.

loadvm *file*   Loads a saved VM state from file *file*. If *file* is not an absolute path, it should point to a file in the fs directory.

storevm *file*   Stores the state of the VM to file *file*. If *file* is not an absolute path, the file is created in the fs directory.

power *on/off*   Turns VM power on or off.

freeze   Suspends the execution of the VM.

unfreeze   Resumes the execution of the VM.

clone *max-vms address-list* Creates copies of the VM on the nodes specified by *address-list.* Copies the content of the `fs` directory to the remote VMs and configures the same device configuration. If the virtual machine is already running, stores the current VM state (as in *storevm*) and loads it in the remote VMs. If `max-vms` is greater than zero, and the number of the specified sessions is bigger than `max-vms`, `clone` pushes its content to up to max-sessions and issues `clone` commands to some of them to clone themselves to the remaining VMs from the list.

The format of the address-list is:

```
address-list = \
    1*(vm-address ',')
vm-address = node-name \
    ['!''port]
   '/'' vm-id
node-name = ANY
port = NUMBER
vm-id = ANY
```

5 Implementation

There are two ways of implementing accesses to programs or system resources as files in Linux, either using Fuse [3] or the 9P [8] protocol. We chose the 9P protocol because it is better suited for communicating with file systems over networks. 9P has also been in use for the past twenty years and is sufficiently hardened to be able to handle various workloads on environments ranging from a single machine to thousands of cluster nodes [5]. Furthermore, our team is well familiarized with 9P through the implementation of V9FS, the kernel module allowing 9P servers to be mounted on a Linux filesystem [10] [4]. It is important to point out, however, that there is no significant barrier to implementing KvmFS using FUSE.

5.1 9P

Representing operating system resources as files is a relatively old concept exploited to some extent in the original UNIX operating system, but it matured extensively with the development and release of the "Plan 9 from Bell-Labs" operating system [12].

"Plan 9 from Bell-Labs" uses a simple, yet very powerful communication protocol to facilitate communication between different parts of the system. The protocol, named "9P" [8], allows heterogeneous resource sharing by allowing servers to build a hierarchy of files corresponding to real or virtual system resources, which then clients access via common (POSIX-like) file operations by sending and receiving 9P messages. The different types of 9P messages are described in Table 1.

There are several benefits of using the 9P protocol:

Simplicity The protocol has only a handful of messages which encompass all major file operations, yet it can be implemented (including the co-routine code explained above) in around 2,000 lines of C code.

Robustness 9P has been in use in the Plan 9 operating system for over 15 years.

Architecture independence 9P has been ported to and used on all major computer architectures.

Scalability Our Xcpu [11] suite uses 9P to control and execute programs on thousands of nodes at the same time.

A 9P *session* between a server and its clients consists of requests by the clients to navigate the server's file and directory hierarchy and responses from the server to those requests. The client initiates a request by issuing a *T-message*, the server responds with an *R-messages*. A 9P *transaction* is the combined act of transmitting a request of particular type by the client and receiving a reply from the server. There may be more than one request outstanding; however, each request requires a response to complete a transaction. There is no limit on the number of transactions in progress for a single session.

Each 9P message contains a sequence of bytes representing the size of the message, the type, the tag (transaction id), control fields depending on the message type, and a UTF-8 encoded payload. Most T-messages contain a 32-bit unsigned integer called *Fid*, used by the client to identify the "current file" on the server, i.e., the last file accessed by the client. Each file in the file system served by our library has an associated element called *Qid* used to uniquely identify it in the file system.

5.2 KvmFS

KvmFS is implemented in C using the SPFS and Spclient [6] libraries for writing 9P2000-compliant userspace file servers and accessing them over a network.

9P type	Description
version	identifies the version of the protocol and indicates the maximum message size the system is prepared to handle
auth	exchanges auth messages to establish an authentication fid used by the attach message
error	indicates that a request (T-message) failed and specifies the reason for the failure
flush	aborts all outstanding requests
attach	initiates a connection to the server
walk	causes the server to change the current file associated with a fid
open	opens a file
create	creates a new file
read	reads from a file
write	writes to a file
clunk	frees a fid that is no longer needed
remove	deletes a file
stat	retrieves information about a file
wstat	modifies information about the file

Table 1: Message types in the 9P protocol

It is a single-threaded code which uses standard networking via the `socket()` routines. Although our implementation is in C, both 9P2000 and KvmFS are language-agnostic and can be reimplemented in any other programming language that has access to networking.

OS Image files used by virtual machines can grow to be quite large (sometimes up to the size of a complete system installation: several gigabytes) and can take a long time to be transferred to a remote node. To start a single VM on all the nodes of a cluster can potentially take upwards of an hour for large clusters, with literally a hundred percent of the time being spent transferring the disk images of the VM either from a head node or from a networked file system such as NFS. To alleviate this problem we can employ tree-based spawning of virtual machines via cloning. During tree-spawning, if an end node has received the complete image (or in some cases a partial image), that node can retransmit the image to another node, potentially located only a hop away on the network. To allow tree-spawns each KvmFS server can also serve as a client to another server by implement-

ing routines which connect over 9P, create new sessions, set-up and start a new VM with the image from the local session. This reduces logarithmically the amount of fetches that need to occur from the head node and significantly increases the scale at which KvmFS can be deployed. We have tested tree-spawn algorithms for small images on several thousand nodes on LANL's clusters.

The total number of lines for KvmFS, not including the SPFS libraries, is less than two thousand lines of code. SPFS itself is 5,158 lines of code, and Spclient is another 2,381 lines of code.

6 Sample Sessions

Several examples of using KvmFS follow. The examples show systems mounted remotely using the v9fs [4] kernel module and consequently being accessed via common shell commands. In the examples below, the names `n1`, `n2`, etc., are names of nodes on our cluster.

6.1 Create a virtual machine

This example creates a virtual machine using two files copied from the home directory. `Disk.img` is set to correspond to hard drive `hda` and `vmstate` is used as a previously saved virtual machine.

```
mount -t 9p n1 /mnt/9
cd /mnt/9
tail -f clone &
cd 0
cp ~/disk.img fs/disk.img
cp ~/vmstate fs/vmstate
echo dev hda disk.img > ctl
echo net 0 00:11:22:33:44:55 > ctl
echo power on freeze > ctl
echo loadvm vmstate > ctl
echo unfreeze > ctl
```

6.2 Migrate a virtual machine to another node

This example shows the migration of a virtual machine from one node to another.

```
mount -t 9p n1 /mnt/9/1
mount -t 9p n2 /mnt/9/2
tail -f /mnt/9/2/clone &
cd /mnt/9/1/0
echo freeze > ctl
echo `clone 0 n2!7777/0' > ctl
echo power off > ctl
```

6.3 Create clones of a virtual machine

This example shows the cloning of a virtual machine onto a new computer.

```
mount -t 9p n1 /mnt/9
cd /mnt/9/0
echo `clone 2 n2!7777/0,\
  n3!7777/0,\
  n4!7777/0` > ctl
```

7 Conclusions And Future Work

We have described the KvmFS file system which presents an interface to virtual machines running on Linux in the form of files accessible locally or remotely. KvmFS allows us to extend the control of the partitioning and running of virtual machines on a computer beyond the system on which the virtual machines are running and onto a networked environment such as a cluster or a computational grid. KvmFS benefits large cluster environments such as the ones in use here, at the Los Alamos National Laboratory, by enabling fine-grained control over the software running on them from a centralized location. Status information regarding the parameters on currently running VMs can also easily be obtained from computers other than the ones they are executing on. Our system also allows checkpointing and migration of VMs to be controlled from a centralized source, thus enabling partitioning schedulers to be built on top of KvmFS.

Future work we have planned for KvmFS is in the area of fine-grained control of the execution parameters of virtual machines running under KvmFS such as their CPU affinity. Also, we plan to integrate KvmFS with existing schedulers at LANL to provide a seamless way of partitioning our clusters.

Another interesting issue we are exploring is exporting the resources of running virtual machines, such as their /proc filesystem, through the KvmFS interface so that processes running under the VM can be controlled externally or even over a network.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, and R. Neugebauer. Xen and the art of virtualization. 2004.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[3] FUSE. Filesystems in userspace. http://fuse.sourceforge.net/.

[4] Eric Van Hensbergen and Latchesar Ionkov. The v9fs project. http://v9fs.sourceforge.net.

[5] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space: Using 9p2000 under linux. In *Freenix Annual Conference*, pages 83–94, 2005.

[6] L. Ionkov. Library for writing 9p2000 compliant user-space file servers. http://sourceforge.net/projects/npfs/.

[7] T.J. Killian. Processes as files. *USENIX Summer 1984 Conf. Proc.*, 1984.

[8] AT&T Bell Laboratories. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[9] R. Meushaw and D. Simard. Nettop: Commercial technology in high-assurance applications. http://www.vmware.com, 2000.

[10] R. Minnich. V9fs: A private name space system for unix and its uses for distributed and cluster computing.

[11] R. Minnich and A. Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *Cluster 2006*, 2006.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[13] D. Presotto and P. Winterbottom. The organization of networks in plan 9. *USENIX Winter 1993 Conf. Proc.*, pages 43–50, 1993.

[14] Qumranet. Kvm: Kernel-based virtualization driver. http://kvm.qumranet.com/kvmwiki/Documents.

Linux-based Ultra Mobile PCs

Rajeev Muralidhar, Hari Seshadri, Krishna Paul, Srividya Karumuri
Mobility Group, Intel Corporation
Rajeev.D.Muralidhar@intel.com, Harinarayanan.Seshadri@intel.com
Krishna.Paul@intel.com, Srividya.Karumuri@intel.com

Abstract

Ultra Mobile PCs present a new class of challenges for system designers since they are perceived to be versatile, low power devices and yet with the full functionality of larger handhelds/laptops. User experience of being ubiquitously connected and access to data anytime and anywhere is a fundamental requirement for this class of devices. However, access to data across wireless interfaces has a direct impact on battery life of such handheld devices. This paper presents detailed analysis of some of Linux file systems and data access across wireless. Based on our observations and analysis, we make some key design recommendations for file systems for Linux-based Ultra-Mobile PCs.

1 Introduction

The new generation of Ultra-Mobile PCs offer consumers significantly better capabilities than ever before to access the Internet, be productive and enjoy their favorite digital content while on the go. Consumers now want and expect access to their personal data and the Internet no matter where they are around the world, and take their laptops/ultra-portable PCs with them. Sharing data and content is an expected feature of any new technology—users want to be connected with the important people in their lives anytime, everywhere they go and want technology to make their lives easier. Some of the important technology features of such Ultra-Mobile PCs (UMPCs) are:

- **Full PC and Internet capabilities**: Full PC capability capable of running mainstream OSes like Linux and Windows, allowing consumers to run familiar applications.

- **Location Adaptibility**: Personalized information and services based on location, environment recognition, and adaptability and interaction with other devices at home, office, or in an automobile while driving.

- **Anytime connectivity**: Connectivity in several ways such as WLAN, WWAN, WPAN, WiMAX, etc. enabling "always reachability" via email, IM, or VoIP.

- **Ultra mobility**: Small, thin, light form factor with high battery life.

In addition to the more generic UMPCs which can essentially compute and communicate with rich productivity features and anytime, anywhere data access, there are other categories of ultra-mobile devices that are targetted to specific usages such as ruggedness (for application in environmental sciences, health/medicine, etc.), affordability (such as education devices), etc.

Regardless of the targeted usage or form factor of such devices, one of the fundamental visions driving pervasive computing research is access to personal and shared data anywhere and anytime. In many ways, this vision is close to being realized with multipe wireless connectivity options such as WLAN, WWAN, WPAN, and the upcoming WiMAX connectivity options to such small, mobile devices. Additionally, with several devices becoming the norm at home, sharing data between these devices in an efficient manner is an important aspect of usage, and it is important that the underlying platform and system support for such usages is done in an energy efficient manner. For example, it is quite likely that a home is equipped with multiple devices such as a desktop, media box (possibly with network connectivity options), home entertainment system with connectivity to a desktop/media storage, laptop, high-end cell phones, and also ultra-mobile PCs, all connected to the Internet through an external broadband connection and internally via high throughput wireless, like the IEEE 802.11n. This is becoming quite common in some of the economies of the world today. In

such a scenario, a UMPC can be used to play locally stored media on the home entertainment device. Alternatively, it could be used to play remote digital media from set top box/desktop locally on the UMPC via wireless/streaming.

In such scenarios, we believe that substantial barriers remain to pervasive, energy efficient data access across wireless. Some of the key challenges are:

- **Low power platform design**: Although devices such as cell phones have matured to provide long talk time and standy battery lives, they lack the full featuredness expected in order to run producitivity applications, mainstream OSes, etc. We believe that the platform design of UMPCs pose significant challenges in order to meet the battery life and full PC capability expectations.

- **Low power communication / connectivity options**: Most connectivity options such as WLAN, WPAN, WWAN are power hungry. Usage scenarios such as media streaming over wireless connections will pose significant limitations on battery life since power-hungry network and storage devices tax the limited battery capacity of UMPCs.

- **Disconnected operation and access to data via local/network caches**: Disconnected operation is a way of life in wireless networks. Some research file systems have explored the use of local and network caches for faster access to data.

- **Energy efficient data sharing and file systems**: Power efficient file systems are critical since mobile data access performance can suffer due to variable storage access times caused by dynamic power management, mobility, and use of heterogeneous storage devices/connectivity options.

This paper focuses on the last aspect above, namely, energy efficiency in file systems, and makes the following contributions.

1. We analyze the platform power consumption of a Linux-based Ultra-mobile PC during different scenarios and quantify the impact of different platform components towards total platform power.

2. We then analyze the popular Network File System (NFS) for some of the common UMPC usage scenarios and identify the power bottlenecks.

3. Subsequently, we analyze one of the recent energy efficient file systems, BlueFS, and understand its impact on platform power.

4. Finally, based on our experiments and analysis, we make recommendations that we believe are critical to designing low power Linux-based ultra-mobile PCs.

This paper is organized as follows: Section 1 is this introduction. Section 2 reviews some of the related work. Section 3 describes the experimental analysis we performed with NFS and BlueFS. Subsequently, in Section 4, we make key observations and design recommendations. We finally conclude the paper with a summary and areas of future work.

2 Related Work

Energy efficiency in file systems is typically not a primary design consideration; this is true for distributed file systems as well. However, several recent projects target energy reduction in local file systems. In [5], the authors provided interfaces to applications to create energy-efficient device access patterns. Moving down the operating system stack, one of the primay components of a file system is its cache—file systems differ substantially in their choice and use of cache hierarchy. For example, NFS [2] uses only the kernel page cache and the file server. The Andrew File System (AFS) [4] adds a single local disk cache; however, it always tries to read data from the disk before going to the server. Coda [1], which is also designed for mobile computing, much like AFS, uses a single disk cache on the client; this cache is always accessed in preference to the server.

BlueFS [3] is different from other file systems in that it uses an adaptive cache hierarchy to reduce energy usage and efficiently integrate portable storage. BlueFS further reduces client energy usage by dynamically monitoring device power states and fetching data from the device that will use the least energy. BlueFS borrows several techniques from Coda including support for disconnected operation and asynchronous reintegration of modifications to the file server. BlueFS also can proactively trigger power mode transitions of storage devices that can lead to improved performance and energy savings. As reported in [3], mechanisms like this can greatly improve energy efficiency on the platform as a whole.

There has been a lot of analysis of power and performance of handhelds, but we believe this is the first analysis of platform power and power efficiency of full-featured ultra-mobile PCs.

3 Experimental Analysis

The experimental setup used consisted of a Linux-based Ultra-mobile PC that was instrumented to enable power measurements using a FLUKE Data acquisition system, 2680A/2686A Data Acquisition System. The following components were measured for power in different scenarios—processor, memory, chipset (GMCH), IO Hub (ICH), Hard drive (PATA), Buses (PCI-E, USB), and peripherals (Audio, WLAN card).
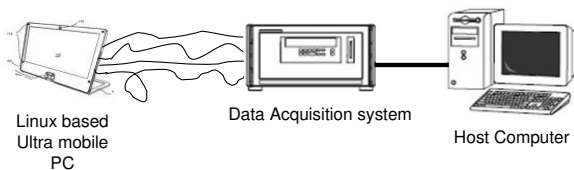
Figure 1: Experimental Setup

The experimental setup is shown in Figure 1 and the system configuration used for the UMPC is shown in Table 1.

The following experiments were performed:

- Idle state

- Audio playback over NFS

- Audio playback over BlueFS

- Rich Media (high-definition movie) playback over NFS

- Rich Media (high-definition movie) playback using BlueFS using local hard drive as cache

- Rich Media (high-definition movie) playback using BlueFS using USB drive as cache

4 Observations and File System Design Recommendations

Based on our analysis of NFS and BlueFS, here are a few observations:
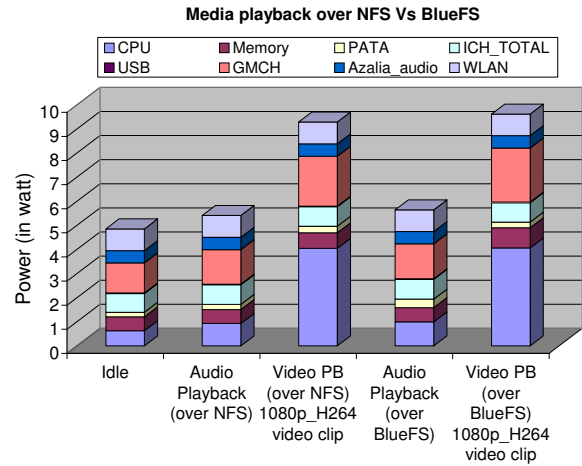
Figure 2: Performance of Media playback over NFS and BlueFS over the wireless link

1. Idle power on the platform was surprising higher, around 4.5W (*Figure 2*), as compared to other OSes which were around 3.2W; a closer look at the data reveals that the idle power for the WLAN component is as same as that of power it consumed when its exercising the work load, leading us to believe that we had an inefficient driver.

2. Another interesting aspect of the data was that the CPU power drawn when the system was idle was about 0.5W and a closer look at the processor's C3 residency state revealed that the CPU has C3 residency of only about 75%. Ideally this should be around 90–95% to get maximum platform idle power. We believe this is a critical piece to resolve to get better power for Linux based platform.

3. In addition to the impact of CPU residency, it is very important that all the devices in the platform support a low power idle state. As seen in our observations above, an inefficient device driver can spoil overall platform idle power and we need to make sure we have well optimized devices and drivers for a low ideal power. In the observation above, if we consider the ideal idle and transmit power for WLAN device, the idle power of the platform improves. This is shown in *Figure 3* which considers the projected ideal WLAN power and shows the performance of Media playback over NFS and BlueFS.

4. **NFS power bottlenecks:** The WLAN power consumption is among the top 3 in the list, next only to CPU and GMCH. This is as expected, all the

Component	Configuration
CPU	Intel® Core Duo 800 MHz
Memory	512 MB DDR2
ICH	ICH7
GMCH	Intel® 945GM Express Chipset Family
Audio	Intel® High Definition Audio
WLAN	Intel® 3945 ABG Network Connection

Table 1: Configuration of Linux-based Ultra-Mobile PC used for experimental analysis

NFS transactions are network centric. Assuming the network stack and the device are well optimized for the power, better platform power saving can be attained through file system specifically designed for low power.
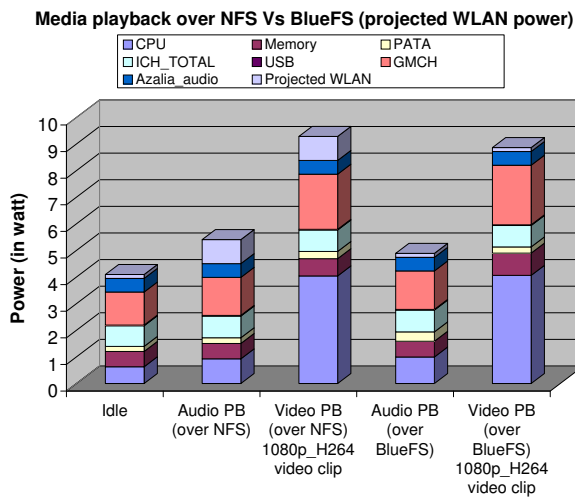
Figure 3: Performance of Media playback over NFS and BlueFS (with projected WLAN power)

4.1 BlueFS impact on Platform Power

1. BlueFS did show more effiency in terms of platform power for the given workload, as compared to NFS. The latency did come down drastically for BlueFS and this clearly brought down the net power consumed by the platform for a given workload to complete.

2. Caching mechanisms also helped in improving the user experience, and overall usage scenario. However, the choice of the caching device does have an impact on the system power. For example, from *Figure 4* BlueFS running over a local disk as cache

consumed less power compared to USB drive as cache, suggesting that if we have an optimized local solid state drive like Robson flash drive [6] as cache for BlueFS will have significant impact on the platform power.

3. We did not see any significant impact on the write performance as compared to NFS though one possible optimization could be to re-evaluate the write to many strategy in terms of power efficiency.
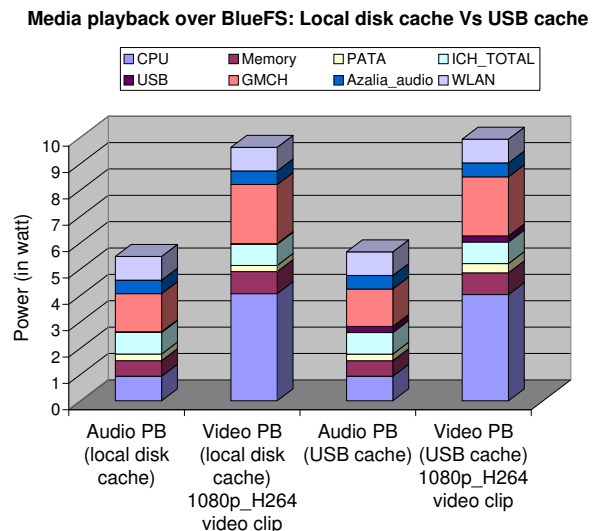
Figure 4: Performance of Media playback over BlueFS using local disk cache and USB cache

4. We also observed that while the Wolverine user daemon dynamically selects the device to read or write on run time, it does not do anything to the devices that are not being used. One possible power saving option would be to modify Wolverine to power down the devices that are not being selected for read or write operation.

5. Caching being a critical piece of BlueFS efficiency, proper focus needs to be made on the cache hierarchy selection. Other power efficient caching technologies like co-operative caching needs to be explored for more optimized power saving.

5 Summary and Future Work

Ultra-Mobile PCs present a new class of challenges for system designers since they are perceived to be high performance, low power devices and yet with the full functionality of larger handhelds/laptops. User experience of being ubiquitously connected and access to data anytime and anywhere is a fundamental requirement for this class of devices. However, access to data across wireless interfaces has a direct impact on battery life of such handheld devices.

This paper presented detailed analysis of the Network File System (NFS) and Blue File System (BlueFS) for specific usage scenarios targetting network data access across wireless interfaces and makes. Based on the power and performance analysis of these file systems, we made some key design recommendations to designers of file systems for Linux-based Ultra-Mobile PCs.

We plan to extend this work further in the following areas:

1. Evaluate BlueFS for multiple storage devices—USB, newer storage mechanisms like Robson Flash memory recently introduced on Intel's newest platforms. We believe that although BlueFS has an excellent mechanism for caching data, the choice of the device will be important in total platform power consumption.

2. Evaluate other wireless interfaces and usage models, specifically mobile digital TV, IP TV, since such scenarios would be critical for ultra-mobile PCs.

6 Acknowledgements

References

[1] J.J. Kistler and M. Satyanarayanan, *Disconnected operation in the Coda File system*, ACM Transactions on Computer Systems, 10(1), February, 1992.

[2] Network Working Group, *NFS: Network File System*, Protocol specification, March 1989. RFC 1094.

[3] Edmund B. Nightingale and Jason Flinn, *Energy-Efficiency and Storage Flexibility in the Blue File System*, Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, CA, December, 2004.

[4] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, *Scale and performance in a distributed file system*, ACM Transactions on Computer Systems, 6(1), February, 1988.

[5] T. Heath, E. Pinheiro, and R. Bianchini, *Application-supported device management for energy and performance*, In Proceedings of the 2002 Workshop on Power-Aware Computer Systems, pages 114–123, February, 2002.

[6] Michael Trainor, *Overcoming Disk Drive Access Bottlenecks with Intel Robson Technology*, Technology at Intel Magazine, Volume 4, Issue 9, December, 2006.

Where is your application stuck?

Shailabh Nagar
IBM India Research Lab
`nagar1234@in.ibm.com`

Balbir Singh   Vivek Kashyap   Chandra Seetharaman

Narasimha Sharoff   Pradipta Banerjee

IBM Linux Technology Center

`{balbir@linux.vnet,vivk@us,sekharan@linux.vnet}.ibm.com,`
`nsharoff@beaverton.ibm.com, bpradipt@in.ibm.com`

Abstract

Xen migration or Container mobility promise better system utiliztion and system performance. But how does one know if the effort will improve the workload's progress? Resource management solutions promise optimal performance tuning. But how does one determine the resources to be reallocated and the impact of the allotment? Most customers develop their own benchmark that is used for purchasing a solution, but how does one know that the bottleneck is not in the customer benchmark?

Per-task delay accounting is a new functionality introduced for the Linux kernel which provides a direct measurement of resource constraints delaying forward progress of applications. Time spent by Linux tasks waiting for CPU time, completion of submitted I/O and in resolving page faults caused by allocated real memory, delay the forward progress of the application being run within the task. Currently these wait times are either unavailable at a per-task granularity or can only by obtained indirectly, through measurement of CPU usage and number of page faults. Indirect measurements are less useful because they make it harder to decide whether low usage of a resource is due to lack of demand from the application or due to resource contention with other tasks/applications.

Direct measurement of per-task delays has several advantages. They provide feedback to resource management applications that control a task's allocation of system resources by altering its CPU priority, I/O priority and real memory limits and enable them to fine tune

these parameters more quickly to adapt to resource management policies and application demand. They are also useful for accurate metering/billing of resource usage which is particularly useful for shared systems such as departmental servers or hosting platforms. For desktop users, these statistics provide a quick way of determining the resource bottleneck, if any, for applications that are not running as fast as expected.

In this paper, we describe the design, implementation and usage of per-task delay accounting functionality currently available in the Linux kernel tree. We demonstrate the utility of the feature by studying the delay profiles of some commonly used applications and how their resource usage can be tuned using the feedback provided. We provide a brief description of the alternative mechanisms proposed to address similar needs.

1 Introduction

Linux is supported on a wide range of systems, from embedded to desktop to mainframe, running a diverse set of applications ranging from business and scientific to office productivity and entertainment. Linux provides multiple metrics and "knobs" for achieving optimal performance in these various deployments. There are a myriad of well known benchmarks and tools available to help gather and analyze and then tune the systems for desired performance.

However, after elaborate tuning as well the applications or system may still perform unsatisfactorily. The resource bottlenecks blocking the progress must be addressed through additional provisioning, resource real-

location or even load-balancing by migration of the application/service to another system.

There is on-going work such as resource groups and 'containers' to re-allocate resources to desired workloads for better system utilization and performance. Xen and Linux containers support the notion of migrating live workloads to consolidate systems when load is low and migrate to other systems for better load-balancing. However, all these mechanisms will work 'blind' without a good method to pinpoint the resources that need tuning.

The delay-accounting framework is built on the observation that: since ultimately the application's progress is most important, the bottlenecks impeding the applications progress must be easily identifiable. The delay-accounting framework therefore gathers the time spent on behalf of the task (or task group) in queues contending for system resources. This information may then be utilised by workload management applications to dynamically increase the desired applicaton's access to the bottlenecked resource by raising its priority, or share of the resource pool, or even initiating service or application migration to a different system.

This paper discusses the design, implementation and utility of the per-task delay accounting framework developed to address this issue. The following sectons outline the use cases

2 Motivation

The traditional focus of operating system accounting is the time spent by a Linux task doing a particular activity e.g. time spent by a task executing on a cpu in system mode, user mode, in interrupts etc. There is also a measure of the system activity that happens e.g. number of system calls, I/O blocks transferred, network packets sent/received. While these metrics are often useful for measuring overall system utilization or for high level detection of performance anomalies, they are not directly usable for determining what is delaying forward progress of a specific application, except perhaps by skilled and experienced system administrators.

Hence there is a need for answering the simple question: what resource, managed by the operating system, is my application waiting for? To start with, the resources of interest can be high level such as cpu time, block I/O

bandwidth and physical memory. These, respectively, translate to knowing how long a Linux task (or groups of tasks) spend in, i.e. get delayed, in

- waiting to run on a CPU after becoming runnable waiting for block

- I/O operations inititated by the task to complete waiting for page

- faults to complete

The information is useful in at least two distinct scenarios:

Simple hand tuning: If a favorite task seems to be spending most of its time waiting for I/O completion, raising its I/O priority (compared to other tasks) may help. Similarly, if a task is mainly waiting on page faults, increasing its RSS limit (or running other memory hogs at a much lower cpu priority) may help alleviate the resource bottleneck.

Workload management: One of the objectives of modern workload management tools is to manage the forward progress of aggregates of tasks which are involved in a given business function. The idea being that if business function A is more important than another one B, the applications (tasks and processes from an OS viewpoint) involved in the former should get preferential access to the OS resources compared to B.

To achieve this, workload managers need to periodically know the bottlenecked resource for all tasks running on the system and use that information, in conjunction with policies that determine prioritization, to adjust the resource priorities like nice values, I/O priorities and RSS limits. Workload managers may also make a decision to shift a given application off a system if it determines that priority boosting isn't helping.

Per-task delay accounting was designed and implemented to meet the above needs. We next describe the design considerations in detail.

3 Design and implementation

The design and implementation of per-task delay accounting had to take the following three major factors into consideration.

3.1 Measurement of delays

The design objective was to measure the most significant sources of delay for a process to the highest accuracy possible, preferably at nanosecond granularity. The approach taken was to take high resolution timestamps at the appropriate kernel functions and accumalate the timestamp differences into per-process data structures.

The delay sources chosen and the manner in which they are collected are:

1. Waiting to run on a cpu after becoming runnable: Here we needed to take timestamps when a process was added to a cpu runqueue and again when it was selected to be run on a cpu. The schedstats codebase, added to help gather cpu scheduler statistics for development and debugging purposes, came in handy since it already had code to gather these timestamps and take their differences. The schedstats functions `sched_info_arrive` and `sched_info_depart` which gathered other statistics that were not of interest to delay accounting had to be refactored to minimize the performance impact. Another decision made to keep the performance impact low was to stick to the jiffie level timestamp resolution used by schedstats instead of higher resolution ones available in the kernel.

2. Waiting for block I/O submitted to complete: Here we had a choice of trying to accurately measure the entire delay in submitting block I/O as well as the delay incurred in the block device performing the I/O. However, it was finally decided that we would measure only the time spent by a process in sleeping for submitted I/O to complete. Measuring the delays in the I/O submission path as well turned out to be quite complex, given the diversity of functions that are involved in block I/O submission as well as the difficulty of correctly attributing the delays to the right process in all these paths. These factors would have necessitated a number of timestamps being collected all over the kernel code which affects maintainability. Moreover, much of the delays seen in I/O submission cannot be changed by the user (other than indirectly by affecting the cpu scheduling of the submitting process) whereas delays incurred after I/O submission can be affected by tweaking the I/O scheduling priority

of the process. Hence it was decided to only measure the time spent in `sched.c/io_schedule()` using high resolution timestamps.

3. Waiting for a page fault to complete We had briefly considered measuring the delays seen in both major and minor page faults but later concentrated only on the former to minimize impact of delay collection code on the virtual memory management code paths. Delays for major page faults are a subset of the block I/O delays which were already being measured so the only change needed was to record the fact that a process was in the middle of a major page fault and use this information at block I/O delay recording time.

Delay accounting also measures and returns an interval-based measurement of time spent running on a cpu. This is more accurate than the sampling-based cpu time measures normally available from the kernel (via a task's utime, stime fields in task struct). Accurate cpu usage times are valuable in accurate workload management. One factor that has to be considered for cpu time is the possibility of the kernel running as a guest OS in a virtualized environment. These "guest OSes" are scheduled by the hypervisor in accordance with its scheduling policies and priorities for the OS instance. As a result, the times spent waiting for a particular resource need to be measured in wall-clock times. In contrast the the utilization of the resource occurs only when the Guest is executing. This differentiation enables an educated assessment of resource allocation (such as additional CPU share) needed by the "guest" or the resources within the the "guest OS."

The delay accounting framework retuns both the real and virtual cpu utilization on virtual systems (currently implemented for LPARs on ppc64) as well as the delays experienced by the individual task groups in the OS instances.

3.2 Storing the delay accounting data

Each of the delay data recorded above needed to be stored taking into account two important constraints:

1. The data structures should be expandable to add other per-task delays that were deemed useful in

future. While the current per-task delay account-
ing only looks at wait for cpu, block I/O and ma-
jor page fault delays, it is well possible that other
delays associated with kernel resource allocation
would be of interest and measured in future.

2. The data should be collected per-task and also ag-
gregated per-process (i.e. per-tgid) within the ker-
nel. This latter design constraint drew a lot of
comments when the delay accounting code was
proposed since it was felt that per-process delays
could as well be accumulated outside the kernel in
userspace. However, we had observed that accurate
measures of per-process delays were very useful
for performance analysis of bottlenecks and also
that accumalating per-task delays in userspace re-
quired far too much overhead and reduced accuracy
due to presence of short-lived tasks in a process.
Hence it was necessary to have a per-process delay
aggregating data structure that would keep the de-
lay data collected for an exiting task and make it
available until all tasks of the process exited.

The taskstats data structure, defined in `include/`
`linux/taskstats.h`, took into account these
constraints. It was versioned, mandated that new
fields would be added at the bottom and also took
into account alignment requirements for 32 and 64
bit architectures.

To meet the second constraint, two copies of
the data structure are maintained. One is per-
task, maintained within the `task_struct`. The
other is per-tgid, maintained within the `signal_`
`struct` associated with a tgid. The fields of struct
taskstats are large enough to serve as an accumala-
tor for per-tgid delays.

3.3   Interface to userspace to supply the data

The interface to access delay accounting data from
userspace formed a large part of the discussion preced-
ing the acceptance of delay accounting in the kernel.
The various design constraints for the interface were:

1. Efficient transfer of large volumes of delay data:
Workload managers that are monitoring the entire
system for performance bottlenecks need to gather
delay statistics from each task periodically. In or-
der to allow this period to be small, it is essential

that the data transfer of delay accounting data be
efficient while handling large volumes (due to po-
tentially large number of tasks). A similar require-
ment, albeit less severe, exists even for monitoring
a single application if its degree of multithreading
is sufficiently high.

2. Handle rapid exit rate without data loss: In order
to do workload management at the level of user-
defined groups of processes, workload managers
need to get the cumulative delays seen by a task
(and a thread group) right up to the time it exits.
Hence delay accounting data needs to be available
even after a task (or thread group) exits. Obviously
the kernel cannot keep such data around for a long
time so the choice was made to use a "push" model
(kernel->user) to send such data out to listening
userspace applications rather than require them to
"pull" the data. In such a scenario, its important to
be able to handle a rapid rate of task/thread group
exits, if not on a sustained basis, atleast for short
bursts, without losing data.

3. Exporting data as text or not: This is another
instance of the classic debate whether such data
should be exported as text through /proc or /sysfs
like interfaces allowing it be directly read in user
space using shell utilities or whether it should be
a structured binary stream requiring special user
space utilities to parse. Given the volume of data
needing to be transferred, the latter option was cho-
sen.

4. Bidirectional read/writes: There was a need for
userspace to send commands and configuration in-
formation to the kernel delay accounting and hence
interfaces like relayfs which lacked a user->kernel
write capability were not usable.

Given these constraints, we decided to use the newly in-
troduced genetlink interface. Genetlink is a generalized
version of the netlink interface. Netlink, which exports
a sockets API, has been used to handle large volumes
of kernel<->user data, primarily for networking related
transfers. But it suffered the limitation of having a lim-
ited number of sockets available for use by different ker-
nel subsystems as well as an API that was too network-
centric for some. Genetlink was created to address these
issues. It multiplexes multiple users over a single netlink
socket and simplifies the API they need to use to effect

bidirectional data transfer. Delay accounting was one of the early adopters of the genetlink mechanism and its usage provided inputs for refining and validating the genetlink interface as well.

Delay accounting handles the rapid exit rate constraint by splitting the delay data sent on exit into per-cpu streams. A listening userspace entity has to explicitly register interest in getting data for a given cpu in the system. Once it does so, it receives the exit delay data for any task which exits while last running on that cpu. This design allows systems with many cpus (which will typically have a correspondingly larger number of exiting tasks) to balance the exit data bandwidth amongst multiple userspace listeners, each listening to a subset of cpus. CPUs are used as a convenient means of dividing up the exit data requirements. They also help when the cpusets mechanism is used to physically partition up machines with very large number of cpus and some of the physical partitions have strict performance requirements that prohibit exit data from being processed. Being able to regulate the sending of exit delay data by the kernel by cpu allows fine-grain control over the performance impact of delay accounting.

Finally, virtual machine technology enables a single system to run multiple OS instances. These "guest OSes" are scheduled by the hypervisor in accordance with its scheduling policies and priorities for the OS instance. As a result, the times spent waiting for a particular resource need to be measured in wall-clock times. In contrast the the utilization of the resource occurs only when the Guest is executing. This differentiation enables an educated assessment of resource allocation (such as additional CPU share) needed by the "guest" or the resources within the the "guest OS."

The delay accounting framework retuns both the real and virtual cpu utilization on virtual systems (currently implemented for LPARs on ppc64) as well as the delays experienced by the individual task groups in the OS instances.

3.4 Delay accounting lifecycle

With the above elements in place, its useful to outline how the delay accounting works in practice. The description is being kept generic without referring to specifics of the interface since that can be obtained elsewhere.

On system startup, the system administrator can optionally start userspace "listeners" who register to listen to exit delay data on one or more cpus. The typical usage is to start one listener listening to all the cpus of the system.

When a task is created (via fork), a taskstats data structures get allocated. If the task is the first one of a thread group, a per-tgid taskstats struct is allocated as well. As the task makes system calls, the delays it encounters get measured and aggregated into both these data structures.

At any point during the task's lifetime, a user can query the delay statistics for a task, or its thread group, via the genetlink interface by sending an appropriate command. The reply contains the taskstats data structure for the task or thread group.

When the task exits, its delay data is sent to any listener which has registered interest in the cpu on which the exit happens. If the task is the last one in its thread group, the accumalated delay data for the thread group is additionally sent to registered listeners.

3.5 Per Task IO Accounting

An important related work are the per-task I/O accounting statistics by Andrew Morton which improve the accuracy of measurement of I/O resource consumption by a task. The CSA infrastructure also supports per-task IO statistics, but the data returned by it, can be incorrect.

CSA accounts for per task IO statistics using data read or written through the `sendfile(2)`, `read(2)`, `readv(2)`, `write(2)`, and `writev(2)` system calls. It does not account for

1. Data read through disk read ahead

2. Page fault initiated reads

3. Sharing data through the page cache. If a page is already dirty and another task T1, writes to it, both the task that first dirtied the page and T1 will be charged for IO.

The following example illustrates the deficiencies of the current per task IO accounting infrastructure. We wroter a sample test application that maps a 1.2GB file and writes to 1GB of the mmap'ed memory. The output below shows the output of `/proc/<pid of test>/io`

```
rchar: 1261
wchar: 237
syscr: 6
syscw: 13
read_bytes: 1048592384
write_bytes: 1317117952
cancelled_write_bytes: 0
```

The gathered statistics indicate that the existing accounting of rchar and wchar present in CSA, failed to capture the IO that had taken place during the test.

The ideal place to account for IO, is the IO submission routine, `submit_bio()`. This works well for accounting data being read in by the task. However, for write operations, we need some place else to account for the following reasons

Writes are usually delayed, which means that it is hard to track the task that initiated the write. Furthermore, the data might become available long after the task has exited.

Write accounting is therefore done at page dirtying time. The routines `__set_page_dirty_nobuffers()` and `__set_page_dirty_buffers()` account the data as written and charge the current task for IO.

A task can also actually cause negative IO, by truncating pagecache data, brought in by another task. Instead of accounting for possible negative write IO, the data is stored in the field called `cancelled_write_bytes`.

Figures 1, 2, and 3 show the call flow graph for read, write, and truncate accounting, respectively.

Communicating the per task IO data to user space is fairly straight forward. It uses the per task taskstats interface for this purpose. The taskstats structure has been expanded to new fields corresponding to the per task IO accounting information. The taskstats interface automatically takes care of sending this data to user space when either a task exits or information is requested from user space.

4 Case studies of performance bottleneck analysis

The following two case studies, taken from a performance analyst who used delay accounting to debug performance issues, illustrate the utility of the mechanism.
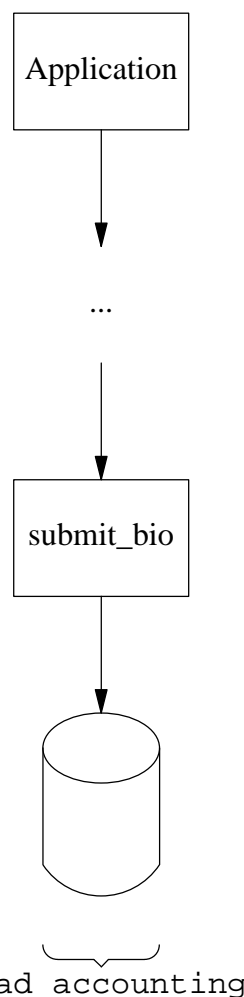
Figure 1: Read IO Accounting

1. Analysing performance issue with an MPI based parallel application:

   This is a brief on how per-task delay accounting was used to find out the root cause for a performance problem of an application running on homogeneous cluster. During the user acceptance test for the cluster this application successfully completed its run within the expected time. However some days later the same application was taking too much time to complete. Nothing changed in the cluster wrt to the configuration. However one strange thing was noted - when the cluster was compleletely isolated from the public network the application completed its run within the expected time. The problem happened only when the cluster was open for use by everyone concerned. This was an important lead but we didn't have any clue on how to proceed. The cluster was pretty large
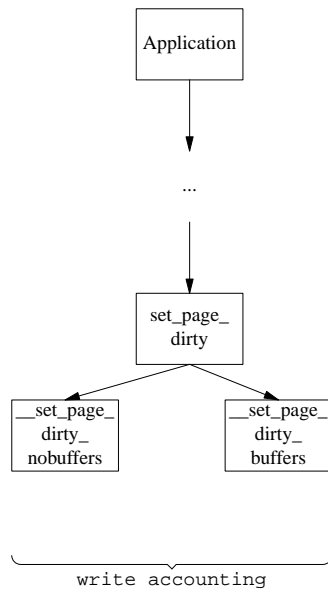
Figure 2: Write IO Accounting

Figure 3: Truncate IO Accounting

and analysing each and every node manually was a tedious task.

Some of the possible things to look for were network communication delays, problems with the job scheduler's resource reservation functionality and system configuration of all the nodes (just to be sure that nothing was changed in between). Even after doing all these activities we were not able to identify the root cause of the problem. We didn't have a clue whether the issue was with any particular node/s or with the entire cluster.

Per-task delay accounting came to our rescue here. We asked the customer not to isolate the cluster and let it be used like in any normal day. The application was run on all the nodes in the cluster and subsequently the getdelays program was run so as to get the delay accounting statistics for this particular application. After completion of the run, the delay accounting statistics from all the nodes were compared and it was found that there was huge IO and CPU delay on one of the nodes in the cluster.

This was affecting the overall performance (execution time) of the application. We then focussed our attention on this particular node. Eventually after monitoring this node for some time, we found that one of the users was directly running interactive jobs on this particular node. This in turn pointed to a security hole in the customer's overall cluster

setup which allowed this user to bypass the access control restrictions put in place and run the jobs directly on a particular node.

Per task delay accounting made the job of isolating the problem a lot easier. An added advantage is that of convincing the customer becomes a lot easier and effective.

Current per task delay accounting can be taken a step forward by including per task network IO delay accounting statistics also and writing a tool based on per-task delay accounting for identifying performance problems in a cluster.

2. There was another instance where per-task delay accounting helped us in dealing with a customer. The customer had a single threaded program on a non-Linux platform. The application was later made multi-threaded on the same platform. The program typically read X number of records in n seconds.

The customer ported the multi-threaded version of the program to Linux (using C programming language). The ported program was reading significantly less number of records in n seconds when compared to the original program on the non-Linux platform (hardware

configuration was same, only the OS was different). Tools like top, vmstat, iostat were not very conclusive. Since source code access for the program was not there, we asked the customer to provide us with both single threaded and multi-threaded version of the program along with the timings and used per-task delay accounting to get the delay stats for both the single-threaded and multi-threaded version of the program. We found that for the multi-threaded version IO delay was on a higher side when compared with the single-threaded version. Pointed this out to the customer (as source code access was not there). Convinced them that they need to relook at the application code.

Per task delay accounting helped us to get specific data pertaining to the multi-threaded program in question.

5   Summary

Detecting the bottlenecks in resource allocation that affect an application's performance on a Linux system is an increasingly important goal for workload management on servers and on desktops. In this paper, we describe per-task delay accounting, a new functionality we contributed to the Linux kernel, that helps identify the resource allocation bottleneck impeding an applications forward progress. We also describe other important related work that improves the accuracy of CPU and I/O bandwidth consumption. Finally we demonstrate how these new mechanisms help identify where applications can get "stuck" within the kernel.

Trusted Secure Embedded Linux

From Hardware Root Of Trust To Mandatory Access Control

Hadi Nahari

MontaVista Software, Inc.

`hnahari@mvista.com`

Abstract

With the ever-increasing presence of Linux implementations in embedded devices (mobile handsets, set-top boxes, headless computing devices, medical equipments, etc.), there is a strong demand for defining the security requirements and augmenting, enhancing, and hardening the operating environment. Currently an estimated 70% of new semiconductor devices are Linux-enabled; such a high growth is accompanied by inevitable security risks, hence the requirement for hardware-based trusted and secure computing environment, enhanced with MAC (Mandatory Access Control) mechanisms for such devices in order to provide appropriate levels of protection. Due to stringent security requirements for resource-constrained embedded devices, establishing trust-chain on hardware root of trust, and deploying MAC mechanisms to balance performance and control are particularly challenging tasks.

This paper presents the status of MontaVista Software's efforts to implement such solutions based on ARM cores that provide separated computing environment, as well as SELinux (Security Enhanced Linux) to provide MAC for embedded devices. The focus will be on practical aspects of hardware integration as well as porting SELinux to resource-constrained devices.

1 Introduction

The defining line between embedded and non-embedded systems is becoming more and more blurred [2]; whether the system has an elaborate UI (User Interface), or if it is operating under a resource-constrained environment are not sufficiently delineating identifiers anymore.

The availability of more computing power at a low cost has also resulted in developers' and manufacturers' interest in adding more functionality to devices that were traditionally either not capable of, or not expected to have them (smart phones, hand-held computing devices, complex medical control systems, navigation gear, etc.). The difficulty in satisfying the security requirements of a software package is proportional to its design complexity. The default access control method in Linux, that is, DAC (Discretionary Access Control) is at best considered *insufficient* for any security-sensitive implementation, hence the increase in demand for MAC in recent years.

HAS (Hardware Assisted Security) has not yet been sufficiently standardized and from the industry adoption perspective, it is still considered to be in its infancy. Yet (or however), HAS is one more item in the toolbox of security architects who need to provide solutions for fundamental problems such as establishing TCB (Trusted Computing Base) using hardware-based root of trust, managing key material, and security governance of the system.

In the past years, there has been an increasing growth in adoption of Linux in various environments. This important phenomenon, which is unparalleled by other operating systems, has resulted in Linux becoming more and more the de-facto operating system for embedded devices. The recent acceleration of this adoption by said-devices is partially due to the highly modular architecture of Linux kernel, and partially due to its maturity, which results in lower COO (Cost Of Ownership.)

One must note, however, that this growth is accompanied by complex and elaborate software solutions build atop embedded Linux devices to address the ever increasing demands of the market, and hence the complex security requirements of such devices. This makes implementing a holistic security strategy for Linux more challenging, especially when one considers the highly varied selection of embedded devices adopting, or planning to adopt Linux; from smart phones, hand-held de-

vices, set top boxes, high-end televisions, medical devices, automotive control, navigation systems, assembly line control devices, missile guidance systems, to the other end of the spectrum such as CGL (Carrier Grade Linux) in the telecoms industry. Such environments each possess a very unique set of security characteristics and requirements, and therefore provide different challenges. Figure 1 shows a typical Linux-based mobile phone architecture.

In this article I will start by describing the most common security requirements in the embedded industry, and follow it by explaining the design principles and reviewing the available technologies that were initially considered viable, both for HAS and MAC, and will propose an architecture based on the selected technologies. Where applicable, I will point out whether the said-method or technology satisfies a subset of the embedded systems (mobile handsets and CGL).

The main focus of this article is to provide:

1. A high-level, architectural overview of a design that provides fundamental and necessary facilities to establish the trustworthiness of the operating system services (via connection to a hardware-based root trust).

2. A mechanism to establish *Effective Containment* (that is, a mechanism to prevent an exploited application from enabling attacks on another application possible) via the MAC offered by SELinux.

Security services provided by higher-level software constructs, such as middleware and frameworks alike are not the focus of this article and will not be discussed.

It is also noteworthy to mention that, where the required security related components exist in the underlying hardware architecture, the proposed design is viable for non-embedded systems as well.

2 Design Constraints

The following are the most commonly considered constraints when designing and developing software components for an embedded device:

1. Memory Footprint

2. Performance Trade-off

Memory footprint is important because a big majority of the embedded devices tend to have limited memory available at run-time. Any security solution for such devices must therefore have an acceptable and low memory footprint.

Performance trade-off is important because the computing power available in an embedded environment is typically low, due to hardware architecture characteristics, as well as issues pertaining power management in battery operated devices. Low power consumption is one of the fundamental reasons based on which ARM is the de-facto architecture for such devices.

3 Design Principles

1. Simplicity

2. Modularity

We have made the design as simple as possible. This is to ensure no unnecessary complexity is introduced into the system and also the overall security analysis of the system is easier to perform.

The proposed architecture is modular. This is to ensure that the design could also be implemented on hardware architectures that lack the security capabilities introduced, and also environments where the types of attacks, or the security assets of the system would not require the high degree of protection provided by this design.

4 Technology Overview

4.1 Secure Boot

Secure Boot (a.k.a. High Assurance Boot) is a technique for verifying and asserting the integrity of an executable image prior to passing the control to it. Assuming the verification mechanism is based on the digital signature of the image being verified, then the reliability of this verification is at best as good as the reliability of the protection mechanism provided in the device for the public key of the image signer.

The most important assumption here is that the code which performs the integrity verification process is itself
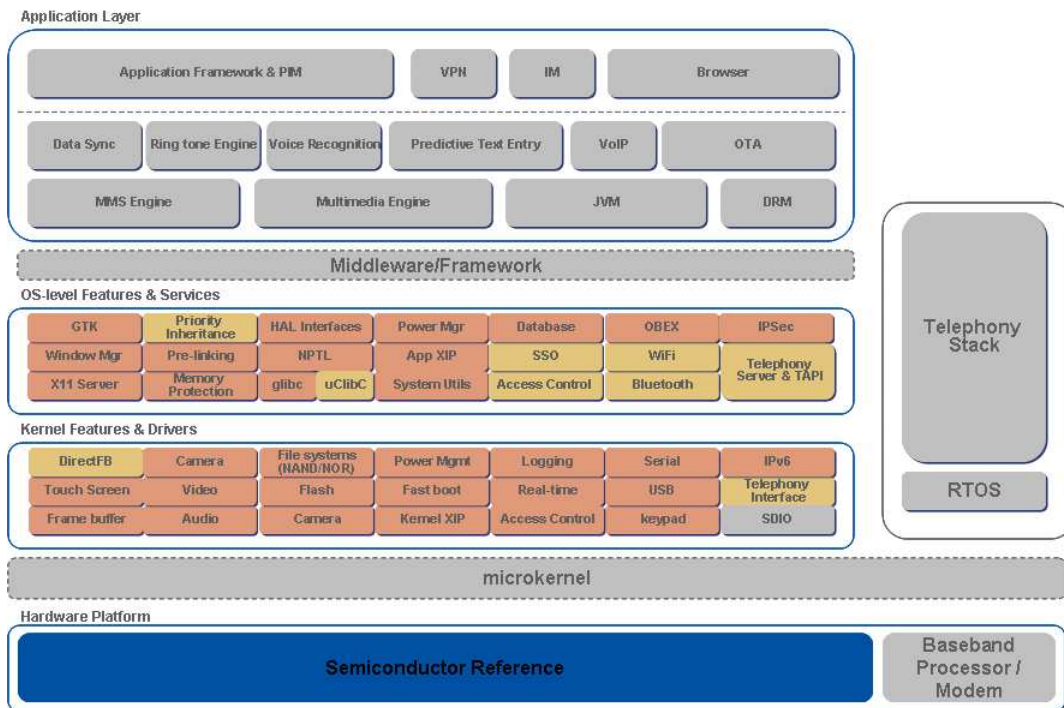
Figure 1: A Typical Linux-based Mobile Phone Architecture

trustworthy. To assert this assumption, the implementations typically put the public key material, as well as the verifier code into non-writable areas of memory, which in turn are protected via a form of hardware protection mechanism. Figure 2 shows a generic Secure Boot architecture.

This design enables the establishment of a chain of trust by ensuring that the trust, on each layer of the system, is based on, and is only based on, the trust on the layer(s) underneath it, all the way down to the hardware security component, which serves as the *Root Of Trust*. If verification fails to succeed at any given stage, the system might be put in a suspended-mode to block possible attacks.

One must note, however, that this architecture, though ensuring the integrity of the operating environment when a *hard boot* occurs, does not guarantee its integrity during the runtime; that is, in case of any *malicious* modification to the operating environment during the runtime, this architecture will not detect it until the next hard boot happens.

4.2 Effective Containment

The "Buffer Overflow" class of attacks is practically impossible to prevent in native environments with no type- or boundary-checking available at runtime. Executing native code in operating environments like Linux makes it specifically susceptible to this category of attacks. This therefore makes exploiting a buffer overflow attack of particular interest to hackers, and successfully mounting such an attack is considered a badge of honor. *Effective Containment*, in this context, is therefore referred to a class of techniques that *contain* (as opposed to *prevent*) such attacks for which there are no *practical* prevention mechanism available. This could be achieved via the use of various software and security technologies. Applying a MAC mechanism is one way to implement effective containment.

4.2.1 Embedded SELinux

One method to achieve a MAC is via implementing RBAC (Role-Based Access Control). NSA's SELinux, among other features such as MLS (Multi Level Security), provides Linux with MAC through RBAC.

Figure 2: A Typical Secure Boot Design

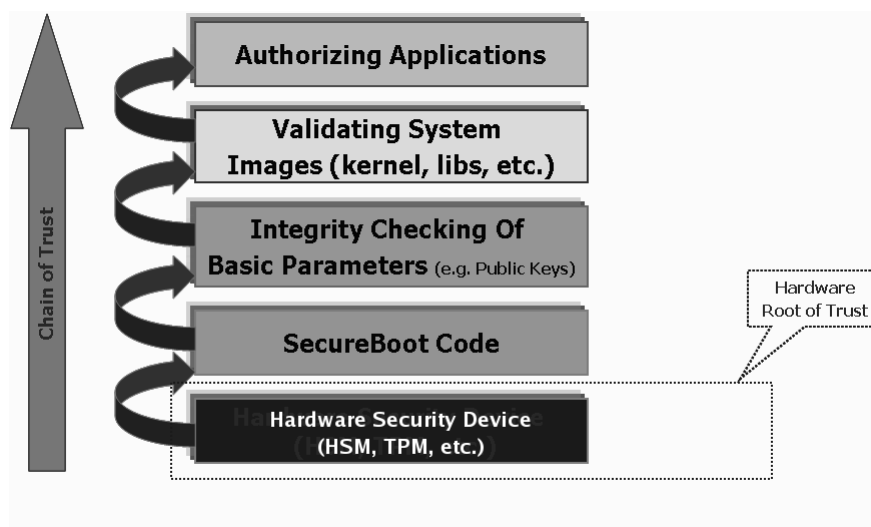SELinux was not originally designed for the ARM architecture, or for embedded devices. There have been, however, previous *and reasonably successful* attempts to port SELinux or parts of it into an embedded device and on an ARM architecture; the most notable of which being by Russell Coker [1].

By adding a MAC mechanism such as SELinux on top of Secure Boot, we will be able to address one of its fundamental shortcomings; providing a level of protection at runtime. Figure 3 shows an architecture, deploying Secure Boot and MAC mechanisms together.

In this design, not only have we accomplished augmenting the Secure Boot mechanism (by way of providing runtime containment), but have also enabled a way to expose hardware-security capabilities (e.g. TPM standard services) to the applications and processes during the system runtime.

As a side note, it is important to mention that RBAC is not the only mechanism to implement MAC. Other implementations exist which might also become suitable for embedded devices; "Tomoyo Linux" and Novell's "AppArmor" are both examples of such solutions that implement a technique called NBAC (Name-Based Access Control). LIDS (Linux Intrusion Detection System) is another example. At the time of writing this article, however, neither of these implementations seem to have been able to gain the traction in the industry, nor by manufacturers of embedded devices to be the default MAC for such devices. This state, however, may change

in the future as the above-mentioned technologies mature.

4.2.2 Multi-core and Virtualization

In recent years, the industry has put more focus on adding more computing power to embedded devices, not only in adding more processing capabilities to each core, but also adding to the number of cores available in hardware architectures. One of the objectives of this expansion is have a dedicated hardware resource to each high-level task, and therefore achieving easier management of software design and implementation through hardware compartmentalization. This approach has resulted in recent growth in implementing virtualization technologies in embedded space. Various types of virtualization techniques exist (hardware- and software-based) which all provide multiple *guest domains*, each assuming total access to the underlying hardware, and conceptually having no awareness of the other guest domains. A fundamental element of any virtualization implementation is a layer called *hypervisor*, which is responsible for mediating the interactions among guest domains, and providing necessary life-cycle management for each guest domain.

As is in most cases, this technique has existed in large-scale enterprise systems prior to embedded space; however, again the implications of this technology in resource-constrained embedded devices are different

Leveraging "Root Of Trust" To Augment "Chain of Trust"

Authorizing Applications

Validating System Images (kernel, libs, etc.)

Integrity Checking Of Basic Parameters (e.g. Public Keys)

SecureBoot Code

Hardware Security Device (HSM, TSM, etc.)

Chain of Trust

Access Control

TPM Services

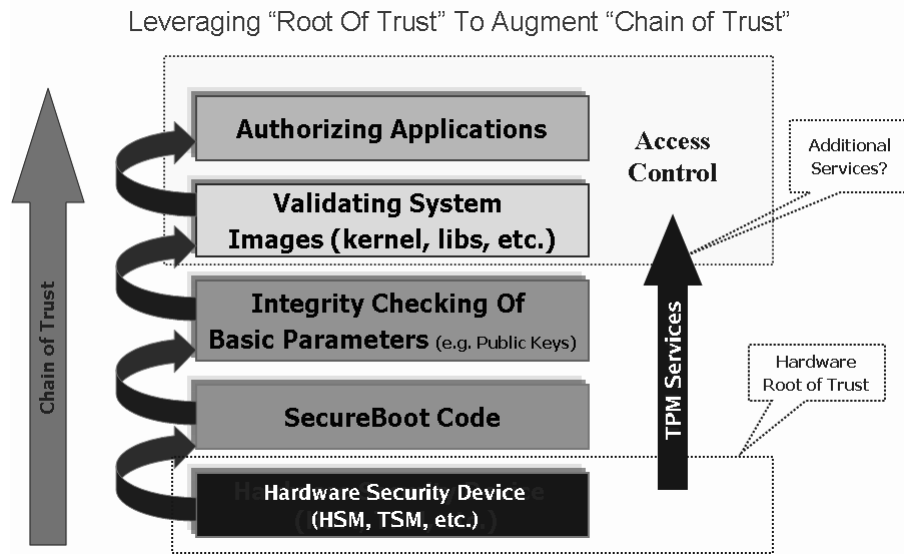Additional Services?

Hardware Root of Trust

Figure 3: Augmenting Secure Boot with Access Control

than those of the enterprise systems. The use cases, however, remain similar.

Any modern design that attempts to provide a security solution for embedded space, therefore needs to assume it might be contained in a virtual, guest domain.

4.2.3 ARM TrustZone Technology

TrustZone is a security technology introduced by ARM Ltd. in its "ARM 1176" core. TrustZone is a technology to provide a hardware-based separation for execution environment, and divide it into two halves; secure and normal worlds. The security-sensitive applications are run executed in a separate memory space which is not accessible to "normal applications." TrustZone is the first ARM architecture to provide hardware-based security in its core. Although *and if done right* this could potentially provide the operating environment with an additional level of security, at its core this could be considered a hardware-based virtualization solution, and is conceptually not a new idea. Furthermore compartmentalized-security and separation of execution environments due to application's security requirements, along with "separation of concerns," have all been well-understood and known concepts in computer science and software engineering for decades [3].

The proposed design in this article considers the availability of ARM TrustZone technology on the underlying

hardware architecture; however, no parts of this design *rely* on such capability. It is also important to mention that we only assume the TrustZone hardware availablity in the underlying architecture; analyzing the *proprietary* software stacks on top of TrustZone hardware that provide additional security capabilities to the applications, is outside the scope of this article.

5 Proposed Architecture

5.1 High-level Analysis

We propose the architecture shown in Figure 4 to address the security requirements discussed in this article.

This design is based on a hardware root of trust, and therefore could provide security as reliable as the method protecting this root. It implements a MAC mechanism based on embedded SELinux, and can do it in a virtualized environment. The proposed design deploys the security capabilities available in hardware (that is, TrustZone hardware) to enforce a separation mechanism among guest domains, via dedicating separate areas of memory to different processes that exist in each domain. The overall security management, such as secure IPC (Inter Process Communication) of such processes, is the responsibility of the VMM (Virtual Machine Monitor) which acts as the hypervisor in this design. This design enables a hardware-enforced separation among processes running in each guest domain,
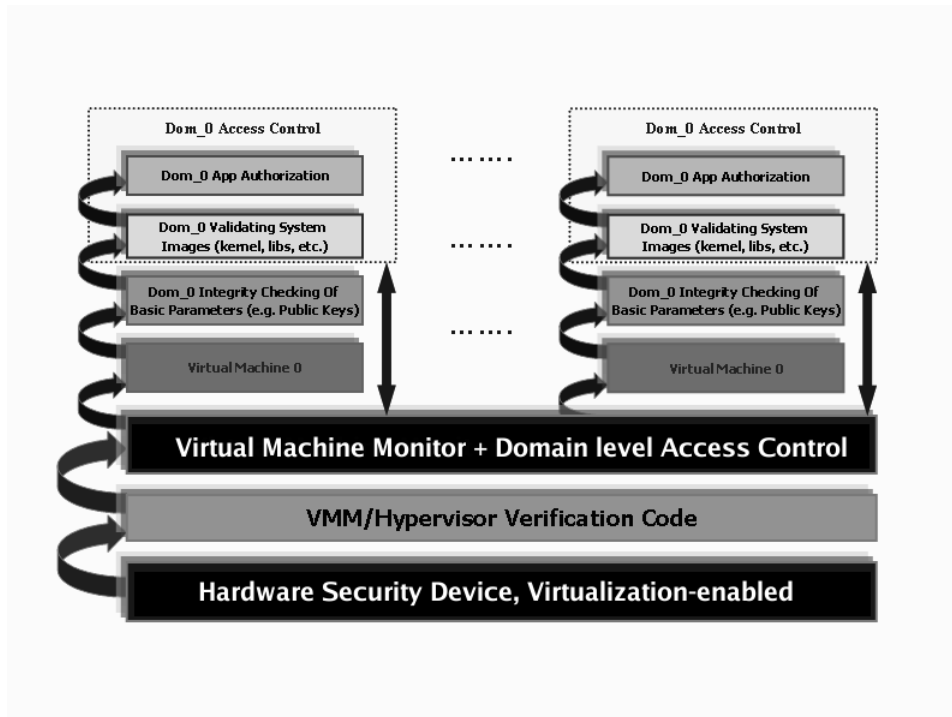
Figure 4: Implementing MAC and Virtualization, Based on Hardware Root Of Trust

with a fine-grained, mandatory access control mechanism provided by SELinux infrastructure. The initial verification of each guest domain happens prior to bringing it up. After each domain is on-line, ensuring its health from security perspective is also provided by the SELinux infrastructure.

It is presumed that due to differences in security requirements during the start-up and runtime, not all the embedded devices would require all the elements provided in this design. This, however, does not indicate a weakness in the architecture, as the main security aspects of the design could be implemented/enabled independently.

6   Conclusion

We proposed a design that deploys a hardware root of trust to provide secure execution of applications in a virtualized environment. We also augmented the design by adding a MAC mechanism to provide enhanced protection to applications and processes at runtime.

On an system which requires and implements all the capabilities provided in this architecture, a thorough analysis must be performed to ensure an appropriate security policy is in place to deploy the SELinux capabilities

efficiently; an unnecessarily comprehensive and restrictive policy has a potential to hamper the overall runtime performance, and increase the memory footprint of the system.

The performance of the hypervisor is also the key in this design, as it is the layer that arbitrates the interactions among the processes which exist in separate guest domains. A fast and high-performing hypervisor is the quintessential key to the successful implementation of this design.

7   Legal Statement

This work represents the personal views of the author and is a technical analysis of an architecture; it does not necessarily represent the views of MontaVista Software, Inc.

Furthermore, the author is not an attorney and makes no judgment or recommendation on legal (and specifically GPL) ramifications of the proposed design; such issues are outside the scope of this article, and should be dealt with via consulting with a GPL attorney/law practitioner.

2007 Linux Symposium, Volume Two • 85

Linux is the registered trademark of Linus Torvalds in the United States of America, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

References

[1] Russell Coker. Porting nsa security enhanced linux to hand-held devices. In *Proceedings of the Linux Symposium*. Ottawa Linux Symposium, July 2003.

[2] William R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the Bounds of Mobile Computing. *IEEE Computer*, 34(4):28–35, April 2001.

[3] Jorrit N. Herder, Herbert Bos, Andrew S. Tenenbaum. A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. Technical report, Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 2006.

Hybrid-Virtualization—Enhanced Virtualization for Linux*

Jun Nakajima and Asit K. Mallick
Intel Open Source Technology Center
`jun.nakajima@intel.com, asit.k.mallick@intel.com`

Abstract

We propose hybrid-virtualization that combines para-virtualization and hardware-assisted virtualization. It can achieve equivalent or better performance than software-only para-virtualization, taking the full advantage of each technology. We implemented a hybrid-virtualization Linux with para-virtualization, which required much fewer modifications to Linux, and yet achieved equivalent or better performance than software-only XenLinux.

The hybrid-virtualization employs para-virtualization for I/O, interrupt controllers, and timer to simplify the system and optimize performance. For CPU virtualization, it allows one to use the same code as in the original kernel.

The other benefits are: One, it can run on broader ranges of VMMs, including Xen and KVM; and Two, it takes full advantage of all the future extensions to hardware including virtualization technologies.

This paper assumes basic understanding of the Linux kernel and virtualization technologies. It provides insights to how the para-virtualization can be extended with hardware assists for virtualization and take advantage of future hardware extension.

1 Introduction

Today x86 Linux already has two levels of para-virtualization layers, including paravirt_t and VMI that communicate with the virtual machine monitor (VMM) to improve performance and efficiency. However, it is true that such (ever-changing) extra interfaces complicate code maintenance of the kernel and create inevitable confusions. In addition, it can create different kernel binaries, potentially for each VMM, such as Xen*, KVM, VMware, etc. Today, paravirt_ops already has **76 operations** for x86, and it would grow and

change over the time. In fact, patches already have been sent to update them while we are writing this paper!

One of the main sources of such growth and modification seems to be different hypervisor implementations in support of para-virtualization.

Para-Virtualization

The para-virtualization is a virtualization technique that presents a software interface to virtual machines that is similar but not identical to that of the underlying hardware. This technique is often used to obtain better performance of guest operating systems running inside a virtual machine.

Para-virtualization can be applicable even to hardware-assisted virtualization as well. Historically, para-virtualization in the Linux community was used to mean modifications to the guest operating system so that it can run in a virtual machine without requiring the hardware-assisted virtualization features. In this paper, we use *software-only para-virtualization* to mean "para-virtualization that obviates hardware-assisted virtualization."

The nature of the modifications used by para-virtualization to the kernel matters. We experienced significant complexity of *software-only para-virtualization* when we ported x86-64 Linux to Xen [4]. The root cause of such complexity is that *software-only para-virtualization* forces the kernel developers to handle the virtual CPU that has significant limitations and different behaviors from the native CPU.

For example, such virtual CPU has completely different systems state such as, it does not have GDT, IDT, LDT, or TSS; completely new interrupt/exception mechanism; or different protection mechanism. And the virtual CPU does not support any privileged instructions, requiring them to be executed by hypercalls.

The protection mechanism often has problems with having a shared kernel address space as there is no ring-level protection when both user and kernel are running at ring 3. This creates an additional overhead of stitching address space between any transition between the application and the kernel.

Additionally, system calls are first intercepted by the Xen (ring 0), and are injected to the guest kernel. Once the guest kernel completes the service, then it needs to go back to the user executing a hypercall (and page table switch because of the reason above).

Benefiting from Para-Virtualization

We also found para-virtualization really simplified and optimized the system. In fact there are still significant cases where the native or *software-only para-virtualization* outperforms full-virtualization using hardware-assisted virtualization especially with I/O or memory intensive workloads (or both).

In this paper we first discuss the advantages and disadvantages of para-virtualization, full-virtualization, and hardware-assisted virtualization. Note that some of these advantages and disadvantages can be combined and complimentary. Second, we discuss the hybrid-virtualization for Linux proposal. Unlike *software-only para-virtualization*, hybrid-virtualization employs hardware-assisted virtualization, and it needs much fewer para-virtualization operations. Third, we discuss the design and implementation. Finally we present some examples of performance data.

2 Para-Virtualization

2.1 Advantages

Obviously para-virtualization is employed to achieve high performance and efficiency. Since para-virtualization typically uses a higher level of APIs that are not available on the underlying hardware, efficiency is also improved.

Time and Idle Handling

"Time" is a notable example. Even in a virtual system, the user expects that the virtual machine maintain the real time, not virtual time! As most operating systems rely on timer interrupts to maintain its time, the system expects timer interrupts even when idle. If timer interrupts are missed, it can affect the time keeping of the operating system. Without para-virtualization, the VMM needs to continue injecting timer interrupts or to inject back-to-back timer interrupts when the guest operating system is scheduled back to run. This is not a reliable or scalable way of virtualization. With para-virtualization, a typical modification is to change the idle code to request the VMM to notify itself in a specified time period. Then time is re-calculated and restored in the guest.

SMP Guests Handling

SMP guest handling is another example. In x86 or x86-64, local APIC is required to support SMP especially because the operating systems need to send IPI (Inter-Processor Interrupt). Figure 1 shows the code for sending IPI on the x86-64 native systems using the flat mode. As you see, the code needs to access the APIC registers a couple of times. Each access to the APIC registers needs to be intercepted for virtualization, causing overhead (often a transition to the VMM).

Para-virtualization can replace such multiple *implicit* requests with a single *explicit* hypercall, achieving faster, simpler, and more efficient implementations.

I/O Device Handling

Writing software that emulates a complete computer, for example, is complex and labor-intensive because various legacy and new devices need to be emulated. With para-virtualization the operating system can operate without such devices, and thus the implementation of the VMM can be simpler.

From the guest operating system's point view, the impacts of such modifications would be limited because the operating system already has the infrastructure that supports layers of I/O services/devices, such as block/character device, PCI device, etc.

2.2 Disadvantages

There are certain advantages of para-virtualization as mentioned above but there are certain disadvantages in Linux.

```
static void flat_send_IPI_mask(cpumask_t cpumask, int vector)
{
...
        /*
         * Wait for idle.
         */
        apic_wait_icr_idle();
        /*
         * prepare target chip field
         */
        cfg = __prepare_ICR2(mask);
        apic_write(APIC_ICR2, cfg);
        /*
         * program the ICR
         */
        cfg = __prepare_ICR(0, vector, APIC_DEST_LOGICAL);
        /*
         * Send the IPI. The write to APIC_ICR fires this off.
         */
        apic_write(APIC_ICR, cfg);
...
}
```

Figure 1: Sending IPI on the native x86-64 systems (flat mode)

Modified CPU Behaviors

One of the fundamental problems, however, is that *software-only para-virtualization* forces the kernel developers to handle the virtual CPU that has significant limitations and different behaviors from the native CPU. And such virtual CPU behaviors can be different on different VMMs because the semantics of the virtual CPU is defined by the VMM. Because of that, kernel developers don't feel comfortable when the kernel code also needs to handle virtual CPUs because they may break virtual CPUs even if the code they write works fine for native CPUs.

Figure 2 shows, for example, part of paravirt_t structure in x86 Linux 2.6.21(-rcX as of today). As you can see, it has operations on TR, GDT, IDT, LDT, the kernel stack, IOPL mask, etc. It is barely possible for a kernel developer to know how those operations can be safely used without understanding the semantics of the virtual CPU, which is defined by each VMM. It also is not guaranteed that the common code between the native and virtual CPU can be cleanly written.

A notable issue is the CPUID instruction because it is available in user mode as well, thus *software-only para-virtualization* inherently requires modifications to the operating system. The CPUID instruction is often used

to detect the CPU capabilities available on the processor. If a user application does so, it may not be possible to modify the application if provided in a binary object.

2.2.1 Overheads

Although para-virtualization is intended to achieve high performance, ironically the protection mechanism technique used by *software-only para-virtualization* can often cause overhead. For example, system calls are first intercepted by the Xen (ring 0), and are injected to the guest kernel. Once the guest kernel completes the service, then it needs to go back to the user executing a hypercall with the page tables switched to protect the guest kernel from the user processes. This means that it is impossible to implement fast system calls.

The same bouncing mechanism is employed when handling exceptions, such as page faults. They are also first intercepted by Xen even if generated purely by user processes.

The guest kernel also loses the global pages for its address translation to protect Xen from the guest kernel.

Note that this overhead can be eliminated in hardware-assisted virtualization because hardware-assisted virtu-

```
struct paravirt_ops
{
...
        void (*load_tr_desc)(void);
        void (*load_gdt)(const struct Xgt_desc_struct *);
        void (*load_idt)(const struct Xgt_desc_struct *);
        void (*store_gdt)(struct Xgt_desc_struct *);
        void (*store_idt)(struct Xgt_desc_struct *);
        void (*set_ldt)(const void *desc, unsigned entries);
        unsigned long (*store_tr)(void);
        void (*load_tls)(struct thread_struct *t, unsigned int cpu);
        void (*write_ldt_entry)(void *dt, int entrynum,
                                    u32 low, u32 high);
        void (*write_gdt_entry)(void *dt, int entrynum,
                                    u32 low, u32 high);
        void (*write_idt_entry)(void *dt, int entrynum,
                                    u32 low, u32 high);
        void (*load_esp0)(struct tss_struct *tss,
                                struct thread_struct *thread);
        void (*set_iopl_mask)(unsigned mask);
...
};
```

Figure 2: The current paravirt_t structure (only virtual CPU part of 76 operations) for x86

alization can provide the same CPU behavior as the native without modifications to the guest operating system.

3 Full-Virtualization

3.1 Advantages

The full-virtualization requires no modifications to the guest operating systems. This attribute itself clearly brings significant value and advantage.

3.2 Disadvantages

Full-virtualization requires one to provide the guest operating systems with an illusion of a complete virtual platform seen within a virtual machine behavior same as a standard PC/server platform. Today, both Xen and KVM need Qemu for PC platform emulation with the CPU being native. For example, its system address space should look like a standard PC/server system address map, and it should support standard PC platform devices (keyboard, mouse, real time clock, disk, floppy, CD-ROM drive, graphics, 8259 programmable interrupt controller, 8254 programmable interval timer, CMOS, etc.), guest BIOS, etc. In addition, we need to provide those virtual devices with the DMA capabilities to obtain optimized performance. Supporting SMP guest OSes further complicates the VMM design and implementation.

In addition, as new technologies emerge, the VMM needs to virtualize more devices and features to minimize functional/performance gaps between the virtual and native systems.

4 Hardware-Assisted Virtualization

The hardware-assisted virtualization is orthogonal to para or full virtualization, and it can be used for the both.

4.1 Advantages

The hardware-assisted virtualization provides virtual machine monitors (VMM) with simpler and robust implementation.

Full-virtualization can be implemented by software only [1], as we see such products such as VMware as well as Virtual PC and Virtual Server from Microsoft today. However, hardware-assisted virtualization such as Intel® Virtualization Technology (simply Intel® VT hereafter) can improve the robustness, and possibly performance.

4.2 Disadvantages

Obviously hardware-assisted virtualization requires a system with the feature, but it is sensible to assume that hardware-assisted virtualization is available on almost all new x86-64-based systems.

Para-virtualization under hardware-assisted virtualization needs to use a certain instruction(s) (such as VMCALL on Intel® VT), which is more costly than the fast system call (such as SYSENTER/SYSEXIT, SYSCALL/SYSRET) used under *software-only para-virtualization*. However, the cost of such instructions will be lower in the near future.

The hardware-assisted virtualization today does not include I/O devices, thus it still needs emulation of I/O devices and possibly para-virtualization of performance-critical I/O devices to reduce frequent interceptions for virtualization.

5 Hybrid-Virtualization for Linux

Hardware-assisted virtualization does simplify the VMM design and allows use of the unmodified Linux as a guest.

There are, however, still significant cases where *software-only para-virtualization* outperforms full-virtualization using hardware-assisted virtualization especially with I/O or memory intensive workloads (or both), which are common among enterprise applications. Those enterprise applications matter to Linux.

For I/O intensive workloads, such performance gaps have been mostly closed by using para-virtualization drivers for network and disk. Those drivers are shared with *software-only para-virtualization*.

For memory intensive workloads, the current production processors with hardware-assisted virtualization does not have capability to virtualize MMU, and the VMM needs to virtualize MMU in software [2]. This causes visible performance gaps between the native or *software-only para-virtualization* and hardware-assisted full-virtualization (See [7], [8]).

5.1 Overview of Hybrid-Virtualization

Hybrid-virtualization that we propose is technically para-virtualization for hardware-assisted virtualization.

However, we use this terminology to avoid any confusions caused by the connotation from *software-only para-virtualization*. And the critical difference is that hybrid-virtualization is simply a set of optimization techniques for hardware-assisted full-virtualization.

5.2 Pseudo Hardware Features

The hybrid-virtualization capabilities are detected and enabled as by the standard Linux for the native as if they were hardware features, i.e. *pseudo hardware feature*, i.e. visible as hardware from the operating system's point of view.

We use the CPUID instruction to detect certain CPU capabilities on the native system, and we include the ones for hybrid-virtualization without breaking the native systems. The CPUID instruction is intercepted by hardware-assisted virtualization so that the VMM can virtualize the CPUID instruction. In other words, the kernel does not know whether the capabilities are implemented in software (i.e., VMM) or hardware (i.e., silicon).

In order to avoid maintenance problems caused by the paravirt layer, the interfaces must be in the lowest level in the Linux that makes the actual operations on the hardware. If a particular VMM needs a higher level or new interface in the kernel, it should be detected and enabled as *pseudo hardware feature* as well, rather than extending or modifying the paravirt layer.

5.3 Common Kernel Binary for the Native and VMMs

One of the goals of hybrid-virtualization is to have the common kernel binary for the native and various VMMs, including the Xen hypervisor, KVM [6], etc. For example, the kernel binary for D0, G, H can be all same in Figure 3.

With the approach above and the minimal paravirt layer, we can achieve the goal.

5.4 Related Works

Ingo Molnar also added simple paravirt ops support (using the shadow CR3 feature of Intel® VT) to improve the context switch of KVM [3]. This technique can be simply incorporated in our hybrid-virtualization.
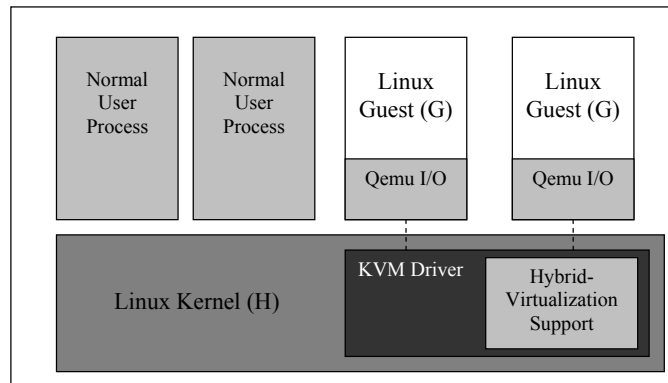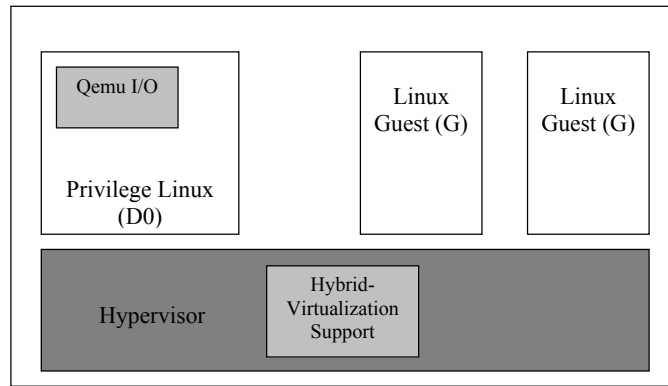
Figure 3: Same Kernel Binary for a hypervisor (D0 and G), KVM (G) and the native (H)

6 Design and Implementation

6.1 Areas for Para-Virtualization

Based of the performance data available in the community and our experiences with x86-64 Linux porting to Xen, we identified the major areas to use para-virtualization, while maintaining the same CPU behavior as the native:

- I/O devices, such as network and disk

- Timer – with para-virtualization, the kernel can have better accounting because of the stolen time for other guests.

- Idle handling – the latest kernel has no idle ticks, but the kernel could even specify the time period for which it will be idle so that the VMM can use the time for other VMs.

- Interrupt controllers

- MMU

- SMP support – the hypervisor also knows which physical CPUs actually have (real) TLBs with information that needs to be flushed. A guest with many virtual CPUs can send many unnecessary IPIs to virtual CPU running other guests. This area is included by the interrupt controller.

In fact all the areas except MMU are straightforward to support in Linux as it already has the proper infrastructures to handle without depending on para-virtualization or virtualization-specific modifications:

- I/O devices – obviously Linux needs to handle various I/O devices today, and various para-virtualization drivers are already available in commercial VMMs.

- Timer – Linux supports various time sources, such as PIT, TSC, HPET,

- Idle handling – Linux already has a mechanism to select the proper idle routine.

- Interrupt controller – the genapic (in x86-64) is a good example.

6.2   MMU Para-Virtualization

This is the outstanding area where Linux benefits significantly today from para-virtualization even with hardware-assisted virtualization.

For our implementation, we used the direct page tables employed by Xen (See [5], [4]). Unlike the shadow page table mode that builds additional (duplicated) page tables for the real translation, the direct page tables are native page tables. The guest operating system use the hypercalls to request the VMM to update the page tables entries.

The paravirt_t layer in x86 already has such operations, and we ported the subset of paravirt operations to x86-64, extending them to the 4-level page tables as shown in Figure 4.

6.2.1   Efficient Page Fault Handling

Although hybrid-virtualization uses the direct page table mode today, it is significantly efficient compared with the one on *software-only para-virtualization* because the page faults can be selectively delivered to guest directly in hardware-assisted virtualization. For example, VMX provides page-fault error-code mask and match fields in the VMCS to filter VM exits due to page-faults based on their cause (reflected in the error-code). We use this functionality so that the guest kernel directly get page faults from *user* processes without causing a VM exit. Note that the majority of page faults are from user processes under practical workloads.

In addition, now the kernel runs in the ring 0 as the native, all the native protection and efficiency, including paging-based protection and global pages, have been returned back to the guest kernel.

6.2.2   Other Optimization Techniques

Since we can run the kernel in ring 0, all the optimization techniques used by the native kernel have been back to the kernel in hybrid-virtualization, including fast system call.

6.3   Detecting Hybrid-Virtualization

The kernel needs to detect whether hybrid-virtualization is available or not (i.e., on the native or unknown VMM that does not support hybrid-virtualization). As we discussed, we use the CPUID (leaf 0x40000000, for example) instruction. The leaf 0x40000000 is not defined on the real hardware, and the kernel can reliably detect the presence of hybrid-virtualization *only in a virtual machine* because the VMM can implement the capabilities of the leaf 0x40000000.

6.4   Hypercall and Setup

Once the hybrid-virtualization capabilities are detected, the kernel can inform the VMM of the address of the page that requires the instruction stream for hypercalls (called "hypercall page"). The request to the VMM is done by writing the address to an MSR returned by the CPUID instruction. Then the VMM actually writes the instruction stream to the page, and then hypercalls will be available via jumping to the hypercall page with the arguments (indexed by the hypercall number).

6.5   Booting and Initialization

The hybrid-virtualization Linux uses the booting code identical to the native at early boot time. It then switches to the direct page table mode if hybrid-virtualization is present. Until that point, the guest kernel needs to use the existing shadow page table mode, and then it switches to the direct page table mode upon a hypercall.

We implemented the **SWITCH_MMU** hypercall for this purpose. Upon that hypercall, the VMM updates the guest page tables so that they can contain host physical address (rather than guest physical address) and write-protect them. Upon completion of the hypercall, the guest needs to use the set of hypercalls to update its page tables, and those are seamlessly incorporated by the paravirt layer.

```
struct paravirt_ops
{
...
        unsigned long (*read_cr3)(void);
        void (*write_cr3)(unsigned long);

        void (*flush_tlb_user)(void);
        void (*flush_tlb_kernel)(void);
        void (*flush_tlb_single)(unsigned long addr);

        void (*alloc_pt)(unsigned long pfn);
        void (*alloc_pd)(unsigned long pfn);

        void (*release_pt)(unsigned long pfn);
        void (*release_pd)(unsigned long pfn);

        void (*set_pte)(pte_t *ptep, pte_t pteval);
        void (*set_pte_at)(struct mm_struct *mm, ...
        pte_t (*ptep_get_and_clear)(struct mm_struct *mm, ...

        void (*set_pmd)(pmd_t *pmdp, pmd_t pmdval);
        void (*set_pud)(pud_t *pudp, pud_t pudval);
        void (*set_pgd)(pgd_t *pgdp, pgd_t pgdval);

        void (*pte_clear)(struct mm_struct *mm, ...
        void (*pmd_clear)(pmd_t *pmdp);
        void (*pud_clear)(pud_t *pudp);
        void (*pgd_clear)(pgd_t *pgdp);

        unsigned long (*pte_val)(pte_t);
        unsigned long (*pmd_val)(pmd_t);
        unsigned long (*pud_val)(pud_t);
        unsigned long (*pgd_val)(pgd_t);

        pte_t (*make_pte)(unsigned long pte);
        pmd_t (*make_pmd)(unsigned long pmd);
        pud_t (*make_pud)(unsigned long pud);
        pgd_t (*make_pgd)(unsigned long pgd);
};
```

Figure 4: The current paravirt_t structure for x86-64 hybrid-virtualization

6.6 Prototype

We implemented hybrid-virtualization x86-64 Linux in Xen, starting from full-virtualization in hardware-assisted virtualization, porting the x86 paravirt (with significant reduction).

We used the existing Xen services for the following:

- I/O devices – virtual block device and network device front-end drivers.

- Timer

- Idle handling

- Interrupt controller

Since the x86-64 Linux uses 2MB pages for the kernel mapping and the current Xen does not support large pages, we needed to add the level 1 pages (page tables) in the kernel code so that **SWITCH_MMU** hypercall can work.

We also reused the code for x86-64 XenLinux virtual MMU code for the paravirt MMU.
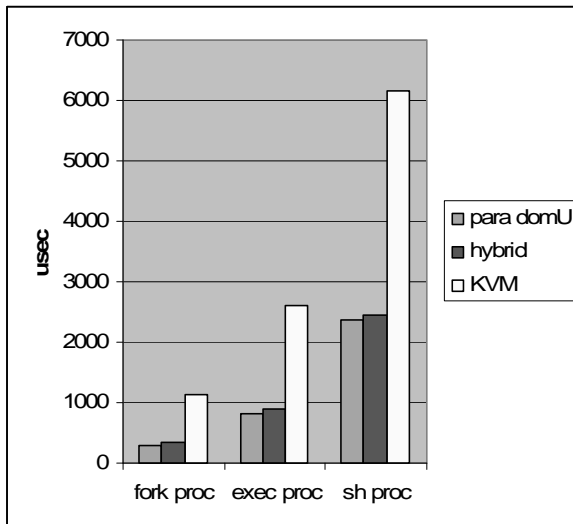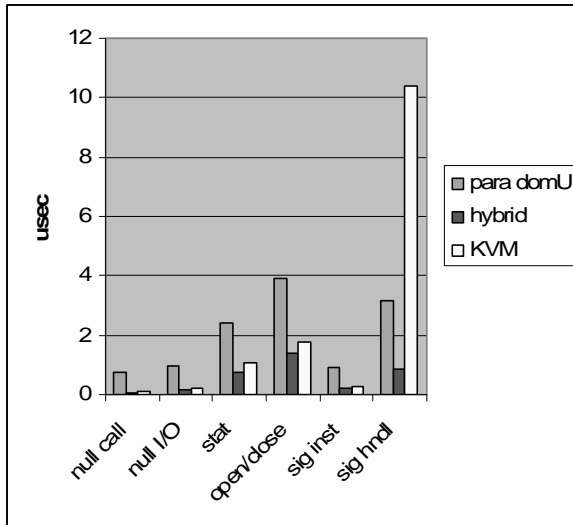
Figure 5: Preliminary Micro-benchmark Results (lm-bench)

7 Performance

Although the cost of hypercalls are slightly higher in hybrid-virtualization, hybrid-virtualization is more efficient than *software-only para-virtualization* because of the retained optimization techniques in the native kernel. In fact, hybrid-virtualization showed the equivalent performance with kernel build, compared with *software-only para-virtualization*, which has near-native performance for that workload.

For micro-benchmarks, hybrid-virtualization showed

visible performance improvements. Figure 5 shows preliminary results. "para-domU" is x86-64 XenLinux with *software-only para-virtualization*, and "hybrid" is the one with hybrid-virtualization. "KVM" is x86-64 Linux running on the latest release (kvm-24, as of today). The smaller are the better, and the absolute numbers are not so relevant.

As of today, we are re-measuring the performance using the latest processors, where we believe the cost of hypercalls are even lower.

8 Conclusion

The hybrid-virtualization is able to combine advantages from both the hardware assisted full virtualization and software-only para-virtualization. The initial prototype results also show performance close to *software-only para-virtualization*. This also provides the added benefit that the same kernel can run under native machines.

Acknowledgment

We would like to thank Andi Kleen and Ingo Molnar for reviewing this paper and providing many useful comments.

Xin Li and Qing He from Intel also have been working on the development of hybrid-virtualization Linux.

References

[1] Virtual Machine Interface (VMI) Specifications. http://www.vmware.com/interfaces/vmi_specs.html.

[2] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu. Extending Xen with Intel®Virtualization Technology. Auguest 2006. http://www.intel.com/technology/itj/2006/v10i3/.

[3] Ingo Molnar. KVM paravirtualization for Linux. 2007. http://lkml.org/lkml/2007/1/5/205.

[4] Jun Nakajima, Asit Mallick, Ian Pratt, and Keir Fraser. X86-64 XenLinux: Architecture, Implementation, and Optimizations. In *Preedings of the Linux Symposium*, July 2006.

[5] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheirmer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the Art of Virtualization. In *Preedings of the Linux Symposium*, July 2005.

[6] Qumranet. KVM: Kernel-based Virtualization Driver. 2006. `http://www.qumranet.com/wp/kvm_wp.pdf/`.

[7] VMware. A Performance Comparison of Hypervisors. 2007. `http://www.vmware.com/pdf/hypervisor_performance.pdf/`.

[8] XenSource. A Performance Comparison of Commercial Hypervisors. 2007. `http://www.xensource.com/files/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf/`.

Readahead: time-travel techniques for desktop and embedded systems

Michael Opdenacker

Free Electrons

`michael@free-electrons.com`

Abstract

Readahead techniques have successfully been used to reduce boot time in recent GNU/Linux distributions like Fedora Core or Ubuntu. However, in embedded systems with scarce RAM, starting a parallel thread reading ahead all the files used in system startup is no longer appropriate. The cached pages could be reclaimed even before accessing the corresponding files.

This paper will first guide you through the heuristics implemented in kernelspace, as well as through the userspace interface for preloading files or just announcing file access patterns. Desktop implementations will be explained and benchmarked. We will then detail Free Electrons' attempts to implement an easy to integrate helper program reading ahead files at the most appropriate time in the execution flow.

This paper and the corresponding presentation target desktop and embedded system developers interested in accelerating the course of Time.

1 Reading ahead: borrowing time from the future

1.1 The page cache

Modern operating system kernels like Linux manage and optimize file access through the *page cache*. When the same file is accessed again, no disk I/O is needed if the file contents are still in the page cache. This dramatically speeds up multiple executions of a program or multiple accesses to the same data files.

Of course, the performance benefits depend on the amount of free RAM. When RAM gets scarce because of allocations from applications, or when the contents of more files have to be loaded in page cache, the kernel has to reclaim the oldest pages in the page cache.

1.2 Reading ahead

The idea of reading ahead is to speed up the access to a file by preloading at least parts of its contents in page cache ahead of time. This can be done when spare I/O resources are available, typically when tasks keep the processor busy. Of course, this requires the ability to predict the future!

Fortunately, the systems we are dealing with are predictable or even totally predictable in some situations!

- Predictions by watching file read patterns. If pages are read from a file in a sequential manner, it makes sense to go on reading the next blocks in the file, even before these blocks are actually requested.

- System startup. The system init sequence doesn't change. The same executables and data files are always read in the same order. Slight variations can still happen after a system upgrade or when the system is booted with different external devices connected to it.

- Applications startup. Every time a program is run, the same shared libraries and some parts of the program file are always loaded. Then, many programs open the same resource or data files at system startup. Of course, file reading behaviour is still subject to changes, according to how the program was started (calling environment, command arguments. . .).

If enough free RAM is available, reading ahead can bring the following benefits:

- Of course, reduced system and application startup time.

- Improved disk throughput. This can be true for storage devices like hard disks which incur a high time cost moving the disk heads between random sectors. Reading ahead feeds the I/O scheduler with more I/O requests to manage. This scheduler can then reorder requests in a more efficient way, grouping a greater number of contiguous disk blocks, and reducing the number of disk head moves. This is much harder to do when disk blocks are just read one by one.

- Better overall utilization for both I/O and processor resources. Extra file I/O is performed when the processor is busy. Context switching, which costs precious CPU cycles, is also reduced when a program no longer needs to sleep waiting for I/O, because the data it is requesting have already been fetched.

2 Kernel space readahead

2.1 Implementation in stock kernels

The description of the Linux kernel readahead mechanism is based on the latest stable version available at the time of this writing, Linux 2.6.20.

When the kernel detects sequential reading on a file, it starts to read the next pages in the file, hoping that the running process will go on reading sequentially.

As shown in Figure 1, the kernel implements this by managing two read windows: the *current* and *ahead* one,

While the application is walking the pages in the current window, I/O is underway on the ahead window. When the current window is fully traversed, it is replaced by the ahead window. A new ahead window is then created, and the corresponding batch of I/O is submitted.

This way, if the process continues to read sequentially, and if enough free memory is available, it should never have to wait for I/O.

Of course, any seek or random I/O turns off this readahead mode.

The kernel actually checks how effective reading ahead is to adjust the size of the new ahead window. If a page cache miss is encountered, it means that some of

its pages were reclaimed before being accessed by the process. In this case, the kernel reduces the size of the ahead window, down to `VM_MIN_READAHEAD` (16 KB). Otherwise, the kernel increases this size, up to `VM_MAX_READAHEAD` (128 KB).

The kernel also keeps track of page cache hits, to detect situations in which the file is partly or fully in page cache. When this happens, readahead is useless and turned off.

Implementation details can be found in the `mm/ readahead.c` file in the kernel sources.[1]

The initial readahead implementation in Linux 2.6 is discussed in the 2004 proceedings [7] of the Ottawa Linux Symposium.

2.2 Adaptive readahead patches

Many improvements to the kernel readahead mechanism have been proposed by Wu Fengguang through the *Adaptive readahead* patchset, since September 2005 (as announced on this LWN article [1].

In addition to the standard sequential reading scenario, this patchset also supports:

- a readahead window which can grow up to 1 MB, depending on the application behaviour and available free memory

- parallel / interleaved sequential scans on one file

- sequential reads across file open/close

- mixed sequential / random accesses

- sparse / skimming sequential read

- backward sequential reading

- delaying readahead if the drive is spinned down in laptop mode

At the time of this writing the latest benchmarks [3] show access time improvements in most cases.

This patchset and its ideas will be described in detail by Wu Fengguang himself at this 2007 edition of the Ottawa Linux Symposium.

---

[1] A very convenient way of studying kernel source files is using a Linux Cross Reference (LXR) website indexing the kernel sources, such as http://lxr.free-electrons.com.

When this page is reached:
- the ahead window becomes the current one
- a new ahead window is created
(possibly shorter or larger)

Current
read offset

Offset in the open file
(scale: pages)

Current read window
(already read ahead)

Ahead read window
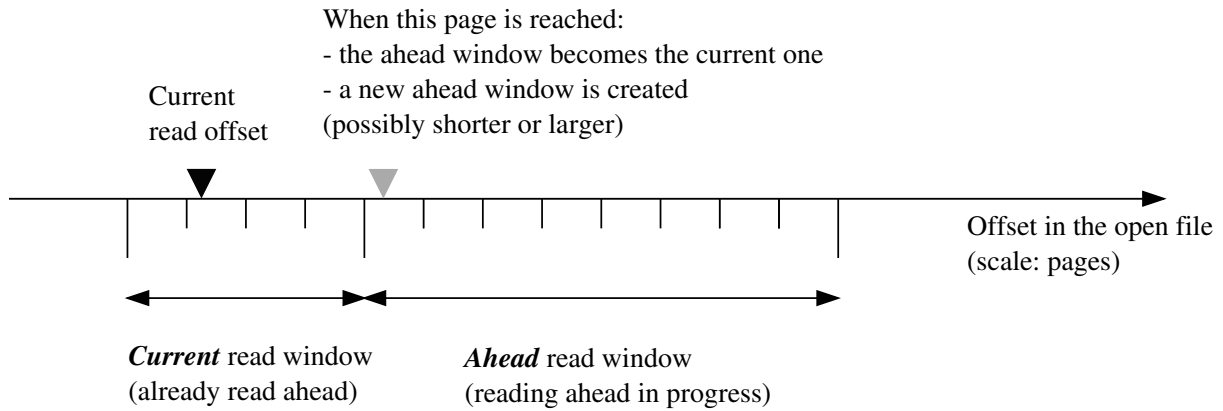(reading ahead in progress)

Figure 1: Stock kernel implementation

3   User-space readahead interface

We've seen how the kernel can do its best to predict the future from recent and present application behaviour, to improve performance.

However, that's about all a general purpose kernel can predict. Fortunately, the Linux kernel allows userspace to let it know its own predictions. Several system call interfaces are available.

3.1   The readahead system call

```
#include <fcntl.h>

ssize_t readahead(
        int fd,
        off64_t *offset,
        size_t count);
```

Given an open file descriptor, this system call allows applications to instruct the kernel to readahead a given segment in the file.

Though any `offset` and `count` parameters can be given, I/O is performed in whole pages. So `offset` is rounded down to a page boundary and bytes are read up to the first page boundary greater than or equal to `offset+count`.

Note that `readahead` blocks until all data have been read. Hence, it is typically called from a parallel thread.

See the manual page for the `readahead` system call [6] for details.

3.2   The fadvise system call

Several variants of this system call exist, depending on your system or GNU/Linux distribution: `posix_fadvise`, `fadvise64`, `fadvise64_64`.

They all have the same prototype though:

```
#define _XOPEN_SOURCE 600
#include <fcntl.h>

int posix_fadvise(
        int fd,
        off_t offset,
        off_t len,
        int advice);
```

Programs can use this system call to announce an intention to access file data in a specific pattern in the future, thus allowing the kernel to perform appropriate optimizations.

Here is how the Linux kernel interprets the possible settings for the `advice` argument:

POSIX_FADV_NORMAL: use the default readahead window size.

POSIX_FADV_SEQUENTIAL: sequential access with increasing file offsets. Double the readahead window size.

POSIX_FADV_RANDOM: random access. Disable readahead.

POSIX_FADV_WILLNEED: the specified data will be accessed in the near future. Initiate a non-blocking read of the specified region into the page cache.

POSIX_FADV_NOREUSE: similar, but the data will just be accessed once.

POSIX_FADV_DONTNEED: attempts to free the cached pages corresponding to the specified region, so that more useful cached pages are not discarded instead.

Note that this system call is not binding: the kernel is free to ignore the given advise.

Full details can be found on the manual page for posix_fadvise [5].

3.3 The madvise system call

```
#include <sys/mman.h>

int madvise(
    void *start,
    size_t length,
    int advice);
```

The madvise system call is very similar to fadvise, but it applies to the address space of a process.

When the specified area maps a section of a file, madvise information can be used by the kernel to readahead pages from disk or to discard page cache pages which the application will not need in the near future.

Full details can be found on the manual page for madvise [4].

3.4 Recommended usage

As the readahead system call is binding, application developers should use it with care, and prefer fadvise and madvise instead.

When multiple parts of a system try to be smart and consume resources while being oblivious to the others, this often hurts overall performance. After all, resource management is the kernel's job. It can be best to let it decide what to do with the hints it receives from multiple sources, balancing the resource needs they imply.

4 Implementations in GNU/Linux distributions

4.1 Ubuntu

Readahead utilities are released through the readahead package. The following description is based on Ubuntu 6.10 (Edgy).

Reading ahead is started early in the system startup by the /etc/init.d/readahead init script. This script mainly calls the /sbin/readahead-list executable, taking as input the /etc/readahead/boot file, which contains the list of files to readahead, one per line.

readahead-list is of course started as a daemon, to proceed as a parallel thread while other init scripts run. readahead-list doesn't just readahead each specified file one by one, it also *orders* them first.

Ordering files is an attempt to read them in the most efficient way, minimizing costly disk seeks. To order two files, their device numbers are first compared. When their device numbers are identical, this means that they belong to the same partition. The numbers of their first block are then compared, and if they are identical, their inode numbers are eventually compared.

The readahead-list package carries another utility: readahead-watch. It is used to create or update the list of files to readahead by watching which files are accessed during system startup.

readahead-watch is called from /etc/init.d/readahead when the profile parameter is given in the kernel command line. It starts watching for all files that are accessed, using the inotify [8] system call. This is a non trivial task, as inotify watches have to be registered for each directory (including subdirectories) in the system.

readahead-watch eventually gets stopped by the /etc/init.d/stop-readahead script, at the very end of system startup. It intercepts this signal and creates the /etc/readahead/boot file.

For the reader's best convenience, C source code for these two utilities and a copy of /etc/readahead/boot can be found on http://free-electrons.com/pub/readahead/ubuntu/6.10/.

4.2 Fedora Core

Readahead utilities are released through the `readahead` package. The following description is based on Fedora Core 6.

The readahead executable is `/usr/sbin/readahead`. Its interface and implementation are similar. It also sorts files in order to minimize disk seeks, with more sophisticated optimizations for the ext2 and ext3 filesystems.

A difference with Ubuntu is that there are two readahead init scripts. The first one is `/etc/init.d/readahead_early`, which is one of the first scripts to be called. It preloads files listed in `/etc/readahead.d/default.early`, corresponding to libraries, executables, and files used by services started by init scripts. The second script, `/etc/init.d/readahead_later`, is one of the last executed scripts. It uses `/etc/readahead.d/default.later`, which mainly corresponds to files used by the graphical desktop and user applications in general.

Another difference with Ubuntu is that the above lists of files are constant and are not automatically generated from application behaviour. They are just shipped in the package. However, the `readahead-check` utility (available in package sources) can be used to generate these files from templates and check for nonexistent files.

Once more, the `readahead.c` source code and a few noteworthy files can be found on `http://free-electrons.com/pub/readahead/fedora-core/6/`.

4.3 Benchmarks

The below benchmarks compare boot time with and without readahead on Ubuntu Edgy (Linux 2.6.17-11, with all updates as of Apr. 12, 2007), and on Fedora Core 6 (2.6.18-1.2798.fc6, without any updates).

Boot time was measured by inserting an init script which just copies `/proc/uptime` to a file. This script was made the very last one to be executed.

`/proc/uptime` contains two figures: the raw uptime in seconds, and the amount of time spent in the idle loop, meaning the CPU was waiting for I/O before being able to do anything else.

Disabling readahead was done by renaming the `/sbin/readahead-list` (Ubuntu) or `/usr/sbin/readahead` programs, so that readahead init scripts couldn't find them any more and exited at the very beginning.

The Fedora Core 6 results are surprising. An explanation is that readahead file lists do not only include files involved in system startup, but also files needed to start the desktop and its applications. Fedora Core readahead is thus meant to reduce the execution time of programs like Firefox or Evolution!

As a consequence, Fedora Core is reading ahead much more files than needed (even if we disable the `readahead-later` step) and it reaches the login screen later than if readahead was not used. The eventual benefits in the time to run applications should still be real. However, they are more difficult to measure.

4.4 Shortcomings

The readahead implementations that we have just covered are fairly simple, but not perfect though.

4.4.1 Reading entire files

A first limitation is that these implementations always preload *entire files*, while the `readahead` system call allows to fetch only specific sections in files.

It's true that it can make sense to assume that plain data files used in system startup are often read in their entirety. However, this is not true at all with executables and shared libraries, for which each page is loaded only when it is needed. This mechanism is called *demand paging*. When a program jumps to a section of its address space which is not in RAM yet, a page fault is raised by the MMU, and the kernel loads the corresponding page from disk.

Using the `top` or `ps` commands, you can check that the actual RAM usage of processes (`RSS` or `RES`) is much smaller than the size of their virtual address space (`VSZ` or `VIRT`).

So, it is a waste of I/O, time, and RAM to load pages in executables and shared libraries which will not be used anyway. However, as we will see in the next section, demand paging is not trivial to trace from userspace.

	boot time	idle time
Ubuntu Edgy without readahead	average: 48.368 s std deviation: 0.153	average: 29.070 s std deviation: 0.281
Ubuntu Edgy with readahead	average: 39.942 s (-17.4 %) std deviation: 1.296	average: 22.3 s (-23.3 %) std deviation: 0.271
Fedora Core 6 without readahead	average: 50.422 s std deviation: 0.496	average: 28.302 s std deviation: 0.374
Fedora Core 6 with readahead	average: 59.858 s (+18.7 %) std deviation: 0.552	average: 35.446 (+20.2 %) std deviation: 0.312

Table 1: Readahead benchmarks on Ubuntu Edgy and Fedora Core 6

4.4.2 Reading ahead too late

Another limitation comes from reading ahead all files in a row, even the ones which are needed at the very end of system startup.

We've seen that files are preloaded according to their location on the disk, and not according to when they are used in system startup. Hence, it could happen that a file needed by a startup script is *accessed before* it is preloaded by the readahead thread.

5 Implementing readahead in embedded systems

5.1 Embedded systems requirements

Embedded systems have specific features and requirements which make desktop implementations not completely appropriate for systems with limited resources.

The main constraint, as explained before, is that free RAM can be scarce. It is no longer appropriate to preload all the files in a row, because some of the readahead pages are likely to be reclaimed before being used. As a consequence, a requirement is to readahead files just a little while before they are used.

Therefore, files should be preloaded according to the order in which they are accessed. Moreover, most embedded systems use flash instead of disk storage. There is no disk seek cost accessing random blocks on storage. Ordering files by disk location is futile.

Still because of the shortness of free RAM, is it also a stronger requirement to preload only the portions of the files which are actually accessed during system startup.

Last but not least, embedded systems also require simple solutions which can translate in lightweight programs and in low cpu usage.

5.2 Existing implementations

Of course, it is possible to reuse code from readahead utilities found in GNU/Linux distributions, to readahead a specific list of files.

Another solution is to use the `readahead` applet that we added to the Busybox toolset (`http://busybox.net`), which is used in most embedded systems. Thanks to this applet, developers can easily add readahead commands to their startup scripts, without having to compile a standalone tool.

5.3 Implementation constraints and plans

Updates, code, benchmarks, and documentation will be available through our readahead project page [2].

5.3.1 Identifying file access patterns

It is easy to identify files which are accessed during startup, either by using `inotify` or by checking the `atime` attribute of files (last access time, when this feature is not disabled at mount time). However, it is much more difficult to trace which sections are accessed in a given file.

When the file is just accessed, not executed, it is still possible to trace the `open`, `read`, `seek`, and `close`

system calls and deduce which parts of each file are accessed. However, this is difficult to implement.[2]

Anyway, when the file is executed (in the case of a program or a shared library), there doesn't seem to be any userspace interface to keep track of accessed file blocks. It is because demand paging is completely transparent to processes.

That's why we started to implement a kernel patch to log all file reads (at the moment by tracing calls to the `vfs_read` function), and demand paging activity (by getting information from the `filemap_nopage` function). This patch also tracks `exec` system calls, by watching the `open_exec` function, for reasons that we will explain in the next section.

This patch logs the following pieces of information for each accessed file:

- inode number,

- device major and minor numbers,

- offset,

- number of bytes read.

Code and more details can be found on our project page [2].

At the time of this writing, this patch is just meant to assess the usefulness of reading ahead only the used sections in a file. If this proves to be profitable, a clean, long term solution will be investigated with the Linux kernel development community.

5.3.2 Postprocessing the file access dump

We are developing a Python script to postprocess file access information dumped from the kernel.

The main need is to translate inode and device numbers into file paths, as the kernel doesn't know about file names.

This is done by identifying the filesystem the inode belong to thanks to major and minor number information. Then, each filesystem containing one of our files is exhaustively traversed to build a lookup table allowing to find a file path for a given inode.

Of course, this can be very costly, but neither data gathering nor this postprocessing is meant to be run on a production system. This will only be done once during development.

5.3.3 Improving readahead in GNU/Linux distributions

Our first experiment will be to make minor changes to the utilities used in GNU/Linux distributions, so that they can process files lists also specifying which parts to readahead in each file.

5.3.4 Towards a generic and efficient implementation

While preloading the right file sections is easy once file access information is available, another requirement is to perform readahead at the right time in the execution flow. As explained before, reading ahead mustn't happen too early, and mustn't happen too late either.

Once more, the challenge is to predict the future by using knowledge about the past.

A very basic approach would be to collect time information together with file access data. However, such information wouldn't be very useful to trigger readahead at the right time, as reading ahead accelerates time. Furthermore, as processes spend less time waiting for I/O, the exact ordering of process execution can be altered.

Thus, what is needed is a way to follow the progress of system startup, and to match the actual events with recorded ones.

A simple idea is to use `inotify` to get access notifications for executables involved in system startup, and match these notifications with recorded `exec` calls.

---

[2] Even tracing these system calls is difficult. System call tracing is usually done on a process and its children with the `strace` command or with the `ptrace` system call that it uses. The problem is that `ptrace` cannot be used for the `init` process, which would have allowed tracing on all running processes at once.

Another, probably simpler solution would be to use *C library interposers*, wrappers around the C library functions used to execute system calls.

event10: first
execution of
/sbin/ifconfig.

event11: second
execution
of /bin/grep.

event12: first
execution of
/usr/bin/dillo

Previous readahead window: 1 MB
of data access recorded after event10.

New readahead window: 1 MB
of data access recorded after event11.

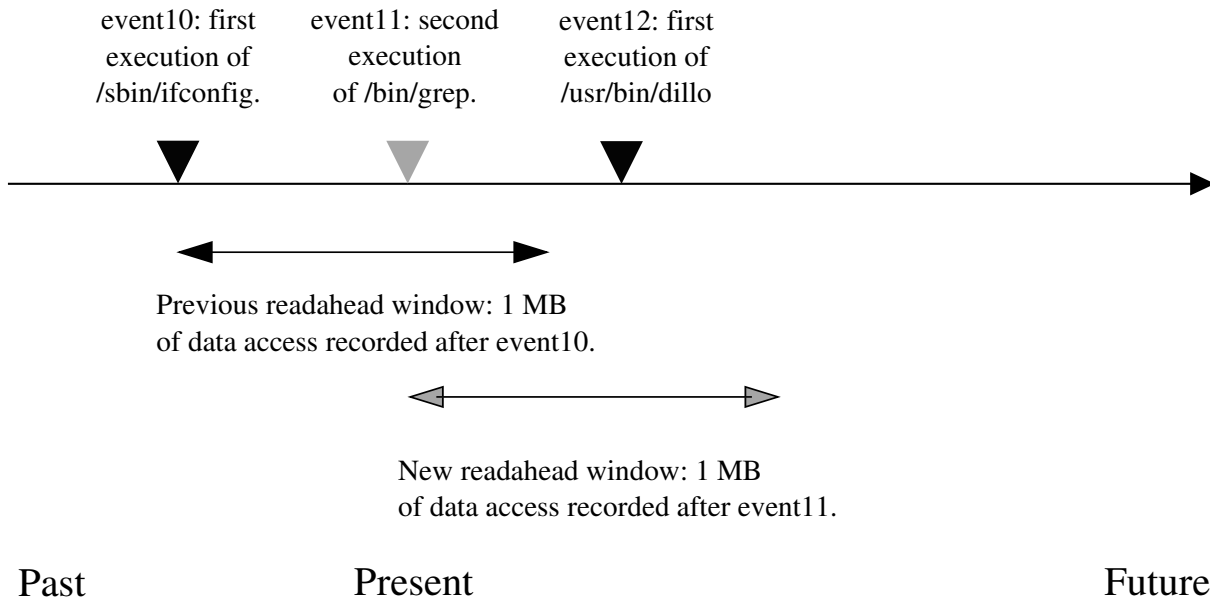Past   Present   Future

Figure 2: Proposed readahead implementation

This would be quite easy to implement, as this would just involve a list of files, without having to register recursive directory based notifications.

As shown in the example in Figure 2, our idea is to manage *readahead windows* of a given data size. In this example, when `event11` is recognized, we create a new readahead window starting from this event, corresponding to 1 MB of recorded disk access starting from this event.

Actually, we would only need to start new readahead I/O from the end of the previous window to the end of the new one. This assumes that the window size is large enough to extend beyond the next event. Otherwise, if readahead windows didn't overlap, there would be parts of the execution flow with no readahead at all.

Within a given window, before firing readahead I/O, we would of course need to remove any duplicate read operations, as well as to merge consecutive ones into single larger ones.

Here are the advantages of this approach:

- Possibility to readahead the same blocks multiple times in the execution flow. This covers the possibility that these blocks are no longer in page cache.

- For each specific system, possibility to tune the window size according to the best achieved results.

- The window size could even be dynamically increased, to make sure it goes beyond the next recorded event.

- If window size is large enough, we expect it to compensate for actual changes in the order of events.

5.3.5 Open issues

Several issues have not been addressed yet in this project.

In particular, we would need a methodology to support package updates in standard distributions. Would file access data harvesting be run again whenever a package involved in system startup is updated? Or should each package carry its own readahead information, requiring a more complex package development process?

5.4 Conclusion

Though the proposed ideas haven't been fully implemented and benchmarked yet, we have identified promising opportunities to reduce system startup time, in a way that both meets the requirements of desktop and embedded Linux systems.

If you are interested in this topic, stay tuned on the project page [2], join the presentation at OLS 2007, discover the first benchmarks on embedded and desktop systems, and share your experience and ideas on accelerating the course of Time.

References

[1] Jonathan Corbet. Lwn article: Adaptive file readahead. `http://lwn.net/Articles/155510/`, October 2005.

[2] Free Electrons. Advanced readahead project. `http://free-electrons.com/community/tools/readahead/`.

[3] WU Fenguang. Linux kernel mailing list: Adaptive readahead v16 benchmarks. `http://lkml.org/lkml/2006/11/25/7`, November 2006.

[4] Linux Manual Pages. madvise(2) - linux man page. `http://www.die.net/doc/linux/man/man2/madvise.2.html`.

[5] Linux Manual Pages. posix_fadvise(2) - linux man page. `http://www.die.net/doc/linux/man/man2/posix_fadvise.2.html`.

[6] Linux Manual Pages. readahead(2) - linux man page. `http://www.die.net/doc/linux/man/man2/readahead.2.html`.

[7] Ram Pai, Badari Pulavarty, and Mingming Cao. Linux 2.6 performance improvement through readahead optimization. In *Ottawa Linux Symposium (OLS)*, 2004. `http://www.linuxsymposium.org/proceedings/reprints/Reprint-Pai-OLS2004.pdf`.

[8] Wikipedia. inotify. `http://en.wikipedia.org/wiki/Inotify`.

Semantic Patches

Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau
EMN
padator@wanadoo.fr

Julia L. Lawall
DIKU
julia@diku.dk

Gilles Muller
EMN
Gilles.Muller@emn.fr

1 Introduction

Device drivers form the glue code between an operating system and its devices. In Linux, device drivers are highly reliant for this on the various Linux internal libraries, which encapsulate generic functionalities related to the various busses and device types. In recent years, these libraries have been evolving rapidly, to address new requirements and improve performance. In response to each evolution, *collateral evolutions* are often required in driver code, to bring the drivers up to date with the new library API. Currently, collateral evolutions are mostly done manually. The large number of drivers, however, implies that this approach is time-consuming and unreliable, leading to subtle errors when modifications are not done consistently.

To address this problem, we propose a scripting language for specifying and automating collateral evolutions. This language offers a WYSIWYG approach to program transformation. In the spirit of Linux development practice, this language is based on the patch syntax. As opposed to traditional patches, our patches are not line-oriented but semantics-oriented, and hence we give them the name *semantic patches*.

This paper gives a tutorial on our semantic patch language, SmPL, and its associated transformation tool, spatch. We first give an idea of the kind of program transformations we target, collateral evolutions, and then present SmPL using an example based on Linux driver code. We then present some further examples of evolutions and collateral evolutions that illustrate other issues in semantic patch development. Finally, we describe the current status of our project and propose some future work. Our work is directed mainly to device driver maintainers, library developers, and kernel janitors, but anyone who has ever performed a repetitive editing task on C code can benefit from it.

2 Evolutions and Collateral Evolutions

The evolutions we consider are those that affect a library API. Elements of a library API that can be affected include functions, both those defined by the library and the callback functions that the library expects to receive from a driver, global variables and constants, types, and macros. A library may also implicitly specify rules for using these elements. Many kinds of changes in the API can result from an evolution that affect one of these elements. For example, functions or macros can change name or gain or lose arguments. Structure types can be reorganized and accesses to them can be encapsulated in getter and setter functions. The protocol for using a sequence of functions, such as up and down can change, as can the protocol for when error checking is needed and what kind of error values should be returned.

Each of these changes requires corresponding collateral evolutions, in all drivers using the API. When a function or macro changes name, all callers need to be updated with the new name. When a function or macro gains or loses arguments, new argument values have to be constructed and old ones have to be dropped in the driver code, respectively. When a structure type is reorganized, all drivers accessing affected fields of that structure have to be updated, either to perform the new field references or to use any introduced getter and setter functions. Changes in protocols may require a whole sequence of modifications, to remove the old code and introduce the new. Many of these collateral evolutions can have a non-local effect, as, for example, changing a new argument value may trigger a whole set of new computations, and changing a protocol may require substantial code restructuring. The interaction of a driver with the API may furthermore include some device-specific aspects. Thus, these changes have to be mapped onto the structure of each affected driver file.

We have characterized evoluations and collateral evolu-

tions in more detail, including numerous examples, in a paper at EuroSys 2006 [1].

3 Semantic Patch Tutorial

In this section, we describe SmPL (Semantic Patch Language), our language for writing semantic patches. To motivate the features of SmPL, we first consider a moderately complex collateral evolution that raises many typical issues. We then present SmPL in terms of this example.

3.1 The "proc_info" evolution

As an example, we consider an evolution and associated collateral evolutions affecting the SCSI API functions `scsi_host_hn_get` and `scsi_host_put`. These functions access and release, respectively, a structure of type `Scsi_Host`, and additionally increment and decrement, respectively, a reference count. In Linux 2.5.71, it was decided that, due to the criticality of the reference count, driver code could not be trusted to use these functions correctly and they were removed from the SCSI API [2].

This evolution had collateral effects on the "proc_info" callback functions defined by SCSI drivers, which call these API functions. Figure 1 shows a slightly simplified excerpt of the traditional patch file updating the proc_info function of `drivers/usb/storage/scsiglue.c`. Similar collateral evolutions were performed in Linux 2.5.71 in 18 other SCSI driver files inside the kernel source tree. To compensate for the removal of `scsi_host_hn_get` and `scsi_host_put`, the SCSI library began in Linux 2.5.71 to pass to these callback functions a `Scsi_Host`-typed structure as an argument. Collateral evolutions were then needed in all the proc_info functions to remove the calls to `scsi_host_hn_get` (line 19 for the `scsiglue.c` driver), and `scsi_host_put` (lines 27 and 42), and to add the new argument (line 4). Those changes in turn entailed the removal of a local variable (line 11) and of null-checking code (line 20-22), as the library is assumed not to call the proc_info function on a null value. Finally, one of the parameters of the proc_info function was dropped (line 6) and every use of this parameter was replaced by a field access (line 33) on the new structure argument.

```
0   --- a/drivers/usb/storage/scsiglue.c
1   +++ b/drivers/usb/storage/scsiglue.c
2   @@ -264,33 +300,21 @@
3   -static int usb_storage_proc_info (
4   +static int usb_storage_proc_info (struct Scsi_Host *hostptr,
5                 char *buffer, char **start, off_t offset,
6   -             int hostno, int inout)
7   +             int inout)
8    {
9      struct us_data *us;
10     char *pos = buffer;
11   - struct Scsi_Host *hostptr;
12     unsigned long f;
13
14     /* if someone is sending us data, just throw it away */
15     if (inout)
16           return offset;
17
18   - /* find our data from the given hostno */
19   - hostptr = scsi_host_hn_get(hostno);
20   - if (!hostptr) {
21   -       return -ESRCH;
22   - }
23     us = (struct us_data*)hostptr->hostdata[0];
24
25     /* if we couldn't find it, we return an error */
26     if (!us) {
27   -       scsi_host_put(hostptr);
28           return -ESRCH;
29     }
30
31     /* print the controller name */
32   - SPRINTF("   Host scsi%d: usb-storage\n", hostno);
33   + SPRINTF("   Host scsi%d: usb-storage\n", hostptr->host_no);
34     /* print product, vendor, and serial number strings */
35     SPRINTF("      Vendor: %s\n", us->vendor);
36
37   @@ -318,9 +342,6 @@
38         *(pos++) = '\n';
39       }
40
41   - /* release the reference count on this host */
42   - scsi_host_put(hostptr);
43
44     /*
45      * Calculate start of next buffer, and return value.
46
```

Figure 1: Simplified excerpt of the patch file from Linux 2.5.70 to Linux 2.5.71

Of the possible API changes identified in Section 2, this example illustrates the dropping of two library functions and changes in the parameter list of a callback function. These changes have non-local effects in the driver code, as the context of the dropped call to `scsi_host_hn_get` must change as well, to eliminate the storage of the result and the subsequent error check, and the value of the dropped parameter must be reconstructed wherever it is used.

3.2 A semantic patch, step by step

We now describe the semantic patch that will perform the previous collateral evolutions, on any of the 19 relevant files inside the kernel source tree, and on any relevant drivers outside the kernel source tree. We first describe step-by-step various excerpts of this semantic

patch, and then present its complete definition in Section 3.3.

3.2.1 Modifiers

The first excerpt adds and removes the affected parameters of the proc_info callback function:

```
  proc_info_func (
+     struct Scsi_Host *hostptr,
      char *buffer, char **start, off_t offset,
-     int hostno,
      int inout) { ... }
```

Like a traditional patch, a semantic patch consists of a sequence of lines, some of which begin with the *modifiers* + and − in the first column. These lines are added or removed, respectively. The remaining lines serve as *context*, to more precisely identify the code that should be modified.

Unlike a traditional patch, a semantic patch must have the form of a complete C-language term (an expression, a statement, a function definition, etc.). Here we are modifying a function, so the semantic patch has the form of a function definition. Because our only goal at this point is to modify the parameter list, we do not care about the function body. Thus, we have represented it with "...". The meaning and use of "..." are described in more detail in Section 3.2.4.

Because of the wide range of possible collateral evolutions, as described in Section 2, collateral evolutions may affect almost any C constructs, such as structures, initializers, function parameters, if statements, in many different ways. SmPL, for flexibility, allows to write almost any C code in a semantic patch and to annotate freely any part of this code with the + and − modifiers. The combination of the unannotated context code with the − code and the combination of the unannotated context code with the + code must, however, each have the form of valid C code, to ensure that the pattern described by the former can match against valid driver code and that the transformation described by the latter will produce valid C code as a result.

Another difference as compared to a traditional patch is that the meaning of a semantic patch is insensitive to newlines, spaces, comments, etc. Thus, the above semantic patch will *match* and *transform* driver code that

has the parameters of the proc_info function all on the same line, spread over multiple lines as in scsiglue.c, or separated by comments. We have split the semantic patch over four lines only to better highlight what is added and removed. We could have equivalently written it as:

```
- proc_info_func(char *buffer, char **start, off_t offset,
-               int hostno, int inout)
+ proc_info_func(struct Scsi_Host *hostptr, char *buffer,
+               char **start, off_t offset, int inout)

    { ... }
```

To apply this semantic patch, it should be stored in a file, *e.g.*, procinfo.spatch. It can then be applied to *e.g.* the set of C files in the current directory using our spatch tool:

spatch *.c < procinfo.spatch.

3.2.2 Metavariables

A traditional patch, like the one in Figure 1, describes a transformation of a specific set of lines in a specific driver. This specificity is due to the fact that a patch hardcodes some information, such as the name of the driver's proc_info callback function. Thus a separate patch is typically needed for every driver. The goal of SmPL on the other hand is to write a generic semantic patch that can transform all the relevant drivers, accommodating the variations among them. In this section and the following ones we describe the features of SmPL that make a semantic patch generic.

In the excerpt of the previous section, the reader may have wondered about the name proc_info_func, which indeed does not match the name of the scsiglue proc_info function, as shown in Figure 1, lines 3 and 4, or the names of any of the proc_info functions in the kernel source tree. Furthermore, the names of the parameters are not necessarily buffer, start, etc.; in particular, the introduced parameter hostptr is sometimes called simply host. To abstract away from these variations, SmPL provides *metavariables*. A metavariable is a variable that matches an arbitrary term in the driver source code. Metavariables are declared before the patch code specifying the transformation, between two @@s, borrowing the notation for delimiting line numbers in a traditional patch (Figure 1, lines 2 and 37). Metavariables are designated as matching terms of a specific kind, such as an identifier, expression,

or `statement`, or terms of a specific type, such as `int` or `off_t`. We call the combination of the declaration of a set of metavariables and a transformation specification a *rule*.

Back to our running example, the previous excerpt is made into a rule as follows:

```
@@
identifier proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
  proc_info_func (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
-       int hostno,
        int inout) { ... }
```

This code now amounts to a complete, valid semantic patch, although it still only performs part of our desired collateral evolution.

3.2.3 Multiple rules and inherited metavariables

The previous excerpt matches and transforms any function with parameters of the specified types. A proc_info function, however, is one that has these properties and interacts with the SCSI library in a specific way, namely by being provided by the driver to the SCSI library on the `proc_info` field of a `SHT` structure (for SCSI Host Template), which from the point of view of the SCSI library represents the device. To specify this constraint, we define another rule that identifies any assignment to such a field in the driver file. SmPL allows a semantic patch to define multiple rules, just as a traditional patch contains multiple regions separated by `@@`. The rules are applied in sequence, with each of them being applied to the entire source code of the driver. In our example we thus define one rule to identify the name of the callback function and another to transform its definition, as follows:

```
@ rule1 @
struct SHT ops;
identifier proc_info_func;
@@
  ops.proc_info = proc_info_func;

@ rule2 @
identifier rule1.proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
```

```
  proc_info_func (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
-       int hostno,
        int inout) { ... }
```

In the new semantic patch, the metavariable `proc_info_func` is defined in the first rule and referenced in the second rule, where we expect it to have the same value, which is enforced by `spatch`. In general, a rule may declare new metavariables and *inherit* metavariables from previous rules. Inheritance is explicit, in that the inherited metavariable must be declared again in the inheriting rule, and is associated with the name of the rule from which its value should be inherited (the rule name is only used in the metavariable declaration, but not in the transformation specification, which retains the form of ordinary C code). To allow this kind of inheritance, we must have means of naming rules. As shown in the semantic patch above, the name of a rule is placed between the two `@@`s at the beginning of a metavariable declaration. A name is optional, and is not needed if the rule does not export any metavariables.

Note that the first rule does not perform any transformations. Instead, its only role is to bind the `proc_info_func` metavariable to constrain the matching of the second rule. Once a metavariable obtains a value it keeps this value until the end of the current rule and in any subsequent rules that inherit it. Metavariables thus not only make a semantic patch generic by abstracting away from details of the driver code, but also allow communicating information and constraints from one part of the semantic patch to another, *e.g.*, from '–' code to '+' code, or from one rule to another.

A metavariable may take on multiple values, if the rule matches at multiple places in the driver code. If such a metavariable is inherited, the inheriting rule is applied once for each possible set of bindings of the metavariables it inherits. For example, in our case, a driver may set the `proc_info` field multiple times, to different functions, in which case rule 2 would be applied multiple times, for the names of each of them.

3.2.4 Sequences

So far, we have only considered the collateral evolutions on the header of the proc_info function. But collateral evolutions are needed in its body as well: deleting the

calls to `scsi_host_hn_get` and `scsi_host_put`, deleting the local variable holding the result of calling `scsi_host_hn_get` and the error checking code on its value. The affected code fragments are scattered throughout the body of the proc_info function and are separated from each other by arbitrary code specific to each SCSI driver. To abstract away from these irrelevant variations, SmPL provides the "`...`" operator, which matches any *sequence* of code. Refining rule2 of the semantic patch to perform these collateral evolutions gives:

```
@ rule2 @
identifier rule1.proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
  proc_info_func (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
-       int hostno,
        int inout) {
    ...
-   struct Scsi_Host *hostptr;
    ...
-   hostptr = scsi_host_hn_get(hostno);
    ...
-   if (!hostptr) { ... return ...; }
    ...
-   scsi_host_put(hostptr);
    ...
  }
```

The second rule of the semantic patch now has the form of the definition of a function that first contains a declaration of the `hostptr` variable, then a call to the function `scsi_host_hn_get`, then an error check, and finally a call to `scsi_host_put` sometime before the end. In practice, however, a proc_info function may *e.g.* contain many calls to `scsi_host_put`, as illustrated by the `scsiglue` example (Figure 1, lines 27 and 42). Closer inspection of the original `scsiglue` source code, however, shows that at execution time, the driver only executes one or the other of these calls to `scsi_host_put`, as the one on line 27 is only executed in an error case, and the one on line 42 is only executed in a non-error case. This is illustrated by Figure 2, which shows part of the control-flow graph of the `scsiglue` proc_info function. Because the execution pattern declare/`scsi_host_hn_get`/error-check/`scsi_host_put` is what must be followed by every SCSI proc_info driver, it is this pattern that the semantic patch should match against. The operator "`...`" thus matches paths in the control-flow graph rather than an arbitrary block of code in the driver source

code. Thus, in practice, a single minus or plus line in the semantic patch can delete or add multiple lines in the source code of the driver.

The transformation specified in a rule is applied on driver code only if the whole rule matches code, not if only parts of the rule match code. Thus, here, the rule only matches proc_info callback functions having 5 parameters of the specified types, *and* the sequence of instructions declare/`scsi_host_hn_get`/error-check/`scsi_host_put`, *and* where these instructions share the use of the same variable, represented in the semantic patch by the repeated use of the same metavariable `hostptr`.

As said in the previous section, the repeated use of the same metavariable, here `hostptr`, can serve multiple purposes. First, it is used here to constrain some transformations by forcing two pieces of code to be equal in the driver code. So, for example, not all conditionals will be removed in the driver, only those testing the local structure returned by `scsi_host_hn_get`. Metavariables are also used to move code from one place to another. Here `hostptr` is used to move the matched local variable name into the parameter list. Metavariables declared as `expression` or `statement` can be used to move more complex terms.

3.2.5 Nested Sequences

The last transformation concerning the proc_info function is the replacement of every reference to the dropped `hostno` parameter by a field access. SmPL provides the `<... ...>` operator to perform such universal replacements. This operator is analogous to the `/g` operator of Perl. In order to avoid having to consider how references to `hostno` may interleave with the calls to `scsi_host_hn_get` and `scsi_host_put`, etc., we define a third rule that simply makes this transformation everywhere it applies:

```
@ rule3 @
identifier rule1.proc_info_func;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
  proc_info_func(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }
```
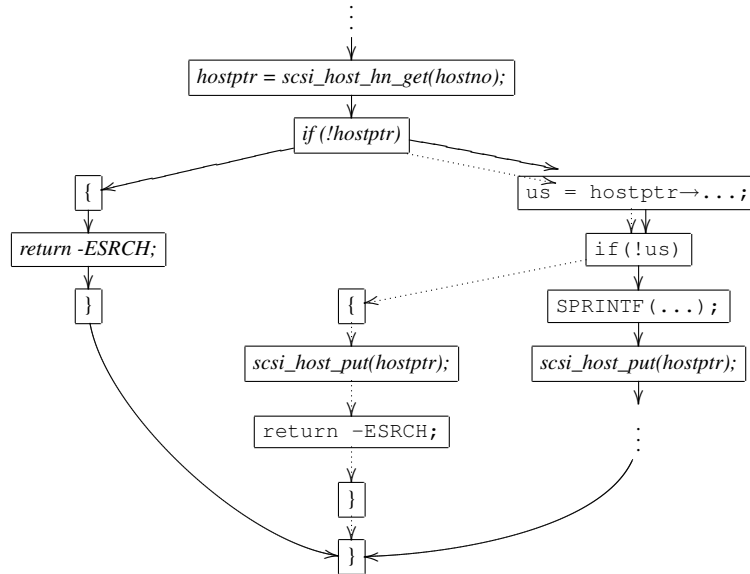
Figure 2: Simplified control-flow graph for part of Figure 1

Note that the operator "...." can be used to represent any kind of sequence. Here, in the function header, it is used to represent a sequence of parameters. It can also be used to provide flexible matching in initializers and structure definitions.

3.2.6 Isomorphisms

We have already mentioned that a semantic patch is insensitive to spacing, indentation and comments. Moreover, by defining sequences in terms of control-flow paths, we abstract away from the various ways of sequencing instructions that exist in C code. These features help make a semantic patch generic, allowing the patch developer to specify only a few scenarios, while spatch handles other scenarios that are semantically equivalent.

Other differences that we would like to abstract away from include variations within the use of specific C constructs. For example, if *x* is any expression that has pointer type, then !*x*, *x* == NULL, and NULL == *x* are all equivalent. For this, we provide a variant of the SmPL syntax for defining *isomorphisms*, sets of syntactically different terms that have the same semantics. The null pointer-test isomorphism is defined in this variant of SmPL as follows:

```
// iso file, not a semantic patch
@@ expression *X; @@
X == NULL <=> !X <=> NULL == X
```

Given this specification, the pattern if(!hostptr) in the semantic patch matches a conditional in the driver code that tests the value of hostptr using any of the listed variants.

In addition to a semantic patch, spatch accepts a file of isomorphisms as an extra argument. A file of isomorphisms is provided with the spatch distribution, which contains 30 equivalences commonly found in driver code. Finally, it is possible to specify that a single rule should use only the isomorphisms in a specific file, *file*, by annotating the rule name with using *file*.

3.3 All Together Now

The complete semantic patch for the proc_info collateral evolutions is shown below. As compared to the rules described above, this semantic patch contains an additional rule, rule4, which adjusts any calls to the proc_info function from within the driver. Note that in this rule, the metavariables that were declared as identifiers in rule2 to represent the parameters of the proc_info function are redeclared as expressions, to represent the proc_info function's arguments.

The second rule has also been slightly modified, in that two lines have been annotated with the "?" operator stating that those lines may or may not be present in the driver. Indeed, many drivers forget to check the return value of scsi_host_hn_get or forget to release

the structure before exiting the function. As previously noted, the latter omission is indeed what motivated the proc_info evolution.

Note that there is no rule for updating the prototype of the proc_info function, if one is contained in the file. When the type of a function changes, `spatch` automatically updates its prototype, if any.

```
@ rule1 @
struct SHT ops;
identifier proc_info_func;
@@
  ops.proc_info = proc_info_func;

@ rule2 @
identifier rule1.proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
  proc_info_func (
+     struct Scsi_Host *hostptr,
      char *buffer, char **start, off_t offset,
-     int hostno,
      int inout) {
    ...
-   struct Scsi_Host *hostptr;
    ...
-   hostptr = scsi_host_hn_get(hostno);
    ...
?-  if (!hostptr) { ... return ...; }
    ...
?-  scsi_host_put(hostptr);
    ...
  }

@ rule3 @
identifier rule1.proc_info_func;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
  proc_info_func(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }

@ rule4 @
identifier rule1.proc_info_func;
identifier func;
expression buffer, start, offset, inout, hostno;
identifier hostptr;
@@
 func(..., struct Scsi_Host *hostptr, ...) {
   <...
    proc_info_func(
+       hostptr,
        buffer, start, offset,,
-       hostno,
        inout)
    ...>
 }
```

On the 30 isomorphisms we have written, 3 of them "apply" to this semantic patch, accommodating many varia-

tions among the 19 drivers inside the kernel source tree. We have already mentioned the different ways to write a test such as `if(!hostptr)` in the previous section. There is also the various ways to assign a value in a field, which can be written `ops.proc_info = fn` as in our semantic patch, or written `ops->proc_info = fn` in some drivers, or written using a global structure initializer. Indeed, the last case was used for the `scsiglue.c` driver as shown by the following excerpt of this driver:

```
struct SHT usb_stor_host_template = {
  /* basic userland interface stuff */
  .name = "usb-storage",
  .proc_name = "usb-storage",
  .proc_info = usb_storage_proc_info,
  .proc_dir = NULL,
```

Finally, braces are not needed in C code when a branch contains only one statement. So, the pattern `{ .. . return ...; }` in rule2 also matches a branch containing only the return statement.

It takes 23 seconds to `spatch` given the whole semantic patch to correctly update the 19 relevant drivers. If run on all the 2404 driver files inside the kernel source tree, it takes `spatch` 3 minutes to correctly update the same 19 drivers.

4 More Features, More Examples

So far we have written and tested 49 semantic patches for collateral evolutions found in the Linux 2.5 and Linux 2.6. By comparing the results produced by the semantic patch to the results produced by the traditional patch, we have found that `spatch` updates 92% of the driver files affected by these collateral evolutions correctly. In the remaining cases, there is typically a problem parsing the driver code, or needed information is missing because `spatch` currently does not parse header files. Parsing the driver code is a particular problem in our case, because our goal is to perform a source-to-source transformation, which means that we have chosen not to expand macros and preprocessor directives, and instead parse them directly.

In this section, we consider some other examples from our test suite, to illustrate some typical issues in semantic patch development.

4.1 Replacing one function name by another

In Linux 2.5.22, the function `end_request` was given a new first argument, of type `struct request *`. In practice, the value of this argument should be the next request from one of the driver's queue, as represented by a reference to the macro `CURRENT`. This collateral evolution affected 27 files spread across the directories `acorn`, `block`, `cdrom`, `ide`, `mtd`, `s390`, `sbus`.

The following semantic patch implements this collateral evolution:

```
@@ expression X; @@
- end_request(X)
+ end_request(CURRENT,X)
```

This semantic patch updates the 27 affected files in the Linux source tree correctly.

This example may seem almost too simple to be worth writing an explicit specification, as one can *e.g.* write a one-line `sed` command that has the same effect. Nevertheless, such solutions are error prone: we found that in the file `drivers/block/swim_iop.c`, the transformation was applied to the function `swimiop_send_request`, which has no relation to this collateral evolution. We conjecture that this is the result of applying a `sed` command, or some similar script, that replaces calls to `end_request` without checking whether this string is part of a more complicated function name. `spatch` enforces the syntactic structure of semantic patch code, allowing matches on identifier, expression, statement, etc. boundaries, rather than simply accepting anything that a superstring of the given pattern.

4.2 Collecting scattered information

In Linux 2.5.7, the function `video_generic_ioctl`, later renamed `video_usercopy`, was introduced to encapsulate the copying to and from user space required by `ioctl` functions. Ioctl functions allow the user level to configure and control a device, as they accept commands from the user level and perform the corresponding action at the kernel level. Without `video_usercopy`, an ioctl function has to use functions such as `copy_from_user` or `get_user` to access data passed in with the command, and functions such as `copy_to_user` or `put_user` to return information to the user level. With `video_usercopy`, the ioctl function receives a pointer to a kernel-level data structure containing the user-level arguments and can modify this data structure to return any values to user level.

Making an ioctl function `video_usercopy`-ready involves the following steps:

- Adding some new parameters to the function.

- Eliminating calls to `copy_from_user`, `put_user`, etc.

- Changing the references to the local structure used by these functions to use the pointer prepared by `video_usercopy`.

The last two points are somewhat complex, because the various commands interpreted by the ioctl function may each have their own requirements with respect to the user-level data. A command may or may not have a user-level argument, and it may or may not return a result to the user level. In the case where there is no use or returned value then no transformation should be performed; in the other cases, the structure containing the user-level argument or result should be converted to a pointer. Furthermore, there are multiple possible copying functions, and there are multiple forms that the references to the copied data can take.

Figure 3 shows a semantic patch implementing this transformation, under the simplifying assumption that the kernel-level representation of the user-level data is stored in a locally declared structure. This semantic patch consists of a single rule that changes the prototype of this function (adding some new variables, as indicated by `fresh identifier`), changes the types of the local structures, and removes the copy functions.

The many variations in an ioctl function noted above are visible in this rule. To express multiple possibilities, SmPL provides a *disjunction operator*, which begins with an open parenthesis in column 0, contains a list of possible patterns separated by a vertical bar in column 0, and then ends with a close parenthesis in column 0. Most of the body of the ioctl function pattern is represented as one large disjunction that considers the possibility of there being both a user-level argument and a user-level return value (lines 14-39), the possibility of there being a user-level argument but no user-level

```
 1 @@
 2 identifier ioctl, dev, cmd, arg, v, fld;
 3 fresh identifier inode, file;
 4 expression E, E1, e1,e2,e3;
 5 type T;
 6 @@
 7   ioctl(
 8 -       struct video_device *dev,
 9 +       struct inode *inode, struct file *file,
10         unsigned int cmd, void *arg) {
11 +     struct video_device *dev = video_devdata(file);
12       ...
13 (
14 -     T v;
15 +     T *v = arg;
16       ...
17 (
18 -     if (copy_from_user(&v,arg,E)) { ... return ...; }
19 |
20 -     if (get_user(v,(T *)arg)) { ... return ...; }
21 )
22     <...
23 (
24 -     v.fld
25 +     v->fld
26 |
27 -     &v
28 +     v
29 |
30 -     v
31 +     *v
32 )
33     ...>
34 (
35 -   if (copy_to_user(arg,&v,E1)) { ... return ...; }
36 |
37 -   if (put_user(v,(T *)arg)) { ... return ...; }
38 )
39     ...
40 |
41 // a copy of the above pattern with the copy_to_user/put_user
42 // pattern dropped
43 |
44 // a copy of the above pattern with the copy_from_user/get_user
45 // pattern dropped
46 |
47   ... when != \(copy_from_user(e1,e2,e3)\|copy_to_user(e1,e2,e3)
48            \|get_user(e1,e2)\|put_user(e1,e2)\)
49 )
50   }
```

Figure 3: Semantic patch for the `video_ usercopy` collateral evolution

return value (elided in comments on line 41), the possibility of there being a user-level return value but no user-level argument (elided in comments on line 41-42), and there being neither a user-level argument nor a user-level return value (line 44-45). These possibilities are considered from top to bottom, with only the first one that matches being applied. This strategy is convenient in this case, because *e.g.* code using both a user-level argument and a user-level return value also matches all of the other patterns. The last cases uses "..." with the construct `when`. The `when` construct indicates a pattern that should not be matched anywhere in the code matched by the associated "...".

Within each of the branches of the outermost disjunction, there are several nested disjunctions. First, another disjunction is used to account for the two kinds of copy functions, `copy_from_user` or `get_user`. This case does not rely on the top-to-bottom strategy, because the patterns are disjoint. Next, between any copying, there is a nest (see Section 3.2.5) replacing the different variations on how to refer to a structure by the pointer-based counterpart. Here again, the ordering of the disjunction is essential, as the final case, `v`, should only be used when the variable is not used in a field access or address expression. Finally, there is a third disjunction allowing either `copy_to_user` or `put_user`.

Like the proc_info semantic patch, this semantic patch relies on isomorphisms. Specifically, the calls to the copy functions may appear alone in a conditional test as shown, or may be compared to 0, and as in the proc_info case, the return pattern in each of the conditional branches can match a single return statement, without braces.

4.3 Collecting scattered information

In Linux 2.6.20, the strategy for creating work queues and setting and invoking their callback functions changed as follows:

- Previously, all work queues were declared with some variant of `INIT_WORK`, and then could choose between delayed or undelayed work dynamically, by using either some variant of `schedule_work` or some variant of `schedule_delayed_work`. Since the changes in Linux 2.6.20, the choice between delayed or undelayed work has to be made statically, by creating the work queue with either `INIT_DELAYED_WORK` or `INIT_WORK`, respectively.

- Previously, creation of a work queue took as arguments a queue, a callback function, and a pointer to the value to be passed as an argument to the callback function. Since 2.6.20, the third argument is dropped, and the callback function is simply passed the work queue as an argument. From this, it can access the local data structure containing the queue, which can itself store whatever information was required by the callback function.

For simplicity, we consider only the case where the work queue is created using `INIT_WORK`, where it is the field of a local structure, and where the callback function

passed to `INIT_WORK` expects this local structure as an argument.

Figure 4 shows the semantic patch. In this semantic patch, we name all of the rules, to ease the presentation, but only those with descriptive names, such as `is_delayed`, are necessary.

The semantic patch is divided into two sections, the first for the case where the work queue is somewhere used with a delaying function such as `schedule_delayed_work` and the second for the case where such a function is not used on the work queue. Both cases can occur within a single driver, for different work queues. The choice between these two variants is made at the first rule, `is_delayed`, using a trick based on metavariable binding. This rule matches all calls to `schedule_delayed_work` and other functions indicating delayed work, for any work queue `&device->fld` and arbitrary task `E`. The next five rules, up to the commented dividing line, refer directly or indirectly to the type of the structure containing the matched work queue `&device->fld`, and thus these rules are only applied to work queues for which the match in `is_delayed` somewhere succeeds. The remaining three rules, at the bottom of the semantic patch, do not depend on the rule `is_delayed`, and thus apply to work queues for which there is no call to any delaying function.

In the first half of the semantic patch, the next task is to convert any call to a non-delaying work queue function to a delaying one, by adding a delay of 0 (`rule2`). The rule `delayed_fn` then changes calls to `INIT_WORK` to calls to `INIT_DELAYED_WORK` and adjusts the argument lists such that the cast on the second argument (the work queue callback function) is dropped and the third argument is dropped completely. Note that the casts on the second and third arguments need not be present in the driver code, thanks to an isomorphism. Next (`rule4`), the work queue is changed from having type `work_struct` to having type `delayed_work`. The last two rules of this section, `rule5` and `rule5a`, update the callback functions identified in the call to `INIT_WORK`. The first, `rule5`, is for the case where the current parameter type is `void *` and the second, `rule5a`, is for the case where the parameter type is the type of the local structure containing the work queue. In both cases, the parameter is given the type `struct work_struct`, and then code using the macro `container_of` is added to the body of the function to reconstruct the original argument value.

```
@ is_delayed @
type local_type; local_type *device; expression E,E1;
identifier fld;
@@
( schedule_delayed_work(&device->fld,E)
| cancel_delayed_work(&device->fld)
| schedule_delayed_work_on(E1,&device->fld,E)
| queue_delayed_work(E1,&device->fld,E)
)

@ rule2 @
is_delayed.local_type *device;
identifier is_delayed.fld; expression E1;
@@
(
- schedule_work(&device->fld)
+ schedule_delayed_work(&device->fld,0)
|
- schedule_work_on(E1,&device->fld)
+ schedule_delayed_work_on(E1,&device->fld,0)
|
- queue_work(E1,&device->fld)
+ queue_delayed_work(E1,&device->fld,0)
)

@ delayed_fn @
type T,T1; identifier is_delayed.fld, fn;
is_delayed.local_type *device;
@@
- INIT_WORK(&device->fld, (T)fn, (T1)device);
+ INIT_DELAYED_WORK(&device->fld, fn);

@ rule4 @
type is_delayed.local_type; identifier is_delayed.fld;
@@
local_type { ...
-       struct work_struct fld;
+       struct delayed_work fld;
        ... };

@ rule5 @
identifier data, delayed_fn.fn, is_delayed.fld;
type T, is_delayed.local_type; fresh identifier work;
@@
- fn(void *data) {
+ fn(struct work_struct *work) {
    <...
-     (T) data
+     container_of(work,local_type,fld.work)
    ...>
  }

@ rule5a @
identifier data, delayed_fn.fn, is_delayed.fld;
type is_delayed.local_type; fresh identifier work;
@@
- fn(local_type *data) {
+ fn(struct work_struct *work) {
+   local_type *data = container_of(work,local_type,fld.work);
    ...
  }
//----------------------------------------------------------
@ non_delayed_fn @
type local_type, T,T1; local_type *device; identifier fld, fn;
@@
- INIT_WORK(&device->fld, (T)fn, (T1)device);
+ INIT_WORK(&device->fld, fn);

@ rule7 @
identifier data, non_delayed_fn.fn, non_delayed_fn.fld;
type T, non_delayed_fn.local_type; fresh identifier work;
@@
- fn(void *data) {
+ fn(struct work_struct *work) {
    <...
-     (T) data
+     container_of(work,local_type,fld)
    ...>
  }

@ rule7a @
identifier data, non_delayed_fn.fn, non_delayed_fn.fld;
type non_delayed_fn.local_type; fresh identifier work;
@@
- fn(local_type *data) {
+ fn(struct work_struct *work) {
+   local_type *data = container_of(work,local_type,fld);
    ...
  }
```

Figure 4: Semantic patch for the `INIT_WORK` collateral evolution

In the second section, calls to `INIT_WORK` for non-delayed work queues have their second and third arguments transformed as in the delayed case. The rules `rule7` and `rule7a` then update the callback functions analogously to rules `rule5` and `rule5a`.

Using the Linux 2.6 git repository [3], we have identified 245 driver files that use work queues. Of these, 45% satisfy the assumptions on which this semantic patch is based. This semantic patch applies correctly to 91% of them. The remaining cases are due to some constructs that are not treated adequately by our approach, to inter-file effects, and to some optimizations made by the programmer that are too special-purpose to be reasonable to add to a generic transformation rule.

5   Conclusion

In this paper we have presented SmPL, our scripting language to automate and document collateral evolutions in Linux device drivers. This language is based on the patch syntax, familiar to Linux developers, but accommodates many variations among drivers. As opposed to a traditional patch, a single semantic patch can update hundreds of drivers at thousands of code sites because of the features of SmPL, including the use of metavariables, isomorphisms, and control-flow paths, which makes a semantic patch generic. We hope that the use of semantic patches will make collateral evolutions in Linux less tedious and more reliable. We also hope that it will help developers with drivers outside the kernel source tree to better cope with the fast evolution of Linux.

Until now we have tried to replay what was already done by Linux programmers. We would like now to interact with the Linux community and really contribute to Linux by implementing or assisting library developers in performing new evolutions and collateral evolutions. As a first step we have subscribed to the janitors kernel mailing list and planned to contribute by automating some known janitorings [4]. We would also like to investigate if SmPL could be used to perform collateral evolutions in other Linux subsystems such as filesystems or network protocols or to perform other kinds of program transformations.

Finally, introducing semantic patches in the development process may lead to new processes, or new tools. For instance, how can semantic patches be integrated in versioning tools such as `git`. We could imagine a versioning tool aware of semantic patches and of the semantics of C, that could for example automatically update new drivers coming from outside the kernel source tree with respect to some recent semantic patches. Semantic patches, due to their degree of genericity, can also help with the problem of conflicts between multiple patches that are developed concurrently and affect some common lines of code, but in an orthogonal way. Finally, for the same reason, semantic patches should be more portable from one Linux version to the next, in the case of a patch that is not immediately accepted into the Linux kernel source tree.

All the semantic patches we have written, as well as a binary version of `spatch`, are available on our website: `http://www.emn.fr/x-info/coccinelle`. Reading those semantic patches can give a better feeling of the expressivity of SmPL. They can also be used as a complement to this tutorial.

References

[1] "Understanding Collateral Evolution in Linux Device Drivers." Yoann Padioleau, Julia L. Lawall, and Gilles Muller. *Proceedings of the ACM SIGOPS EuroSys 2006 Conference*, Leuven, Belgium, April, 2006, pages 59–71.

[2] `http://lwn.net/Articles/36311/`.

[3] `http://git.kernel.org/git/?p=linux/ kernel/git/torvalds/linux-2.6.git;a= summary`.

[4] `http://kernelnewbies.org/ KernelJanitors/Todo`.

cpuidle—Do nothing, efficiently…

Venkatesh Pallipadi
Shaohua Li
Intel Open Source Technology Center
`{venkatesh.pallipadi|shaohua.li}@intel.com`

Adam Belay
Novell, Inc.
`abelay@novell.com`

Abstract

Most of the focus in Linux processor power management today has been on power managing the processor while it is active: `cpufreq`, which changes the processor frequency and/or voltage and manages the processor performance levels and power consumption based on processor load. Another dimension of processor power management is processor 'idling' power.

Almost all mobile processors in the marketplace today support the concept of multiple processor idle states with varying amounts of power consumed in those idle states. Each such state will have an entry-exit latency associated with it. In general, there is a lot of attention shifting towards idle platform power and new platforms/processors are supporting multiple idle states with different power and wakeup latency characteristics. This emphasis on idle power and different processors supporting different number of idle states and different ways of entering these states, necessitates the need for a generic Linux kernel framework to manage idle processors.

This paper covers `cpuidle`, an effort towards a generic processor idle management framework in Linux kernel. The goal is to have a clean interface for any processor hardware to make use of different processor idle levels and also provide abstraction between idle-drivers and idle-governors allowing independent development of drivers and governors. The target audiences are the developers who are keen to experiment with new idle governors on top of `cpuidle`, and developers who wants to use the `cpuidle` driver infrastructure in various architectures, and any one else who is keen to know about `cpuidle`.

1 Introduction

Almost all the mobile processors today support multiple idle states and the trend is spreading as processor power management and system power management gain importance for a variety of reasons.

In typical system usage models, processor(s) spend a lot of their time idling (like while you are reading this paper on your laptop, with your favorite pdf-reader). Thus any power saved when system is idle will have big returns in terms of battery life, heat generated in the system, need for cooling, etc.

But there is a trade-off between idling power and amount of state a processor saves and the amount of time it takes to enter and exit from this idle state. The idle enter-exit latency, if it is too high, may be visible with media applications like a DVD player. Such usage models will limit the usage of a particular idle state on the processor running this application, even though the idle state is power efficient. Similarly, if a processor idle state does not preserve the the contents of the processor's cache, some particular application which has some idle time may notice a performance degradation when this particular idle state is used.

In order to manage this trade-off effectively, the kernel needs to know the characteristics of all idle states and also should understand the currently running applications, and should take a well-informed decision about what idle state it wants to enter when processor goes to idle.

To do this effectively and cleanly, there is a preliminary requirement of having clean and simple interfaces. Such an interface can provide consistent information to the user and ease the innovation and development in the area of processor idle management.

`cpuidle` is a an effort in this direction and this paper provides insight into `cpuidle`. We start section 2 with a background on processor power management and idle states. Section 3 provides the design description of `cpuidle`. Section 4 talks about all the develop-

ments and advancements happening in `cpuidle` and some conclusions in section 5.

2 Background

2.1 Processor Power management

Processor power management can be broadly classified into two classes.

Processor active – various states a processor can be in while actively executing and retiring instructions. Processor frequency scaling, in which a processor can run at different frequencies and or voltages falls under this class. So does processor thermal throttling, where processor runs slower due to duty cycle throttling.

Linux `cpufreq`, extensively discussed in [4], [6], and [5], is a generic infrastructure that handles CPU frequency scaling.

Processor idle – various states a processor can be in while it is idle and not retiring any instructions. The states here differ in amount of power the processor consumes while being in that state and also the latency to enter-exit this low-power idle state. There may also be other differences like preserving the processor state across these idle states, etc. based on a specific processor. For example, a processor may only flush L1 cache in one idle state, but may flush L1 and L2 caches in another idle state. There can also be differences around when an idle state can be entered and what its impact will be on other logical or physical processors in the system.

2.2 Processor idle states

Currently, most of the processors in mobile and handheld segments support multiple idle states. The prime objective here is to provide a more power-efficient system with longer battery life or fewer cooling requirements. This feature is slowly moving up the chain into desktops and servers. This is much like processor frequency scaling which was mostly present in mobile processors a few years back, to most of the servers supporting that feature today. Recent EnergyStar idle power regulations [2] are tending to make this faster, making this feature more common across a range of systems.

2.3 Current Processor idle state support

Below is a short summary of current processor idle state management in Linux 2.6.21 [3].

ACPI based idle states For the remainder of this section we restrict our attention to idle state support as in i386 (and x86-64) architectures.

In i386 (and x86-64) architectures, there is support for ACPI-based [1] processor idle states. These states are referred to as C-states in ACPI terminology. Each of the ACPI C-states is characterised by its power consumption and wakeup latency, and also based on preservation of the processor state, while in this C-state. ACPI-based platforms will report processor idle capability to Linux using ACPI interfaces. A platform can dynamically change the number of C-states supported, based on different platform parameters such as whether it is running on battery or AC power.

The current Linux support for such idle states is fully embedded in the `drivers/acpi` directory along with all ACPI support code. Code here detects the C-states available at boot time, handles any changes to the number of C-states during run time, and has simplistic policy to choose a particular C-state to enter into whenever a CPU goes idle. This code includes various platform-specific bits, specific workarounds for platform ACPI bugs, and also a `/proc`-based interface exporting the C-state-related information to userspace.

Arch specific idle—i386 and x86-64 i386 and x86-64 (and also ia64) have some architecture-specific processor idle management that does not depend on ACPI. On i386 and x86-64, it includes support for `poll_idle`, `halt_idle`, and `mwait_idle`. `poll_idle` is a polling-based idle loop, which is not really power efficient, but will have very little wakeup overhead. `halt_idle` is based on the x86 `hlt` instruction, and `mwait_idle` is based on the `monitor mwait` pair of instructions. There are specific static rules regarding which of these idle routines will be used on any system, based on boot options and hardware capabilities. Further, boot options across x86 and x86-64 are not the same for these three idle routines.

Arch-specific idle—other architectures There are various other architectures that have their own

code for processor idle state management. This includes ia64 with PAL halt and PAL light halt, Power with nap and doze modes, and idle support for different platforms in the ARM architecture. Each of these types of support for idle states also comes with its own set of boot parameters and/or `/proc` or `/sys` interfaces to user-space.

Bottom line There is very little sharing of code and sharing of idle management policies across architectures. Processor idle state management and various boot options, etc., are duplicated; this results in code duplication and maintenance overhead. This, as well as the increasing focus on processor idle power in platforms, highlights the need for a generic processor idle framework in Linux kernel.

3   Basic `cpuidle` infrastructure

Figure 1 gives a high-level overview of the `cpuidle` architecture. The basic idea behind `cpuidle` is to separate the idle state management policies from hardware-specific idle state drivers. At this level, the `cpuidle` model has similarities with `cpufreq` [6].

3.1   `cpuidle` core

The `cpuidle` core provides a set of generic interfaces to manage processor idle features.

3.1.1   `cpuidle` data structures

A per-cpu `cpuidle_device` structure holds information about the number of idle states supported by each processor, information about each of those idle state (in an array of `cpuidle_state struct`), and the status of this device, among other things.

`cpuidle_state` is a structure that contains information about each individual state, power usage, exit latency, usage statistics of the state, etc.

`cpuidle` core maintains separate linked lists of all registered drivers, all registered governors, and all detected devices.

`cpuidle_lock` is the lone mutex that handles all SMP orderings within `cpuidle`.

3.1.2   Initialization and Registration

Drivers can register and unregister with `cpuidle` core using `cpuidle_register_driver` and `cpuidle_unregister_driver`. Governors can register and unregister using `cpuidle_register_governor` and `cpuidle_unregister_governor`. Each cpu device gets detected on cpu `add_device` callback of `cpu_sysdev`. If there is a currently active governor and active driver, then the device gets initialized with those governor and driver.

3.1.3   Idle handling

`cpuidle` core has an idle handler, `cpuidle_idle_call()`, that gets plugged into an architecture-independent `pm_idle` function pointer, that will be used by each individual processor when it goes idle. Just before going into idle, the governor selects the best idle state to go into. And then `cpuidle` invokes the entry point for that particular state in the `cpuidle` driver. On returning from that state, there is an optional governor callback for the governor to capture information about idle state residency.

3.1.4   Handling system state change

The number and type of idle states can vary dynamically based on a given system state, like battery- or AC-powered, etc. Such a system state change notification goes to the idle driver, which will invoke `cpuidle_force_redetect()` in the `cpuidle` core. This results in the idle handler being temporarily uninstalled and the idle states being re-detected by the driver, followed by re-initialization of the governor state to take note of this change.

3.2   Design guidelines

There were few conscious design decisions/trade-offs in `cpuidle`.

3.2.1   `cpu_idle_wait`

To make sure we do not take a lock during the normal idle routine entry-exit, and to be able to safely change
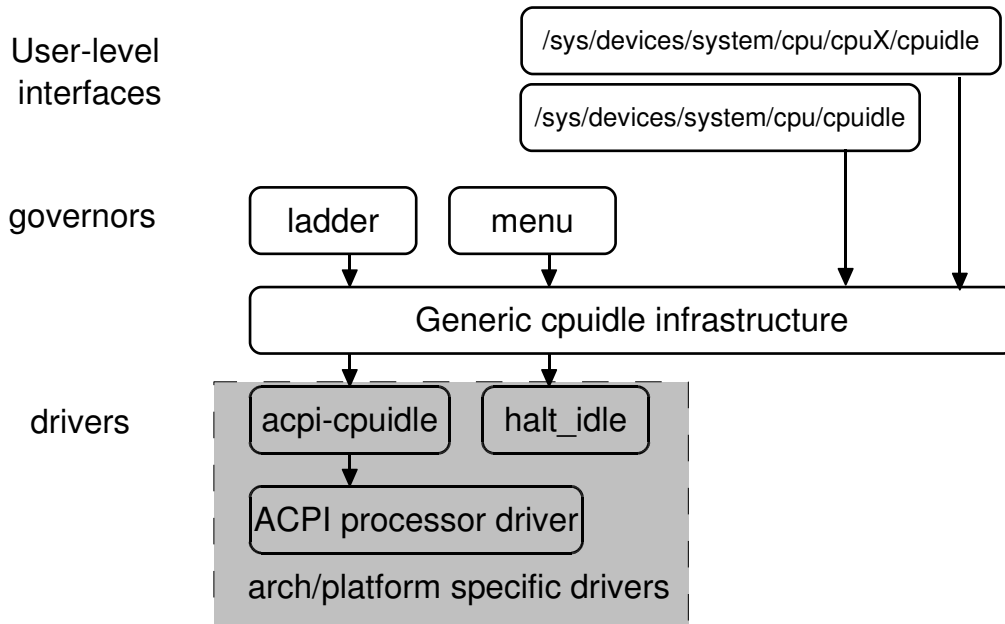
Figure 1: `cpuidle` overview

the governor/driver at run time, `cpu_idle_wait` was used. Note that changing of drivers/governors is an uncommon event which will not be performance-sensitive.

3.2.2 system-level governor and driver

Should `cpuidle` support a single driver and single governor for the whole system, or should they be per-cpu? Considering the advantage of keeping things simple with a system-level governor and driver with respect to usage of per-cpu-based governor and driver, it was decided to have a single system-level governor and driver.

3.2.3 No `cpu_hotplug_lock` in `cpuidle`

Learning from experiences of `cpufreq` and `cpu_hotplug_lock`, `cpuidle` avoids using `cpu_hotplug_lock` in the entire subsystem. This in fact resulted in a cleaner self-contained SMP and hotplug synchronization model for `cpuidle`.

3.2.4 Runtime governor/driver switching

Even though runtime switching of the governor and driver can result in potential wrong usages by the end-users, `cpuidle` supports runtime switching of the governor or driver, mostly to help developers and testers of

`cpuidle`. In the future, this switching of driver and governor can be disabled by default, in order to avoid incorrect usage.

3.3 driver interface

The `cpuidle_register_driver` uses a structure that defines the `cpuidle` driver interface:

```
struct cpuidle_driver {
 char            name[CPUIDLE_NAME_LEN];
 struct list_head  driver_list;

 int  (*init)    (struct cpuidle_device *dev);
 void (*exit)    (struct cpuidle_device *dev);
 int  (*redetect) (struct cpuidle_device *dev);

 int  (*bm_check) (void);

 struct module    *owner;
};
```

`init()` is a callback, called by `cpuidle` to initialize each device in the system with this specific driver. `exit()` is called to exit this particular driver for each device. The `redetect()` callback is used to re-detect the device states, on certain system state changes. `bm_check()` is used to note the bus mastering status on the device. In `init()`, the driver has to initialize all the states for the particular device and handle the total state count for that device.

```
struct cpuidle_state {
 char name[CPUIDLE_NAME_LEN];
 void *driver_data;

 unsigned int flags;
 unsigned int exit_latency; /* in US */
 unsigned int power_usage;  /* in mW */
 unsigned int target_residency; /* in US */

 unsigned int usage;
 unsigned int time; /* in US */

 int (*enter) (struct cpuidle_device *dev,
           struct cpuidle_state *state);

 struct kobject kobj;
};
```

`enter()` is the callback used to actually enter this idle state. `exit_latency` and `power_usage` will be characteristic of the idle state. `flags` denote generic capabilities, features, and bugs of the idle state. `usage` is the count of times this idle state is invoked, and `time` is time spent in this state.

`cpuidle_register_driver()` and `cpuidle_unregister_driver()` are used to register and unregister (respectively) a driver with `cpuidle`. `cpuidle_force_detect()` is used by the driver to force the `cpuidle` core to re-detect all the device states (e.g., after a system state change).

3.4 governor interface

```
struct cpuidle_governor {
 char             name[CPUIDLE_NAME_LEN];
 struct list_head  governor_list;

 int  (*init)      (struct cpuidle_device *dev);
 void (*exit)      (struct cpuidle_device *dev);
 void (*scan)      (struct cpuidle_device *dev);

 int  (*select)    (struct cpuidle_device *dev);
 void (*reflect)   (struct cpuidle_device *dev);

 struct module     *owner;
};
```

`init()` is a callback, called by `cpuidle`, to initialize each governor with a specific device. `exit()` is called to exit this governor for a device.

`scan()` is called on a re-detect of the states in the device. This provides an opportunity for the governor to note the changes in states during a driver re-detect.

`select()` is called before each idle entry by a device, for the governor to make a state selection for the idle call. `reflect()` is called after an idle exit, for the governor to capture information about idle state residency. Note that time spent in the governor's `reflect()` is in the critical path (on exit from idle, before starting the work) and hence has to be fast.

`cpuidle_register_governor()` and `cpuidle_unregister_governor()` are used to register and unregister (respectively) a governor with `cpuidle`. `cpuidle_get_bm_activity()` gets the information about `bm` activity, which can be used by the governor during its `select` routine.

3.5 Userspace interface

`cpuidle` userspace interfaces are split at the following two places in `/sys`.

3.5.1 System-generic information

This information is under `/sys/devices/system/cpu/cpuidle/`.

  `available_drivers` is a read-only interface that lists all the drivers that have successfully registered with `cpuidle`.

  `current_driver` is a read-write interface that contains the current active `cpuidle` driver. By writing a new value to this interface, the idle driver can be changed at run time.

  `available_governors` is a read-only interface that lists all the governors that have successfully registered with `cpuidle`.

  `current_governor` is a read-write interface that contains the current active `cpuidle` governor. By writing a new value to this interface, the idle governor can be changed at run-time.

  Note there can be single governor and single driver for all processors in the system.

3.5.2 Per-cpu information

This information is under `/sys/devices/system/cpu/cpuX/cpuidle/` where X=0,1,2,.... For each idle state *Y* supported by the current driver, the following read-only information can be seen under `sysfs`.

stateY/usage: Shows the count of number of times this idle state has been entered since the last driver `init` or `redetect`.

stateY/time: Shows the amount of time spent in this idle state in uS. item`stateY/latency`: Shows the wakeup latency for this state.

stateY/power: Shows the typical power consumed when CPU enters this state in mW.

3.6 Configuring and using `cpuidle`

To configure `cpuidle`, select:

```
Main Kernel Config
  Power management options (ACPI, APM) --->
    CPU idle PM support --->
      [ ] CPU idle PM support
```

Once `CPU idle PM` is selected, there will be further options for various governors supported in the kernel, which can then be selected.

```
<*> 'ladder' governor (NEW)
<*> 'menu' governor (NEW)
```

Currently `cpuidle` is supported only on i386 and x86-64, with an ACPI-based idle driver.

4 `cpuidle` advancements

The current `cpuidle` changes are the beginning of things to come. There are a few things under development and discussion.

4.1 New governors

The `ladder` governor takes a step-wise approach to selecting an idle state. Although this works fine with periodic tick-based kernels, this step-wise model will not work very well with tickless kernels. The kernel can go idle for a long time without a periodic timer tick and it may not get a chance to step-down the ladder to the deep idle state whenever it goes idle.

A new idle governor to handle this, called the `menu` governor, is being worked on. The `menu` governor looks at different parameters like what the expected sleep time

is (as seen by dyntick), latency requirements, previous C-state residency, `max_cstate` requirement, and `bm` activity, etc., and then picks the deepest possible idle state straight away. This governor aims at getting maximum possible power advantage with little impact on performance.

4.2 Power data

Power/Performance data with various idle policies will be provided at the time of presentation of this paper.

4.3 Future Work

Below is some of the items from the `cpuidle` to-do list. The list below is not exhaustive. Specifically, if you don't find your favorite architecture mentioned here and you would like to use cpuidle on your architecture, let the authors of this paper know about it.

Today, CPU logical offline does not take CPU to its deepest idle state. There are thoughts about using `cpuidle` to enter the deepest idle state when a CPU is logically offlined.

`cpuidle` needs to be more flexible with regards to different non-ACPI-based idle drivers supported, and also support run-time switching across these drivers.

Make `cpuidle` simple by default, and make it use the right driver and right governor for a platform by using a rating scheme for drivers and governors. This will avoid all the issues with users/distributions needing to configure `cpuidle` at every boot.

Experiment with different governors to find the most power/performance efficient governor for specific platforms. This will be an ongoing exercise as more platforms support multiple idle states and use the `cpuidle` infrastructure.

5 Conclusion

The authors hope that `cpuidle` infrastructure enables Linux to have a platform-independent, generic infrastructure for processor idle management. Such an infrastructure will simplify support of idle states on specific hardware by making it possible to write a simple plug-in driver. Additionally, such an infrastructure will

simplify the writing of idle governors, and hopefully will increase experimentation and innovation in idle governors—something similar to the frequency governors that resulted from the `cpufreq` infrastructure.

6 Acknowledgements

Thanks to the developers and testers in the community who took time to comment on, report issues with, and contribute to `cpuidle` in various ways. Special thanks to Len Brown for providing the feedback, directions, and constant support.

References

[1] Acpi in linux.
   `http://acpi.sourceforge.net`.

[2] Energy star - office equipment - computers.
   `http://www.energystar.gov`.

[3] Linux 2.6.21. `http://www.kernel.org`.

[4] Linux kernel cpufreq subsystem.
   `http://www.kernel.org/pub/linux/`
   `utils/kernel/cpufreq/cpufreq.html`.

[5] Dominik Brodowski. Current trend in linux kernel power management, linuxtag 2005. `http:`
   `//www.free-it.de/archiv/talks_`
   `2005/paper-11017/paper-11017.pdf`.

[6] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor, ols 2006.
   `http://www.linuxsymposium.org/`
   `2006/linuxsymposium_procv2.pdf`.

My bandwidth is wider than yours

Ultra Wideband, Wireless USB, and WiNET in Linux*

Iñaky Pérez-González
Open source Technology Center, Intel Corporation
`inaky.perez-gonzalez@intel.com`

Abstract

Imagine a radio technology that gives you 480 Mbps at short range. Imagine that you don't have to keep all those cables around to connect your gadgets. Make it low-power, too. You even want to be able to stream content from your super-duper cell phone to your way-too-many inches flat TV. Top it off, make it an open standard.

It's not just a dream: we have it and is called Ultra-Wide-Band, and it comes with many toppings of your choice (Wireless USB and WiNET; Bluetooth and 1394 following) to help remove almost every data cable that makes your day miserable. And Linux already supports it.

1 What is all this?

UWB is a high speed, short range radio technology intended to be the common backbone for higher level protocols. It aims to replace most of the data cables in desktop systems, home theaters, and other kinds of PAN-like interconnects. It has been defined by the WiMedia consortium after a long fight in IEEE over the underlying implementation and now it is ECMA-368 and is back in IEEE for further standardization.

It provides all the blocks (delivery of payloads, neighborhood and bandwidth management, encryption support, {broad,multi,uni}cast, etc.) needed by higher level protocols to build on top without any central infrastructure.

Wireless USB sits on top of UWB, where it allocates bandwidth and establishes a *virtual cable*; WUSB devices connect to the host in the same master-slave fashion as wired USB. The security of the cable is replaced with strong encryption and an authentication process to rule out snooping and man-in-the-middle attacks. Backwards compatibility is maintained, so we can reuse all of our drivers with slight modifications in the core host stack.

WiNET snaps an Ethernet frame over a UWB payload and adds a few protocols for bridging; devices cluster in different WiNET networks (similar to WIFI ad-hoc) and also includes authentication and strong encryption.

Other high level protocols can build on top of UWB (Bluetooth 3.0 is planning to do so, for example).

2 Ultra Wide Band

This protocol is designed for being low-power (as in little usage and efficient), with good facilities for QoS and streaming (mainly centered in audio and video) and providing strong cryptography on the transport to compensate for the open medium.
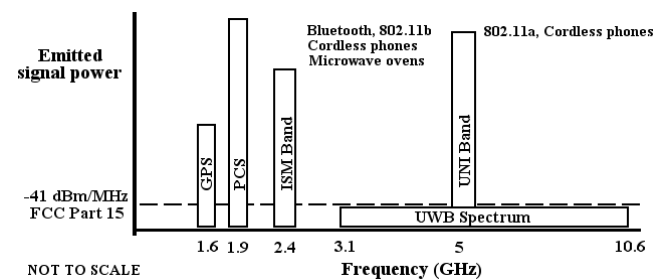

Figure 1: UWB's spectrum usage

Ultra-Wide-Band operates over the unlicensed 3.1 to 10.6 GHz band, transferring at data rates from 53Mbps to 480Mbps;[1] high rates reach up to 3 meters; lower rates, all the way to 10 meters. These are split in fourteen 528 Mhz bands; these are grouped in five band

---

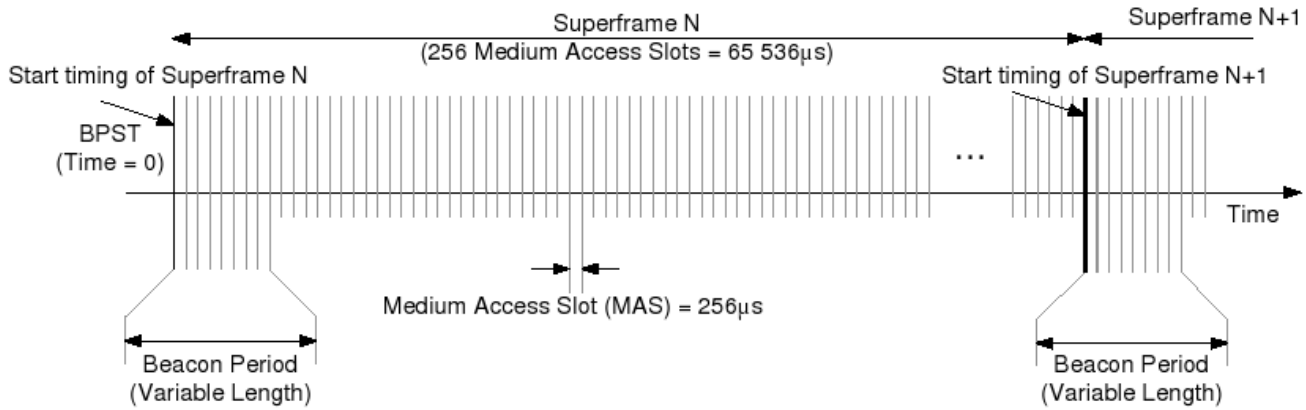[1] 53.3, 80, 106.7, 160, 200, 320, 400, and 480 Mbps, not all mandatory.

Figure 2: Division of time in UWB (credit: ECMA-368 Fig 3)

groups (*channels*) (composed of three bands each except the last one, which are two). Data is encoded MB-OFDM[2] over 122 sub-carriers (100 data, 10 guard, 12 pilot).

The power emission is as low as the maximum specified in the FCC Part 15 limit for interference: $-41dBm/Mhz$ ($0.074\mu W/MHz$), being the practical radiated power about $100\mu W/band$ ($-10dBm$). This is more or less three thousand times less than a cell phone[3] and allows UWB to appear as noise to other devices.

Time is divided in *superframes* (see Figure 2), composed of 256 *media allocation slots* (MAS), $256\mu s$ each. Thus a *superframe* is about $65ms$. The *MAS* is the basic bandwidth allocation unit.

The *superframe* starts with the *beacon period*, which is divided in $85\mu s$ *beacon slots* (96 maximum, about 32 MAS slots). The first beacon slots are used for *signalling*; when a new device wants to join it senses for it to be empty and transmits its beacon until it is assigned another empty slot.

It is important to note that devices don't have a common concept of *start of the superframe*. There might be an offset and devices coordinate among themselves to synchronize in a common start of superframe. That is called a *beacon group*, a group of devices that beacon during a shared beacon period at the beginning of the same superframe. This becomes a complication for a device B
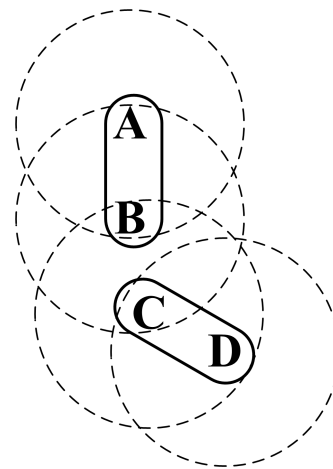
Figure 3: Hidden neighbours

that can hear beacons from A and C without A and C being able to listen to each other.

If A and B are beaconing in the same beacon group and C gets in range of B (Figure 3), C might be beaconing at a time different to that of A and B.[4] B recognizes C's beacon as an *alien beacon* and tells A about which slots C is using; thus A and B don't try to use those slots to transmit (as their tranmission would get mixed with C's).

The rules for use of the media are extremely simple. A device might only transmit:

- Its beacon, during the beacon slot assigned to it.

---

[2] Multiband Orthogonal Frequency Division Modulation.

[3] Roughly calculated, about five orders of magnitude less than WIFI.

[4] Especially if it has a beacon group formed with device D, for example.

Figure 4: Full WiNET topology

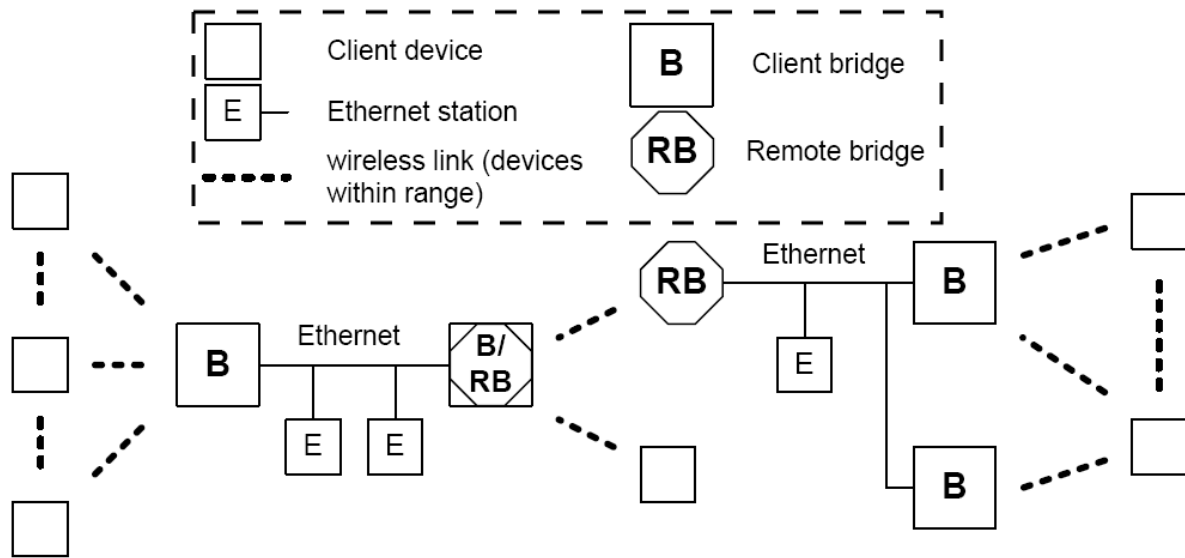• Data, during the MAS slots reserved by it. This is called *Distributed Reservation Protocol* (DRP), basically a TDMA model. It involves a negotiation among the devices on which devices own which MAS slots. Each device can define a maximum of eight static *streams* (or allocations) with this method. They are fixed (as in reserved bandwidth) until dropped.

• Data, when the media is not being used. Called *Prioritized Contention Access*, it is a CSMA technology: sense the carrier and if empty, transmit. Eight different priority levels are defined for which devices contend based on the prioritization they give to their data.

UWB aims to provide a secure enough media, tamper- and snoop-proof. It is implemented using AES-128/CCM, one-time pads, and 4-way handshakes for devising pair-wise and group-wise temporal keys (secret establishment/authentication to avoid man-in-the-middle attacks is left to the higher level protocols).

Consideration is given to power saving. Devices can switch off their radio until they have to transmit/receive (beacon or data), even advertising to each other that they are suspending beaconing for an amount of time. Transmission rates and emission power can be adjusted to decrease consumption of energy, for example, for devices in close proximity.

In general, UWB becomes a flexible low-level protocol for building on top. It offers enough flexibility for all kinds of media and data, and almost no restrictions in mobility (other than range) and usage models.

3 WiNET: IP over UWB

WiNET slaps an Ethernet payload over UWB frame to achieve the same functionality that is possible with Ethernet: IP, bridging, etc.

In concept, it is very similar to WIFI, except for the short range (10m max; a brick wall will stop it short[5]) and the nonexistence of access points (all is ad-hoc). It is faster and more power-efficient for PAN usage models.

WiNET-capable devices group in *WiNET Service Sets* (WSSs).[6] Devices may belong to more than one at the same time.

Security is provided using UWB's framework,[7] which provides data integrity and privacy; there are association methods to avoid man-in-the-middle attacks when establishing a trust relationship (using numeric comparison or simple password comparison).

---

[5] Which is an advantage in many usage models
[6] Roughly equivalent to the *ESSID* in WIFI terms.
[7] AES-128/CCM, 4-way handshakes to generate pair and group wise keys, one time pads.

QoS is achieved by reserving fixed point-to-point streams allocated with the UWB Dynamic Reservation Protocol (for example, the bandwidth required to stream audio to the living room speakers is known ahead of time) or by mapping IP traffic prioritization into the UWB Prioritized Channel Access traffic levels.

As well, 802.1D bridging services are provided (see Figure 4) to allow different WiNET segments to be bridged. This is useful, for example, for providing wireless network connectivity in high density urban dwellings, where the high number of apartments would make a WIFI access point on each unfeasible. Under this model, an Ethernet backbone connects WiNET *access points* that mobile devices connect to while roaming through the apartment without losing connectivity.

4 Wireless USB

Wireless USB by itself is very simple: remove the cable from USB 2.0, put in its place an UWB radio. The rest (at the high level) remains the same; with a few modifications to the USB core stack we can (in theory) reuse most of our already written drivers for mass storage, video, audio, etc.

WUSB still maintains the master/slave model of the wired version (even if UWB is peer to peer). The WUSB host creates a static bandwidth allocation with the Distributed Reservation Protocol, and in there it creates a *WUSB Channel*. Devices that connect to that channel define (along with the host) a *WUSB Cluster*.
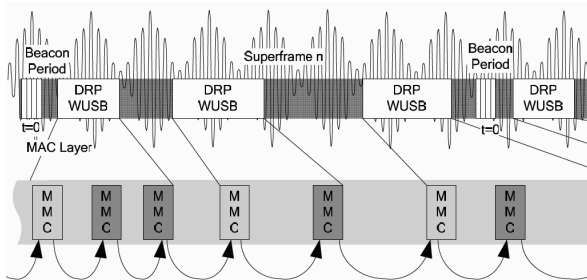

Figure 5: A WUSB Channel (credit: WUSB1.0 Fig 4-4)

At the beginning of each allocated period of time, the host emits an MMC (*Microscheduled Management Command*). This is a data structure composed of *information elements* (IEs) that specifies the length of the allocated period, which devices can transmit and when,

gives time for devices to query the host (*device notifications*, DNs[8]), and provides a link to the next MMC, which prefixes another allocation period.

This model allows WUSB devices to be simplified, as they don't have to understand (unless desired) UWB, beaconing, DRP, or PCA. They just look for MMCs and follow the links, getting their I/O control information from them.

Cable-based concepts, such as reset, connect, and disconnect are implemented via signalling in the *device notification* time slots (for device to host) and in the MMC information elements (host to device).

4.1 Wireless USB security

WUSB security builds on top of UWB's security framework.[9]

The trust relationships are established via a *Connection Context*, which is composed of a *Connection Host ID* (CHID), *Connection Device ID* (CDID), and *Connection Key* (CK). The CHID uniquely identifies the host, the CDID the device, and the CK is the shared secret. Both host and device keep the same CC as proof of trust.

Establishing the secure connection is done via the 4-way handshake process. When a device is directed by the user to connect to the host, it looks for the CHID broadcasted by it, and then looks up in its internal CC tables for the CDID it was assigned by that host. Then host and device prove to each other that they have the CK without actually exchanging it and derive a pair-wise temporal key. The host issues a new group key and issues it to all devices, including the new one. As well, all keys expire after a certain number of messages have been encoded with them, time at which new ones are renegotiated using another 4-way handshake.

When there is no Connection Context established, WUSB specifies methods for the authentication/pairing process:

- **Cable Based Association:** For devices that can connect with a cable, it is used to establish trust by transmitting the connection context using a *Cable-Based Association Framework* [USB] interface.

---

[8]Unlike wired USB, in WUSB devices can initiate transactions to the host.

[9]AES-128/CCM, 4-way handshakes, pair and group wise keys, one time pads.

• **Numeric Association:** Use Diffie-Hellman to create a temporary secure channel and avoid man-in-the middle attacks by having the device and the host present a short (two to four digits) number to the user. If they match, the user confirms the pairing. Limited range and explicit user conditioning make up for the lack of strength in a four-digit decimal hash.

Devices can keep more than one Connection Context in non-volatile memory, so that there is no need for re-authenticating when moving a device from one host to another.

5 The hardware

Hardware comes in the shape of a UWB Radio Controller (*RC*) with support on top for the higher level protocols. There is specialization, however: a low-power device might sacrifice some functionality to save power; a PC-side controller would offer full support.

We will concentrate mostly on the host side, as we just want to use the devices.

5.1 USB dongles: HWAs or Host Wire Adapters

Defined by the Wireless USB specification, HWAs consist of a UWB radio controller and a WUSB host controller all connected via USB to the host system. Other extra interfaces are possible (for example, for WiNET).

WUSB traffic to/from WUSB devices is piped through wired USB. Imagine a USB controller connected via USB instead of PCI.

Its main intent is to enable legacy systems to get seamless UWB and WUSB connectivity. The drawback is the high overhead of piping USB traffic over USB.

5.2 Wireless USB hubs: DWAs or Device Wire Adapters

Defined by the Wireless USB specification, a DWA is composed of a hub for USB wired devices whose upstream connection is wireless USB. Similarly to the HWA, think of a USB host controller that is connected to the system via *Wireless USB*, not PCI.

This is intended to connect your wired devices to your Wireless-USB-enabled laptop; as well, legacy applications will use much of this. Take an existing USB chipset for some kind of device, put in front a DWA adapter, and suddenly your device is "wireless."[10]

It has the same drawbacks as HWA regarding performance.

5.3 PCI (and friends) connected adapters: WHCI

Defined by the Wireless Host Controller Interface, the brother of EHCI. It is a Wireless USB host controller plugged straight to the PCI bus, which, as HWA, might contain other interfaces (again, such as a WiNET interface).

This is intended for new systems or those where a PCI or mini-PCI card can be deployed easily. It gives the best performance, as there is no extra overhead for delivering the final data to its destination (as in HWA/DWA).

Exercise for the reader: what happens when you connect a WUSB printer to a HWA, the HWA to a DWA, and the DWA to a WHCI, finally to your PC?

6 Linux support

Full-featured Linux support for UWB, Wireless USB, and WiNET will consist of:

• A UWB Stack to provide radio neighborhood and bandwidth management.

• Drivers for HWA (USB dongle) and WHCI (PCI) UWB Radio Controllers plugging into the UWB stack.

• A Wireless USB stack providing two abstractions used by the three different kinds of host controllers (Wireless USB Host Controller and Wire Adapter). It also includes security, authorization/ pairing management, and seamless integration into the main USB stack.

• Drivers for HWA (USB dongle) and WHCI (PCI) Wireless USB host controllers, as well as for the DWA (Wireless USB hub) host controller.

---

[10]And yes, if you connect twenty of these, you'll have twenty USB hosts in your machine.
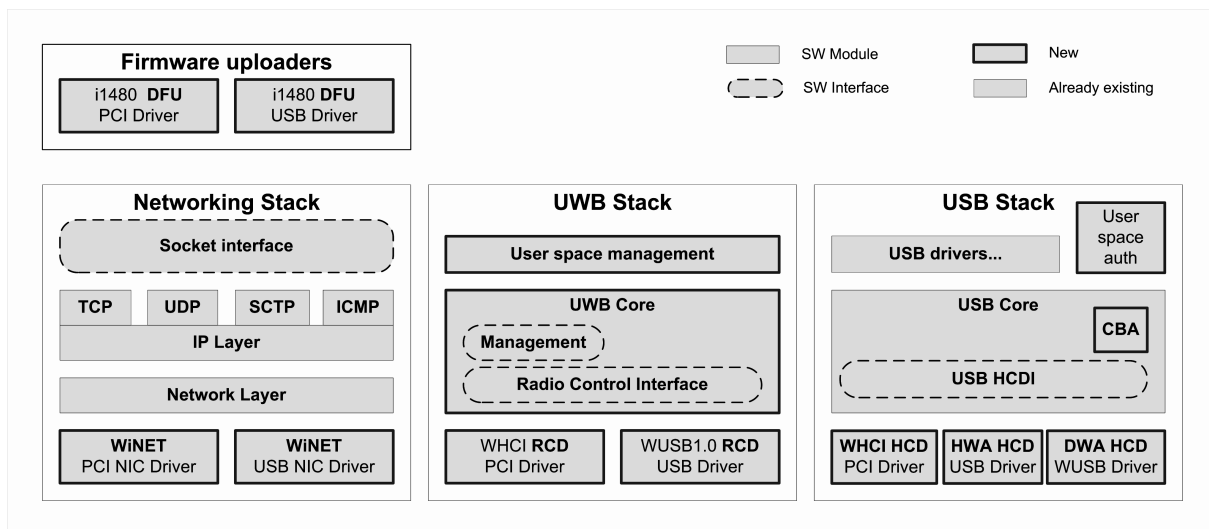
Figure 6: Linux's support for UWB/WUSB/WiNET

- Driver for WUSB Cable-Based Association, as well as support for WUSB numeric association.

- WiNET drivers: Our efforts are also to implement, using the Intel® Wireless UWB Link 1480, network device drivers for the USB and PCI form factors.

This adds up to two stacks and six drivers; eight, if we count the WiNET drivers (Figure 6).

6.1 Current status

In general, the driver set is quite stable and usable with the available hardware (which is very scarce).

The Linux UWB stack has implemented most of the basic management features, allowing the discovery of remote devices and a Distributed Reservation Protocol bandwidth negotiator. It can work with devices implementing WUSB 1.0 and WHCI 0.95.[11] Radio control drivers have been implemented for HWA (`hwa-rc`) and WHCI (`whc-rc`).

As of April 2007, only the HWA Wireless USB host controller is implemented, in a limited fashion; only control, bulk, and interrupt transfers work. However, it is possible to use an AL-4500 mass storage evaluation device made by Alereon. Authorization/pairing is being implemented, making the low-level Cable-Based Association driver complete (user space glue is now done manually).

We have also developed drivers for the Intel 1480 Wireless UWB Link WiNET interface, which are quite stable as of now.

In general, there is still a lot of work to be done. The UWB stack still needs to handle a lot of complex situations, like alien beacons, suspend and resume handling in the UWB media, selection of ideal rate and power transmission parameters, and fine tuning of the bandwidth allocator. The Wireless USB stack requires a complete transfer scheduler and implementation of isochronous support. The driver for DWA needs to be put together (with pieces from HWA) and a driver for the WHCI WUSB host controller has to be created.

6.2 Sample usage scenarios

All interaction between the user and the local radio controllers happens through sysfs:

```
# cd /sys
# ls -N bus/uwb/devices/
uwb0/
# ls -N class/uwb_rc/
uwb0/
```

---

[11]Both specs are mostly identical; however, as WHCI is developing on aspects that were not yet known when WUSB 1.0 was finalized, errata will be issued to correct them.

The radio is kept off by default; we could start beaconing to announce ourselves to others in one of the supported channels[12] on station A:

```
A# cd /sys/class/uwb_rc
A# echo 13 0 > uwb0/beacon
```

If now a user starts scanning on station B:

```
B# echo 13 0 > uwb0/scan
```

It will find station A's beacon and will be announced by the kernel; entries will be created in `sysfs`:

```
B# tail /var/log/kern.log
...
uwb-rc uwb0: uwb device \
(mac 00:14:a5:cb:6f:54 dev f8:5c)\
 connected to usb 1-4.4:1.0
...
B# ls -N /sys/bus/uwb/devices/
uwb0/
f8:5c/
```

This indicates that a remote UWB device with address `f8:5c` has been detected. At this point, the devices are not *connected*, but just B is listening for A's beacon. For them to be able to exchange information, B needs to also beacon so A knows about it—they need to be linked in the same *beacon group*. That we accomplish by asking B to beacon against A's beacon (in the same beacon period):

```
B# echo f8:5c > uwb0/beacon
```

Now we'll see in station B a similar message (`uwb device (...) connected`) as well as an entry with its 16-bit address in `/sys/bus/uwb/devices`. In the case of the Intel 1480 Wireless UWB Link (USB form factor), we could now load the WiNET driver (`i1480u-winet`) and configure network connections on both sides:

```
# modprobe i1480u-winet
# ifconfig winet0 192.168.2.1
```

---

[12]Most hardware known to our team supports channels 13, 14, and 15.

Connecting Wireless USB devices

If we had any Wireless USB devices, we would have to tell the WUSB controller to create a WUSB Channel; assuming controller `usb6` is the one corresponding to our radio controller:

```
# cd /sys/class/usb_host/usb_host6
# echo <16 byte CHID>    \
      0001               \
      mylinux-wusb-01  \
  > wusb_chid
```

With this we have given a 16-byte CHID to the driver, which has configured it into the host so it broadcasts a WUSB channel named `mylinux-wusb-01`. Now devices that have been paired with this CHID before will request connection to the host when we ask them to (for now we allow all of them to connect):

```
new variable speed Wireless USB \
device using hwa-hc and address 2
```

From now on, this behaves like yet another USB device. There are still no controls for selecting rate or power.

If no transactions are done to the device or the device doesn't ping back to the host's keep-alives, then it will be disconnected. The timeout is specified in a per-host basis in file `/sys/class/usb_host/X/wusb_trust_timeout`. Most devices we've seen until now don't implement keep-alives, so this value has to be set high to avoid their disconnection.

7 Conclusion

We have done a quick description of Ultra Wideband, Wireless USB, and WiNET, a set of technologies that aim at replacing all the cables that clutter desktops and living rooms. Designed with streaming and power efficiency in mind, they also provide security comparable to that of the cable.

We have also described how Linux implements support for it, its current status, and how to use it. There is basic support working, but a lot is still to be done.

We expect a huge increase of consumer electronics devices of all kinds (cellphones, cameras, computing, audio, video. . .) supporting this set of technologies in upcoming years. And with Linux playing an all-the-time

more important role in those embedded applications, as well as in the desktop, it is key that it supports them as soon as they hit the streets en masse.

References

[linuxuwb.org] Linux UWB/WUSB/WiNET,
    `http://linuxuwb.org`

[WiMedia] WiMedia Alliance,
    `http://wimedia.org`

[ECMA368] Ecma International, *ECMA-368 High Rate Ultra Wideband PHY and MAC standard, 1.0*,
    `http://www.ecma-international.org`

[WiMedia] WiMedia Alliance, *WiNET specification* (still not publicly available)
    `http://wimedia.org`

[WUSB1.0] USB Implementors Forum, *Wireless USB 1.0 specification*, `http://usb.org`

[WAM1.0] USB Implementors Forum, *Association Models supplement to the Certified Wireless Universal Serial Bus Specification, 1.0*,
    `http://usb.org`

[WHCI] Intel Corporation, *Wireless Host Controller Interface, 0.95*, `http://intel.com`

Zumastor Linux Storage Server

Daniel Phillips
Google, Inc.
`phillips@google.com`

Abstract

Zumastor provides Linux with network storage functionality suited to a medium scale enterprise storage role: live volume backup, remote volume replication, user accessible volume snapshots and integration with Kerberized network filesystems. This paper examines the design and functionality of the major system components involved. Particular attention is paid to the subjects of optimizing server performance using NVRAM, and reducing the amount of network bandwidth required for remote application using various forms of compression. Future optimization strategies are discussed. Benchmark results are presented for currently implemented optimizations.

1 Introduction

Linux has done quite well at the edges of corporate networks—web servers, firewalls, print servers and other services that rely on well standardized protocols—but has made scant progress towards the center, where shared file servers define the workflow of a modern organization. The barriers are largely technical in that Linux storage capabilities, notably live filesystem snapshot and backup, have historically fallen short of the specialized proprietary offerings to which users have become accustomed.

Zumastor provides Linux with network storage functionality suited to medium scale enterprise storage roles: live volume backup, remote volume replication, user accessible volume snapshots and integration with Kerberized network filesystems. Zumastor takes the form of a set of packages that can be added to any Linux server and will happily coexist with other roles that the server may serve. There are two major components: the ddsnap virtual block device, which provides base snapshot and replication functionality, and the Zumastor volume monitor, which presents the administrator with a simple command line interface to manage volume snapshotting and replication. We first examine the low level components in order to gain an understanding of the design approach used, its capabilities and limitations.

2 The ddsnap Virtual Block Device

The ddsnap virtual block device provides multiple read/write volume snapshots. As opposed to the incumbent lvm snapshot facility, ddsnap does not suffer from degraded write performance as number of snapshots increases and does not require a separate underlying physical volume for each snapshot. A ddsnap virtual device requires two underlying volumes: an origin volume and a snapshot store. The origin, as with the lvm snapshot, is a normal volume except that write access is virtualized in order to protect snapshotted data. The snapshot store contains data copied from the origin in in the process of protecting snapshotted data. Metadata in the snapshot store forms a btree to map logical snapshot addresses to data chunks in the snapshot store.

The snapshot store also contains bitmap blocks that control allocation of space in the snapshot store, a list of currently held snapshots, a superblock to provide configuration information, and a journal for atomic and durable updates to the metadata. The ddsnap snapshot store resembles a simple filesystem, but with only a single directory, the btree, indexed by the logical address of a snapshot. Each leaf of the btree contains a list of logical chunk entries; each logical chunk entry in the btree contains a list of one or more physical chunk addresses; and for each physical chunk address, a bitmap indicates which snapshots currently share that physical chunk. Share bits are of a fixed, 64 bit size, hence the ddsnap limitation of 64 simultaneous snapshots. This limitation may be removed in the future with a redesign of the btree leaf format.

2.1 Client-server design

DDsnap was conceived as a cluster snapshot—*DD* stands for *distributed data*—with a client-server architecture. Each ddsnap client (a device mapper device) provides access to either the underlying, physical origin volume or to some snapshot of the origin volume. To implement the copy-before-write snapshot strategy, a ddsnap origin client communicates with a userspace server over a socket, requesting permission from this snapshot server before writing any block so that the server may move any data shared with a snapshot to the snapshot store. Similarly, a ddsnap snapshot client requests permission from the snapshot server before writing any block so that space may be allocated in the snapshot store for the write. A snapshot client also requests permission before reading snapshot data, to learn the physical location of the data and to serialize the read against origin writes, preventing the data being read from moving to a new location during the read. On the other hand, an origin client need not consult the server before reading, yielding performance nearly identical to the underlying physical volume for origin reads.

Since all synchronization with the snapshot server is carried out via messages, the server need not reside on the same node as an origin or snapshot client, although with the current single-node storage application it always does. Some more efficient means of synchronization than messages over a socket could be adopted for a single node configuration, however messaging overhead has not proved particularly bothersome, with a message inter-arrival time measured in microseconds due to asynchronous streaming. Some functionality required for clustering, such as failing over the server, uploading client read locks in the process, is not required for single-node use, but imposes no overhead by its presence.

Creating or deleting a snapshot is triggered by sending a message to the snapshot server, as are a number of other operations such as obtaining snapshot store usage statistics and obtaining a list of changed blocks for replication. The *ddsnap* command utility provides a commandline syntax for this. There is one snapshot server for each snapshot store, so to specify which snapshot server the command is for, the user gives the name of the respective server control socket. A small extra complexity imposed by the cluster design is the need for a ddsnap *agent*, whose purpose on a cluster is to act as a node's central cluster management communication point, but which serves no useful purpose on a single node. It is likely that the functionality of agent and snapshot server will be combined in future, somewhat simplifying the setup of a ddsnap snapshot server. In any event, the possibility of scaling up the Zumastor design using clustering must be viewed as attractive.

2.2 Read/Write Snapshots

Like lvm snapshots, ddsnap snapshots are read/write. Sometimes the question is raised: why? Isn't it unnatural to write to a snapshot? The answer is, writable snapshots come nearly for free, and they do have their uses. For example, virtualization software such as Bochs, QEMU, UML or Xen might wish to base multiple VM images on the same hard disk image.[1] The copy-on-write property of a read/write snapshot gives each VM a private copy in its own snapshot of data that it has written. In the context of zumastor, a root volume could be served over NFS to a number of diskless workstations, so each workstation is able to modify modify part of its own copy while continuing to share the unmodified part.

2.3 Snapshotted Volume IO Performance

Like the incumbent lvm snapshot, origin read performance is nearly identical to native read performance, because origin reads are simply passed through to the underlying volume.

As with the incumbent lvm snapshot, ddsnap uses a copy-before-write scheme where snapshotted data must be copied from the origin to the snapshot store the first time the origin chunk is written to after a new snapshot. This can degrade write performance markedly under some loads. Keeping this degradation to a tolerable level has motivated considerable design effort, and work will continue in this area. With the help of several optimization techniques discussed below, a satisfactory subjective experience is attained for the current application: serving network storage.

Compared to the incumbent lvm snapshot, the ddsnap snapshot design requires more writes to update the metadata, typically five writes per newly allocated physical chunk:

---

[1] http://en.wikipedia.org/wiki/Copy-on-write

1. Write allocation bitmap block to journal.

2. Write modified btree leaf to journal.

3. Write journal commit block.

4. Write allocation bitmap block to store.

5. Write modified btree leaf to store.

This update scheme is far from optimal and is likely to be redesigned at some point, but for now a cruder approach is adopted: add some nonvolatile memory (NVRAM) to the server.

The presence of a relatively small amount of nonvolatile RAM can accelerate write performance in a number of ways. One way we use NVRAM in Zumastor is for snapshot metadata. By placing snapshot metadata in NVRAM we reduce the cost of writing to snapshotted volume locations significantly, particularly since ddsnap in its current incarnation is not very careful about minimizing metadata writes. Unfortunately, this also limits the maximum size of the btree, and hence the amount of snapshot data that can be stored. This limit lies roughly in the range of 150 gigabytes of 4K snapshot chunks per gigabyte of NVRAM. NVRAM is fairly costly, so accommodating a large snapshot store be expensive. Luckily, much can be done to improve the compactness of the btree, a subject for another paper.

Using NVRAM, snapshot performance is no worse than the incumbent lvm snapshot, however the size of the btree and hence the amount of data that can be stored in the snapshot store is limited by the amount of NVRAM available. Future work will relax this limitation.

2.3.1 Filesystem Journal in NVRAM

For filesystems that support separate journals, the journal may be placed in NVRAM. If the filesystem is further configured to journal data writes as well as metadata, a write transaction will be signalled complete as soon as it has been entered into the journal, long before being flushed to underlying storage. At least until the journal fills up, this entirely masks the effect of slower writes to the underlying volume. The practical effect of this has not yet been measured.

2.3.2 Effect of Chunk Size and Number of Snapshots on Write Performance

Untar time on the native (Ext3) filesystem is about 14 seconds. Figure 1 shows that untar time on the virtual block device with no snapshots held is about 20 seconds, or slower by a factor of 1.43. This represents the overhead of synchronizing with the snapshot server, and should be quite tractable to optimization. Snapshotted untar time ranges from about 3.5 times to nearly 10 times slower than native untar time.

Figure 1 also shows the effect of number of currently held snapshots on write performance and of varying the snapshot chunk size. At each step of the test, a tar archive of the kernel source is unpacked to a new directory and a snapshot is taken. We see that (except for the very first snapshot) the untar time is scarcely affected by the number of snapshots. For the smallest chunk size, 4K, we see that untar time does rise very slightly with the number of snapshots, which we may attribute to increased seek time within the btree metadata. As chunk size increases, so does performance. With a snapshot store chunk size of 128KB, the untar runs nearly three times faster.

2.3.3 Effect of NVRAM on Write Performance

Figure 2 shows the effect of placing the snapshot data in NVRAM. Write performance is dramatically improved, and as before, number of snapshots has little or no effect. Interestingly, the largest chunk size tested, 128KB, is no longer the fastest; we see best performance with 64K chunk size. The reason for this remains to be investigated, however this is good news because a smaller chunk size improves snapshot store utilization. Write performance has improved to about 2 to 5 times slower than native write performance, depending on chunk size.

2.3.4 NVRAM Journal compared to NFS Write Log

NVRAM is sometimes used to implement a NFS write log, where each incoming NFS write is copied to the write log and immediately acknowledged, before being written to the underlying filesystem. Compared to the strategy of putting the filesystem journal in NVRAM, performance should be almost the same: in either case,

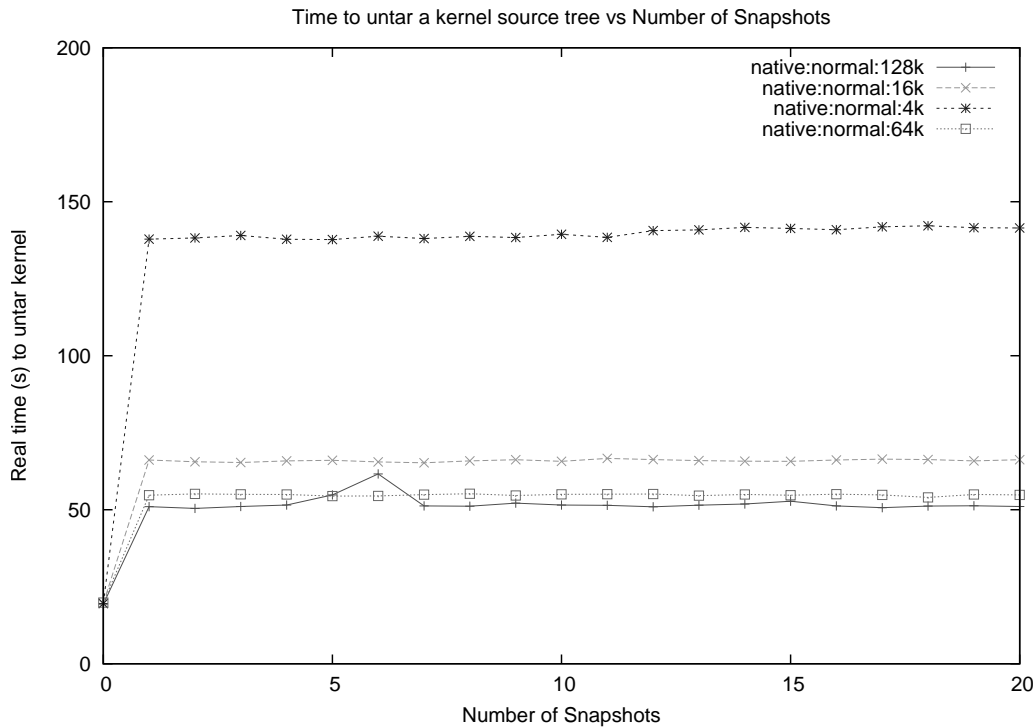Time to untar a kernel source tree vs Number of Snapshots

Figure 1: Write performance without NVRAM

a write is acknowledged immediately after being written to NVRAM. There may be a small difference in the overhead of executing a filesystem operation as opposed to a potentially simpler transaction log operation, however the filesystem code involved is highly optimized and the difference is likely to be small. On the other hand, the transaction log requires an additional data copy into the log, likely negating any execution path efficiency advantage. It is clear which strategy requires less implementation effort.

2.3.5 Ongoing Optimization Efforts

A number of opportunities for further volume snapshot write optimization remain to be investigated. For example, it has been theorized that writing to a snapshot instead of the origin can improve write performance a great deal by eliminating the need to copy before writing. If read performance from a snapshot can be maintained, then perhaps it would be a better idea to serve a master volume from a snapshot than an origin volume.

3 Volume Replication

Volume replication creates a periodically updated copy of a master volume at some remote location. Replication

differs from mirroring in two ways: 1) changes to the remote volume are batched into volume deltas so that multiple changes to the same location are collapsed into a single change and 2) a write operation on the master is not required to wait for write completion on the remote volume. Batching the changes also allows more effective compression of volume deltas, and because the changes are sorted by logical address, applying a delta to a remote volume requires less disk seeking than applying each write to a mirror member in write completion order. Replication is thus suited to situations where the master and remote volume are not on the same local network, which would exhibit intolerable remote write latency if mirrored. High latency links also tend to be relatively slow, so there is much to be gained by good compression of volume deltas.

Zumastor implements remote replication via a two step process: 1) Compute difference list; 2) Generate delta. To generate the difference list for a given pair of snapshots, the ddsnap server scans through the btree to find all snapshot chunks that belong to one snapshot and not the other, which indicates that the data for the corresponding chunks was written at different times and is most probably different. To generate the delta, a ddsnap utility runs through the difference list reading data from one or both of the snapshots which is incorporated into

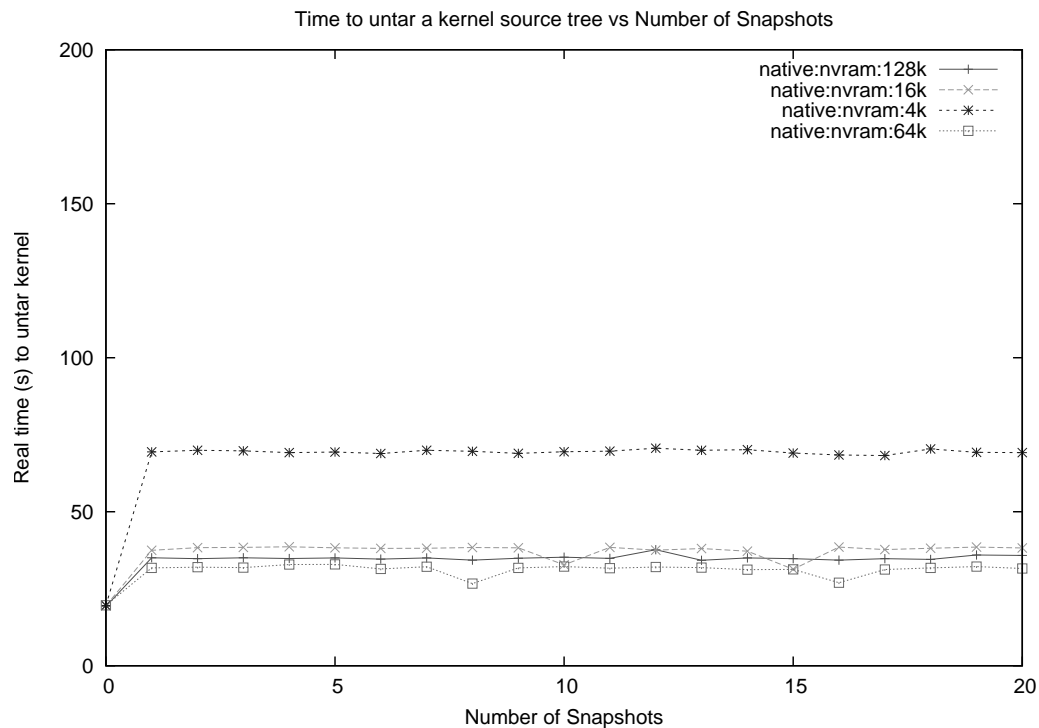Time to untar a kernel source tree vs Number of Snapshots

Figure 2: Write performance with NVRAM

the output delta file. To allow for streaming replication, each volume delta is composed of a number of extents, each corresponding to some number of contiguous logical chunks.

A variety of compression options are available for delta generation. A "raw" delta incorporates only literal data from the destination snapshot. An "xdelta" delta computes the binary difference between source and destination snapshot. Raw delta is faster to generate and requires less disk IO, is faster to apply to the target volume, and is more robust in the sense that the code is very simple. Computing an xdelta delta requires more CPU and disk bandwidth but should result in a smaller delta that is faster to transmit.

A volume delta may be generated either as a file or as a TCP stream. Volume replication can be carried out manually using delta files:

1. Generate a snapshot delta as a file

2. Transmit the delta or physically transport it to the downstream host

3. Apply the delta to the origin volume of the downstream host

It is comforting to be able to place a snapshot delta and to know that the replication algorithm is easy enough to carry out by hand, which might be important in some special situation. For example, even if network connectivity is lost, volume replication can still be carried out by physically transporting storage media containing a delta file.

Zumastor uses ddsnap's streaming replication facility, where change lists and delta files are never actually stored, but streamed from the source to target host and applied to the target volume as a stream. This saves a potentially large amount of disk space that would be otherwise be required to store a delta file on both the source and target host.

From time to time it is necessary to replicate an entire volume, for example when initializing a replication target. This ability is provided via a "full volume delta" that generates a raw, compressed delta as if every logical chunk had appeared in the difference list. Incidentally, only the method of delta generation is affected by this option, not the delta file format.

To apply a volume delta, ddsnap overwrites each chunk of the target volume with the new data encoded in the delta in the case of a raw delta, or reads the source snapshot and applies the binary difference to it in the case

of xdelta. Clearly, it is required that the source snapshot exist on the downstream host and be identical to the source snapshot on the upstream host.

To create the initial conditions for replication:

1. Ensure that upstream and downstream origin volumes are identical, for example by copying one to the other

2. Snapshot the upstream and downstream volumes

The need to copy an entire volume over the network in the first step can be avoided in some common cases. When a filesystem is first created, it is easy to zero both upstream and downstream volumes, which sets them to an identical state. The filesystem is then created after step 2. above, so that relatively few changed blocks are transmitted in the first replication cycle. In the case where the downstream volume is known to be similar, but not identical to the upstream volume (possibly as a result of an earlier hardware or software failure) then the remote volume differencing utility rdiff may be used to transmit a minimal set of changes downstream.

Now, for each replication cycle:

1. Set a new snapshot on the upstream volume.

2. Generate the delta from old to new upstream snapshot.

3. Transmit the delta downstream.

4. Set a new snapshot on the downstream volume (downstream origin and new snapshot are now identical to the old upstream snapshot).

5. Apply the delta to the downstream origin (downstream origin is now identical to the new upstream snapshot).

For the streaming case, step 4 is done earlier so that the transmit and apply may iterate:

1. Set a new snapshot on upstream and downstream volumes.

2. Generate the delta from old to new upstream snapshot.

3. Transmit the next extent of the delta downstream.

4. Apply the delta extent to the downstream origin.

5. Repeat at 3 until done.

For streaming replication, a server is started via ddsnap on the target host to receive the snapshot delta and apply it to the downstream origin.

Fortunately for most users, all these steps are handled transparently by the Zumastor volume manager, described below.

Multi level replication from a master volume to a chain of downstream volumes is handled by the same algorithm. We require only that two snapshots of the master volume be available on the upstream volume and that the older of the two also be present on the downstream volume. A volume may be replicated to multiple targets at any level in the chain. In general, volume replication topology is a tree, with the master volume at the root and an arbitrary number of target volumes at interior and leaf nodes. Only the master is writable; all the target volumes are read-only.

3.1 Delta Compression

Compression of delta extents is available as an option, either using zlib (gzip) or in the case of xdelta, an additional Huffman encoding stage. A compressed xdelta difference should normally be more compact than gzipped literal data, however, one can construct cases where the reverse is true. A further (extravagant) "best" compression option computes both the gzip and xdelta compression for a given extent and use the smaller for the output delta. Which combination of delta generation options is best depends largely on the amount of network bandwidth available.

Figure 3 illustrates the effect of various compression options on delta size. For this test, the following steps are performed:

1. Set snapshot 0.

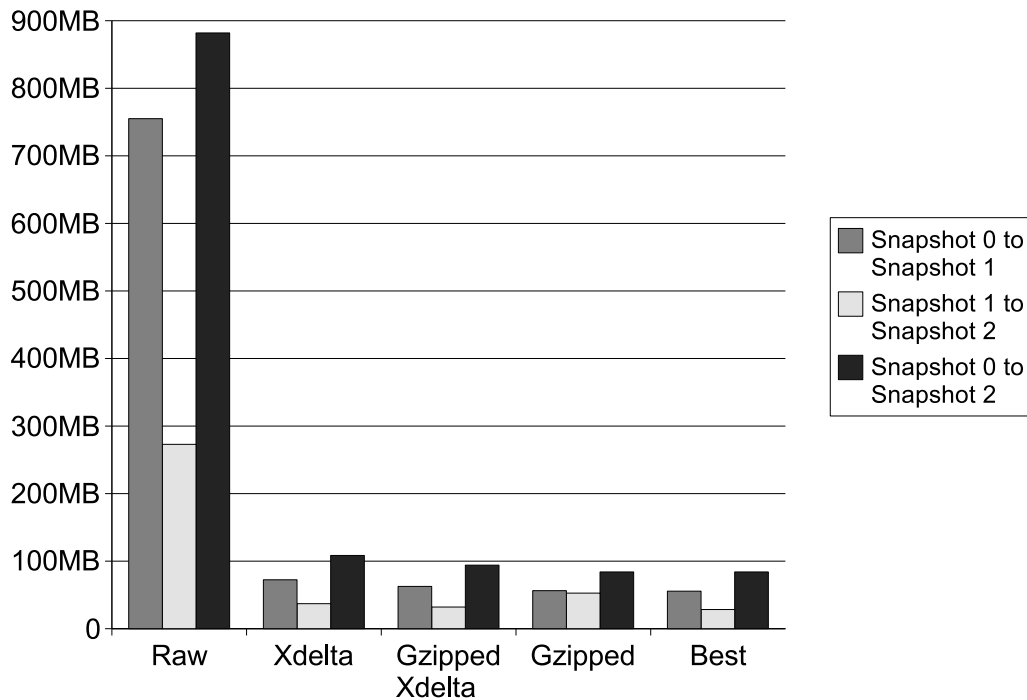2. Untar a kernel tree.

3. Set snapshot 1.

Delta Size

Figure 3: Delta Compression Effectiveness

4. Apply a (large) patch yielding the next major kernel release.

5. Set snapshot 2.

Three delta files are generated, the two incremental deltas from snapshot 0 to snapshot 1 and from snapshot 1 to snapshot 2, and the cumulative delta from snapshot 0 to snapshot 2. We observe a very large improvement in delta size, ranging up to a factor of 10 as compared to the uncompressed delta.

XDelta performs considerably better on the snapshot 1 to snapshot 2 delta, which is not surprising because this delta captures the effect of changing many files as opposed to adding new files to the filesystem, so it is only on this delta that there are many opportunities to take advantage of similarity between the two snapshots.

The "best" method gives significantly better compression on the snapshot 1 to snapshot 2 delta, which indicates that some delta extents compress better with gzip than they do with xdelta. (As pointed out by the author of xdelta, this may be due to suboptimal use of compression options available within xdelta.) Figure 4 re-

expresses the delta sizes of Figure 3 as compression ratios, ranging from 5 to 13 for the various loads.

Delta compression directly affects the time required to transmit a delta over a network. This is particularly important when replicating large volumes over relatively low bandwidth network links, as is typically the case. The faster a delta can be transmitted, the fresher the remote copy will be. We can talk about the "churn rate" of a volume, that is, the rate at which it changes. This could easily be in the neighborhood of 10% a day, which for a 100 gigabyte disk would be 10 gigabytes. Transmitting a delta of that size over a 10 megabit link would require 10000 seconds, or about three hours. An 800 gigabyte volume with 10% churn a day would require more than a day to transmit the delta, so the following delta will incorporate even more than 10% churn, and take even longer. In other words, once replication falls behind, it rapidly falls further and further behind, until eventually nearly all the volume is being replicated on each cycle, which for our example will take a rather inconvenient number of days.

In summary, good delta compression not only improves the freshness of replicated data, it delays the point at
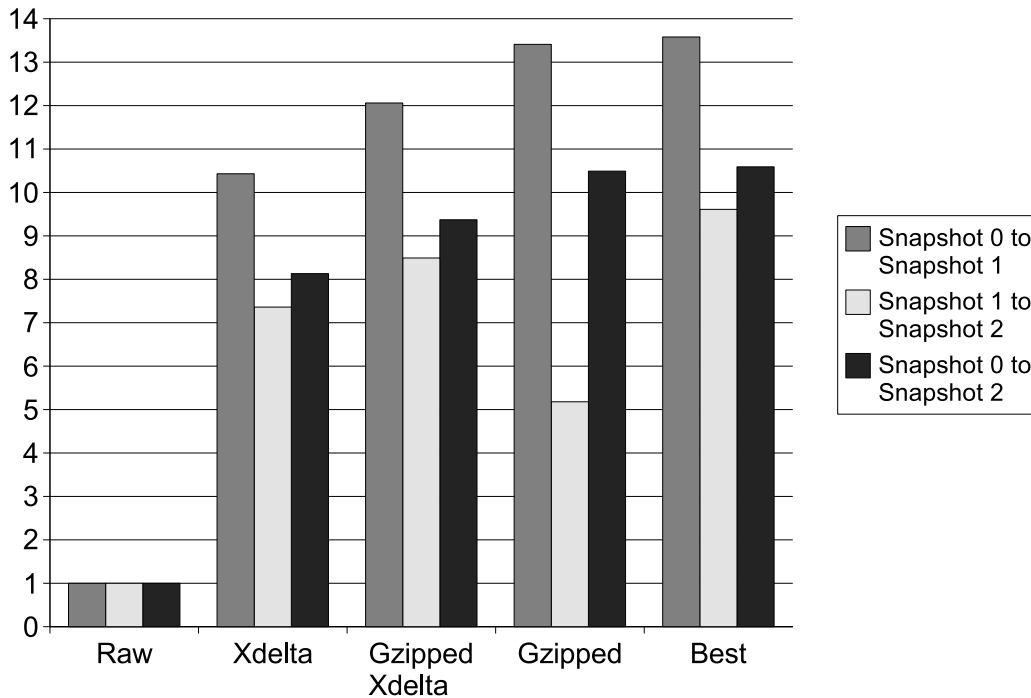
Delta Compression Ratio

Figure 4: Delta Compression Effectiveness

which replication lag begins to feed on itself and enables timely replication of larger, busier volumes over lower speed links.

3.2 NFS Snapshot Rollover

Exporting a replicated volume via NFS sounds easy except that we expect the filesystem to change state "spontaneously" each time a new volume delta arrives, without requiring current NFS clients to close their TCP connections. To create the effect of jumping the filesystem from state to state as if somebody had been editing the filesystem locally, we need to unmount the old snapshot and mount the new snapshot so that future NFS accesses will be to the new snapshot. The problem is, the Linux server will cache some elements of the client connection state such as the file handle of the filesystem root, which pins the filesystem and prevents it from being unmounted.

Zumastor solves this problem by introducing a nfsd suspend/resume operation. This flushes all cached client state which forces the use count of the exported filesystem snapshot to one, the mount point. This allows the old snapshot to be unmounted and the new snapshot to

be mounted in its place, before resuming. Interestingly, the patch to accomplish this is only a few lines, because most of the functionality to accomplish it already existed.

3.3 Incremental Backup using Delta Files

Ddsnap delta files are not just useful for replication, they can also be used for incremental backup. From time to time, a "full volume" delta file can be written to tape, followed periodically by a number of incremental deltas. This should achieve very rapid backup and economical use of tape media, particularly if deltas are generated with more aggressive compression options. To restore, a full (compressed) volume must be retrieved from tape, along with some number of delta files, which are applied sequentially to arrive at a volume state as some particular point in time. Restoring a single file would be a very slow process, however it is also expected to be a rare event. It is more important that backup be fast, so that it is done often.

4   Zumastor volume monitor

The Zumastor volume monitor takes care of most of the chores of setting up virtual block devices, including making socket connections between the ddsnap user space and kernel components, creating the virtual devices with dmsetup, organizing mount points and mounting volumes. It maintains a simple database implemented as a directory tree that stores the configuration and operating status of each volume on a particular host, and provides the administrator with a simple set of commands for adding and removing volumes from the database, and defining their operational configuration. Finally, it takes care of scheduling snapshots and initiating replication cycles.

The Zumastor volume database is organized by volumes, where each volume is completely independent from the others, not even sharing daemons. Each Zumastor volume is either a master or a target. If a master, it has a replication schedule. If a target, then it has an upstream source. In either case, it may have any number of replication targets. Each replication target has exactly one source, which prevents cycles and also allows automatic checking that the correct source is replicating to the correct target.

The replication topology for each volume is completely independent. A given host may offer write/write access to volumes that are replicated to other hosts and read-only access to volumes replicated to it from other hosts. So for example, two servers at widely separated geographic locations might each replicate a volume to the other, which not provides a means of sharing data, but also provides a significant degree of redundancy, particularly if each server backs up both its own read/write volume and the replicated read-only volume to tape.

The replication topology for each volume is a tree, where only the master (root of the tree) behaves differently from the other nodes. The master generates snapshots either periodically or on command. Whenever one of its target hosts is ready to receive a new snapshot delta, the master creates a new snapshot and replicates it to the target, ensuring that the downstream host receives as fresh as possible a view of the master volume. On all other hosts, snapshot deltas are received from an upsteam source and simply passed along down the chain.

Zumastor replication is integrated with NFS in the sense that Zumastor knows how to suspend NFS while it re-mounts a replicated volume to the latest snapshot, effecting the snapshot rollover described above.

5   The Future

In the future, Zumastor will continue to gain new functionality and improve upon existing functionality. It would be nice to have a graphical front end to the database, and a web interface. It would be nice to see the state of a whole collection of Zumastor servers together in one place, including the state of any replication cycles in progress. It would be natural to integrate more volume management features into Zumastor, such as volume resizing. Zumastor ought to be able to mirror itself to a local machine and fail over NFS service transparently. Zumastor should offer its own incremental backup using delta files. Another creative use of delta files would be to offer access to "nearline" snapshots, where a series of archived reverse deltas are applied to go back further in time than is practical with purely online snapshots.

There is still plenty of room for performance optimization. There is a lot more that can be done with NVRAM, and things can be done to improve the performance without NVRAM, perhaps making some of Zumastor's replication and backup capabilities practical for use on normal workstations and the cheapest of the cheap servers.

All in all, there remains plenty of work to do and plenty of motivation for doing it.

Cleaning up the Linux Desktop Audio Mess

Lennart Poettering

Red Hat, Inc.

`lennart@poettering.net`

Abstract

Desktop audio on Linux is a mess. There are just too many competing, incompatible sound systems around. Most current audio applications have to support every sound system in parallel and thus ship with sound abstraction layers with a more or less large number of back-end plug-ins. JACK clients are incompatible with ALSA clients, which in turn are incompatible with OSS clients, which in turn are incompatible with ESD clients, and so on. "Incompatible" often means "exclusive;" e.g., if an OSS application gets access to the audio hardware, all ALSA applications cannot access it.

Apple MacOS X has CoreAudio, Microsoft Windows XP has a new user-space audio layer; both manage to provide comprehensive APIs that make almost every user happy, ranging from desktop users to pro audio people. Both systems provide fairly modern, easy-to-use audio APIs, and a vast range of features including desktop audio "bling."

On Linux we should be able to provide the same: a common solution that works on the desktop, in networked thin-client setups and in pro audio environments, scaling from mobile phones to desktop PCs and high-end audio hardware.

1 Fixing the Linux Audio Stack

In my talk, I want to discuss what we can do to clean up the mess that desktop audio on Linux is: why we need a user-space sound system, what it should look like, how we need to deal with the special requirements of networked audio and pro-audio stuff, and how we should expose the sound system to applications for allowing Compiz-style desktop "bling"—but for audio. I then will introduce the PulseAudio sound server as an attempt to fix the Linux audio mess.

PulseAudio already provides compatibility with 90% of all current Linux audio software. It features low-latency audio processing and network transparency in an extensible desktop sound server. PulseAudio is now part of many distributions, and is likely to become the default sound system on Fedora and Ubuntu desktops in the next releases of these distributions.

The talk will mostly focus on the user-space side of Linux audio, specifically on the low-level interface between hardware drivers and user-space applications.

2 Current State of Linux Audio

The current state of audio on Linux and other Free Software desktops is quite positive in some areas, but in many other areas, it is unfortunately very poor. Several competing audio systems and APIs are available. However, none of them is useful in all types of applications, nor does any meet the goals of being easy-to-use, scalable, modern, clean, portable, and complete. Most of these APIs and systems conflict in one way or another.

On the other hand, we have a few components and APIs for specific purposes that are well accepted and cleanly designed (e.g., LADSPA, JACK). Also, Linux-based systems can offer a few features that are not available on competing, proprietary systems. Among them is network transparent audio and relatively low-latency scheduling.

While competing, proprietary systems currently lack a few features the Linux audio stack can offer, they managed to provide a single (specific to the respective OS) well-accepted API that avoids the balkanisation we currently have on Free Software desktops. Most notably, Apple MacOS X has CoreAudio which is useful for the desktop as well as for professional audio applications. Microsoft Windows Vista, on the other hand, now ships a new user-space audio layer, which also fulfills many of the above requirements for modern audio systems and APIs.

In the following sections, I will quickly introduce the systems that are currently available and used on Linux desktop, their specific features, and their drawbacks.

2.1   Advanced Linux Sound Architecture (ALSA)

The ALSA system [2] has become the most widely accepted audio layer for Linux. However, ALSA, both as audio system and as API, has its share of problems:

- The ALSA user-space API is relatively complicated.

- ALSA is not available on anything but Linux.

- The ALSA API makes certain assumptions about sound devices that are only true for hardware devices. Implementing an ALSA plug-in for virtual devices ("software" devices) is not doable without nasty hacks.

- Not "high-level" enough for many situations.

- `dmix` is bug-ridden and incomplete.

- Resampling is very low quality.

- You need different configurations for normal desktop use (`dmix`) and pro audio use (no `dmix`).

On the other hand, it also has some real advantages over other solutions:

- It is available on virtually every modern Linux installation.

- It is very powerful.

- It is (to a certain degree) extendable.

2.2   Open Sound System (OSS)

OSS [7] has been the predecessor of ALSA in the Linux kernel. ALSA provides a certain degree of compatibility with OSS. Besides that the API is available on several other Unixes. OSS is a relatively "old" API, and thus has a number of limitations:

- Doesn't offer all the functionality that modern sound hardware provides which needs to be supported by the software (such as no surround sound, no float samples).

- Not high-level enough for almost all situations, since it doesn't provide sample format or sample rate conversions. Most software silently assumes that S16NE samples at 44100Hz are available on all systems, which is no longer the case today.

- Hardly portable to non-Unix systems.

- `ioctl()`-based interface is not type-safe, not the most user-friendly.

- Incompatible with everything else; every application gets exclusive access to the sound device, thus blocking all other applications from accessing it simultaneously.

- Almost impossible to virtualize correctly and comprehensively. Hacks like `esddsp`, `aoss`, `artsdsp` have proven to not work.

- Applications silently assume the availability of certain driver functionality that is not necessarily available in all setups. Most prominently, the 3D game Quake doesn't run with drivers that don't support `mmap()`-based access to the DMA audio buffer.

- No network transparency, no support for desktop "bling."

The good things:

- Relatively easy to use;

- Very well accepted, even beyond Linux;

- Feels very Unix-ish.

2.3   JACK

The JACK Audio Connection Kit [3] is a sound server for professional audio purposes. As such, it is well accepted in the pro-audio world. Its emphasis, besides playback of audio through a local sound card, is streaming audio data between applications.

Plusses:

- Easy to use;

- Powerful functionality;

- It is a real sound server;

- It is very well accepted in the pro-audio world.

Drawbacks:

- Only floating point samples;

- Fixed sampling rate;

- Somewhat awkward semantics which makes it unusable as a desktop audio server (i.e., server doesn't start playback automatically, needs a manual "start" command);

- Not useful on embedded machines;

- No network transparency.

2.4 aRts

The KDE sound server aRts [5] is no longer actively developed and has been orphaned by its developer. Having a full music synthesiser as desktop sound server might not be such a good idea, anyway.

2.5 EsounD

The Enlightened Sound Daemon (EsounD or ESD) [4] has been the audio daemon of choice of the GNOME desktop environment since GNOME 1.0 times. Besides basic mixing and network transparency capabilities, it doesn't offer much. Latency querying, low-latency behaviour, and surround sound are not available at all. It is thus hardly useful for anything beyond basic music playback or playing event sounds ("bing!").

2.6 PortAudio

The PortAudio API [6] is a cross-platform abstraction layer for audio hardware access. As such it sits on top of other audio systems, like OSS, ALSA, ESD, the Windows audio stack, or MacOS X's CoreAudio. PortAudio has not been designed with networked audio devices in mind, and also doesn't provide the necessary functionality for clean integration into a desktop sound server. PortAudio has never experienced wide adoption.

3 What we Need

As shown above, none of the currently available audio systems and APIs can provide all that is necessary on a modern desktop environment. I will now define four major goals which a new desktop audio system should try to achieve.

3.1 A Widely Accepted, Modern, Portable, Easy-to-Use, Powerful, and Complete Audio API

None of the described Linux audio APIs fulfills all requirements that are expected from a modern audio API. On the other hand, Apple's CoreAudio and Microsoft's new Windows Vista user-space audio layer reach this goal, for the most part. More precisely, a modern sound API for Free Software desktops should fulfill the following requirements:

- Completeness: an audio API should be a general-purpose interface; it should be suitable for simple audio playback as well as professional audio production.

- Scalability: usable on all kinds of different systems, ranging from embedded systems to modern desktop PCs and pro audio workstations.

- Modernness: provide good integration into the Free Software desktop ecosystem.

- Proper support for networked and "software" (virtual) devices, besides traditional hardware devices.

- Portability: the audio API should be portable across different operating systems.

- Easy-to-use for both the user, and for the programmer. This includes a certain degree of automatic fine-tuning, to provide optimal functionality with "zero configuration."

3.2 Routing and Filtering Audio in Software

Classic audio systems such as OSS are designed to provide an abstract API around hardware devices. A modern audio system should provide features beyond that:

- It needs to be possible to play back multiple audio streams simultaneously, so that they are mixed in real time.

- Applications should be able to hook into what is currently being played back.

- Before audio is written to an output device, it might be transferred over the network to another machine.

- Before audio is played back some kind of post-processing might take place.

- Audio streams might need to be re-routed during playback.

3.3 Desktop "Bling"

Free Software desktops currently lack an audio counterpart for the well known window manager Compiz. A modern desktop audio systems should be able to provide:

- Separate per-application and per-window volumes.

- Soft fade-ins and fade-outs of music streams.

- Automatically increasing the volume of the application window in the foreground, decreasing the volume of the application window in the background.

- Forward a stop/start request to any music-playing applications if a VoIP call takes place.

- Remember per-application and per-window volumes and devices.

- Reroute audio to a different audio device on-the-fly without interruption, from within the window manager.

- Do "hot" switching between audio devices whenever a new device becomes available. For example, when a USB headset is plugged in, automatically start using it by switching an in-progress VoIP call over to the new headset.

3.4   A Compatible Sound System

Besides providing the features mentioned above, a new sound system for Linux also needs to retain a large degree of compatibility with all the currently available systems and APIs, as much as possible. Optimally, all currently available Linux audio software should work simultaneously and without manual intervention. A major task is to marry the pro-audio and desktop audio worlds into a single audio system.

4   What PulseAudio already provides

The PulseAudio [1] sound server is our attempt to reach the four aforementioned goals. It is a user-space sound server that provides network transparency, all kinds of desktop "bling," relatively low-latency, and is extensible through modules. It sits atop of OSS and ALSA sound devices and routes and filters audio data, possibly over the network.

PulseAudio is intended to be a replacement for systems like ESD or aRts. The former is entirely superseded; PulseAudio may be installed as drop-in replacement for EsounD on GNOME desktops.

PulseAudio ships with a large set of modules (plug-ins):

- Driver modules (i.e., accessing OSS, ALSA, Win32, Solaris drivers).

- Protocol support (i.e., native TCP-based protocol, EsounD protocol, RTP).

- Integration into LIRC, support for multimedia keyboards.

- Desktop integration (i.e., hooks into the X11 system for authentication and redirecting the X11 bell).

- Integration with JACK, EsounD.

- Zeroconf support, using Avahi.

- Management: Automatically restore volumes, devices of playback streams, move a stream to a different device if its original devices becomes unavailable due to a hot-plug event.

- Auto-configuration: integration with HAL for automatic and dynamic configuration of the sound server based on the available hardware.

- Combination of multiple audio devices into a single audio device while synchronising audio clocks.

PulseAudio is not intended to be a competitor to JACK, GStreamer, Helix, KDE Phonon, or Xine. Quite the opposite: we already provide good integration into JACK, GStreamer, and Xine. We have different goals.

The PulseAudio core is carefully optimised for speed and low latency. Local clients may exchange data with the PulseAudio audio server over shared memory data transfer. The PulseAudio sound server will never copy audio data blocks around in memory unless it is absolutely necessary. Most audio data operations are based on `liboil`'s support for the extended instruction sets of modern CPUs (MMX, SSE, AltiVec).

Currently the emphasis for PulseAudio is on networked audio, where it offers the most comprehensive functionality.

PulseAudio support is already available in a large number of applications. For others, we have prepared patches. Currently we have native plug-ins, drivers, patches, and compatibility for Xine, MPlayer, GStreamer, `libao`, XMMS, Audacious, ALSA, OSS (using `$LD_PRELOAD`), EsounD, Music Player Daemon (MPD), and the Adobe Flash player.

PulseAudio has a small number of graphical utility applications:

- Volume Control.

- Panel Applet (for quickly changing the output device, selecting it from a list of Zeroconf-announced audio devices from the network).

- Volume Meter.

- Preferences panel, for a user-friendly configuration of advanced PulseAudio functionality.

- A management console to introspect a PulseAudio server's internals.

5 PulseAudio Internals

5.1 Buffering Model

PulseAudio offers a powerful buffering model which is an extension of the model Jim Gettys pioneered in the networked audio server AF [9]. In contrast to traditional buffering models it offers flexible buffering control, allowing large buffers—which is useful for networked audio systems—while still providing quick response to external events. It allows absolute and relative addressing of samples in the audio buffer and supports a notion of "zero latency." Samples that have already been pushed into the playback buffer may be rewritten at any time.

5.2 Zero-Copy Memory Management

Audio data in the PulseAudio sound server is stored in reference-counted memory blocks. Audio data queues contain only references to these memory blocks instead of the audio data itself. This provides the advantage of minimising copying of audio data in memory, and also saves memory. In fact the PA core is written in a way that, in most cases, data arriving on a network socket is written directly into the sound card DMA hardware buffer without spending time in bounce buffers or similar. This helps to keep memory usage down and allows very low-latency audio processing.

5.3 Shared Memory Data Transfer

Local clients can exchange audio data with a local PulseAudio daemon through shared-memory IPC. Every process allocates a shared memory segment where it stores the audio data it wants to transfer. Then, when the data is sent to another process, the recipient receives only the information necessary to find the data in that segment. The recipient maps the segment of the originator in read-only mode and accesses the data.

The shared memory data transfer is the natural extension of the aforementioned zero-copy memory management, for communication between processes.

5.4 Synchronisation

Multiple streams can be synchronised together on the server side. If this is done, it is guaranteed that the playback indexes of these streams never deviate. Clients can label stream channels freely (e.g., "left," "right," "rear-left," "rear-right," and so on). Together with the aforementioned buffering model, this allows implementation of flexible server-side multi-track mixing.

5.5 Buffer Underrun Handling

PulseAudio does its best to ensure that buffer underruns have no influence on the time axis. Two modes are available: In the first mode, playback pauses when a buffer underrun happens. This is the mode that is usually available in audio APIs such as OSS. In the second mode playback never stops, and if data is available again, enough data is skipped so that the time function experiences no discontinuities.

6 Where we are going

While the PulseAudio project in the current state fulfills a large part of the aforementioned requirements, it is not complete yet. Compatibility with many sound systems, a wide range of desktop audio "bling," networked audio, and low-latency behaviour are already available in current versions of PulseAudio. However we are still lacking in other areas:

- Better low-latency integration into JACK.

- Some further low-latency fixes can be made.

- Better portability to systems which don't support floating-point numbers.

- The client API PulseAudio currently offers is comparatively complicated and difficult to use.

Besides these items, there are also a lot of minor issues to be solved in the PulseAudio project. In the following subsections, I quickly describe the areas we are currently working on.

6.1 Threaded Core

The current PulseAudio core is mostly single-threaded. In most situations this is not a problem, since we carefully make sure that no operation blocks for longer than necessary. However, if more than one audio device is used by a single PulseAudio instance, or when extremely low latencies must be reached, this may become a problem. Thus the PulseAudio core is currently being moved to a more threaded design: every PulseAudio module that is important in low-latency situations will run its own event loop. Communication between those separate event loops, the main event loop, and other local clients is (mostly) done in a wait-free fashion (mostly not lock-free, however). The design we are currently pursuing allows a step-by-step upgrade to this new functionality.

6.2 `libsydney`

During this year's Foundation of Open Media Software (FOMS) conference in January in Sydney, Australia, the most vocally expressed disappointment in the Linux audio world is the lack of a single well-defined, powerful audio API which fulfills the requirements of a modern audio API as outlined above. Since the current native PulseAudio API is powerful but unfortunately overly complex, we took the opportunity to define a new API at that conference. People from Xiph, Nokia, and I sat down to design a new API.

Of course, it might appear as a paradox to try fix the balkanisation of Linux audio APIs by adding yet another one, but given the circumstances, and after careful consideration, we decided to pursue this path. This new API was given the new name `libsydney` [8], named after the city we designed the first version of the API in. `libsydney` will become the only supported API in future PulseAudio versions. Besides working on top of PulseAudio, it will natively support ALSA, OSS, Win32, and Solaris targets. This means that developing a client for PulseAudio will offer cross-platform support for free. `libsydney` is currently in development; by the time of the OLS conference, an initial public version will be made available.

6.3 Multi-User

Currently the PulseAudio audio server is intended to be run as a session daemon. This becomes a problem if multiple users are logged into a single machine simultaneously. Before PulseAudio can be adopted by modern distributions, some kind of hand-over of the underlying audio devices will need to be implemented to support these multi-user setups properly.

7 Where you can get it

PulseAudio is already available in a large number of distributions, including Fedora, Debian, and Ubuntu. Since it is a drop-in replacement for EsounD, it is trivial to install it and use it as the desktop sound server in GNOME.

Alternatively, you may download a version from our web site [1].

It is planned for PulseAudio to replace EsounD in the default install in the next versions of Fedora and Ubuntu.

8 Who we are

PulseAudio has been and is being developed by Lennart Poettering (Red Hat, Inc.) and Pierre Ossman (Cendio AB).

References

[1] PulseAudio, http://pulseaudio.org/

[2] ALSA, http://alsa-project.org/

[3] JACK Audio Connection Kit, http://jackaudio.org/

[4] EsounD, http://www.tux.org/~ricdude/overview.html

[5] aRts, http://www.arts-project.org/

[6] PortAudio, http://www.portaudio.com/

[7] Open Sound System, http://www.opensound.com/oss.html

[8] libsydney, http://0pointer.de/cgi-bin/viewcvs.cgi/trunk/?root=libsydney

[9] AF, http://tns-www.lcs.mit.edu/vs/audiofile.html

Linux-VServer

Resource Efficient OS-Level Virtualization

Herbert Pötzl
`herbert@13thfloor.at`

Marc E. Fiuczynski
`mef@cs.princeton.edu`

Abstract

Linux-VServer is a lightweight virtualization system used to create many independent *containers* under a common Linux kernel. To applications and the user of a Linux-VServer based system, such a container appears just like a separate host.

The Linux-Vserver approach to kernel subsystem containerization is based on the concept of *context isolation*. The kernel is modified to isolate a container into a separate, logical execution context such that it cannot see or impact processes, files, network traffic, global IPC/SHM, etc., belonging to another container.

Linux-VServer has been around for several years and its fundamental design goal is to be an extremely low overhead yet highly flexible production quality solution. It is actively used in situations requiring strong isolation where overall system efficiency is important, such as web hosting centers, server consolidation, high performance clusters, and embedded systems.

1 Introduction

This paper describes Linux-VServer, which is a virtualization approach that applies *context isolation* techniques to the Linux kernel in order to create lightweight container instances. Its implementation consists of a separate kernel patch set that adds approximately 17K lines of code to the Linux kernel. Due to its architecture independent nature it has been validated to work on eight different processor architectures (x86, sparc, alpha, ppc, arm, mips, etc.). While relatively lean in terms of overall size, Linux-VServer touches roughly 460 existing kernel files—representing a non-trivial software-engineering task. Linux-VServer is an efficient and flexible solution that is broadly used for both *hosting* and *sandboxing* scenarios.

Hosting scenarios such as web hosting centers providing Virtual Private Servers (VPS) and HPC clusters need to isolate different groups of users and their applications from each other. Linux-VServer has been in production use for several years by numerous VPS hosting centers around the world. Furthermore, it has been in use since 2003 by PlanetLab (www.planet-lab.org), which is a geographically distributed research facility consisting of roughly 750 machines located in more than 30 countries. Due to its efficiency, a large number of VPSs can be robustly hosted on a single machine. For example, the average PlanetLab machine today has a 2.4Ghz x86 processor, 1GB RAM, and 100GB of disk space and typically hosts anywhere from 40-100 live VPSes. Hosting centers typically use even more powerful servers and it is not uncommon for them to pack 200 VPSes onto a single machine.

Server consolidation is another hosting scenario where isolation between independent application services (e.g., db, dns, web, print) improves overall system robustness. Failures or misbehavior of one service should not impact the performance of another. While hypervisors like Xen and VMware typically dominate the server consolidation space, a Linux-VServer solution may be better when resource efficiency and raw performance are required. For example, an exciting development is that the One Laptop Per Child (OLPC) project has recently decided to use Linux-VServer for their gateway servers that will provide services such as print, web, blog, void, backup, mesh/wifi, etc., to their $100 laptops at school premises. These OLPC gateways servers will be based on low cost / low power consuming embedded systems hardware, and for this reason a resource efficient solution like Linux-VServer rather than Xen was chosen.

Sandboxing scenarios for generic application plugins are emerging on mobile terminals and web browsers to isolate arbitrary plugins–not just Java lets–downloaded by the user. For its laptops OLPC has designed a security framework, called bitfrost [1], and has decided to

utilize Linux-VServer as its sandboxing solution, isolating the various activities from each other and protecting the *main* system from all harm.

Of course, Linux-VServer is not the only approach to implementing containers on Linux. Alternatives include non-mainlined solutions such as OpenVZ, Virtuozzo, and Ensim; and, there is an active community of developers working on kernel patches to incrementally containerize the mainline kernel. While these approaches differ at the implementation level, they all typically focus in onto a single point within the overall containerization design spectrum: *system virtualization*. Benchmarks run on these alternative approaches to Linux-VServer reveal non-trivial time and space overhead, which we believe are fundamentally due to their focus on system virtualization. In contrast, we have found with Linux-VServer that using a context isolation approach to containerize critical kernel subsystems yields neglible overhead compared to a vanilla kernel—and in most cases performance of Linux-VServer and Linux is indistinguishable.

This paper has two goals: 1) serve as a gentle introduction to container-based system for the general Linux community, and 2) highlight both the benefits (and drawbacks) of our context isolation approach to kernel containerization. The next section presents a high-level overview of container-based systems and then describes the Linux-VServer approach in further detail. Section 3 evaluates efficiency of Linux-VServer. Finally, Section 4 offers some concluding remarks.

2 Linux-VServer Design Approach

This section provides an overview of container-based systems, describes the general techniques used to achieve isolation, and presents the mechanisms with which Linux-VServer implements these techniques.

2.1 Container-based System Overview

At a high-level, a container-based system provides a shared, virtualized OS image, including a unique root file system, a set of system executables and libraries, and resources (cpu, memory, storage, etc.) assigned to the container when it is created. Each container can be "booted" and "shut down" just like a regular operating system, and "rebooted" in only seconds when necessary.
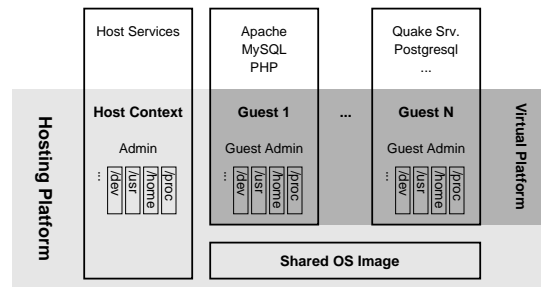

Figure 1: Container-based Platform

To applications and the user of a container-based system, the container appears just like a separate linux system.

Figure 1 depicts a container-based system, which is comprised of two basic platform groupings. The hosting platform consists essentially of the shared OS image and a privileged *host context*. This is the context that a system administrator uses to manage containers. The virtual platform is the view of the system as seen by the *guest containers*. Applications running in a guest container work just as they would on a corresponding non-container-based system. That is, they have their own root file system, IP addresses, `/dev`, `/proc`, etc.

The subsequent sections will focus on the main contributions that Linux-VServer makes, rather than being exhaustively describing all required kernel modifications.

2.2 Kernel Containerization

This section describes the kernel subsystem enhancements that Linux-VServer makes to support containers. These enhancements are designed to be low overhead, flexible, as well as to enhance security in order to properly confine applications into a container. For CPU scheduling, Linux-VServer introduces a novel filtering technique in order to support fair-share, work-conserving, or hard limit container scheduling. However, in terms of managing system resources such as storage space, io bandwidth, for Linux-VServer it is mostly an exercise of leveraging existing Linux resource management and accounting facilities.

2.2.1 Guest Filesystems

Guest container filesystems could either be implemented using loop-back mounted images—as is typical

for qemu, xen, vmware, and uml based systems—or by simply using the native file systems and using chroot. The main benefit of using a chroot-ed filesystem over the loop-back mounted images is performance. Guests can read/write files at native filesystem speed. However, there are two drawbacks: 1) chroot() information is volatile and therefore only provides weak confinement, and 2) chroot-ed filesystems may lead to significant duplication of common files. Linux-VServer addresses both of these problems, as one of its central objectives is to support containers in a resource efficient manner that performs as well as native Linux.

Filesystem Chroot Barrier

Because chroot() information is volatile, it is simple to escape from a chroot-ed environment, which would be bad from a security perspective when one wants to maintain the invariant that processes are confined within their containers filesystem. This invariant is nice to have when using containers for generic hosting scenarios, but clearly is required for sandboxing scenario. To appreciate how easy it is to escape conventional chroot() confinement, consider the following three simple steps: a) create or open a file and retain the file-descriptor, b) chroot into a subdirectory at equal or lower level with regards to the file, which causes the 'root' to be moved 'down' in the filesystem, and then c) use fchdir() on the file descriptor to escape from that 'new' root, which lets the process escape from the 'old' root as well, as this was lost in the last chroot() system call.

To address this problem Linux-VServer uses a special file attribute called the *chroot barrier*. The above trick does not work when this barrier is set on the root directory of a chroot-ed filesystem, as it prevents unauthorized modification and escape from the chroot confinement.

Filesystem Unification

To appreciate the second drawback mentioned above, consider that systems with dozens or maybe even hundreds of containers based on the same Linux distribution will unnecessarily duplicate many common files. This duplication occurs simply to maintain a degree of separation between containers, as it would be difficult using conventional Linux filesystem techniques to ensure safe sharing of files between containers.

To address this problem Linux-VServer implements a disk space saving technique by using a simple unification technique applied to whole files. The basic approach is that files common to more than one container, which are rarely going to change (e.g., like libraries and binaries from similar OS distributions), can be hard linked on a shared filesystem. This is possible because the guest containers can safely share filesystem objects (inodes).

The only drawback with hard linking files is that without additional measures, a container could (un)intentionally destroy or modify such shared files, which in turn would harm/interfere other containers.

This can easily be addressed by adding an immutable attribute to the file, which then can be safely shared between two containers. In order to ensure that a container cannot modify such a file directly, the Linux *capability* to modify this attribute is removed from the set of capabilities given to a guest container.

However, removing or updating a file with immutable link attribute set from inside a guest container would be impossible. To remove the file the additional "permission to unlink" attribute needs to be set. With this alone an application running inside a container could manually implement a *poor man's* CoW system by: copying the original file, making modifications to the copy, unlinking the original file, and renaming the copy the original filename.

This technique was actually used by older Linux-VServer based systems, but this caused some incompatibilities with programs that make in-place modifications to files. To address this problem, Linux-VServer introduced *CoW link breaking* which treats shared hard-linked files as copy-on-write (CoW) candidates. When a container attempts to mutate a CoW marked file, the kernel will create a private copy of the file for the container.

Such CoW marked files belonging to more than one container are called 'unified' and the process of finding common files and preparing them in this way is called *Filesystem Unification*. Unification is done as an out-of-band operation by a process run in the root container— typically a cron job that intelligent walks all of the containers' filesystems looking for identical files to unify.

The principal reason for doing filesystem unification is reduced resource consumption, not simplified administration. While a typical Linux distribution install will consume about 500MB of disk space, our experience is that after unification the incremental disk space required

when creating a new container based on the same distribution is on the order of a few megabytes.

It is straightforward to see that this technique reduces required disk space, but probably more importantly it improves memory mappings for shared libraries, reduces inode caches, slab memory for kernel structures, etc. Section 3.1 quantifies these benefits using a real world example.

2.2.2 Process Isolation

Linux-VServer uses the global PID space across all containers. Its approach is to hide all processes outside a container's scope, and prohibits any unwanted interaction between a process inside a container and a process belonging to another container. This separation requires the extension of some existing kernel data structures in order for them to: a) become aware to which container they belong, and b) differentiate between identical UIDs used by different containers. To work around false assumptions made by some user-space tools (like pstree) that the `init` process has to exist and have PID 1, Linux-VServer also provides a per container mapping from an arbitrary PID to a fake init process with PID 1.

When a Linux-VServer based system boots, by default all processes belong to the *host context*. To simplify system administration, this host context acts like a normal Linux system and doesn't expose any details about the guests, except for a few proc entries. However, to allow for a global process view, Linux-VServer defines a special *spectator context* that can peek at all processes at once. Both the host and spectator context are only logical containers—i.e., unlike guest containers, they are not implemented by kernel datastructures.

A side effect of this approach is that process migration from one container to another container *on the same host* is achieved by changing its container association and updating the corresponding per-container resource usage statistics such NPROC, NOFILE, RSS, ANON, MEMLOCK, etc.

The benefit to this isolation approach for the systems process abstraction is twofold: 1) that it scales well with a large number of contexts, 2) most critical-path logic manipulating processes and PIDs remain unchanged. The drawback is that one cannot as cleanly implement container migration, checkpoint and resume, because

it may not be possibly to re-instantiate processes with the same PID. To overcome this drawback, alternative container-based systems virtualize the PID space on a per container basis.

We hope to positively influence the proposed kernel mainlining of containerized PID space support such that depending on the usage scenario it is possible to choose Linux-VServer style isolation, virtualization, or a hybrid thereof.

2.2.3 Network Isolation

Linux-VServer does not fully virtualize the networking subsystem. Rather, it shares the networking subsystem (route tables, IP tables, etc.) between all containers, but restricts containers to bind sockets to a subset of host IPs specified either at container creation or dynamically by the host administrator. This has the drawback that it does not let containers change their route table entries or IP tables rules. However, it was a deliberate design decision, as it inherently lets Linux-VServer containers achieve native networking performance.

For Linux-VServer's *network isolation* approach several issues have to be considered; for example, the fact that bindings to special addresses like IPADDR_ANY or the local host address have to be handled to avoid having one container receive or snoop traffic belonging to another container. The approach to get this right involves tagging packets with the appropriate container identifier and incorporating the appropriate filters in the networking stack to ensure only the right container can receive them. Extensive benchmarks reveal that the overhead of this approach is minimal as high-speed networking performance is indistinguishable between a native Linux system and one enhanced with Linux-VServer regardless of the number of concurrently active containers.

In contrast, the best network L2 or L3 virtualization approaches as implemented in alternative container-based systems impose significant CPU overhead when scaling the number of concurrent, high-performance containers on a system. While network virtualization is a highly flexible and nice feature, our experience is that it is not required for all usage scenarios. For this reason, we believe that our network isolation approach should be a feature that high-performance containers should be permitted to select at run time.

Again, we hope to positively influence the proposed kernel mainlining of network containerization support such that depending on the usage scenario it is possible to choose Linux-VServer style isolation, virtualization, or a hybrid thereof.

2.2.4 CPU Isolation

Linux-VServer implements CPU isolation by overlaying a *token bucket* scheme on top of the standard Linux CPU scheduler. Each container has a token bucket that accumulates tokens at a specified rate; every timer tick, the container that owns the running process is charged one token. A container that runs out of tokens has its processes removed from the run-queue until its bucket accumulates a minimum amount of tokens. This token bucket scheme can be used to provide fair sharing **and/or** work-conserving CPU reservations. It can also enforce hard limits (i.e., an upper bound), as is popularly used by VPS hosting centers to limit the number of cycles a container can consume—even when the system has idle cycles available.

The rate that tokens accumulate at in a container's bucket depends on whether the container has a *reservation* and/or a *share*. A container with a reservation accumulates tokens at its reserved rate: for example, a container with a 10% reservation gets 100 tokens per second, since a token entitles it to run a process for one millisecond. A container with a share that has runnable processes will be scheduled before the idle task is scheduled, and only when all containers with reservations have been honored. The end result is that the CPU capacity is effectively partitioned between the two classes of containers: containers with reservations get what they've reserved, and containers with shares split the unreserved capacity of the machine proportionally. Of course, a container can have both a reservation (e.g., 10%) and a fair share (e.g., 1/10 of idle capacity).

2.2.5 Network QoS

The Hierarchical Token Bucket (`htb`) queuing discipline of the Linux Traffic Control facility (`tc`) [2] can be used to provide network bandwidth reservations and fair service. For containers that have their own IP addresses, the `htb` kernel support just works without modifications.

However, when containers share an IP address, as is done by PlanetLab, it is necessary to track packets in order to apply a queuing discipline to a containers flow of network traffic. This is accomplished by tagging packets sent by a container with its context id in the kernel. Then, for each container, a token bucket is created with a *reserved rate* and a *share*: the former indicates the amount of outgoing bandwidth dedicated to that container, and the latter governs how the container shares bandwidth beyond its reservation. The `htb` queuing discipline then allows each container to send packets at the reserved rate of its token bucket, and fairly distributes the excess capacity to other containers in proportion to their shares. Therefore, a container can be given a capped reservation (by specifying a reservation but no share), "fair best effort" service (by specifying a share with no reservation), or a work-conserving reservation (by specifying both).

2.2.6 Disk QoS

Disk I/O is managed in Linux-VServer using the standard Linux CFQ ("completely fair queuing") I/O scheduler. The CFQ scheduler attempts to divide the bandwidth of each block device fairly among the containers performing I/O to that device.

2.2.7 Storage Limits

Linux-VServer provides the ability to associate limits to the amount of memory and disk storage a container can acquire. For disk storage one can specify limits on the maximum number of disk blocks and inodes a container can allocate. For memory, a variety of different limits can be set, controlling the Resident Set Size and Virtual Memory assigned to each context.

Note that fixed upper bounds on RSS are not appropriate for usage scenarios where administrators wish to overbook containers. In this case, one option is to let containers compete for memory, and use a watchdog daemon to recover from overload cases—for example by killing the container using the most physical memory. PlanetLab [3] is one example where memory is a particularly scarce resource, and memory limits without overbooking are impractical: given that there are up to 90 active containers on a PlanetLab server, this would imply a tiny 10MB allocation for each container

on the typical PlanetLab server with 1GB of memory. Instead, PlanetLab provides basic memory isolation between containers by running a simple watchdog daemon, called `pl_mom`, which resets the container consuming the most physical memory when swap has almost filled. This penalizes the memory hog while keeping the system running for everyone else, and is effective for the workloads that PlanetLab supports. A similar technique is apparently used by managed web hosting companies.

3 Evaluation

In a prior publication [5], we compared in further detail the performance of Linux-VServer with both vanilla Linux and Xen 3.0 using lmbench, iperf, and dd as microbenchmarks and kernel compile, dbench, postmark, osdb as synthetic macrobenchmarks. Our results from that paper revealed that Linux-VServer has in the worst case a 4 percent overhead when compared to an unvirtualized, vanilla Linux kernel; however, in most cases Linux-VServer is nearly identical in performance and in a few lucky cases—due to gratuitous cache effects—Linux-VServer consistently outperforms vanilla kernel. Please consult our other paper [5] for these details.

This section explores the efficiency of Linux-VServer. We refer to the combination of scale and performance as the *efficiency* of the system, since these metrics correspond directly to how well the virtualizing system orchestrates the available physical resources for a given workload. All experiments are run on HP Proliant servers with dual core processors, 2MB caches, 4GB RAM, and 7.2k RPM SATA-100 disks.

3.1 Filesystem Unification

As discussed in Section 2.2.1, Linux-VServer supports a unique filesystem unification model. The reason for doing filesystem unification is to reduce disk space consumption, but more importantly it reduces system resource consumption. We use a real world example to demonstrate this benefit from filesystem unification.

We configure guest containers with Mandriva 2007. The disk footprint of a non-unified installation is 150MB per guest. An activated guest runs a complement of daemons and services such as syslog, crond, sshd, apache, postfix and postgresql—a typical configuration used in VPS hosting centers.

We evaluate the effectiveness of filesystem unification using two tests. The first consists of starting 200 separate guests one after the other measuring memory consumption. The second test is identical to the first, except before all guests are started, their filesystems are unified. For the latter test, the disk footprint of each unified guest reduces from 150MB to 10MB, resulting in 140MB of common and thus shared data on disk.

Tables 1 and 2 summarize the results for this test, categorizing how much time it takes for the guest to start up and how much memory it consumes categorized by memory type such as active, buffer, cache, slab, etc. What these columns in the two tables reveal is that the kernel inherently shares memory for shared libraries, binaries, etc. due to the unification (i.e., hard linking) of files.

Table 3 compares the rows with the 200th guest, which shows the difference and overhead percentage in consumed memory as well as the time required to start the same number of guest container. These differences are significant!

Of course, techniques exist to recoup redundant memory resources (e.g., VMware's content-based memory sharing used in its ESX product line [6]). However, such techniques require the system to actively seek out redundant memory by computing hash keys page data, etc., which introduces non-trivial overhead. In contrast, with the filesystem unification approach we inherently obtain this benefit.

As a variation of our prior experiment, we have also measured starting the 200 guest containers in parallel. The results for this experiment are shown in Figure 2. The first thing to note in the figure is that the parallel startup of 200 separate guests causes the machine to spend most of the very long startup (approx. 110min) paging in and out data (libraries, executeables, shared files), which causes the cpu to hang in iowait most of the time (light blue area in the cpu graphs) rendering the system almost unresponsive. In contrast, the same startup with 200 unified guests is rather fast (approx. 20min), and most of the startup time is spent on actual guest processes.

3.2 Networking

We also evaluated the efficiency of network operations by comparing 2.6.20 based kernels, one unmodified kernel, and one patched with Linux-VServer 2.2. Two sets

Guest	Time	Active	Buffers	Cache	Anon	Mapped	Slab	Recl.	Unrecl.
001	0	16364	2600	20716	4748	3460	8164	2456	5708
002	7	30700	3816	42112	9052	8200	11056	3884	7172
003	13	44640	4872	62112	13364	12872	13248	5268	7980
…	…	…	…	…	…	…	…	…	…
198	1585	2093424	153400	2399560	849696	924760	414892	246572	168320
199	1593	2103368	151540	2394048	854020	929660	415300	246324	168976
200	1599	2113004	149272	2382964	858344	934336	415528	245896	169632

Table 1: Memory Consumption—Separate Guests

Guest	Time	Active	Buffers	Cache	Anon	Mapped	Slab	Recl.	Unrecl.
001	0	16576	2620	20948	4760	3444	8232	2520	5712
002	10	31368	4672	74956	9068	8140	12976	5760	7216
003	14	38888	5364	110508	13368	9696	16516	8360	8156
…	…	…	…	…	…	…	…	…	…
198	1304	1172124	88468	2492268	850452	307596	384560	232988	151572
199	1313	1178876	88896	2488476	854840	309092	385384	233064	152320
200	1322	1184368	88568	2483208	858988	310640	386256	233388	152868

Table 2: Memory Consumption—Unified Guests

Attribute	Difference	Overhead
Time	277 s	21.0 %
Active	928848 k	79.5 %
Buffers	60724 k	70.7 %
Cache	100012 k	-4.2 %
Anon	632 k	0.0 %
Mapped	623680 k	203.0 %
Slab	29340 k	7.8 %
Recl.	12572 k	5.4 %
Unrecl.	16768 k	11.4 %

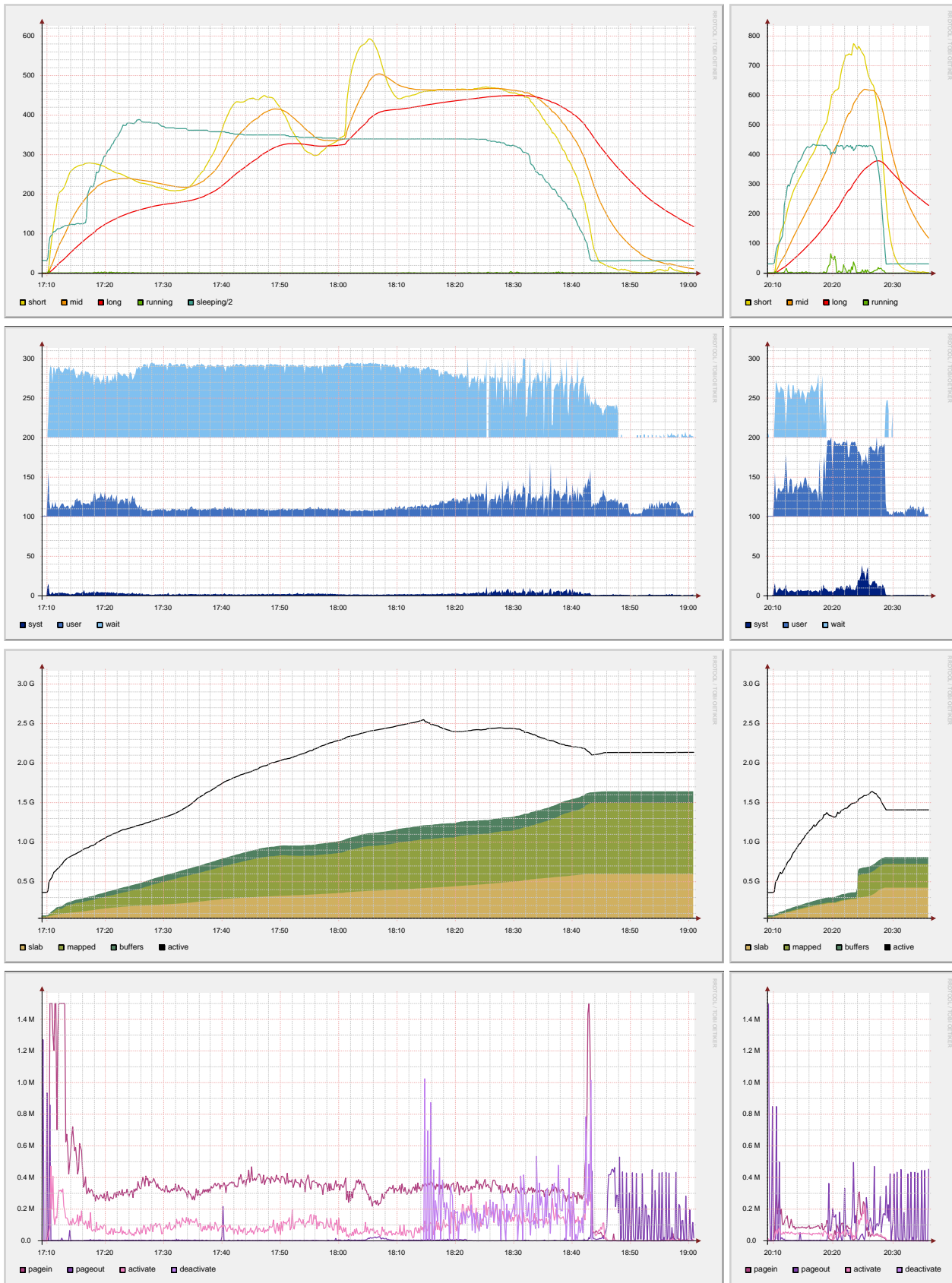Table 3: Overhead Unified vs. Separate

Figure 2: Parallel Startup of 200 Guests—Separate (left) vs. Unified (right)

of experiments were conducted: the first measuring the throughput of packets sent and received over the network for a CPU bound workload, and the second assessing the cost of using multiple heavily loaded Linux-VServer containers concurrently. While the former evaluates the ability of a single container to saturate a high-speed network link, the latter measures the efficiency of Linux-VServer's utilization of the underlying hardware. In both experiments, fixed-size UDP packets were exchanged between containers and a remote system on the local network. HP Proliant servers were used for both the sender and receiver, connected through a Gigabit network, equipped with 2.4Ghz Xeon 3060.

The first set of experiments demonstrated that the performance of Linux-VServer is equivalent to that of vanilla Linux. This is in line with our prior iperf-based TCP throughput results [5].

The second set of experiments involved continually increasing the number of clients confined in Linux-VServer based containers sending UDP packets as in the previous experiment. We made two observations: 1) the CPU utilization of Linux-VServer containers for packet sizes with which the network was saturated was marginally higher (77% as opposed to 72%), and 2) the increase in the CPU utilization of Linux-VServer, and the threshold beyond which it saturated the CPU was identical to that of Native Linux as containers were added.

These experiments suggest that for average workloads, the degradation of performance using the Linux-VServer network isolation approach is marginal. Furthermore, Linux-VServer can scale to multiple concurrent containers exchanging data at high rates with a performance comparable to native Linux.

3.3 CPU Fair Share and Reservations

To investigate both CPU isolation of a single resource and resource guarantees, we use a combination of CPU intensive tasks. Hourglass is a synthetic real-time application useful for investigating scheduling behavior at microsecond granularity [4]. It is CPU-bound and involves no I/O.

Eight containers are run simultaneously. Each container runs an instance of hourglass, which records contiguous periods of time scheduled. Because hourglass uses no

I/O, we may infer from the gaps in its time-line that either another container is running or the virtualized system is running on behalf of another container, in a context switch for instance. The aggregate CPU time recorded by all tests is within 1% of system capacity.

We evaluated two experiments: 1) all containers are given the same fair share of CPU time, and 2) one of the containers is given a reservation of $1/4^{th}$ of overall CPU time. For the first experiment, VServer for both UP and SMP systems do a good job at scheduling the CPU among the containers such that each receive approximately one eights of the available time.

For the second experiment we observe that the CPU scheduler for Linux-VServer achieves the requested reservation within 1%. Specifically, the container having requested $1/4^{th}$ of overall CPU time receives 25.16% and 49.88% on UP and SMP systems, respectively. The remaining CPU time is fairly shared amongst the other seven containers.

4   Conclusion

Virtualization technology in general benefits a wide variety of usage scenarios. It promises such features as configuration independence, software interoperability, better overall system utilization, and resource guarantees. This paper described the Linux-VServer approach to providing these features while balancing the tension between strong isolation of co-located containers with efficient sharing of the physical resources on which the containers are hosted.

Linux-VServer maintains a small kernel footprint, but it is not yet feature complete as it lacks support for true network virtualization and container migration. These are features that ease management and draw users to hypervisors such as Xen and VMware, particularly in the server consolidation and hosting scenarios. There is an active community of developers working towards adding these features to the mainline Linux kernel, which we expect will be straightforward to integrate with Linux-VServer.

In the mean time, for managed web hosting, PlanetLab, the OLPC laptop and gateway server, embedded systems, etc., the trade-off between isolation and efficiency is of paramount importance. We believe that Linux-VServer hits a sweet spot in the containerization design

space, as it provides for strong isolation and it performs equally with native Linux kernels in most cases.

References

[1] Ivan Krstic. System security on the One Laptop per Child's XO laptop: the Bitfrost security platform. `http://wiki.laptop.org/go/Bitfrost`.

[2] Linux Advanced Routing and Traffic Control. `http://lartc.org/`.

[3] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[4] John Regehr. Inferring scheduling behavior with hourglass. In *In Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference*, June 2002.

[5] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. 2nd EUROSYS*, Lisboa, Portugal, March 2007.

[6] Carl Waldspurger. Memory resource management in vmware esx server. In *Proc. 5th OSDI*, Boston, MA, Dec 2002.

Internals of the RT Patch

Steven Rostedt
Red Hat, Inc.
srostedt@redhat.com
rostedt@goodmis.org

Darren V. Hart
IBM Linux Technology Center
dvhltc@us.ibm.com

Abstract

Steven Rostedt (srostedt@redhat.com)

Over the past few years, Ingo Molnar and others have worked diligently to turn the Linux kernel into a viable Real-Time platform. This work is kept in a patch that is held on Ingo's page of the Red Hat web site [7] and is referred to in this document as **the RT patch**. As the RT patch is reaching maturity, and slowly slipping into the upstream kernel, this paper takes you into the depths of the RT patch and explains exactly what it is going on. It explains Priority Inheritance, the conversion of Interrupt Service Routines into threads, and transforming spin_locks into mutexes and why this all matters. This paper is directed toward kernel developers that may eventually need to understand Real-Time (RT) concepts and help them deal with design and development changes as the mainline kernel heads towards a full fledge Real-Time Operating System (RTOS). This paper will offer some advice to help them avoid pitfalls that may come as the mainline kernel comes closer to an actual RTOS.

The RT patch has not only been beneficial to those in the Real-Time industry, but many improvements to the mainline kernel have come out of the RT patch. Some of these improvements range from race conditions that were fixed to reimplementation of major infrastructures.[1] The cleaner the mainline kernel is, the easier it is to convert it to an RTOS. When a change is made to the RT patch that is also beneficial to the mainline kernel, those changes are sent as patches to be incorporated into mainline.

1 The Purpose of a Real-Time Operating System

The goal of a Real-Time Operating System is to create a predictable and deterministic environment. The primary purpose is not to increase the speed of the system, or lower the latency between an action and response, although both of these increase the quality of a Real-Time Operating System. The primary purpose is to eliminate "surprises." A Real-Time system gives control to the user such that they can develop a system in which they can calculate the actions of the system under any given load with deterministic results. Increasing performance and lowering latencies help in this regard, but they are only second to deterministic behavior. A common misconception is that an RTOS will improve throughput and overall performance. A quality RTOS still maintains good performance, but an RTOS will sacrifice throughput for predictability.

To illustrate this concept, let's take a look at a hypothetical algorithm that on a non Real-Time Operating System, can complete some calculation in 250 microseconds on average. An RTOS on the same machine may take 300 microseconds for that same calculation. The difference is that an RTOS can guarantee that the worst case time to complete the calculation is known in advanced, and the time to complete the calculation will not go above that limit.[2] The non-RTOS can not guarantee a maximum upper limit time to complete that algorithm. The non-RTOS may perform it in 250 microseconds 99.9% of the time, but 0.1% of the time, it might take 2 milliseconds to complete. This is totally unacceptable for an RTOS, and may result in system failure. For example, that calculation may determine if a device driver needs to activate some trigger that must be set within 340 microseconds or the machine will lock up. So we see that a non-RTOS may have a better average

---

[1] such as hrtimers and generic IRQs

[2] when performed by the highest priority thread.

performance than an RTOS, but an RTOS guarantees to meet its execution time deadlines.

The above demonstrates an upper bound requirement for completing a calculation. An RTOS must also implement the requirement of response time. For example, a system may have to react to an asynchronous event. The event may be caused by an external stimulus (hitting a big red button) or something that comes from inside the system (a timer interrupt). An RTOS can guarantee a maximum response time from the time the stimulant occurs to the time the reaction takes place.

1.1 Latencies

The time between an event is expected to occur and the time it actually does is called **latency**. The event may be an external stimulus that wants a response, or a thread that has just woken up and needs to be scheduled. The following is the different kinds and causes of latencies and these terms will be used later in this paper.

- `Interrupt Latency` — The time between an interrupt triggering and when it is actually serviced.

- `Wakeup Latency` — The time between the highest priority task being woken up and the time it actually starts to run. This also can be called `Scheduling Latency`.

- `Priority Inversion` — The time a high priority thread must wait for a resource owned by a lower priority thread.

- `Interrupt Inversion` — The time a high priority thread must wait for an interrupt to perform a task that is of lower priority.

Interrupt latency is the easiest to measure since it corresponds tightly to the time interrupts are disabled. Of course, there is also the time that it takes to make it to the actual service routine, but that is usually a constant value.[3] The duration between the waking of a high priority process and it actually running is also a latency. This sometimes includes interrupt latency since waking of a process is usually due to some external event.

Priority inversion is not a latency but the effect of priority inversion causes latency. The amount of time a thread must wait on a lower priority thread is the latency due to priority inversion. Priority inversion can not be prevented, but an RTOS must prevent unbounded priority inversion. There are several methods to address unbounded priority inversion, and Section 6 explains the method used by the RT patch.

Interrupt inversion is a type of priority inversion where a thread waits on an interrupt handler servicing a lower priority task. What makes this unique, is that a thread is waiting on an interrupt context that can not be preempted, as opposed to a thread that can be preempted and scheduled out. Section 2 explains how threaded interrupts address this issue.

2 Threaded Interrupts

As mentioned in Section 1.1, one of the causes of latency involves interrupts servicing lower priority tasks. A high priority task should not be greatly affected by a low priority task, for example, doing heavy disk IO. With the normal interrupt handling in the mainline kernel, the servicing of devices like hard-drive interrupts can cause large latencies for all tasks. The RT patch uses threaded interrupt service routines to address this issue.

When a device driver requests an IRQ, a thread is created to service this interrupt line.[4] Only one thread can be created per interrupt line. Shared interrupts are still handled by a single thread. The thread basically performs the following:

```
while (!kthread_should_stop()) {
  set_current_state
          (TASK_INTERRUPTIBLE);
  do_hardirq(desc);
  cond_resched();
  schedule();
}
```

Here's the flow that occurs when an interrupt is triggered:

The architecture function `do_IRQ()`[5] calls one of the following chip handlers:

---

[3]except with the RT kernel, see Section 2.

[4]See `kernel/irq/manage.c do_irqd`.

[5]See `arch/<arch>/kernel/irq.c`. (May be different in some architectures.)

- `handle_simple_irq`

- `handle_level_irq`

- `handle_fasteoi_irq`

- `handle_edge_irq`

- `handle_percpu_irq`

Each of these sets the IRQ descriptor's status flag `IRQ_INPROGRESS`, and then calls `redirect_hardirq()`.

`redirect_hardirq()` checks if threaded interrupts are enabled, and if the current IRQ is threaded (the IRQ flag `IRQ_NODELAY` is not set) then the associated thread (`do_irqd`) is awaken. The interrupt line is masked and the interrupt exits. The cause of the interrupt has not been handled yet, but since the interrupt line has been masked, that interrupt will not trigger again. When the interrupt thread is scheduled, it will handle the interrupt, clear the `IRQ_INPROGRESS` status flag, and unmask the interrupt line.

The interrupt priority inversion latency time is only the time from the triggering of the interrupt, the masking of the interrupt line, the waking of the interrupt thread, and returning back to the interrupted code, which takes on a modern computer system a few microseconds. With the RT patch, a thread may be given a higher priority than a device handler interrupt thread, so when the device triggers an interrupt, the interrupt priority inversion latency is only the masking of the interrupt line and waking the interrupt thread that will handle that interrupt. Since the high priority thread may be of a higher priority than the interrupt thread, the high priority thread will not have to wait for the device handler that caused that interrupt.

2.1 Hard IRQs That Stay Hard

It is important to note that there are cases where an interrupt service routine is not converted into a thread. Most notable example of this is the timer interrupt. The timer interrupt is handled in true interrupt context, and is not serviced by a thread. This makes sense since the timer interrupt controls the triggering of time events, such as, the scheduling of most threads.

A device can also specify that its interrupt handler shall be a true interrupt by setting the interrupt descriptor flag

`IRQ_NODELAY`. This will force the interrupt handler to run in interrupt context and not as a thread. Also note that an `IRQ_NODELAY` interrupt can not be shared with threaded interrupt handlers. The only time that `IRQ_NODELAY` should be used is if the handler does very little and does not grab any spin_locks. If the handler acquires spin_locks, it will crash the system in full `CONFIG_PREEMPT_RT` mode.[6]

It is recommended never to use the `IRQ_NODELAY` flag unless you fully understand the RT patch. The RT patch takes advantage of the fact that interrupt handlers run as threads, and allows for code that is used by interrupt handlers, that would normally never schedule, to schedule.

2.2 Soft IRQs

Not only do hard interrupts run as threads, but all soft IRQs do as well. In the current mainline kernel,[7] soft IRQS are usually handled on exit of a hard interrupt. They can happen anytime interrupts and soft IRQs are enabled. Sometimes when a large number of soft IRQs need to be handled, they are pushed off to the `ksoftirqd` thread to complete them. But a soft IRQ handler can not assume that it will be running in a threaded context.

In the RT patch, the soft IRQs are only handled in a thread. Furthermore, they are split amongst several threads. Each soft IRQ has its own thread to handle them. This way, the system administrator can control the priority of individual soft IRQ threads.

Here's a snapshot of soft IRQ and hard IRQ threads, using `ps -eo pid,pri,rtprio,cmd`.

---

[6]see Section 4.
[7]2.6.21 as the time of this writing.

PID	PRI	RTPRIO	CMD
4	90	50	[softirq-high/0]
5	90	50	[softirq-timer/0]
6	90	50	[softirq-net-tx/]
7	90	50	[softirq-net-rx/]
8	90	50	[softirq-block/0]
9	90	50	[softirq-tasklet]
10	90	50	[softirq-sched/0]
11	90	50	[softirq-hrtimer]
12	90	50	[softirq-rcu/0]
304	90	50	[IRQ-8]
347	90	50	[IRQ-15]
381	90	50	[IRQ-12]
382	90	50	[IRQ-1]
393	90	50	[IRQ-4]
400	90	50	[IRQ-16]
401	90	50	[IRQ-18]
402	90	50	[IRQ-17]
413	90	50	[IRQ-19]

3 Kernel Preemption

A critical section in the kernel is a series of operations that must be performed atomically. If a thread accesses a critical section while another thread is accessing it, data can be corrupted or the system may become unstable. Therefore, critical sections that can not be performed atomically by the hardware, must provide mutual exclusion to these areas. Mutual exclusion to a critical section may be implemented on a uniprocessor (UP) system by simply preventing the thread that accesses the section from being scheduled out (disable preemption). On a symmetric multiprocessor (SMP) system, disabling preemption is not enough. A thread on another CPU might access the critical section. On SMP systems, critical sections are also protected with locks.

An SMP system prevents concurrent access to a critical section by surrounding it with spin_locks. If one CPU has a thread accessing a critical section when another CPU's thread wants to access that same critical section, the second thread will perform a busy loop (spin) until the previous thread leaves that critical section.

A preemptive kernel must also protect those same critical sections from one thread accessing the section before another thread has left it. Preemption must be disabled while a thread is accessing a critical section, otherwise another thread may be scheduled and access that same critical section.

Linux, prior to the 2.5 kernel, was a non-preemptive kernel. That means that whenever a thread was running in kernel context (a user application making a system call) that thread would not be scheduled out unless it volunteered to schedule (calls the scheduler function). In the development of the 2.5 kernel, Robert Love introduced kernel preemption [2]. Robert Love realized that the critical sections that are protected by spin_locks for SMP systems, are also the same sections that must be protected from preemption. Love modified the kernel to allow preemption even when a thread is in kernel context. Love used the spin_locks to mark the critical sections that must disable preemption.[8]

The 2.6 Linux kernel has an option to enable kernel preemption. Kernel preemption has improved reaction time and lowered latencies. Although kernel preemption has brought Linux one step closer to an RTOS, Love's implementation contains a large bottleneck. A high priority process must still wait on a lower priority process while it is in a critical section, even if that same high priority process did not need to access that section.

4 Sleeping Spin Locks

Spin_locks are relatively fast. The idea behind a spin_lock is to protect critical sections that are very short. A spin_lock is considered fast compared to sleeping locks because it avoids the overhead of a schedule. If the time to run the code in a critical section is shorter than the time of a context switch, it is reasonable to use a spin_lock, and on contention, spin in a busy loop, while waiting for a thread on another CPU to release the spin_lock.

Since spin_locks may cause a thread on another CPU to enter a busy loop, extra care must be given with the use of spin_locks. A spin_lock that can be taken in interrupt context must always be taken with interrupts disabled. If an interrupt context handler that acquires a spin_lock is triggered while the current thread holds that same spin_lock, then the system will deadlock. The interrupt handler will spin on the spin_lock waiting for the lock to be released, but unfortunately, that same interrupt handler is preventing the thread that holds the spin_lock from releasing it.

A problem with the use of spin_locks in the Linux kernel is that they also protect large critical sections. With

---

[8]other areas must also be protected by preemption (e.g., interrupt context).

the use of nested spin_locks and large sections being protected by them, the latencies caused by spin_locks become a big problem for an RTOS. To address this, the RT patch converts most spin_locks into a mutex (sleeping lock).

By converting spin_locks into mutexes, the RT patch also enables preemption within these critical sections. If a thread tries to access a critical section while another thread is accessing it, it will now be scheduled out and sleep until the mutex protecting the critical section is released.

When the kernel is configured to have sleeping spin_locks, interrupt handlers must also be converted to threads since interrupt handlers also use spin_locks. Sleeping spin_locks are not compatible with not-threaded interrupts, since only a threaded interrupt may schedule. If a device interrupt handler uses a spin_lock and also sets the interrupt flag `IRQ_NODELAY`, the system will crash if the interrupt handler tries to acquire the spin_lock when it is already taken.

Some spin_locks in the RT kernel must remain a busy loop, and not be converted into a sleeping spin_lock. With the use of type definitions, the RT patch can magically convert nearly all spin_locks into sleeping mutexes, and leave other spin_locks alone.

5 The Spin Lock Maze

To avoid having to touch every spin_lock in the kernel, Ingo Molnar developed a way to use the latest gcc extensions to determine if a spin_lock should be used as a mutex, or stay as a busy loop.[9] There are places in the kernel that must still keep a true spin_lock, such as the scheduler and the implementation of mutexes themselves. When a spin_lock must remain a spin_lock the RT patch just needs to change the type of the spin_lock from `spinlock_t` to `raw_spinlock_t`. All the actual spin_lock function calls will determine at compile time which type of spin_lock should be used. If the spin_lock function's parameter is of the type `spinlock_t` it will become a mutex. If a spin_lock function's parameter is of the type `raw_spinlock_t` it will stay a busy loop (as well as disable preemption).

Looking into the header files of spinlock.h will drive a normal person mad. The macros defined in those

headers are created to actually facilitate the code by not having to figure out whether a spin_lock function is for a mutex or a busy loop. The header files, unfortunately, are quite complex. To make it easier to understand, I will not show the actual macros that make up the spin_lock function, but, instead, I will show what it looks like evaluated slightly.

```
#define spin_lock(lock)
  if (TYPE_EQUAL((lock),
raw_spinlock_t))
     __spin_lock(lock);
  else if (TYPE_EQUAL((lock),
spinlock_t))
     _spin_lock(lock);
  else __bad_spinlock_type();
```

The `TYPE_EQUAL` is defined as `__builtin_types_ compatible_p(typeof(lock), type *)` which is a gcc internal command that handles the condition at compile time. The `__bad_spinlock_type` function is not actually defined, if something other than a spinlock_t or raw_spinlock_t is passed to a spin_lock function the compiler will complain.

The `__spin_lock()`[10] acts like the original spin_lock function, and the `_spin_lock()`[11] evaluates to the mutex, `rt_spin_lock()`, defined in `kernel/rtmutex. c`.

6 Priority Inheritance

The most devastating latency that can occur in an RTOS is unbounded priority inversion. As mentioned earlier, priority inversion occurs when a high priority thread must wait on a lower priority thread before it can run. This usually occurs when a resource is shared between high and low priority threads, and the high priority thread needs to take the resource while the low priority thread holds it. Priority inversion is natural and can not be completely prevented. What we must prevent is unbounded priority inversion. That is when a high priority thread can wait an undetermined amount of time for the lower priority thread to release the resource.

The classic example of unbounded priority inversion takes place with three threads, each having a different priority. As shown in Figure 1, the CPU usage of three

---

[9]also called raw_spin_lock.

[10]prefixed with two underscores.

[11]prefixed with one underscore.

threads, A (highest priority) B (middle priority), and C (lowest priority). Thread C starts off and holds some lock, then thread A wakes up and preempts thread C. Thread A tries to take a resource that is held by thread C and is blocked. Thread C continues but is later preempted by thread B before thread C could release the resource that thread A is blocked on. Thread B is of higher priority than thread C but lower priority than thread A. By preempting thread C it is in essence preempting thread A. Since we have no idea how long thread B will run, thread A is now blocked for an undetermined amount of time. This is what is known as **unbounded priority inversion**.

There are different approaches to preventing unbounded priority inversion. One way is just simply by design. That is to carefully control what resources are shared as well as what threads can run at certain times. This is usually only feasible by small systems that can be completely audited for misbehaving threads. The Linux kernel is far too big and complex for this approach. Another approach is priority ceiling [3], where each resource (lock) knows the highest priority thread that will acquire it. When a thread takes a resource, it is temporarily boosted to the priority of that resource while it holds the resource. This prevents any other thread that might acquire that resource from preempting this thread. Since pretty much any resource or lock in the Linux kernel may be taken by any thread, you might as well just keep preemption off while a resource is held. This would include sleeping locks (mutexes) as well.

What the RT patch implements is **Priority Inheritance** (PI). This approach scales well with large projects, although it is usually criticized that the algorithms to implement PI are too complex and error prone. PI algorithms have matured and it is easier to audit the PI algorithm than the entire kernel. The basic idea of PI is that when a thread blocks on a resource that is owned by a lower priority thread, the lower priority thread inherits the priority of the blocked thread. This way the lower priority thread can not be preempted by threads that are of lower priority than the blocked thread. Figure 2 shows the same situation as Figure 1 but this time with PI implemented.

The priority inheritance algorithm used by the RT patch is indeed complex, but it has been vigorously tested and used in production environments. For a detailed explanation of the design of the PI algorithm used not only by the RT patch but also by the current mainline kernel PI

futex, see the kernel source documentation [9].

7 What's Good for RT is Good for the Kernel

The problem with prior implementations of RT getting accepted into mainline Linux, was that too much was done independently from mainline development, or was focused strictly on the niche RT market. Large intrusive changes were made throughout the kernel in ways that were not acceptable by most of the kernel maintainers. Finally, one day Ingo Molnar noticed the benefits of RT and started developing a small project that would incorporate RT methods into the Linux kernel. Molnar, being a kernel maintainer, could look at RT from a more general point of view, and not just from that of a niche market. His approach was not to force the Linux kernel into the RT world, but rather to bring the beneficial parts of the RT world to Linux.

One of the largest problems back then (2004) was this nasty lock that was all over the Linux kernel. This lock is know as the Big Kernel Lock (BKL). The BKL was introduced into Linux as the first attempt to bring Linux to the multiprocessor environment. The BLK would protect concurrent accesses of critical sections from threads running on separate CPUs. The BKL was just one big lock to protect everything. Since then, spin_locks have been introduced to separate non related critical sections. But there are still large portions of Linux code that is still protected by the BKL.

The BKL was a spin_lock that was large and intrusive, and would cause large latencies. Not only was it a spinning lock, but it was also recursive.[12] It also had the non-intuitive characteristic that a process holding a BKL is allowed to voluntarily sleep. Spin locks could cause systems to lock up if a thread were to sleep while holding one, but the BKL was special, in that the scheduler magically released the BKL, and would reacquire the lock when the thread resumes.

Molnar developed a way to preempt this lock [6]. He changed the BKL from a spin lock into a mutex. To preserve the same semantics, the BKL would be released if the thread voluntarily scheduled, but not when the thread was preempted. Allowing threads to be preempted while holding the BKL greatly reduced the scheduling latencies of the kernel.

---

[12] Allowed the same owner to take it again while holding it, as long as it released the lock the same number of times.
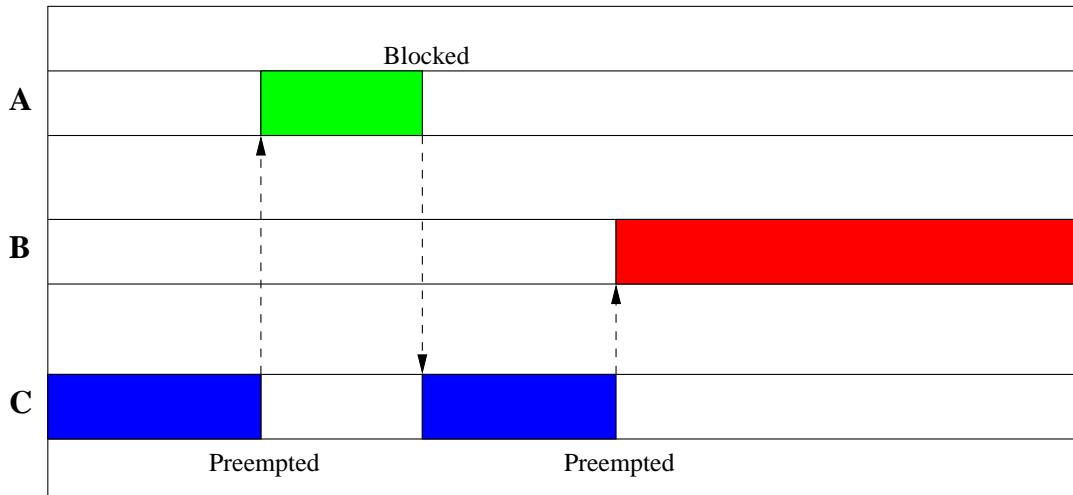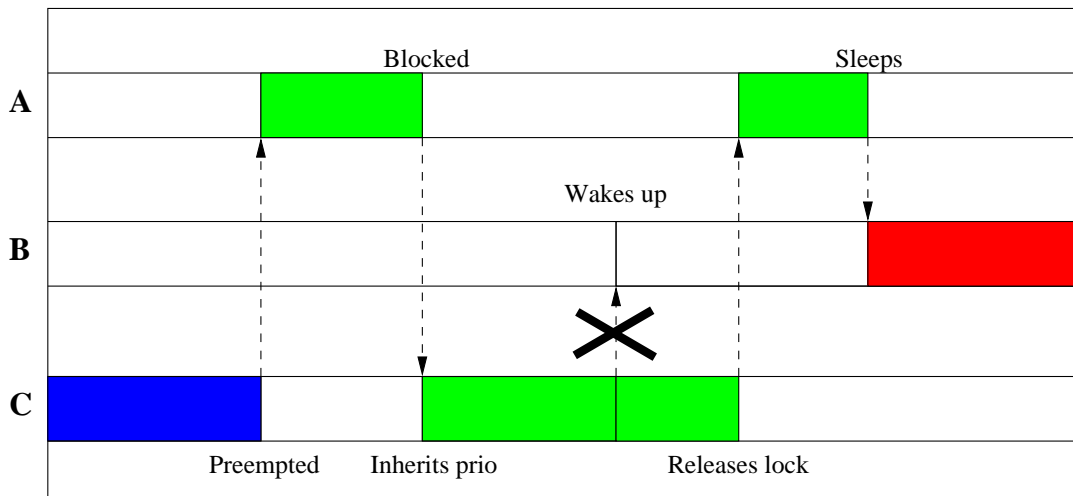
Figure 1: Priority Inversion

Figure 2: Priority Inheritance

7.1 Death of the Semaphore

Semaphores are powerful primitives that allow one or more threads access to a resource. The funny thing is, they were seldom used for multiple accesses. Most of the time they are used for maintaining mutual exclusion to a resource or to coordinate two or more threads. For coordination, one thread would down (lock) a semaphore and then wait on it.[13] Some other thread, after completing some task, would up (unlock) the semaphore to let the waiting threads know the task was completed.

The latter can be replaced by a `completion` [4]. A `completion` is a primitive that has the purpose of notifying one or more threads when another thread has completed an event. The use of semaphores for this purpose is now obsolete and should be replaced with `completions`.

There was nothing available in the mainline kernel to replace the semaphore for a single mutual exclusion lock. The ability of a semaphore to handle the case of multiple threads accessing a resource produces an overhead when only acting as a mutex. Most of the time this overhead is unnecessary. Molnar implemented a new primitive for the kernel called **mutex**. The simpler design of the mutex makes it much cleaner and slightly faster than a

---

[13]usually the semaphore was just created as locked.

semaphore. A mutex only allows a single thread access to a critical section, so it is simpler to implement and faster than a semaphore. So the addition of the mutex to the mainline kernel was a benefit for all. But Molnar had another motive for implementing the mutex (which most people could have guessed).

Semaphores, with the property of allowing more than one thread into a critical section, have no concept of an owner. Indeed, one thread may even release a semaphore while another thread acquires it.[14] A semaphore is never bound to a thread. A mutex, on the other hand, always has a one-to-one relationship with a thread when locked. The mutex may be owned by one, and only one, thread at a time. This is key to the RT kernel, as it is one of the requirements of the PI algorithm. That is, a lock may have one, and only one, owner. This ensures that a priority inheritance chain stays a single path, and does not branch with multiple lock owners needing to inherit a priority of a waiting thread.

7.2 The Path to RT

Every mainline release includes more code from the RT patch. Some of the changes are simply clean-ups and fixes for race conditions. An RTOS on a single CPU system exposes race conditions much easier than an 8 way SMP system. The race windows are larger due to the preemptive scheduling model. Although race conditions are easier to exposed on an RT kernel, those same race conditions still exist in the mainline kernel running on an SMP system. There have been arguments that some of the fixed race conditions have never been reported, so they most likely have never occurred. More likely, these race conditions have crashed some server somewhere, but since the race window is small, the crash is hard to reproduce. The crash would have been considered an anomaly and ignored. With the RT patch exposing rare bugs and the RT patch maintainers sending in fixes, the mainline kernel has become more stable with fewer of these seemingly unreproducable crashes.

In addition to clean-ups and bug fixes, several major features have been already incorporated into the mainline kernel.

- `gettimeofday` (2.6.18) – John Stultz's redesign of the time infrastructure.

---
[14]in the case of completions.

- `Generic IRQS` (2.6.18) – Ingo Molnar's consolidation of the IRQ code to unify all the architectures.

- `High Resolution Timers Pt. 1 (2.6.16)` – Thomas Gleixner's separation of timers from timeouts.

- `High Resolution Timers Pt. 2 (2.6.21)` – Thomas Gleixner's change to the timer resolution. Now clock event resolution is no longer bound to jiffies, but to the underlining hardware itself.

One of the key features for incorporating the RT patch into mainline is PI. Linus Torvalds has previously stated that he would never allow PI to be incorporated into Linux. The PI algorithm used by the RT patch can also be used by user applications. Linux implements a user mutex that can create, acquire, and release the mutex lock completely in user space. This type of mutex is known as a **futex** (fast mutex) [1]. The futex only enters the kernel on contention. RT user applications require that the futex also implements PI and this is best done within the kernel. Torvalds allowed the PI algorithm to be incorporated into Linux (2.6.18), but only for the use with futexes.

Fortunately, the core PI algorithm that made it into the mainline Linux kernel is the same algorithm that is used by the RT patch itself. This is key to getting the rest of the RT patch upstream and also brings the RT patch closer to mainline, and facilitates the RT patch maintenance.

8 RT is the Gun to Shoot Yourself With

The RT patch is all about determinism and being able to have full control of the kernel. But, like being root, the more power you give the user the more likely they will destroy themselves with it. A common mistake for novice RT application developers is writing code like the following:

```
x = 0;
/* let another thread set x */
while (!x)
    sched_yield();
```

Running the above with the highest priority on an RTOS, and wondering why the system suddenly freezes.

This paper is not about user applications and an RT kernel, but the focus is on developers working within the kernel and needing to understand the consequences of their code when someone configures in full RT.

8.1 yield() is Deadly

As with the above user land code, the kernel has a similar devil called `yield()`. Before using this, make sure you truly understand what it is that you are doing. There are only a few locations in the kernel that have legitimate uses of `yield()`. Remember in the RT kernel, even interrupts may be starved by some device driver thread looping on a `yield()`. `yield()` is usually related to something that can also be accomplished by implementing a `completion`.

Any kind of spinning loop is dangerous in an RTOS. Similar to using a busy loop spin_lock without disabling interrupts, and having that same lock used in an interrupt context handler, a spinning loop might starve the thread that will stop the loop. The following code that tries to prevent a reverse lock deadlock is no longer safe with the RT patch:

```
retry:
  spin_lock(A);
  if (!spin_trylock(B)) {
    spin_unlock(A);
    goto retry;
  }
```

Note: The code in `fs/jbd/commit.c` has such a situation.

8.2 rwlocks are Evil

Rwlocks are a favorite with many kernel developers. But there are consequences with using them. The rwlocks (for those that don't already know), allow multiple readers into a critical section and only one writer. A writer is only allowed in when no readers are accessing that area. Note, that rwlocks which are also implemented as busy loops on the current mainline kernel, are now sleeping mutexes in the RT kernel.[15]

Any type of read/write lock needs to be careful, since readers can starve out a writer, or writers can starve

out the readers. But read/write locks are even more of a problem in the RT kernel. As explained in Section 6, PI is used to prevent unbounded priority inversion. Read/write locks do not have a concept of ownership. Multiple threads can read a critical section at the same time, and if a high priority writer were to need access, it would not be able to boost all the readers at that moment. The RT kernel is also about determinism, and known measurable latencies. The time a writer must wait, even if it were possible to boost all readers, would be the time the read lock is held multiplied by all the readers that currently have that lock.[16]

Presently, to solve this issue in the RT kernel, the rwlocks are not simply converted into a sleeping lock like spin_locks are. Read_locks are transformed into a recursive mutex so that two different threads can **not** enter a read section at the same time. But read_locks still remain recursive locks, meaning that the same thread can acquire the same read_lock multiple times as long as it releases the lock the same number of times it acquires it. So in the RT kernel, even read_locks are serialized among each other. RT is about predictable results, over performance. This is one of those cases that the overall performance of the system may suffer a little to keep guaranteed performance high.

8.3 Read Write seqlocks are Mischievous

As with rwlocks, read/write seqlocks can also cause a headache. These are not converted in the RT kernel. So it is even more important to understand the usage of these locks. Recently, the RT developers came across a regression with the introduction of some of the new time code that added more instances of the `xtime_lock`. This lock uses read/write seqlocks to protect it. The way the read/write seqlocks work, is that the read side enters a loop starting with `read_seqlock()` and ending with `read_sequnlock()`. If no writes occurred between the two, the `read_sequnlock()` returns zero, otherwise it returns non-zero. If something other than zero is returned by the `read_seqlock()`, the loop continues and the read is performed again.

The issue with the read/write seqlocks is that you can have multiple writes occur during the read seqlock. If the design of the seqlocks is not carefully thought out, you could starve the read lock. The situation with the

---

[15]mainline kernel also has sleeping rwlocks implemented with `up_read` and `down_read`.

[16]Some may currently be sleeping.

`xtime_lock` was even present in the 2.6.21-rc series. The `xtime_lock` should only be written to on one CPU, but a change that was made in the -rc series that allowed the `xtime_lock` to be written to on any CPU. Thus, one CPU could be reading the `xtime_lock` but all the other CPUs could be queueing up to write to it. Thus the latency of the read_seqlock is not only the time the read_seqlock is held, but also the sum of all the write_seqlocks that are run on each CPU. A poorly designed read/write seqlock implementation could even repeat the write_seqlocks for the CPUs. That is to say, while CPU 1 is doing the read_seqlock loop, CPU 2 does a write_seqlock, then CPU 3 does a write_seqlock, then CPU 4 does a write_seqlock, and by this time, CPU 2 is doing another write_seqlock. All along, leaving CPU 1 continually spinning on that read_seqlock.

8.4 Interrupt Handlers Are No Longer Supreme

Another gotcha for drivers that are running on the RT kernel is the assumption that the interrupt handler will occur when interrupts are enabled. As described in section threaded-interrupts, the interrupt service routines are now carried out with threads. This includes handlers that are run as soft IRQs (e.g., net-tx and net-rx). If a driver for some reason needs a service to go off periodically so that the device won't lockup, it can not rely on an interrupt or soft IRQ to go off at a reasonable time. There may be cases that the RT setup will have a thread at a higher priority than all the interrupt handlers. It is likely that this thread will run for a long period of time, and thus, starve out all interrupts.[17] If it is necessary for a device driver to periodically tickle the device then it must create its own kernel thread and put it up at the highest priority available.

9 Who Uses RT?

The RT patch has not only been around for development but there are also many users of it, and that number is constantly growing.

The audio folks found out that the RT patch has significantly helped them in their recordings (`http://ccrma.stanford.edu/planetccrma/software`).

IBM, Red Hat and Raytheon are bringing the RT patch to the Department of Defense (DoD).

(`http://www-03.ibm.com/press/us/en/pressrelease/21033.wss`)

Financial institutions are expressing interest in using the RT kernel to ensure dependably consistent transaction times. This is increasingly important due to recently enacted trading regulations [8].

With the growing acceptance of the RT patch, it won't be long before the full patch is in the mainline kernel, and anyone can easily enjoy the enhancements that the RT patch brings to Linux.

10 RT Benchmarks

Darren V. Hart (`dvhltc@us.ibm.com`)

Determinism and latency are the key metrics used to discuss the suitability of a real-time operating system. IBM's Linux Technology Center has contributed several test cases and benchmarks which test these metrics in a number of ways. The results that follow are a small sampling that illustrates the features of the RT patch as well as the progress being made merging these features into the mainline Linux kernel. The tests were run on a 4 CPU Opteron system with a background load of `make -j8` 2.6.16 kernel build. Source for the tests used are linked to from the RT Community Wiki.[18] Full details of these results are available online [5].

10.1 gettimeofday() Latency

With their dependence on precise response times, real-time systems are prone to making numerous system calls to determine the current time. A deterministic implementation of `gettimeofday()` is critical. The `gtod_latency` test measures the difference between the time reported in pairs of consecutive `gettimeofday()` calls.

The scatter plots for 2.6.18 (Figure 3) and 2.6.18-rt7 (Figure 4) illustrate the reduced latency and improved determinism the RT patch brings to the `gettimeofday()` system call. The mainline kernel experiences a 208 us maximum latency, with a number of samples well above 20 us. Contrast that with the 17 us maximum latency of the 2.6.18-rt7 kernel (with the vast majority of samples under 10 us).

---

[17]besides the timer interrupt.

[18]`http://rt.wiki.kernel.org/index.php/IBM_Test_Cases`

10.2 Periodic Scheduling

Real-time systems often create high priority periodic threads. These threads perform a very small amount of work that must be performed at precise intervals. The results from the `sched_latency` measure the scheduling latency of a high priority periodic thread, with a 5 ms period.

Prior to the high resolution timer (hrtimers) work, timer latencies were only good to about three times the period of the periodic timer tick period (about 3ms with HZ=1000). This level of resolution makes accurate scheduling of periodic threads impractical since a task needing to be scheduled even a single microsecond after the timer tick would have to wait until the next tick, as illustrated in the latency histogram for 2.6.16 (Figure 5).[19] With the hrtimers patch included, the RT kernel demonstrates low microsecond accuracy (Figure 6), with a max scheduling latency of 25 us. The mainline 2.6.21 kernel has incorporated the hrtimers patch.

10.3 Asynchronous Event Handling

As discussed in Section 2, real-time systems depend on deterministic response times to asynchronous events. `async_handler` measures the latency of waking a thread waiting on an event. The events are generated using POSIX conditional variables.

Without the RT patch, the 2.6.20 kernel experiences a wide range of latencies while attempting to wake the event handler (Figure 7), with a standard deviation of 3.69 us. 2.6.20-rt8 improves on the mainline results, reducing the standard deviation to 1.16 us (Figure 8). While there is still some work to be done to reduce the maximum latency, the RT patch has greatly improved the deterministic behavior of asynchronous event handling.

References

[1] Futex. `http://en.wikipedia.org/wiki/Futex`.

[2] Preemptible kernel patch makes it into linux kernel v2.5.4-pre6. `http://www.linuxdevices.com/news/NS3989618385.html`.

[3] Priority ceiling protocol. `http://en.wikipedia.org/wiki/Priority_ceiling_protocol`.

[4] Jonathan Corbet. Driver porting: completion events. `http://lwn.net/Articles/23993/`.

[5] Darren V. Hart. Ols 2007 - real-time linux latency comparisons. `http://www.kernel.org/pub/linux/kernel/people/dvhart/ols2007`.

[6] Ingo Molnar. `http://lwn.net/Articles/102216/`.

[7] Ingo Molnar. Rt patch. `http://people.redhat.com/mingo/realtime-preempt`.

[8] "The Trade News". Reg nms is driving broker-dealer investment in speed and storage technology. `http://www.thetradenews.com/regulation-compliance/compliance/671`.

[9] Steven Rostedt. Rt mutex design. `Documentation/rt-mutex-design.txt`.

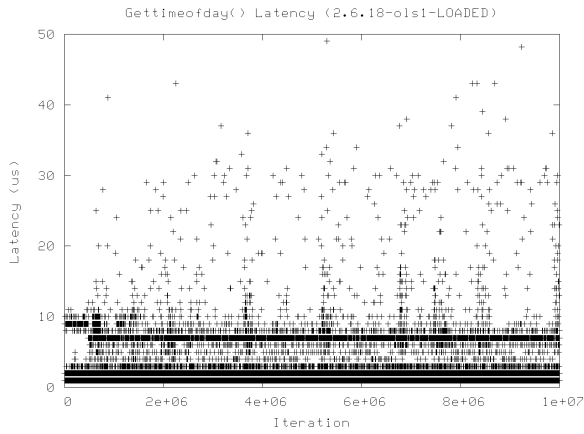[19]2.6.16 was used as later mainline kernels were unable to complete the test.

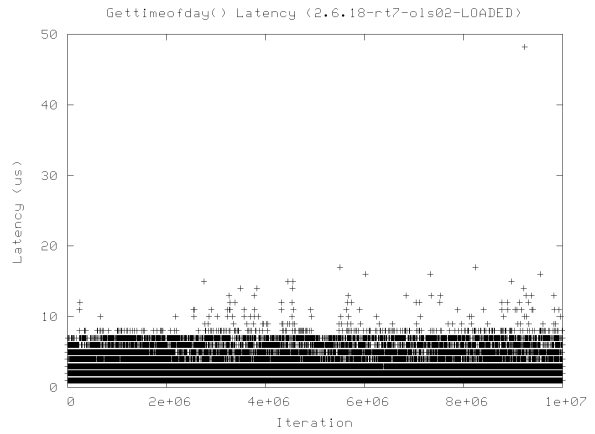Figure 3: 2.6.18 gtod_latency scatter plot
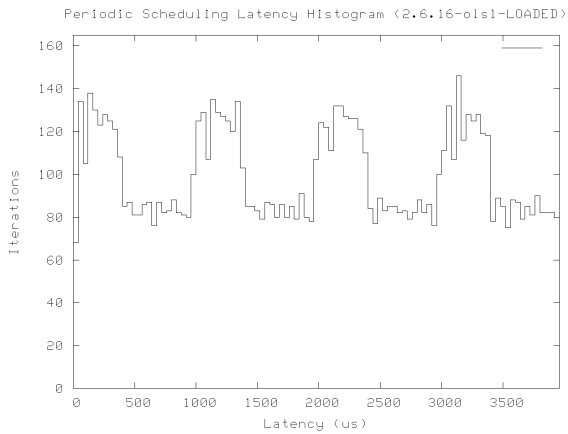
Figure 4: 2.6.18-rt7 gtod_latency scatter plot

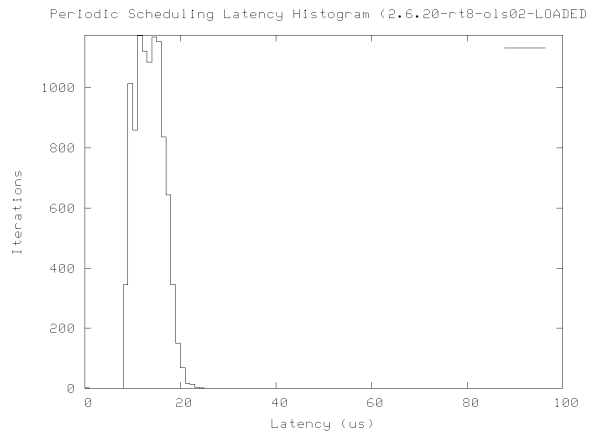Figure 5: 2.6.16 sched_latency histogram

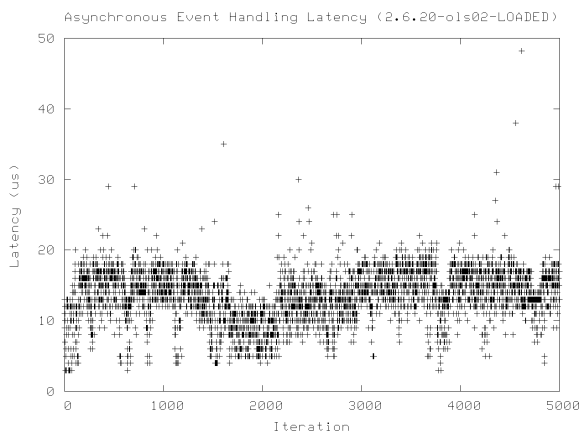Figure 6: 2.6.20-rt8 sched_latency histogram
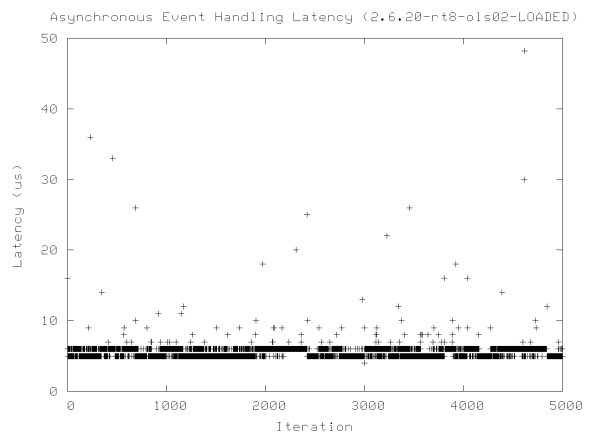
Figure 7: 2.6.20 async_handler scatter plot

Figure 8: 2.6.20-rt8 async_handler scatter plot

lguest: Implementing the little Linux hypervisor

Rusty Russell

IBM OzLabs

`rusty@rustcorp.com.au`

Abstract

Lguest is a small x86 32-bit Linux hypervisor for running Linux under Linux, and demonstrating the paravirtualization abilities in Linux since 2.6.20. At around 5,000 lines of code including utilities, it also serves as an excellent springboard for mastering the theory and practice of x86 virtualization.

This talk will cover the philosophy of lguest and then dive into the implementation details as they stand at this point in time. Operating System experience is required, but x86 knowledge isn't. By the time the talk is finished, you should have a good grounding in the range of implementation issues facing all virtualization technologies on Intel, such as Xen and KVM. You should also be inspired to create your own hypervisor, using your own pets as the logo.

1 Introduction

Around last year's OLS I was having discussions with various technical people about Linux support for paravirtualization, and Xen in particular. (Paravirtualization is where the guest knows it's being run under a hypervisor, and changes its behaviour).

I wanted the Linux kernel to support Xen, without wedding the Linux kernel to its interface: it seemed to me that now Xen showed that Free Software virtualization wasn't hard, we'd see other virtualization technologies worth supporting. There was already VMWare's proposed VMI standard, for example, but that wasn't a proven ABI either.

The result was *paravirt_ops*. This is a single structure which encapsulates all the sensitive instructions which a hypervisor might want to override. This was very similar to the VMI proposal by Zach Amsden, but some of the functions which Xen or VMI wanted were non-obvious to me.

So, I decided to write a trivial, self-contained Linux-on-Linux hypervisor. It would live in the Linux kernel source, run the same kernel for guests as for host, and be as simple as possible. This would serve as a third testbed for `paravirt_ops`.

2 Post-rationale for lguest

There turned out to be other benefits to writing such a hypervisor.

- It turned out to be around 5,000 lines, including the 1,000 lines of userspace code. This means it is small enough to be read and understood by kernel coders.

- It provides a complete in-tree example of how to use `paravirt_ops`.

- It provides a simple way to demonstrate the effects of a new Linux paravirtualization feature: you only need to patch one place to show the new feature and how to use it.

- It provides a testbed for new ideas. As other hypervisors rush to "productize" and nail down APIs and ABIs, lguest can be changed from kernel to kernel. Remember, lguest only promises to run the matching guests and host (i.e., no ABI).

My final point is more social than technical. I said that Xen had shown that Free Software paravirtualization was possible, but there was also some concern that its lead and "buzz" risked sucking up the groundwater from under other Free hypervisors: why start your own when Xen is so far ahead? Yet this field desperately needs more independent implementations and experimentation. Creating a tiny hackable hypervisor seemed to be the best way to encourage that.

As it turned out, I needn't have worried too much. The KVM project came along while I was polishing my patches, and slid straight into the kernel. KVM uses a similar "linux-centric" approach to lguest. But on the bright side, writing lguest taught me far more than I ever thought I'd know about the horribly warty x86 architecture.

3 Comparing With Other Hypervisors

As you'd expect from its simplicity, lguest has fewer features than any other hypervisor you're likely to have heard of. It doesn't currently support SMP guests, suspend and resume, or 64-bit. Glauber de Oliveira Costa and Steven Rostedt are hacking on lguest64 furiously, and suspend and resume are high on the TODO list.

Lguest only runs matching host and guest kernels. Other hypervisors aim to run different Operating Systems as guests. Some also do full virtualization, where unmodified OSs can be guests, but both KVM and Xen require newer chips with virtualization features to do this.

Lguest is slower than other hypervisors, though not always noticeably so: it depends on workload.

On the other hand, lguest is currently 5,004 lines for a total of 2,009 semicolons. (Note that the documentation patch adds another 3,157 lines of comments.) This includes the 983 lines (408 semicolons) of userspace code.

The code size of KVM and Xen are hard to compare to this: both have *features*, such as 64-bit support. Xen includes IA-64 support, and KVM includes all of qemu (yet doesn't use most of it).

Nonetheless it is instructive to note that KVM 19 is 274,630 lines for a total of 99,595 semicolons. Xen unstable (14854:039daabebad5) is 753,242 lines and 233,995 semicolons (the 53,620 lines of python don't carry their weight in semicolons properly, however).

4 Lguest Code: A Whirlwind Tour

lguest consists of five parts:

1. The guest `paravirt_ops` implementation,

2. The launcher which creates and supplies external I/O for the guest,

3. The switcher which flips the CPU between host and guest,

4. The host module (lg.ko) which sets up the switcher and handles the kernel side of things for the launcher, and

5. The awesome documentation which spans the code.[1]

4.1 Guest Code

How does the kernel know it's an lguest guest? The first code the x86 kernel runs are is `startup_32` in `head.S`. This tests if paging is already enabled: if it is, we know we're under some kind of hypervisor. We end up trying all the registered `paravirt_probe` functions, and end up in the one in `drivers/lguest/lguest.c`. Here's the guest, file-by-file:

drivers/lguest/lguest.c Guests know that they can't do privileged operations such as disable interrupts: they have to ask the host to do such things via *hypercalls*. This file consists of all the replacements for such low-level native hardware operations: we replace the `struct paravirt_ops` pointers with these.

drivers/lguest/lguest_asm.S The guest needs several assembler routines for low-level things and placing them all in `lguest.c` was a little ugly.

drivers/lguest/lguest_bus.c Lguest guests use a very simple bus for devices. It's a simple array of device descriptors contained just above the top of normal memory. The lguest bus is 80% tedious boilerplate code.

drivers/char/hvc_lguest.c A trivial console driver: we use lguest's DMA mechanism to send bytes out, and register a DMA buffer to receive bytes in. It is assumed to be present and available from the very beginning of boot.

drivers/block/lguest_blk.c A simple block driver which appears as `/dev/lgba`, `lgbb`, `lgbc`, etc. The mechanism is simple: we place the information about the request in the device page,

---

[1]Documentation was awesome at time of this writing. It may have rotted by time of reading.

then use the SEND_DMA hypercall (containing the data for a write, or an empty "ping" DMA for a read).

drivers/net/lguest_net.c This is very simple, a virtual network driver. The only trick is that it can talk directly to multiple other recipients (i.e., other guests on the same network). It can also be used with only the host on the network.

4.2 Launcher Code

The launcher sits in the Documentation/lguest directory: as lguest has no ABI, it needs to live in the kernel tree with the code. It is a simple program which lays out the "physical" memory for the new guest by mapping the kernel image and the virtual devices, then reads repeatedly from /dev/lguest to run the guest. The read returns when a signal is received or the guest sends DMA out to the launcher.

The only trick: the Makefile links it statically at a high address, so it will be clear of the guest memory region. It means that each guest cannot have more than 2.5G of memory on a normally configured host.

4.3 Switcher Code

Compiled as part of the "lg.ko" module, this is the code which sits at 0xFFC00000 to do the low-level guest-host switch. It is as simple as it can be made, but it's naturally very specific to x86.

4.4 Host Module: lg.ko

It is important to that lguest be "just another" Linux kernel module. Being able to simply insert a module and start a new guest provides a "low commitment" path to virtualization. Not only is this consistent with lguest's experimental aims, but it has potential to open new scenarios to apply virtualization.

drivers/lguest/lguest_user.c This contains all the /dev/lguest code, whereby the userspace launcher controls and communicates with the guest. For example, the first write will tell us the memory size, pagetable, entry point, and kernel address offset. A read will run the guest until a

signal is pending (-EINTR), or the guest does a DMA out to the launcher. Writes are also used to get a DMA buffer registered by the guest, and to send the guest an interrupt.

drivers/lguest/io.c The I/O mechanism in lguest is simple yet flexible, allowing the guest to talk to the launcher program or directly to another guest. It uses familiar concepts of DMA and interrupts, plus some neat code stolen from futexes.

drivers/lguest/core.c This contains run_ guest() which actually calls into the host↔guest switcher and analyzes the return, such as determining if the guest wants the host to do something. This file also contains useful helper routines, and a couple of non-obvious setup and teardown pieces which were implemented after days of debugging pain.

drivers/lguest/hypercalls.c Just as userspace programs request kernel operations via a system call, the guest requests host operations through a "hypercall." As you'd expect, this code is basically one big switch statement.

drivers/lguest/segments.c The x86 architecture has segments, which involve a table of descriptors which can be used to do funky things with virtual address interpretation. The segment handling code consists of simple sanity checks.

drivers/lguest/page_tables.c The guest provides a virtual-to-physical mapping, but the host can neither trust it nor use it: we verify and convert it here to point the hardware to the actual guest pages when running the guest. This technique is referred to as *shadow pagetables*.

drivers/lguest/interrupts_and_traps.c This file deals with Guest interrupts and traps. There are three classes of interrupts:

1. Real hardware interrupts which occur while we're running the guest,

2. Interrupts for virtual devices attached to the guest, and

3. Traps and faults from the guest.

Real hardware interrupts must be delivered to the host, not the guest. Virtual interrupts must be delivered to the guest, but we make them look just like real hardware would deliver them. Traps from

the guest can be set up to go directly back into the guest, but sometimes the host wants to see them first, so we also have a way of "reflecting" them into the guest as if they had been delivered to it directly.

4.5 The Documentation

The documentation is in seven parts, as outlined in `drivers/lguest/README`. It uses a simple script in `Documentation/lguest` to output interwoven code and comments in literate programming style. It took me two weeks to write (although it did lead to many cleanups along the way). Currently the results take up about 120 pages, so it is appropriately described throughout as a heroic journey. From the README file:

Our Quest is in seven parts:

Preparation: In which our potential hero is flown quickly over the landscape for a taste of its scope. Suitable for the armchair coders and other such persons of faint constitution.

Guest: Where we encounter the first tantalising wisps of code, and come to understand the details of the life of a Guest kernel.

Drivers: Whereby the Guest finds its voice and become useful, and our understanding of the Guest is completed.

Launcher: Where we trace back to the creation of the Guest, and thus begin our understanding of the Host.

Host: Where we master the Host code, through a long and tortuous journey. Indeed, it is here that our hero is tested in the Bit of Despair.

Switcher: Where our understanding of the intertwined nature of Guests and Hosts is completed.

Mastery: Where our fully fledged hero grapples with the Great Question: "What next?"

5 Benchmarks

I wrote a simple extensible GPL'd benchmark program called virtbench.[2] It's a little primitive at the moment,

---

[2] `http://ozlabs.org/~rusty/virtbench`

but it's designed to guide optimization efforts for hypervisor authors. Here are the current results for a native run on a UP host with 512M of RAM and the same configuration running under lguest (on the same Host, with 3G of RAM). Note that these results are continually improving, and are obsolete by the time you read them.

Test Name	Native	Lguest	Factor
Context switch via pipe	2413 ns	6200 ns	2.6
One Copy-on-Write fault	3555 ns	9822 ns	2.8
Exec client once	302 us	776 us	2.6
One fork/exit/ wait	120 us	407 us	3.7
One int-0x80 syscall	269 ns	266 ns	1.0
One syscall via libc	127 ns	276 ns	2.2
Two PTE updates	1802 ns	6042 ns	3.4
256KB read from disk	33333 us	41725 us	1.3
One disk read	113 us	185 us	1.6
Inter-guest ping-pong	53850 ns	149795 ns	2.8
Inter-guest 4MB TCP	16352 us	334437 us	20
Inter-guest 4MB sendfile	10906 us	309791 us	28
Kernel Compile	10m39	13m48s	1.3

Table 1: Virtbench and kernel compile times

6 Future Work

There is an infinite amount of future work to be done. It includes:

1. More work on the I/O model.

2. More optimizations generally.

3. `NO_HZ` support.

4. Framebuffer support.

5. 64-bit support.

6. SMP guest support.

7. A better web page.

7   Conclusion

Lguest has shown that writing a hypervisor for Linux isn't difficult, and that even a minimal hypervisor can have reasonable performance. It remains to be seen how useful lguest will be, but my hope is that it will become a testing ground for Linux virtualization technologies, a useful basis for niche hypervisor applications, and an excellent way for coders to get their feet wet when starting to explore Linux virtualization.

ext4 online defragmentation

Takashi Sato

NEC Software Tohoku, Ltd.

`sho@tnes.nec.co.jp`

Abstract

ext4 greatly extends the filesystem size to 1024PB compared to 16TB in ext3, and it is capable of storing many huge files. Previous study has shown that fragmentation can cause performance degradation for such large filesystems, and it is important to allocate blocks as contiguous as possible to improve I/O performance.

In this paper, the design and implementation of an online defragmentation extension for ext4 is proposed. It is designed to solve three types of fragmentation, namely single file fragmentation, relevant file fragmentation, and free space fragmentation. This paper reports its design, implementation, and performance measurement results.

1 Introduction

When a filesystem has been used for a long time, disk blocks used to store file data are separated into discontiguous areas (fragments). This can be caused by concurrent writes from multiple processes or by the kernel not being able to find a contiguous area of free blocks large enough to store the data.

If a file is broken up into many fragments, file read performance can suffer greatly, due to the large number of disk seeks required to read the data. Ard Biesheuvel has shown the effect of fragmentation on file system performance [1]. In his paper, file read performance decreases as the amount of free disk space becomes small because of fragmentation. This occurs regardless of the filesystem type being used.

ext3 is currently used as the standard filesystem in Linux, and ext4 is under development within the Linux community as the next generation filesystem. ext4 greatly extends the filesystem size to 1024PB compared to 16TB in ext3, and it is capable of storing many huge files. Therefore it is important to allocate blocks as contiguous as possible to improve the I/O performance.

In this paper, an online defragmentation feature for ext4 is proposed, which can solve the fragmentation problem on a mounted filesystem.

Section 2 describes various features in current Linux filesystems to reduce fragmentation, and shows how much fragmentation occurs when multiple processes writes simultaneously to disk. Section 3 explains the design of the proposed online defragmentation method and Section 4 describes its implementation. Section 5 shows the performance benchmark results. Existing work on defragmentation is shown in Section 6. Section 7 contains the summary and future work.

2 Fragmentations in Filesystem

Filesystem on Linux have the following features to reduce occurrence of fragments when writing a file.

- Delayed allocation

  The decision of where to allocate the block on the disk is delayed until just before when the data is stored to disk in order to allocate blocks as contiguously as possible (Figure 1). This feature is already implemented in XFS and is under discussion for ext4.

- Block reservation

  Contiguous blocks are reserved right after the last allocated block, in order to use them for successive block allocation as shown in Figure 2. This feature is already implemented in ext4.

Although ext4 and XFS already have these features implemented to reduce fragmentation, writing multiple files in parallel still causes fragmentation and decrease in file read performance. Figure 3 shows the difference in file read performance between the following two cases:
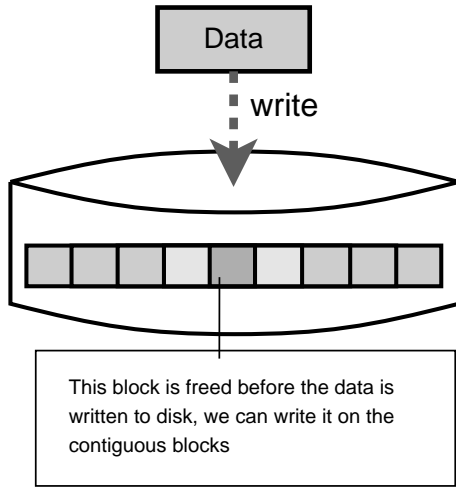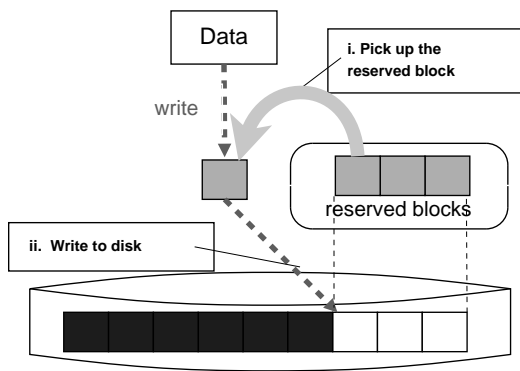
Figure 1: Delayed allocation

Figure 2: Block reservation

- Create the file by writing 32 1GB files sequentially

- Create the file by writing 32 1GB files from 32 threads concurrently

File read performance decreases about 15% for ext3, and 16.5% for XFS.

Currently ext4 does not have any defragmentation feature, so fragmentation will not be resolved until the file is removed. Online defragmentation for ext4 is necessary to solve this problem.

3 Design of Online Defragmentation

3.1 Types of Fragmentation

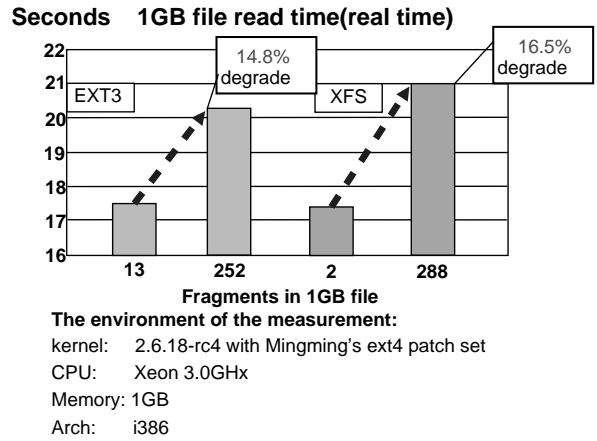Fragmentation could be classified into the following three types [2].

Figure 3: The influence of fragments

- Single file fragmentation
  Single file fragmentation occurs when a single file is broken into multiple pieces. This decreases the performance of accessing a single file.

- Relevant file fragmentation
  Relevant file fragmentation occurs when relevant files, which are often accessed together by applications are allocated separately on the filesystem. This decreases the performance of applications which access many small files.

- Free space fragmentation
  Free space fragmentation occurs when the filesystem has many small free areas and there is no large free area consisting of contiguous blocks. This will make the other two types of fragmentation more likely to occur.

Online defragmentation should be able to solve these three types of fragmentation.

3.2 Single File Fragmentation

Single file fragmentation could be solved by moving file data to contiguous free blocks as shown in Figure 4.

Defragmentation for a single file is done in the following steps (Figure 5).

1. Create a temporary inode.

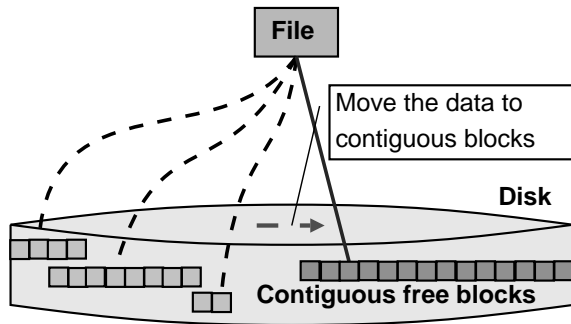2. Allocate contiguous blocks to temporary file as in Figure 5(a).

Figure 4: Defragment for a single file

3. Move file data for each page as described in Figure 5(b). The following sequence of events should be committed to the journal atomically.

   (a) Read data from original file to memory page.

   (b) Swap the blocks between the original inode and the temporary inode.

   (c) Write data in memory page to new block.
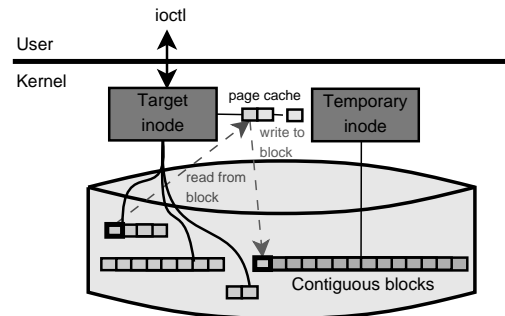
3.3 Relevant File Fragmentation

Relevant file fragmentation could be solved by moving the files under the specified directory close together with the block containing the directory data as presented in Figure 6.

Defragmentation for relevant files is done in the following steps (Figure 7).

1. The defragmentation program asks the kernel for the physical block number of the first block of the specified directory. The kernel retrieves the physical block number of the extent which is pointed by the inode of the directory and returns that value as in Figure 7(a).

2. For each file under the directory, create a temporary inode.

3. The defragmentation command passes the physical block number obtained in step 1 to the kernel. The kernel searches for the nearest free block after the given block number, and allocates that block to the temporary inode as described in Figure 7(b).

4. Move file data for each page, by using the same method as for single file defragmentation.
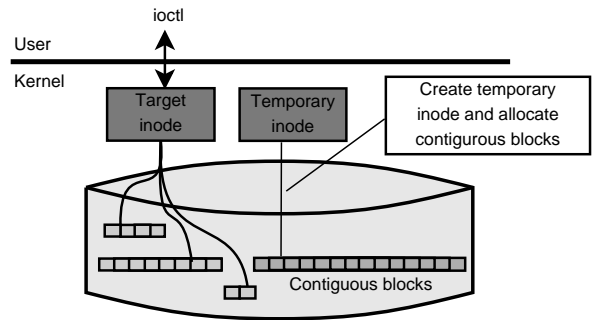
(a) Allocate contiguous blocks

(b) Replace data blocks

Figure 5: Resolving single file fragmentation
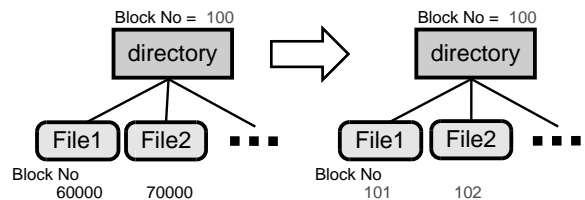
Figure 6: Defragment for the relevant files

5. Repeat steps 2 to 4 for all files under the specified directory.

3.4 Free Space Fragmentation

If the filesystem has insufficient contiguous free blocks, the other files are moved to make sufficient space to allocate contiguous blocks for the target file. Free space defragmentation is done in the following steps (Figure 8).

1. Find out the block group number to which the target file belongs. This could be calculated by using the number of inodes in a group and the inode number as below.

$$groupNumber = \frac{inodeNumber}{iNodesPerGroup}$$

(a) Get block mapping
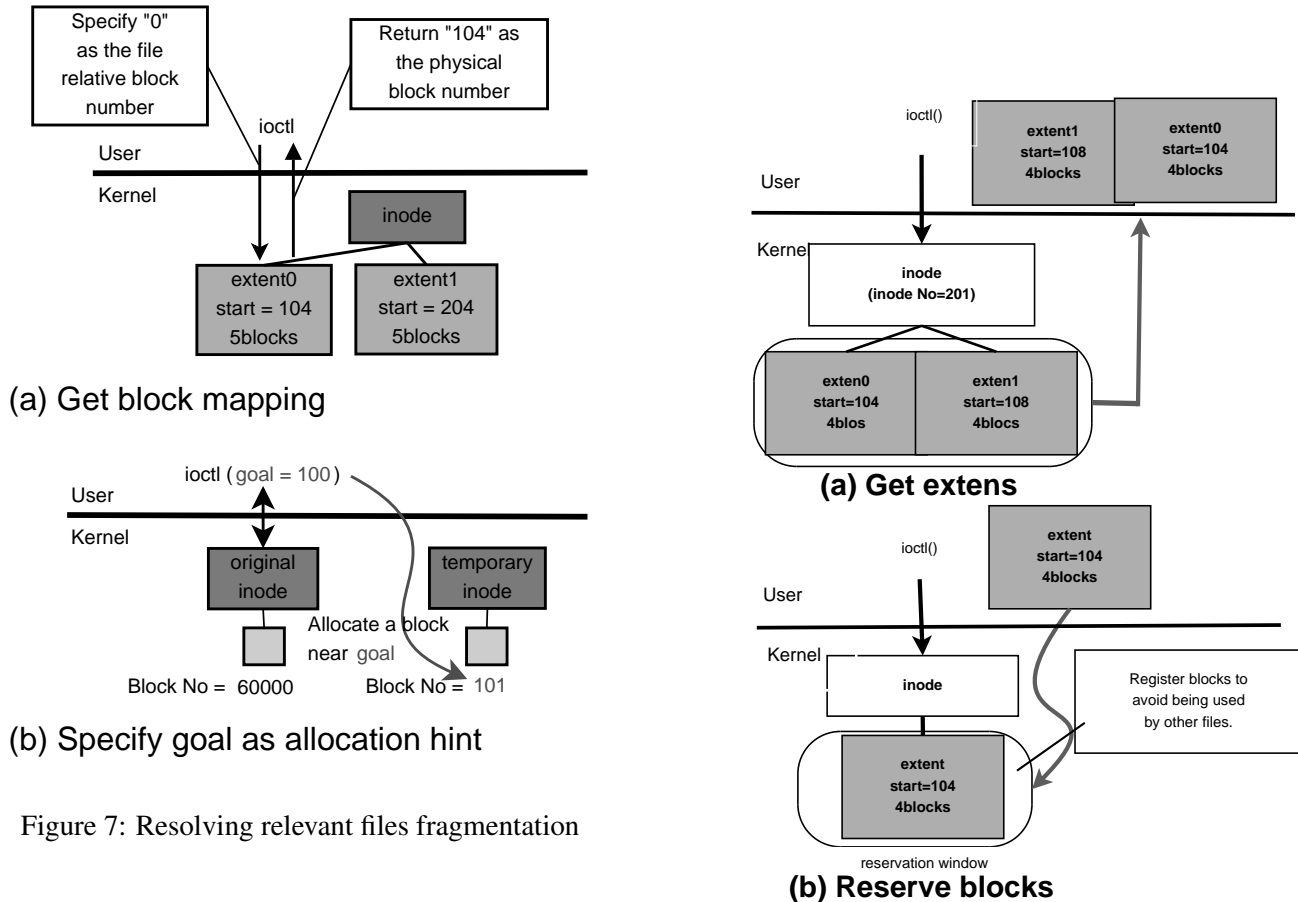

(b) Specify goal as allocation hint

Figure 7: Resolving relevant files fragmentation

2. Get the extent information of all the files belonging to the target group and choose the combination of files to evict for allocating contiguous blocks large enough to hold the target file as in Figure 8(a).

3. Reserve the chosen blocks so that it will not be used by other processes. This is achieved by the kernel registering the specified blocks to the reservation window for each file as presented in Figure 8(b).

4. Move the data in the files chosen in step 2 to other block groups. The destination block group could be either specified by the defragmentation command or use the block group which is farthest away from the target group as in Figure 8(c). The file data is moved using the same method as for single file defragmentation.

5. Move the data in the target file to the blocks which have been just freed. The kernel allocates the freed blocks to the temporary inode and moves the file data using the same method as for single file defragmentation as in Figure 8(d).
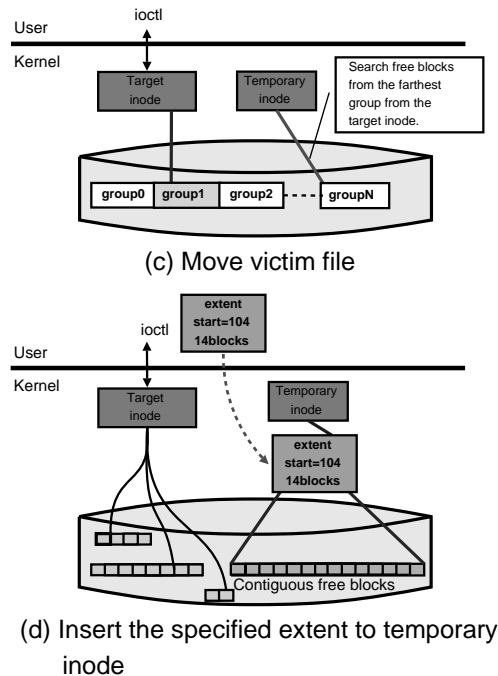

(a) Get extens

(b) Reserve blocks

(c) Move victim file

(d) Insert the specified extent to temporary inode

Figure 8: Resolving free space fragmentation

4 Implementation

The following five ioctls provide the kernel function for online defragmentation. These ioctls use Alex Tomas's patch [3] to implement multi-block allocation feature to search and allocate contiguous free blocks. Alex's multi-block allocation and the proposed five ioctls are explained below.

The ioctls described in Section 4.3 to 4.6 are still under development, and have not been tested. Also, moving file data for each page is currently not registered to the journal atomically. This will be fixed by registering the sequence of procedures for replacing blocks to the same transaction.

4.1 Multi-block allocation

Alex's multi-block allocation patch [3] introduces a bitmap called "buddy bitmap" to manage the contiguous blocks for each group in an inode which is pointed from the on-memory superblock (Figure 9).

The buddy bitmap is divided into areas to keep bitmap of contiguous free blocks with length of powers of two, e.g. bitmap for 2 contiguous free blocks, bitmap for 4 contiguous free blocks, etc. The kernel can quickly find contiguous free blocks of the desired size by using the buddy bitmap. For example, when the kernel requests 14 contiguous free blocks, it searches in the area for 8, 4, and 2 contiguous free blocks and allocates 14 contiguous free blocks on disk.
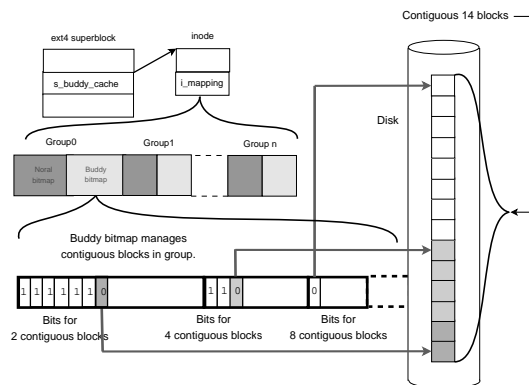

Figure 9: Multi-block allocation

4.2 Moving Target File Data (EXT4_IOC_DEFRAG)

This ioctl moves file data from fragmented blocks to the destination. It uses the structures `ext4_ext_defrag_data` and `ext4_extent_data` shown in Figure 10 for input parameters. The behavior of this ioctl differs depending on the type of defragmentation it is used for.

- Single file fragmentation
  Both the start offset of defragmentation (*start_offset*) and the target size (*defrag_size*) need to be specified. When both *goal* and *ext.len* are set to 0, the kernel searches for contiguous free blocks starting from the first block of the block group which the target file's inode belongs to, and replaces the target file's block with the free blocks.

- Relevant file fragmentation
  In addition to *start_offset* and *defrag_size*, *goal* should be set to the physical block number of the first block in the specified directory. When *goal* is set to a positive number, the kernel searches for free blocks starting from the specified block, and replaces the target file's blocks with the nearest contiguous free blocks.

- Free space fragmentation
  In addition to *start_offset* and *defrag_size*, *ext* should be set to the information of the extent which represents the new contiguous area for replacement. When *ext.len* is set to a positive number, the kernel replaces the target file's blocks with the blocks in *ext*

Since I am still designing the implementation of the following four ioctls, I haven't tested them yet.

4.3 Get Group Information (EXT4_IOC_GROUP_ INFO)

This ioctl gets the group information. There is no input parameter. The kernel gets the number of blocks in a group (*s_blocks_per_group*) and the number of the inodes (*s_inodes_per_group*) from ext4 memory superblock (*ext4_sb_info*) and returns them with the structure (*ext4_group_data*) in Figure 11.

blocks_per_group is not used in the current implementation, but it is returned in case of future use.

4.4 Get Extent Information (EXT4_IOC_GET_EXTENTS_INO)

This ioctl gets the extent information. The structure shown in Figure 12 is used for both input and output. The command sets the first extent number of the target extent to *entries*. The kernel sets the number of returned extents upon return. Since there might be very large number of extents in a file, the kernel returns extents up to *max_entries* specified as input. If the number of extents is larger than *max_entries*, the command can get all extents by calling this ioctl multiple times with updated *entries*.

4.5 Block Reservation (EXT4_IOC_RESERVE_BLOCK)

This ioctl is used to reserve the blocks, so that it will not be used by other processes. The blocks to reserve is specified in the extent information (*ext4_extent_data*). The kernel reserves the blocks using the existing block reservation function.

4.6 Move Victim File (EXT4_IOC_MOVE_VICTIM)

This ioctl moves a file from the block group where it belongs to other block groups. *ext4_extents_info* structure is used for input parameter. *ino* stores the inode number of the target file, and *entries* holds the number of extents specified in *ext*. *ext* points to the array of extents which specify the areas which should be moved, and *goal* contains the physical block number of the destination.

The kernel searches for a contiguous free area starting from the block specified by *goal*, and replaces the target file's blocks with the nearest contiguous free blocks. If *goal* is 0, it searches from the first block of the block group which is farthest away from the block group which contains the target file's inode.

5 Performance Measurement Results

5.1 Single File Fragmentation

Fifty fragmented 1GB files were created. Read performance was measured before and after defragmentation. Performance measurement result is shown in Table 1. In this case, defragmentation resulted in 25% improvement in file read performance.

	Fragments	I/O performance (Sec)
Before defrag	12175	618.3
After defrag	800	460.6

Table 1: The measurement result of the defragmentation for a single file

5.2 Relevant File Fragmentation

The Linux kernel source code for 2.6.19-rc6 (20,000 files) was extracted on disk. Time required to run find command on the kernel source was measured before and after defragmentation. Performance measurement result is shown in Table 2. In this case, defragmentation resulted in 29% performance improvement.

	I/O performance (Sec)
Before defrag	42.2
After defrag	30.0

Table 2: The measurement result of the defragmentation for relevant files

Defragmentation for free space fragmentation is still under development. Performance will be measured once it has been completed.

6 Related Work

Jan Kara has proposed an online defragmentation enhancement for ext3. In his patch [4], a new ioctl to exchange the blocks in the original file with newly allocated contiguous blocks is proposed. The implementation is in experimental status and still lacks features such as searching contiguous blocks. It neither supports defragmentation for relevant files nor defragmentation for free space fragmentation.

During the discussion in linux-ext4 mailing list, there were many comments that there should be a common interface across all filesystems for features such as free block search and block exchange. And in his mail [5], a special filesystem to access filesystem meta-data was proposed. For instance, the extent information could be accessed by reading the file data/extents. Also the blocks used to store file data could be exchanged to contiguous blocks by writing the inode number of the temporary file which holds the newly allocated contiguous

```
struct ext4_ext_defrag_data {
        // The offset of the starting point (input)
        ext4_fsblk_t start_offset;
        // The target size for the defragmentation (input)
        ext4_fsblk_t defrag_size;
        // The physical block number of the starting
        // point for searching contiguous free blocks (input)
        ext4_fsblk_t goal
        // The extent information of the destination.
        struct ext4_extent_data ext;
}

struct ext4_extent_data {
        // The first logical block number
        ext4_fsblk_t block;
        // The first physical block number
        ext4_fsblk_t start;
        // The number of blocks in this extent
        int        len;
}
```

Figure 10: Structures used for ioctl (EXT4_IOC_DEFRAG)

```
struct ext4_group_data {
        // The number of inodes in a group
        int inodes_per_group
        // The number of blocks in a group
        int blocks_per_group;
}
```

Figure 11: Structures used for ioctl (EXT4_IOC_GROUP_INFO)

```
struct ext4_extents_info {
        //inode number (input)
        unsigned long long ino;
        //The max number of extents which can be held (input)
        int   max_entries;
        //The first extent number of the target extents (input)
        //The number of the returned extents (output)
        int   entries;
        //The array of extents (output)
        //NUM is the same value as max_entries
        struct ext4_extent_data ext[NUM];
}
```

Figure 12: Structures used for ioctl (EXT4_IOC_GET_EXTENTS_INO)

blocks to data/reloc. During the discussion, two demerits were found. One is that common procedures across all filesystems are very few. The other is that there are a lot of overheads for encoding informations in both userspace and the kernel. Therefore this discussion has gone away. This discussion is for unifying interface and is not for the implementation for the online defragmentation which is explained by this paper.

7 Conclusion

Online defragmentation for ext4 has been proposed and implemented. Performance measurement has shown that defragmentation for a single file can improve read performance by 25% on a fragmented 1GB file. For relevant file fragmentation, defragmentation resulted in 29% performance improvement for accessing all file in the Linux source tree.

Online defragmentation is a promising feature to improve performance on a large filesystems such as ext4. I will continue development on the features which have not been completed yet.

There are also work to be done in the following areas:

- Decrease performance penalty on running processes
  Since it is possible for defragmentation to purge data on the page cache which other processes might reference later, defragmentation may decrease performance of running processes. Using fadvice to alleviate the performance penalty may be one idea.

- Automate defragmentation
  To reduce system administrator's effort, it is necessary to automate defragmentation. This can be realized by the following procedure.

  1. The kernel notifies the occurrence of fragments to user space.

  2. The user space daemon catches the notification and executes the defragmentation command.

References

[1] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert, "The Effects of Filesystem Fragmentation," in *Proceedings of the Linux Symposium*, Ottawa, 2006, Vol. 1, pp. 193–208, http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdf.

[2] "File system fragmentation." http://en.wikipedia.org/wiki/File_system_fragmentation.

[3] Alex Tomas. "[RFC] delayed allocation, mballoc, etc." http://marc.info/?l=linux-ext4&m=116493228301966&w=2.

[4] Jan Kara. "[RFC] Ext3 online defrag," http://marc.info/?l=linux-fsdevel&m=116160640814410&w=2.

[5] Jan Kara. "[RFC] Defragmentation interface," http://marc.info/?l=linux-ext4&m=116247851712898&w=2.

The Hiker Project: An Application Framework for Mobile Linux Devices

David "Lefty" Schlesinger
ACCESS Systems Americas, Inc.
`lefty@{hikerproject.org,access-company.com}`

Abstract

The characteristics of mobile devices are typically an order of magnitude different than desktop systems: CPUs run at megahertz, not gigahertz; memory comes in megabytes, not gigabytes; screen sizes are small and input methods are constrained; however, there are billions of mobile devices sold each year, as opposed to millions of desktop systems. Creating a third-party developer ecosystem for such devices requires that fragmentation be reduced, which in turn demands high-quality solutions to the common problems faced by applications on such devices. The Hiker Project's application framework components present such solutions in a number of key areas: application lifecycle management, task-to-task and task-to-user notifications for a variety of events, handling of structured data (such as appointments or contact information) and transfer of such data between devices, management of global preferences and settings, and general security issues. Together, these components comprise an "application framework," allowing the development of applications which can seamlessly and transparently interoperate and share information.

ACCESS Co., Ltd., originally developed the Hiker Project components for use in their "ACCESS Linux Platform" product, but recently released them under an open source license for the benefit and use of the open source community. This paper will describe, in detail, the components which make up the Hiker Project, discuss their use in a variety of real-world contexts, examine the proliferation of open source-based mobile devices and the tremendous opportunity for applications developers which this growth represents.

1 The Need for, and Benefits of, a Mobile Framework

The typical cell phone of today is generally equivalent in power, in various dimensions, to a desktop system of four or five years ago. Cell phones increasingly have CPUs running at close to half a gigahertz, dynamic RAM of 64 megabytes and beyond, and storage, typically semiconductor-based, in capacities of several gigabytes. Current MP3 players can have disk capacities well beyond what was typical on laptop systems only two or three years ago. The usage models for cell phones and other mobile devices, however, tend to be very different than that of desktop systems.

The bulk of applications-related development effort on open source-based systems has been primarily focused on servers and desktop devices which have a distinct usage model which does not adapt well to smaller, more constrained mobile devices. In order to effectively use open source-based systems to create a mobile device, and particularly to enable general applications development for such a device, a number of additional services are needed.

The term *application framework* means different things to different people. To some, it is the GUI toolkit that is used. To others, it is the window manager. Still others see it as the conventions for starting and running a process (e.g. Linux's `fork()` and `exec()` calls; C programs have an entry point called "main" that takes a couple of parameters and returns an integer; etc.).

When we use the term "application framework," we intend it to mean the set of libraries and utilities that support and control applications running on the platform. Why are additional libraries needed to control applications? Why not just use the same conventions as on a PC: choose programs off a start menu and explicitly end an application by choosing the "exit" menu item or closing its window?

The reason is the different "use model" on handheld devices. PCs have large screens that can accommodate many windows, full keyboards, and general pointing devices. A cell phone typically has a screen fewer than three inches diagonal (although pixel resolutions

can be equivalent to QVGA and higher), a complement of under twenty keys and a "five-way" pointing device or a "jog wheel." Based on experience refining Garnet (formerly known as "Palm OS"), we believe that there should be only one active window on a typical handheld device. As another example, when the user starts a new application, the previous application should automatically save its work and state, and then exit.

Similarly, the optimal conceptual organization of data is different on mobile devices. Rather than an "application/document" paradigm, using an explicit, tree-structured file system, a "task-oriented" approach, where data is inherent to the task, is more natural on these devices. Tasks, as opposed to applications, are short-lived and focused: making a call, reading an SMS message, creating a contact or appointment, etc. Occasionally, tasks will be longer-lived: browsing the web, or viewing media content. Another typical attribute of such devices is regular, unscheduled interruptions: a low-battery warning, an incoming email, a stock or news alert.

As well as the task of managing the lifecycle of programs (launching, running, stopping), the application framework must also help with distributing and installing applications. The conventions are simple: an application and all supporting files (images, data, localizations, etc.) are rolled up into a single file known as a "bundle." Bundles are convenient for users and third party developers, and allow software to be passed around and downloaded as an atomic object.

A third task of the application framework is to support common utility operations for applications, such as communication between applications, keeping track of which applications handle which kind of content, and dealing with unscheduled events like phone calls or instant messages.

The final task of the application framework is to implement a secure environment for software. That means an environment which resists attempts by one application to interfere with another (this hardening is called "application sandboxing"). The secure environment also supports security policies for permission-based access to resources. For example, part of a policy might be "only applications from the vendor are allowed network access." The security policy is implemented using a Linux security module.

The Hiker components then, broadly speaking, focus on several key areas:

- presenting a common view of applications to the user (*Application Manager* and *Bundle Manager*),

- communicating time-based and asynchronous events between applications or between an application and the user (*Notification Manager*, *Alarm Manager*, *Attention Manager*);

- interoperability of applications and sharing of information between them (*Exchange Manager*, *Global Settings*), and

- performing these functions in a secure context (*Security Policy Framework/Hiker Security Module*)

The *Abstract IPC* service is used by these components in order to simplify their implementation and allow them to generally take advantage of improved underlying mechanisms through a single gating set of APIs. Taken together, they provide mechanisms to allow seamless interoperability and sharing of information between suites of applications in a trusted environment.

These components are intended to offer several concrete benefits to the development community:

- They provide real reference implementations that can serve as the basis for application developers who want to write interoperable applications for mobile devices.

- They (hopefully) help to jump-start activities related to mobile devices in several key areas (e.g. security) by filling a number of current gaps in the services available to applications in that space.

- They can help to generally increase interest in application development for open source-based mobile devices

- They might encourage other companies to both participate in these projects and to contribute new projects as well.

1.1 Access to the Project, Licensing Terms, etc.

Full source code and other reference material for Hiker can be found at the Hiker Project web site, `http://www.hikerproject.org`.

Sources are currently available as tarballs, but we expect to be putting a source code repository up in the near future. Several mailing lists are available, and a TRAC-based bug reporting/wiki system will be in place shortly. A large amount of detailed API documentation, generated by Doxygen, is also available on the site.

Hiker is dual-licensed under the Mozilla Public License, v. 1.1 and the Library General Public License, v. 2, with the exception of the LSM-based portion of the Security Framework, which, as an in-kernel component, is licensed under GPL v. 2.

2 The Application Manager

The Application Manager controls application lifetime and execution and is the component that is responsible for maintaining the simple application task model that users expect on a phone. The application manager starts and stops applications as appropriate, ensures that the current running application controls the main display, maintains a list of running applications, places applications in the background, and prevents multiple launches of a single application.

The Application Manager is initiated at system start-up and provides the following services:

- Application launching mechanism and management of application lifecycle.

- Routing of launch requests to the existing instance (if the application is already running).

- Coordination of the "main UI app" – retires the current application when a new one is launched, and launches a default when needed.

- May also provide default behavior for certain important system events (i.e., hard key handling, clamshell open/close).

The Application Manager runs in its own process. Applications typically run in their own processes and control their own UI. This simple mapping of applications to processes provides a secure, stable model for application execution. To maximize utility on small screen form factors, the Application Manager will preserve the standard "Garnet OS" behavior of having one main UI application at a time, with that application having drawing control of the main panel of the screen (exclusive of status bars, etc.) When the user runs a new application, the system will generally ask the current one to exit (although there is a facility for applications which need to continue execution in the background). Also, as in legacy "Garnet OS" and desktop operating systems like Mac OS X, only a single instantiation of any given application at time will be running.

The Application Manager handles the high-level operations of launching applications, and provides a number of APIs to applications and to the system for access to these services. It maintains a list of currently running applications, and keeps track of which one is the "primary" or "main UI" application. This special status is used to coordinate the display and hiding of UI when the user moves between applications. Note that an application that is not the main UI application may still put up UI under exceptional circumstances, it is simply recommended that this be done only occasionally (for example to ask the user for a password), and that it be in the form of a modal dialog. The user's attention is generally focused on the main UI application, and so UI from background applications is often an interruption. It is expected that most applications will not run in background mode.

3 The Bundle Manager

The Bundle Manager combines a file format for distributing single files that contain applications and all their dependencies (application name, icon, localized resources, dependant libraries, etc.) along with functions for installing, removing, and enumerating application "bundles." The Bundle Manager takes care of allocation of per-application storage locations in writable storage in the file system, and providing access to localized resources contained within the bundle. Through the Bundle Manager enumeration mechanism, multiple application types are merged and can be launched through a common mechanism.

The Bundle Manager is the system component responsible for controlling how applications, and supplemental data for applications (libraries, resources, etc.), are
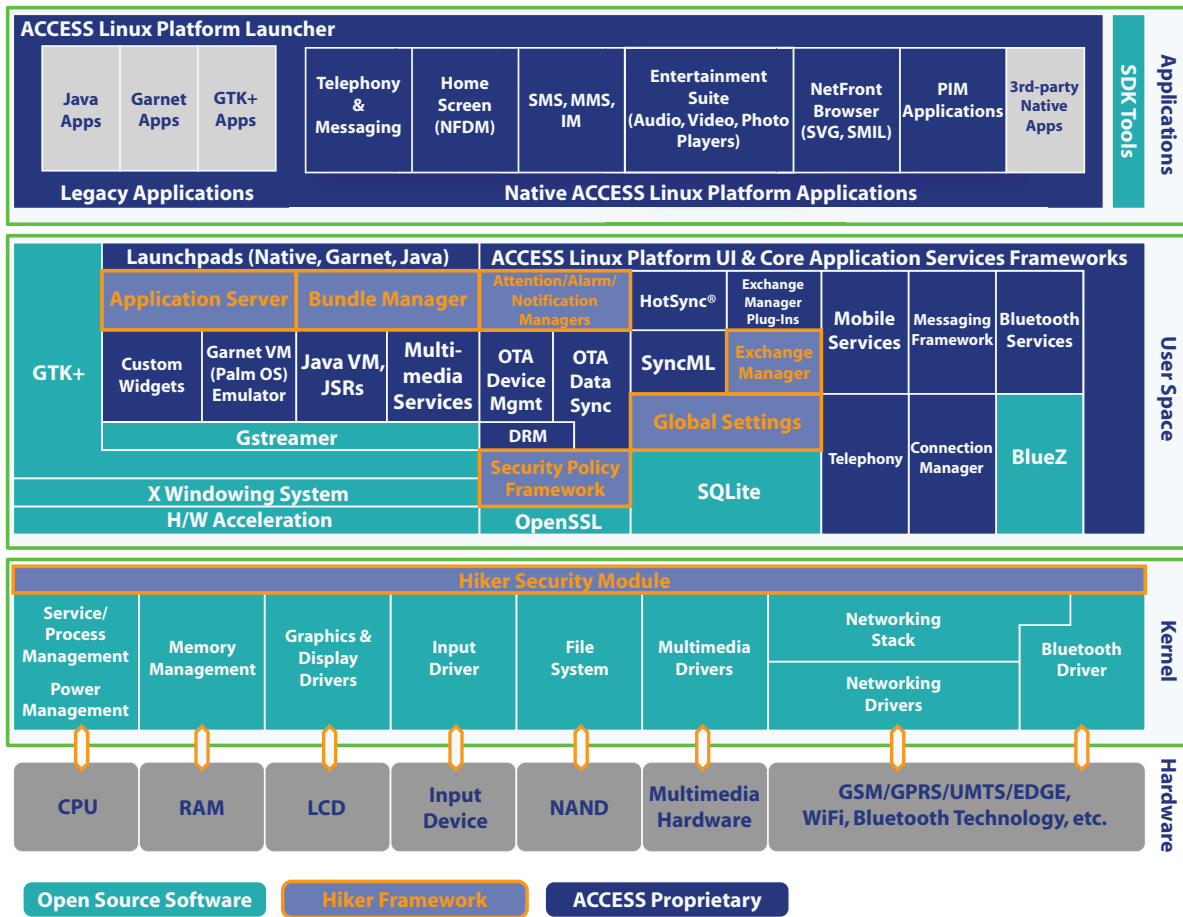
Figure 1: *The Hiker Components in the ACCESS Linux Platform*

loaded onto a system using the Application Manager and other framework components, manipulated, transmitted off of the system, and removed.

The Bundle Manager is a mid-level library and server which provides easy access to application resources for developers, as well as maintaining state about bundles present in the system.

The Bundle Manager is designed around the notion of "bundles" as concrete immutable lumps of information which are managed on a device, where each bundle can contain an arbitrary amount of data in a format appropriate to that bundle. Each bundle type is defined both in terms of how it is stored on the device, and as a "flattened" format suitable for transmission as a stream out of the device. (These formats may be the same, different, or overlap at various times).

The Bundle Manager is the channel through which all third-party applications are distributed and loaded on to

a device. The server component of the Bundle Manager is intended to be the only software on the device which has permission to access the bundle folder on the internal filesystem, requiring interaction with the Bundle Manager for installation or removal.

The Bundle Manager provides consistent mechanisms for retrieving resources, both localized and unlocalized (i.e., loading the files or other bundle contents, perhaps in a localized folder-name) from bundles.

4   The Notification Manager

Notification Manager provides a mechanism for sending programmatic notifications of unsolicited system events to applications. Applications can register to receive particular types of notifications that they are interested in. The Notification Manager can deliver notifications not only to currently running applications, but also to ap-

plications that are registered to receive them but are not currently running.

Notifications are general system level or user application level events like application installed/uninstalled, card inserted/removed, file system mounted/unmount, incoming call, clamshell opened/closed, time changed, locale changed, low power, or device going to sleep/waking up. The Notification Manager has a client/server architecture.

4.1 Notification Manager server

The Notification Manager server is a persistent thread in a separate system process which keeps track of all registered notifications and broadcasts notifications to registered clients. The Notification Manager server also communicates with the Package Manager and Application Server.

4.2 Notification Manager client library

Client processes call APIs in the Notification Manager client library to

1. register to receive notifications,

2. unregister previously registered notifications,

3. signal the completion of a notification, and

4. broadcast notifications.

The Notification Manager client library uses the Abstract IPC framework to communicate with the Notification Manager server.

4.3 What the Notification Manager is not

1. The Notification Manager should not be used for application specific or directed notifications like alarms or find.

2. The Notification Manager facilitates the sending and receiving of notifications but it does not itself broadcast notifications (individual component owners are responsible for broadcasting their own notifications).

5 The Alarm Manager

The Alarm Manager provides a mechanism to notify applications of real time alarm (i.e. time-based) events. Both currently running and non-running applications can receive events from the Alarm Manager. The Alarm Manager does not control presentation to the user—the action taken by an application in response to an alarm is defined by the application.

The Alarm Manager:

- Works with power management facilities to register the first timer;

- Calls the Application Manager to launch the applications for which an alarm is due;

- Supports multiple alarms per application; and

- Stores its alarm database in SQLite for persistence.

The Alarm Manager has no UI of its own; applications that need to bring an alarm to the user's attention must do this through the Attention Manager. The Alarm Manager doesn't provide reminder dialog boxes, and it doesn't play the alarm sound. Applications should use the Attention Manager to interact with the user.

6 The Attention Manager

The Attention Manager manages system events that need to be presented to the user, such as a low battery condition or an incoming call (rather than programmatic events delivered to other applications like the service provided by the Notification Manager). The Attention Manager uses a priority scheme to manage presentation of items needing attention to the user. The infrastructure used by applications and system services to ask for attentions and the storage of the currently pending list of events requiring the user's attention is separate from the actual presentation of the events to the user.

The Attention Manager is a standard facility by which applications can tell the user that something of significance has occurred. The Attention Manager is responsible only for being a nexus for such events and interacting with the user in regards to these events; it is not responsible for generating the events.

The Attention Manager provides both a single alert dialog and maintains a list of all "alert-like" events. Together these improve the user experience by first getting the user's attention when needed, then allowing the user to deal with the attention or dismiss for review later. By handling it this way, it is no longer necessary to click through a series of old alert dialogs. Often the user doesn't care about most of the missed appointments or phone calls—although he might care about a few of them. Without the Attention Manager, the user cannot selectively dismiss or follow up on the alert events but would instead have to deal with each alert dialog in turn.

Applications have complete control over the types and level of attention they can ask for.

Typical flow of an attention event:

- An application (e.g. the calendar of "Date Book") requests the Alarm Manager to awaken it at some time in the future.

- The Alarm Manager simply sends an event to an application when that future point in time is reached. The application can then post an event to the Attention Manager with the appropriate priority.

- The Attention Manager will present the appropriate alert dialog based on the event type and priority.

- The Attention Manager is designed solely to interact with the user when an event must be brought to the user's attention.

6.1 When the Attention Manager isn't appropriate

The Attention Manager is specifically designed for attempts to get attentions that can be effectively handled or suspended. The Attention Manager also doesn't attempt to replace error messages. Applications should use modal dialogs and other existing OS facilities to handle these cases.

The Attention Manager is also not intended to act as a "To Do" or "Tasks" application, nor act as a "universal in-box." Applications must make it clear that an item appearing in the Attention Manager is simply a reminder, and that dismissing it neither deletes nor cancels the item itself. That is, saying "OK" to an attention message

regarding an upcoming appointment does not delete the appointment, and dismissing an SMS reminder does not delete the SMS message from the SMS inbox.

The Attention Manager is not an event logger, nor an event history mechanism. It contains only the state of active attention events.

The Attention Manager is organized in order to meet specific design goals, which are:

- To separate UI from attention event logging mechanisms;

- To provide sufficient configurability so that a licensee may alter both the appearance and behavior of the posted events;

- To maintain persistent store of events that will survive a soft reset;

- To be responsive (need to be quick from alert to UI—prime example is incoming call); and

- To minimize memory usage (i.e. try to get as small of memory foot print as possible).

The server model currently utilized relies on an init script to start a small daemon. The daemon accepts IPC requests to post, update, query or delete events and maintain such event state in a database to provide persistent storage for the events. The daemon launches the attention UI application which will display the appropriate alert dialog. If the snooze action is chosen for an event (provided that a snooze action is associated with the event), the Attention UI application will call the Alarm manager to schedule a wakeup. The wakeup will be in the form of a launch or relaunch via the Application server. In this model, the attention status gadget is assumed to be polling the Attention Manager daemon for active event state and using that information in displaying status. If the attention status gadget is "clicked" on, it starts the Attention UI through the Application Server.

6.2 Features

The Attention Manager implements the following major components:

- attention events;

- an API library for posting, updating, deleting, and querying events;

- a server through which events are posted, retrieved and managed;

- a UI application that displays the alert dialogs;

- an event database and associated DML API; and

- a status gadget for the status bar.

7   Abstract IPC

The Abstract IPC service provides a lightweight, robust interface to a simple message-based IPC mechanism. The actual implementation can be layered on top of other IPC mechanisms. The current implementation is based on Unix sockets, but this mechanism can be layered on other IPC mechanisms if required. The current implementation has a peer-to-peer architecture that minimizes context switches, an important feature on some popular embedded architectures.

The Abstract IPC Service comprises an API for a simple interprocess communication (IPC) mechanism used by the framework components described in this paper.

The goals of this design include:

- Independence from underlying implementation mechanisms (e.g. pipes, sockets, D-BUS, etc.);

- A simple, easy to use send/receive message API;

- Support for marshalling/unmarshalling message data; and

- Minimization of context switches by sending messages directly between processes without passing through an intermediary process.

The IPC mechanism is based on a single server process exchanging messages with one or more clients. The server process creates a channel; clients connect to the channel and receive a connection pointer they can use to send/receive messages to the server. The format of the messages is completely up to the server and clients of the channel.

Communication can be synchronous or asynchronous. When done asynchronously, processes receive messages via a callback mechanism that works through the gLib main loop.

8   Security Policy Framework and Hiker Security Module

The Security Policy Framework (SPF) is the component which controls the security policy for the device. The actual policy used by the framework is created by a licensee and can be updated. Policy is flexible and separate from the mechanisms used to enforce it. The policy can express a wide (and extensible) range of policy attributes. Typical elements of a policy address use of file system resources, network resources, password restriction policies, access to network services, etc. Each policy is a combination of these attributes and is tied to a particular digital signature.

Applications are checked for a digital signature (including no signature or a self-signature) and an appropriate security policy is applied to the application. One of the policy decisions that can be made by the framework is whether the user should be consulted—this allows for end-users to control access to various types of data on the device and ensure that malicious applications will not access this data covertly. Other types of decisions are allow/deny which may be more appropriate for a carrier to use to protect access to network resources, etc.

The Hiker Security Module is a kernel level enforcement component that works in concert with the Security Policy Framework. The Security Module controls the actual access to files, devices and network resources. Because it is an in-kernel component, the Security Module is released under the GPL.

There must be some user control on who is allowed to connect to the user's device and request action from it. Security is mostly based on the user control at the following levels:

- There will be default handlers (implemented, for example, by in the box PIM applications) for a bunch of published standard services. In the event where a third party application would try to register a duplicate handler, and the first handler declared it wanted to be unique, the user would be alerted

and asked to arbitrate which application should be the installed handler. The user is the authority that tells the system which handler wins in case of conflict. This security works whichever handler installs first. In a model where all installed applications are signed and therefore trusted, we can rely on what the application do when they register.

- When there is a non-authenticated incoming connection, the user is asked to authorize the connection. Local is obviously initiated by the user and is always valid. IR is considered authenticated, as the user must explicitly direct his device to the initiator. Paired Bluetooth is authenticated by definition. SMS is authenticated by the mobile network. TCP may be considered authenticate if the source IP address figures in the table of trusted sources (although this may be discussed as it is easy to spoof the source IP). TCP may also be configured to require a challenge password before it accepts to read from the connection.

- Above the connection level authentication, handlers may also require permission from the user before they perform their action. This is handler specific. "get vCard" is obviously a very good candidate to user authorization.

9 The Exchange Manager

The Exchange Manager is a central broker to manage inter-application/inter-device communication. Requests to the Exchange Manager contain verbs ("get", "store", "play"), data (qualified by mime-type) as well as other parameters used to identify the specific item to be affected by the request. Use cases of the Exchange Manager include beaming a contact to another device, taking a picture using the camera from an MMS application, looking up a vCard based on caller ID, viewing an email attachment, etc.

The Exchange Manager is an extensible framework: new handlers can be created for new data types and actions as well as for new transports (e.g. IR, Local, Bluetooth, SMS, TCP/IP, etc.).

There is a need for any application to be able to perform different tasks (such as play, get, store, print, etc.) on several types of data. This component offers a simple, yet expandable, API that lets any application defer the actual handling of the data to whoever declared he was the handler for this action/type-of-data pair. In addition, the destination of the request (where the action will actually be performed) can be specified as the local device or a remote location. This opens up new universal possibilities such as directly send a vCard to someone through Internet while being on the phone with him.

This component also implements the legacy Palm OS "Garnet" Exchange Manager functionality (send data to a local or remote application). This simply corresponds to the action *store*

The Java VM also implements a similar functionality (JSR211—CHAPI). It is foreseen that the Java and native components will interoperate. In other words, a Java application will be able to send a vCard to a remote device through IR, as well as receive data from a local native application, for some examples.

9.1 Features

The purpose of this component is to enable an application or system component to request the system to perform some action on some data, without knowing who will carry and fulfill the request. In addition, the client can request that the action be performed either on the local device or on some specific other destination (mobile phone or desktop PC, etc.), using any available transport. This opens up new interesting scenarios that were not possible to do before.

An application that implements a service it wants to make available to others registers a handler to tell the system it can perform this specific action on this specific type of data. The handler may also specify that it will accept only local request, or that it will accept all requests. Each registered handler is valid for only one combination of action/data type. The action/data type is defined by a verb, and a MIME type (e.g. "store – text/vCard").

Transport modules are responsible to carry the request from the source to the destination device. When the request arrives at the destination (which may be the local machine), the transport will hand it to the Exchange Manager who will then invoke the corresponding action handler to do the actual work. A result may be sent back to the initiator, which means it is also possible to use this component to retrieve some data (not only send

it), or pass some data and get it back modified in some way. An obvious use case would be to use your phone to lookup a contact in your desktop PC address book, or retrieve a contact's photo and name given its phone number.

Handlers can be registered or unregistered at any time. A board game would register the "moveplayer" action (a handler to receive other players moves) when it is launched, and would send its own moves by requesting this same action from the other user device. The game would unregister the handler when it quits. Note that this example does not mean Exchange Manager could be used as a network media to handle more than peer to peer exchanges.

An application cannot verify the availability and identity of a handler and this would not make sense in the general usage (it is the goal of the Exchange Manager to be able to have an unknown handler execute a request). If an application needs to authenticate the handler, then this means it looks for a very precise handler it knows and in this case, it could use, for example, data encryption to ensure only the person with the right key can understand the request.

Transports are independent modules that can also be added or removed at any time. Typical transports would include Local, IrDA, Bluetooth, SMS, and TCP. Non-Hiker destination systems must run at least an Exchange Manager daemon and transports as well. Except for the handler invocation part, this should be the same code for any Linux platform (the sharedlib, daemon and transports only use standard Linux and GTK services). It would be easy, for example, to implement a new encrypted transport, should the need arise. The library provides a standard UI dialog to let the user select a transport and enter the relevant parameters. If the transport information is missing in the request, the Exchange Manager will itself pop this UI up at the time it needs the information. If the transport determines that the destination address is missing, it will also popup an address selection UI.

Verbs can be accompanied by parameters. Only a few parameters are common to all verbs (e.g. a human-readable description of the data that may be used to ask the user if he accepts what he is receiving). Parameters are passed as a tag/value pair (the value being int or string). Some are mandatory but most are optional and depend on the specific handler definition. Using pa-

rameters, the result of an action handler can be precisely customized to the client needs.

A non-exhaustive list of actions that will be defined include *store*, *get*, *print*, and *play*.

Parameters can be used by the action handler to find out how exactly it should perform its action, or by the transport module to get the destination address or other transport-specific information.

An Exchange Manager daemon is started at boot time (or at any other time). The daemon is used to listen for incoming action request for all transports. When a transport has an incoming request ready, the daemon dispatches it to the right action handler. The action handler then performs its duty, and returns the result back to the transport. The answer is then sent back to the originator.

When the originator is local, the user is never asked to accept the incoming data. When the originator is remote, whether a user confirmation is required will depend on the transport being used. For IrDA, it is assumed that the user implicitly accepts as he directs his device toward the emitter (it is still to be decided if we ignore the possibility that someone would be able to beam something to you without your consent while you have the device turned on). For Bluetooth, it depends whether the connection is paired or not. For SMS, the mobile network identifies the originator (in addition, there is no "connection" with SMS). For TCP, the transport configuration will tell whether the originator IP is authorized, and it will ask if not.

For handlers that are essentially data consumers and don't return anything (like "store"), it may make sense to let multiple handlers register for the same verb/data. For this reason, it is left to the handler to specify if it must be unique or not at registration time. In case it must be unique and someone tries to register a second handler for the same verb/data, the user would be notified and he would have the responsibility to arbitrate which of the two handlers should be the active one.

To maintain transfer compatibility with phones or legacy Garnet OS devices, the initiator may set a parameter to tell he wants to use Obex as the transport. In this case, the only verb allowed is "store." The transport plug-in will recognize this parameter and send the data using the OpenObex daemon. As well, a transport plug-in can also receive data from the OpenObex daemon. It would

treat that like an incoming connectionless "store" action on the received data.

For all other combinations of action/data type, the transport protocol is ours (or third party in case of third party transport). Obex protocol is handled by the OpenObex library. SMS NBS transport is handled in the SMS component. Some specific Bluetooth profiles (e.g. basic imaging profile) could also be handled by the Bluetooth transport in order to maximize compatibility with other kinds of devices.

An "Exchange request" is the only entity an application works with. It is an opaque structure that contains all the information characterizing a request: the verb, the parameters, the data reference and the destination. There are APIs to set and get all of them. The data itself can be specified in multiple ways: file descriptor (data will be read from this fd by the transport) or URL (URL is sent, and action handler will access data through the URL).

10   Global Settings

The Global Settings component provides a common API and storage for all applications and services to access user preferences (fonts, sizes, themes) and other application settings and configuration data. Global Settings are hierarchical and could be used, e.g., as the basis for OMA Device Management settings storage. The component is designed to support the security requirements of OMA Device Management. The storage for Global Settings uses the recently open-sourced libsqlfs project layered on SQLite. Global Settings provides generic, non-mobile specific, storage.

The Global Settings service provides functions for storing user preferences. It provides APIs for the setting and getting of software configurations (typically key-value pairs such as "font size: 12 points"). The settings keys may form a hierarchy like a directory tree, with each key comparable to a file or directory in a file system, e.g. `applications/datemanager/fontsize`. Indeed, Global Settings is actually implemented on top of an abstract POSIX file system: each key has a value and meta data such as a user id, a group id, and an access permission.

To accomplish this, Global Settings utilizes libsqlfs, a service component which creates the abstraction of a POSIX file system within an SQLite database file. An initial implementation was attempted using gconf. To support OMA Device Management requirements, we dropped the gconf design, which could not provide security over keys.

The Global Settings service has two parts: the daemon (aka server) and the client library. You could store settings by simply having a server process. However, to make it trivial for the clients to talk to the server, we also provide a client library which handles the IPC to the server. The client library is linked in with the client application.

The client library and the daemon communicate via Abstract IPC. The daemon does the actual I/O for the data; it links with the SQLite library and does the key content reads and writes on SQLite databases, with the tree hierarchy actually implemented using relational tables through SQL. SQLite provides no effective access control, so the daemon uses Unix file access control on the database file to exclude everyone else. The daemon also keeps track of the users and groups that are allowed to access certain keys, and enforces access control. The SQLite database files will be only readable and writable by the daemon process.

10.1   Data Model

1. We expose key/value pairs where values amount to the "contents" of the key and can be arbitrary data of arbitrary size.

2. Key descriptions (in multiple languages) are supported only by convention (see below).

3. We support settings that are user, group, or other readable/writable (or not); the user and group identities are based on these of the system and are granted by the system. The Global Settings service does not give special meanings to any specific group or user id.

The possible key names or key strings form a key space, which is similar to the space of file paths. The key strings are also called key paths. Each key path can be absolute or relative; absolute key or key paths must start with /, and relative key paths are relative to a "current directory" or a "cwd." There are APIs to get and to set the "cwd."

A "directory" is a key which contains other keys. A directory is similar to the interior nodes in the OMA Device Management Tree definition. To simplify our design, we disallow a directory key from having its own content. So if you add a child key to an existing key which has no value (or having the null value), that key becomes a directory and attempts to set the content for that key will fail in the future. Permissions for directories thus follow the semantics of POSIX file system.

A directory key can be created in two ways:

- A directory is created if a request is made to create a key that would be a child or a further descent of the directory

- Or a key can be created with an explicit API for this purpose

Key names follow a standard convention to allow grouping of similar attributes under the same part of the key hierarchy. We follow the GConf conventions as closely as possible for application preferences and system settings, with consideration for device-specific standards like /dm for OMA Device Management.

Some typical keys are represented below:

- `/dm` for OMA Device Management

- `/capabilities` for "is java installed," "is Garnet OS emulation installed," etc.

- `/packages/com.access.apps.app1` for preferences of "app1"

The Global Settings service is not meant for storage of security-sensitive data such as passwords or private keys. The data in the Global Settings are only protected by POSIX file permissions and are not encrypted or otherwise protected!

Global Settings implements the POSIX file permission (user, group and others, readable, writeable or executable) model on the keys and the key hierarchy. So a run time process has to acquire the appropriate user or group IDs to access a key the same way it access a file with the same POSIX permission bits in the host file system.

Otherwise, Global Settings places no meanings on the uids and gids, except the uid 0 which has permissions for everything, as root. The SUID bit is not honored and ignored for Global Settings.

The x bits in directories determine if the directories can be entered; the x bits in non-directory keys are ignored and have no meaning currently.

The uid and gids of keys are modifiable according to the following rules:

- The owner id cannot be changed except by the root.

- The group id cannot be changed except by the root or the owner.

Global Settings has reserved rooms for extra attributes for keys, currently not implemented but these could be used for access control lists or some other meta data but I treat such usage as the best to be avoided; if POSIX permissions provide all that's needed it is the best as it is simple and efficient.

With the above convention for keys, it becomes desirable for the applications or specific device (drivers) to create its key hierarchy during the installation time (or at the point of system image creation). Once created a hierarchy may be protected from access by applications other than the designated applications.

The initial key installation follow these steps:

- A group id is pre-assigned to a particular application (group). This assignment is to be guaranteed at run time by the Bundle Manager and system security.

- After an application is installed, it, or the Bundle Manager, will use a Global Settings tool or equivalent APIs to copy the key data from the default settings XML file into the Global Settings database and properly set the user and the group ids and the permissions for these keys.

- Later the Global Settings service ensures that the key hierarchy is accessible only to applications with the right group membership or the user id. The Unix file permission style permissions will be the only security mechanism protecting the key and key values; Global Settings does not provide encryption-based protection mechanism.

The Global Settings daemon is started at system boot time. Packages need to include their associated default preferences and these need to be installed in the correct fashion for the underlying preferences system.

Clients that wish to be informed of changes in settings can register a callback with the Notification Manager. The Global Settings daemon will send Notification Manager a string representing each key that it changes. Notification Manager will notify each application that has registered to be told about changes to that key. (Notification Manager will notify each application that registered for that key, or a prefix of that key. For example, an application that registered for `oma/apps` would be notified when any of the following keys changed: `oma/apps/calendar/fontsize`, `oma/apps/date/background-image`, or `oma/apps`.

Following the conventions of the Notification Manager, each key change notification has the "notification type" (or "notify type") of a string of the form `/alp/globalsettings/keychange/` + the key string. For example, a change of the key `oma/apps/calendar/fontsize` will invoke a broadcast notification call with the notification type `/alp/globalsettings/keychange/oma/apps/calendar/fontsize`.

The Notification Manager currently is responsible for monitoring the key changes in a directory tree; it implements this by checking if a changed key has, as a prefix, a substring which matches the representative key of a key subspace being monitored for changes by some application. For example, the previous key change example will invoke change notifications for applications which want to know key changes in the `/oma/apps/calendar/` key directory. The Notification Manager is the actual component which checks this and invoke the notification callback in client applications.

11   Conclusion

Mobile devices, and the applications running on them, represent a new frontier for open source developers. Worldwide cell phone sales are approaching one billion a year, with "smart phones" (that is, phones whose capabilities can be enhanced "post-platform," subsequent to their purchase by the consumer) showing the largest rate of growth. Open source-based operating systems have been of increasing interest as a foundation for work in the space.

By 2010, it's estimated that the number of "smart phones" sold per year will exceed the number of desktop systems. As many as half of those phones will be running Linux-based system, according to some analysts.

Linux and other open source software has taken on increased significance in this context. While a number of Linux-based phones have been shipped (primarily in China and other Pacific Rim geographies), few of them have been truly "open" systems, that is, the ability to develop software for and incorporate it onto such devices has been terribly limited.

Further, because the usage model for these devices is so different from the usual desktop-oriented paradigms, there are significant gaps in service-level functionality, particularly in areas such as abstraction of varying execution environments for the user, packaging of applications and related resources and metadata, information interchange between applications as well as between devices, and general approaches to security.

The security component, in particular, is key to the effort to foster a "third-party developer ecosystem" for mobile devices. Because of their very nature, as well as the regulatory structure within which they exist, certain operational characteristics of devices such as cell phones must be guaranteed. This is true even in the case of errant or malicious additional software being installed on the device.

A paper of this length can only provide a high-level overview of the components involved and their potential characteristics, usage, and interrelationships. A great deal more detailed information is available on the project web site.

Hiker is an attempt to address the most significant of the gaps discussed at the outset of this paper as a way of reducing fragmentation and encouraging participation in the mobile applications development context. The Hiker Project is intended to be a open, community effort. While ACCESS engineers are responsible for the initial implementation, the project is intended for the use of all developers interested in mobile applications development, and their participation in improving the framework is invited.

Getting maximum mileage out of tickless

Suresh Siddha   Venkatesh Pallipadi   Arjan Van De Ven
Intel Open Source Technology Center
{suresh.b.siddha|venkatesh.pallipadi|arjan.van.de.ven}@intel.com

Abstract

Now that the tickless(/dynticks) infrastructure is integrated into the base kernel, this paper talks about various add on changes that makes tickless kernels more effective.

Tickless kernel pose some hardware challenges that were primarily exposed by the requirement of continuously running per-CPU timer. We will discuss how this issue was resolved by using HPET in a new mode. Eliminating idle periodic ticks causes kernel process scheduler not do idle balance as frequently as it would do otherwise. We provide insight into how this tricky issue of saving power with minimal impact on performance, is resolved in tickless kernel.

We will also look at the kernel and user level daemons and drivers, polling for things with their own timers and its side effect on overall system idle time, with suggestions on how to make these daemons and drivers tickless-friendly.

1   Introduction

Traditionally, SMP Linux (just as most other operating systems) kernel uses a global timer (present in the chipset of the platform) to generate a periodic event for time keeping (for managing time of the day) and periodic local CPU timer event for managing per-CPU *timers*. *Timers* are basically events that the kernel wants to happen at a specified time in the future (either for its own use or on behalf of an application). An example of such a timer is the blinking cursor; the GUI wants the cursor to blink every 1.2 seconds so it wants to schedule an operation every 600 milliseconds into the future. In the pre tickless kernel, per-CPU "timers" were implemented using the periodic timer interrupt on each CPU. When the timer interrupt happens, the kernel checks if any of the scheduled events on that CPU are due and if so, it performs the operation associated with that event.

Old Linux kernels used a 100 Hz frequency (every 10 milliseconds) timer interrupt, newer Linux uses a 1000 Hz (1ms) or 250 Hz (4ms) timer interrupt. The frequency of the timer interrupt determines how fine grained you can schedule future events. Since the kernel knows this frequency, it also knows how to keep time: every 100 / 250 / 1000 timer interrupts another second has passed.

This periodic timer event is often called "the timer tick." This timer tick is nice and simple but has two severe downsides: First of all, this timer tick happens periodically irrespective of the processor state (idle Vs busy) and if the processor is idle, it has to wake up from its power saving sleep state every 1, 4, or 10 milliseconds, which costs quite a bit of energy (and hence battery life in laptops). Second of all, in a virtualization environment, if a system has 50 guests that each have a 1000 Hz timer tick, the total system will have a 50,000 effective ticks. This is not workable and highly limits the number of possible guests.

Tickless(/Dynticks) kernel is designed to solve these downsides. This paper will briefly look into the current tickless kernel implementation and then look in detail about the various add-ons the authors added to make the tickless kernel more effective. Section 2 will look into the current tickless kernel infrastructure. Section 3 will look at the impact of different timers in the kernel and its impact on tickless kernel. Section 4 will address the hardware challenges that were exposed by the requirement of per-CPU timer.

Section 5 will talk about the impact on process load balancing in tickless idle kernels and the problem resolution. Section 6 will look at the impact of the different timers (and polling) in user level daemons and drivers on the power savings aspect of the tickless kernel.

2 Tickless Kernel

Tickless kernel, as the name suggests, is a kernel without the regular timer tick. Tickless kernel depends on architecture independent generic clocksource and clockevent management framework, which got into the mainline in the recent past [7]. Generic timeofday and clocksource management framework moved lot of timekeeping code into architecture independent portion of code, with architecture portion reduced to defining and managing low level hardware pieces of clocksources. While clock sources provide read access to the monotonically increasing time value, clock event devices are used to schedule the next event interrupt(s). The setup and selection of the event devices is hardwired into the architecture dependent code. The clock events provides a generic framework to manage clock event devices and their usage for the various clock event driven kernel functionalities.

Linux kernel typically provided timers which are of tick (HZ) based timer resolution. clockevents framework in the newer kernels enabled the smooth implementation of high resolution timers across architectures. Depending on the clock source and clock event devices that are available in the system, kernel switches to the hrtimer [8] mode and per CPU periodic timer tick functionality is also provided by the per CPU hrtimers (managed by per-CPU clockevent device).

Hrtimer based periodic tick enabled the functionality of tickless idle kernel. When a CPU goes into idle state, timer framework evaluates the next scheduled timer event and in case that the next event is further away than the next periodic tick, it reprograms the per-CPU clockevent device to this future event. This will allow the idle CPU to go into longer idle sleeps without the unnecessary interruption by the periodic tick.

The current solution in 2.6.21 [4] eliminates the tick during idle and as such it is not a full tickless or dynamic tick implementation. However, eliminating the tick during idle is a great first step forward. Future trend however is to completely remove the tick, irrespective of the busy or idle CPU state.

2.1 Dynticks effectiveness data

Effectiveness of dynticks can be measured using different system information like power consumption, idle time etc. Following sections of this paper looks at the dynticks effectiveness in terms of three measurements done on a totally idle system.

- The number of interrupts per second.

- The number of timer events per second.

- Average amount of time CPU stays in idle state upon each idle state entry.

These measurements, collectively, gives an easy approximation to actual system power and battery consumption.

These measurements reported were taken on a totally idle system, few minutes after reboot. The number of interrupts in the system, per second, is computed by taking the average from output of `vmstat`. The value reported is the number of interrupts per second on the whole system (all CPUs). The number of events reported is from `/proc/timer_stats` and the value reported is events per second. Average CPU idle time per call is the average amount of time spent in a CPU idle state per entry into idle. The time reported is in uS and this data is obtained by using `/proc/acpi/processor/ CPU*/power`. All these three measurements are at the system level and includes the activity of all CPUs.

The system used for the measurement is a Mobile reference system with Intel® Core™ 2 Duo CPUs (2 CPU cores), running i386 kernel with a HZ rate of 1000.

2.1.1 Baseline measurement

Table 1 show the data with and without dynticks enabled.

As the data indicates, dyntick eliminates the CPU timer ticks when CPUs idle and hence reduces the number of interrupts in the system drastically. This reduction in number of interrupts also increases the amount of time CPU spends in an idle state, once entering that idle state.

Is this the best that can be done or is there scope to do more changes within the kernel to reduce the number of interrupts and/or reduce the number of events? Following sections addresses this specific question.

	# interrupts	#events	Avg CPU idle residency (uS)
With ticks	2002	59.59	651
Tickless	118	60.60	10161

Table 1: System activity during idle with and without periodic ticks

3 Keeping Kernel Quiet

One of the noticeable issue with the use of timers, inside core kernel, in drivers and also in userspace, is that of staggered timers. Each user of a timer does not know about other timers that may be setup on the system at the particular time and picks a particular time based on its own usage restrictions. Most of the time, this specific usage restriction does not mandate a strict hard time value. But, each timer user inadvertently end up setting the timer on their own and hence resulting in a bunch of staggered timers at the system level.

To prevent this in kernel space, a new API `__round_jiffies()` is introduced [6] in the kernel. This API rounds the jiffy timeout value to the nearest second. All users of timeout, who are really not interested in precise timeout should use this API while setting their timeout. This will cause all such timers to coalesce and expire at the same jiffy, preventing the staggered interrupts.

Another specific issue inside the kernel that needed special attention are the timer interrupts from drivers which are only important when CPU is busy and not really as important to wake the CPU from idle and service the timer. Such timer can tolerate the latency until some other important timer or interrupt comes along, at which time this timer can be serviced as well. Classic example of this usage model is cpufreq ondemand governor.

ondemand governor monitors each processor utilization at periodic intervals (many times per second) and tries to manage the processor frequency, keeping it close to the processor utilization. When a processor is totally idle, there is no pressing need for ondemand to periodically wakeup the processor just to look at its utilization. To resolve this issue, a new API of `deferrable timers` was introduced [1] in the recent kernel.

Deferrable timer is a timer that works as a normal timer when processor is active, but will be deferred to a later time when processor is idle. The timers thus deferred will be handled when processor eventually comes out of idle due to a non-deferrable timer or any other interrupts.

Deferrable timer is implemented using a special state bit in the timer structure, overloaded over one of the fields in the current structure, which maintains the nature of the timer (deferrable or not). This state bit is preserved as the timer moves around in various per-CPU queues and `__next_timer_interrupt()` skips over all the deferrable timers and picks the timer event for next non-deferrable timer in the timer queue.

This API works very well with ondemand, reducing the number of interrupts and number of events on an idle system as shown in the Table 2. As shown, this feature nearly doubles the average CPU idle residency time, on a totally idle system. This API may also have limited usages with other timers inside the kernel like machine check or cache reap timers and has the potential to reduce the number of wakeups on an idle system further down.

Note that this timer is only available for in kernel usage at this point and usage for user apps is not yet conceptualized. It is not straight forward to extend this to userspace as user timer are not directly associated with per-CPU timer queues and also there can be various dependencies across multiple timers, which can result in timer reordering depending on what CPU they are scheduled on and whether that CPU is idle or not.

4 Platform Timer Event Sources

Dynticks depends on having a per-CPU timer event source. On x86, LAPIC timer can be used as a dependable timer event source. But, when the platform supports low power processor idle states (ACPI C2, C3 states), on most current platforms LAPIC timer stops ticking while CPU is in one of the low power idle states. Dynticks uses a nice workaround to address this issue, with the concept of broadcast timer. Broadcast timer is an always working timer, that will be shared across a

	# interrupts	#events	Avg CPU idle residency (uS)
Ondemand	118	60.60	10161
Ondemand + deferrable timer	89	17.17	20312

Table 2: System activity during idle with and without deferrable timer usage in ondemand

pool of processors and has the responsibility to send a local APIC timer interrupt to wakeup any processor in the pool. Such broadcast timers are platform timers like PIT or HPET [3].

PIT/8254: is a platform timer that can run either in one-shot or periodic mode. It has a frequency of 1193182 Hz and can have a maximum timeout of 27462 uS.

HPET: Is based on newer standard, has a set of memory mapped timers. These timers are programmable to work in different modes and frequency of this timer is based on the particular hardware. On our system under test, HPET runs at 14318179 Hz freq and can have a max timeout of more than 3 seconds (with 32-bit timer).

HPET is superior than PIT, in terms of max timeout value and thus can reduce the number of interrupts when the system is idle. Most of the platforms today has built in HPET timers in the chipset. Unfortunately, very few of the platforms today enable HPET timer and/or advertise the existence of HPET timer to OS using the ACPI tables. As a result, Linux kernel ends up using PIT for broadcast timer. This is not an issue on platforms which do not support low power idle state (e.g., today's servers) as they have always working local APIC time. But, this does cause issue on laptops that typically support low power idle states.

4.1 Force detection of HPET

To resolve this issue of BIOS not enabling/advertising HPET, there is a kernel feature to forcefully detect and enable HPET on various chipsets, using chipset specific PCI quirks. Once that is done, HPET can be used instead of PIT to reduce the number of interrupts on an idle system further. On our test platform, data that we got after forcefully detecting and enabling HPET timer is in Table 3.

Note that this patch to force enable HPET was in proposal-review state at the time of writing this paper and was not available in any standard kernel yet.

4.2 HPET as a per-CPU timer

Linux kernel uses "legacy replacement" mode of HPET timer today to generate timer events. In this mode, HPET appears like legacy PIT and RTC to OS generating interrupts on IRQ0 and IRQ8 for HPET channel 0 and channel 1 timer respectively. There is a further optimization possible, where HPET can be programmed in "standard" interrupt delivery mode and use different channels of HPET to send "per-CPU" interrupt to different processors. This will help laptops that have 2 logical CPUs and at least 2 HPET timer channels available. Different channels of HPET can be used to program timer for each CPU, thereby avoiding the need for broadcast timer altogether and eliminating the LAPIC timers as well. This feature, which is still under development at the time of writing this paper, brings an incremental benefit on systems with more than one logical CPUs and that support deep idle states (most laptop systems with dual core processors). Table 4 shows the data with and without this feature on such a system.

In comparison to base tickless data (as shown in Table 1), features we have talked so far (as shown in Tables 2, 3, and 4) have demonstrated the increase in idle residency time by approximately 7 times.

5 Idle Process Load balancing

In the regular kernel, one of the usages of the periodic timer tick on each processor is to perform periodic process load balancing on that particular processor. If there is a process load imbalance, load balancer will pull the process load from a busiest CPU, ensuring that the load is equally distributed among the available processors in the system. Load balancing period and the subset of processors which will participate in the load balancing will depend on different factors, like the busy state of the processor doing the load balance, the hierarchy of scheduler domains that this processor is part of and the

	# interrupts	#events	Avg CPU idle residency (uS)
PIT	89	17.17	20312
HPET	32	15.15	56451

Table 3: System activity during idle with PIT Vs HPET

	# interrupts	#events	Avg CPU idle residency (uS)
global HPET	32	15.15	56451
percpu HPET	22	15.15	73514

Table 4: System activity during idle with global vs. per-CPU HPET channels

Garbage collector	Perf Regression
parallel	6.3%
gencon	7.7%

Table 5: SPECjbb2000 performance regression with Tickless kernel. Tickless Idle load balancing enhancements recovered this performance regression.

process load in the system. Compared to busy CPUs, idle CPUs perform load balancing more often (mainly because it has nothing else to do and can immediately start executing the pulled load, the load otherwise was waiting for CPU cycles on another busy CPU). In addition to the fairness, this will help improve the system throughput. As such, idle load balancing plays a very important role.

Because of the absence of the periodic timer tick in tickless kernel, idle CPU will potentially sleep for longer time. This extended sleep will delay the periodic load balancing and as such the idle load balancing in the system doesn't happen as often as it does in the earlier kernels. This will present a throughput and latency issue, especially for server workloads.

To measure the performance impact, some experiments were conducted on an 8 CPU core system (dual package system with quad-core) using SPECjbb2000 benchmark in a 512MB heap and 8 warehouses configuration. Performance regression with tickless 2.6.21-rc6-rt0 kernel [5] is shown in Table 5.

Recovering this performance regression in the tickless kernel with no impact on power savings is tricky and

challenging. There were some discussions happened in the Linux kernel community on this topic last year, where two main approaches were discussed.

First approach is to increase the back off interval of periodic idle load balancing. Regular Linux kernel already does some sort of backoff (increasing the load balance period up to a maximum amount) when the CPU doing the load balance at a particular sched domain finds that the load at that level is already balanced. And if there is an imbalance, the periodic interval gets reset to the minimum level. Different sched domains in the scheduler domain hierarchy uses different minimum and maximum busy/idle intervals and this back off period increases as one goes up in the scheduler domain hierarchy. Current back off intervals are selected in such a fashion that there are not too many or too less load balancing attempts, so that there is no overdoing the work when the system is well balanced and also react in reasonable amount of time, when the load changes in the system.

To fix the performance regression, this approach suggests to further increase the backoff interval for all the levels in the scheduler domain hierarchy but still retaining the periodic load balancing on each CPU (by registering a new periodic timer event which will trigger the periodic load balancing). Defining the interval increase will be tricky and if it is too much, then the response time will also be high and won't be able to respond for sudden changes in the load. If it is small, then it won't be able to save power, as the periodic load balancing will wake up the idle CPU often.

Second mechanism is some sort of a watchdog mech-

anism where the busy CPU will trigger the load balance on an idle CPU. This mechanism will be making changes to the busy load balancing (which will be doing more load balancing work, while the current busy task on that CPU is eagerly waiting for the CPU cycles). Busy load balancing is quite infrequent compared to idle load balancing attempts. Similar to the first mechanism, this mechanism also won't be able to respond quickly to changes in load. And also figuring out that a CPU is heavily loaded and where that extra load need to moved, is some what difficult job, especially so in the case of hierarchical scheduler domains.

This paper proposes a third route which nominates an owner among the idle CPUs, which does the idle load balancing (ILB) on behalf of the other idle CPUs in the system. This ILB owner will have the periodic tick active in idle state and uses this tick to do load balancing on behalf of all the idle CPUs that it tracks, while the other idle CPUs will be in long tickless sleeps. If there is an imbalance, ILB owner will wakeup the appropriate CPU from its idle sleep. Once all the CPUs in the system are in idle state, periodic tick on the ILB owner will also stop (as there is no other CPU generating load and hence no reason for checking the load imbalance). New idle load balancing owner will be selected again, as soon as there is a busy CPU in the system.

This solution is in 2.6.21 -mm kernels and the experiments showed that this patch completely recovered the performance regression seen earlier (Table 5) in the tickless kernels with SPECjbb workload. If this ILB owner selection is done carefully (like an idle core in a busy package), one can minimize the power wasted also.

6   Keeping Userspace Quiet

Tickless idle kernel alone is not sufficient to enable the idle processor to go into long and deep sleeps. In addition to the kernel space, applications in the user space also need to quite down during idle periods, which will ensure that the whole system goes to long sleeps, ultimately saving power (and thus enhancing battery life in case of laptops).

Dave Jones' OLS2006 talk [9] entitled "Why Userspace Sucks" revealed to the Linux community that the userspace really doesn't quiet down as one would hope for, on an otherwise idle system. Number of applications and daemons wakeup at frequent intervals (even on a completely idle system) for performing a periodic activity like polling a device, cursor blinking, querying for a status change to modify the graphical icon accordingly and so on (for more information about the mischeivous application behaviors look into references [9, 10, 2]).

A Number of fixes went into applications and libraries over the course of the last year to fix these problems [2]. Instead of polling periodically for checking status changes, applications and deamons should use some sort of event notification where ever possible and perform the actions based on the triggered event. For example, the `hal` daemon used to poll very frequently to check for media changes and thus making the idle processor wakeup often. Newer SATA hardware supports a feature called Asynchronous Notification, which will notify the host at a media change event. With the recent changes in the community, `hal` daemon will avoid the polling on platforms which has the support of this asynchronous notification.

Even in the case where the application has to rely on periodic timers, application should use intelligent mechanisms to avoid/minimize the periodic timers when possible. Instead of having scattered timers across the period of time, it will be best to group them and expire the bunch of timers at the same instance. This grouping will minimize the number of instances a processor will be wokenup, while still servicing the same number of timers. This will enable the processor to sleep longer and go into the lowest power state that is possible.

For example all gnome applications use the glib timer `g_timeout_add()` API for their timers, which expire at scattered instances. A second API, `g_timeout_add_seconds()` has now been added which causes all recurring timers to happen at the start of the second, enabling the system wide grouping of timers. The start of the second is offset by a value which is system wide but system-specific to prevent all Linux machines on the internet doing network traffic at the same time.

New tools are getting developed [10] for identifying the applications (and even the kernel level components) which behave badly by wakingup more often than required. These tools use the kernel interfaces (like `/proc/timer_stats`, `/proc/acpi/processor/CPU*/power`, `/proc/interrupts`) and report the biggest offenders and also the average C-state residency information of the processor. Developers should use these tools to identify if their application is

on the hitlist and if so, fix them based on the above mentioned guidelines.

7 Conclusions

Tickless kernel infrastructure is the first and important step forward in making the idle system go into long and deep sleeps. Now that this is integrated into Linux kernel, enhancements mentioned in this paper will increase the mileage out of the tickless kernel, by minimizing the unnecessary wakeups in an otherwise idle system. Going forward, responsibility of saving power lies with both system and user level software. As an evolutionary step, in coming days we can expect the Linux kernel to be fully (both during idle and busy) dynamic tick capable.

8 Acknowledgments

Thanks to all the Linux community members who contributed, reviewed and provided feedback to various tickless add on features mentioned in this paper.

References

[1] Deferrable timers.
`http://lwn.net/Articles/228143/`.

[2] Dependency tree for bug 204948: Userspace sucks (wakeups). `https:
//bugzilla.redhat.com/bugzilla/
showdependencytree.cgi?id=204948`.

[3] High precision event timers specification.
`http://www.intel.com/technology/
architecture/hpetspec.htm`.

[4] Linux 2.6.21. `http://www.kernel.org`.

[5] Realtime preempt patches.
`http://people.redhat.com/mingo/
realtime-preempt/`.

[6] Round jiffies infrastructure. `http:
//lkml.org/lkml/2006/10/10/189`.

[7] Thomas Gleixner and Ingo Molnar. Dynamic ticks.
`http://lwn.net/Articles/202319/`.

[8] Thomas Gleixner and Douglas Neihaus. High resolution timers ols 2006.
`https://ols2006.108.redhat.com/
reprints/gleixner-reprint.pdf`.

[9] Dave Jones. Why userspace sucks ols 2006.
`https://ols2006.108.redhat.com/
reprints/jones-reprint.pdf`.

[10] Arjan Van De Ven. Programming for low power usage.
`http://conferences.oreillynet.
com/cs/os2007/view/e_sess/12958`.

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.
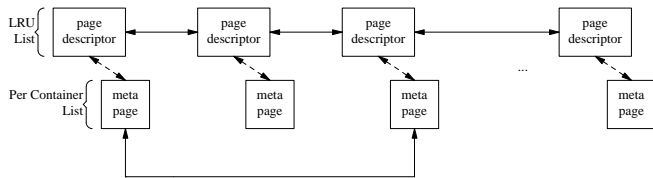
Figure 1: RSS controller with per container list

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.
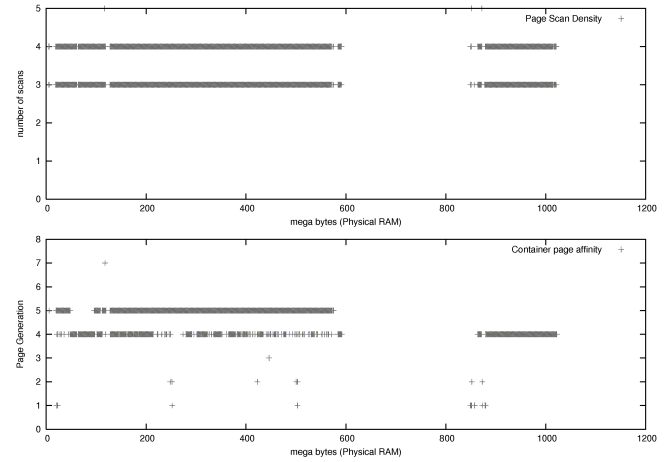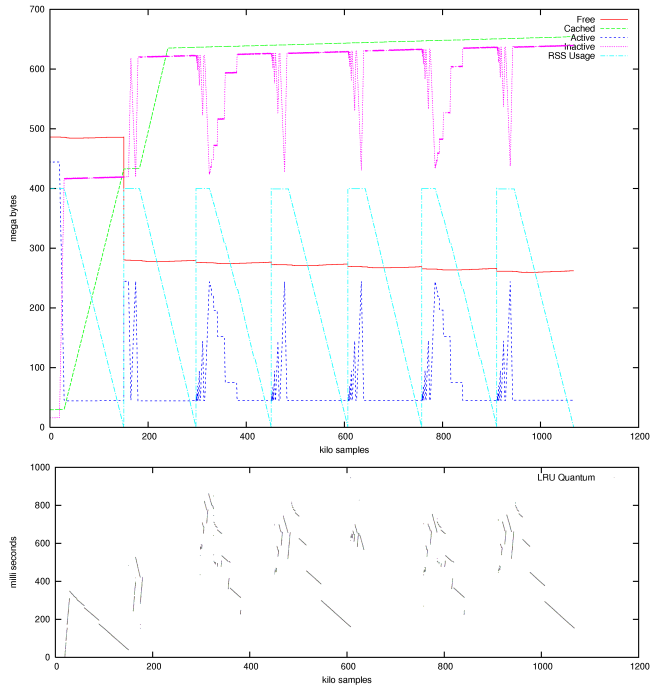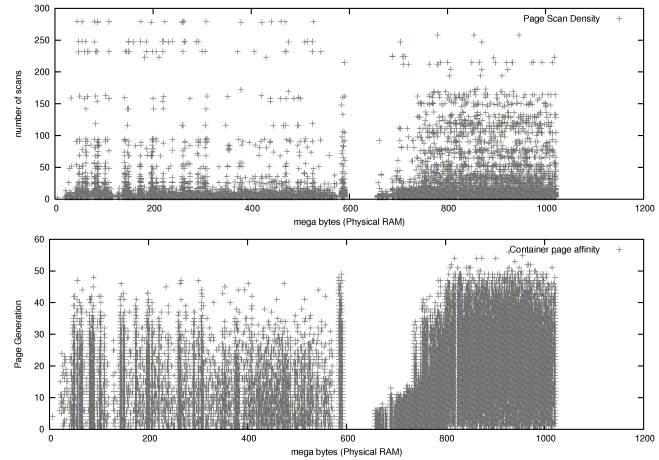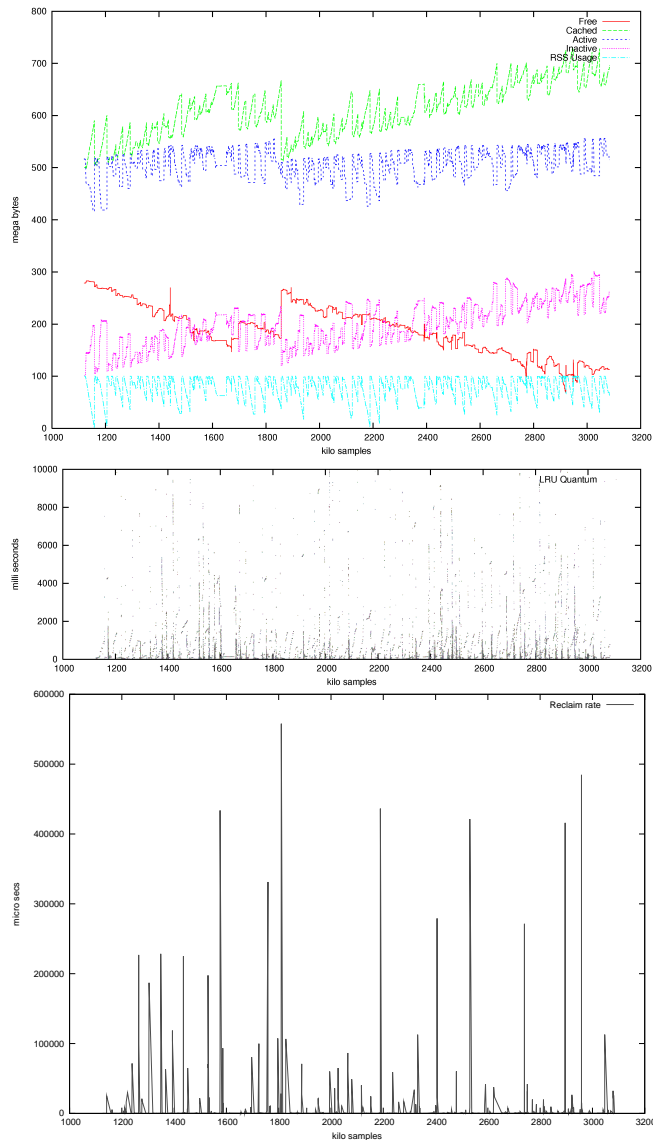
Observations:

- 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.
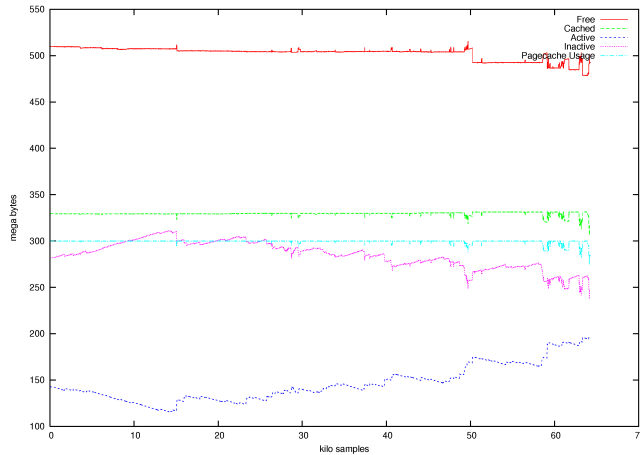
Observations:

- 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.
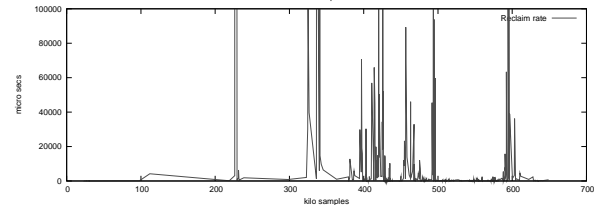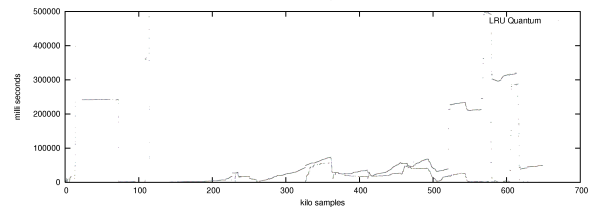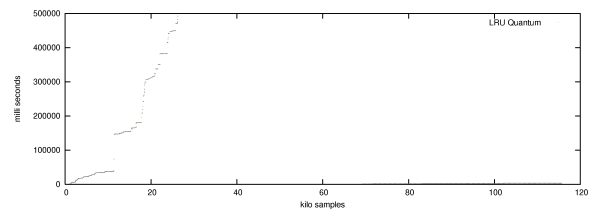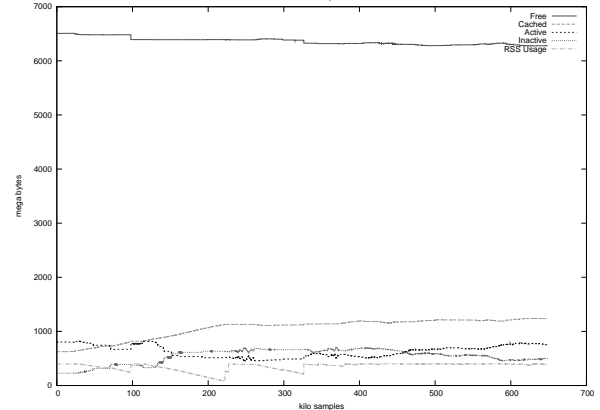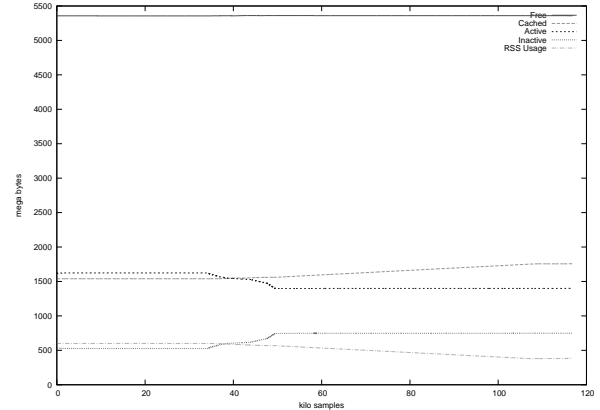
Observations:

- Pagecache controller would not reclaim all page-cache 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.

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 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.

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.

Kernel Support for Stackable File Systems

Josef Sipek, Yiannis Pericleous, and Erez Zadok
Stony Brook University
`{jsipek,yiannos,ezk}@fsl.cs.sunysb.edu`

Abstract

Although it is now possible to use stackable (layered) file systems in Linux, there are several issues that should be addressed to make stacking more reliable and efficient. To support stacking properly, some changes to the VFS and VM subsystems will be required. In this paper, we discuss some of the issues and solutions proposed at the Linux Storage and Filesystems workshop in February 2007, our ongoing work on stacking support for Linux, and our progress on several particular stackable file systems.

1 Introduction

A stackable (layered) file system is a file system that does not store data itself. Instead, it uses another file system for its storage. We call the stackable file system the *upper* file system, and the file systems it stacks on top of the *lower* file systems.

Although it is now possible to use stackable file systems, a number of issues should be addressed to improve file system stacking reliability and efficiency. The Linux kernel VFS was not designed with file system stacking in mind, and therefore it comes as no surprise that supporting stacking properly will require some changes to the VFS and VM subsystems.

We use eCryptfs and Unionfs as the example stackable file systems to cover both linear and fan-out stacking, respectively.

eCryptfs is a cryptographic file system for Linux that stacks on top of existing file systems. It provides functionality similar to that of GnuPG, except that encrypting and decrypting the data is transparent to the application [1, 3, 2].

Unionfs is a stackable file system that presents a series of directories (branches) from different file systems as one virtual directory, as specified by the user. This is commonly referred to as namespace unification. Previous publications [4, 5, 6] provide detailed description and some possible use cases.

Both eCryptfs and Unionfs are based on the FiST stackable file system templates, which provide support for layering over a single directory [7]. As shown in Figures 1(a) and 1(b), the kernel's VFS is responsible for dispatching file-system–related system calls to the appropriate file system. To the VFS, a stackable file system appears as if it were a standard file system. However, instead of storing or retrieving data, a stackable file system passes calls down to lower-level file systems. In this scenario, NFS is used as a lower-level file system, but any file system can be used to store the data as well (e.g., Ext2, Ext3, Reiserfs, SQUASHFS, isofs, `tmpfs`, etc.).

To the lower-level file systems, a stackable file system appears as if it were the VFS. Stackable file system development can be difficult because the file system must adhere to the conventions of both the file systems for processing VFS calls, and of the VFS for making VFS calls.

Without kernel support, stackable file systems suffer from inherent cache coherency problems. These issues can be divided into two categories: (1) data coherency of the page cache contents, and (2) meta-data coherency of the `dentry` and `inode` caches. Changes to the VFS and the stackable file systems are required to remedy these problems.

Moreover, `lockdep`, the in-kernel lock validator, "observes" and maps all locking rules as they dynamically occur, as triggered by the kernel's natural use of locks (spinlocks, rwlocks, mutexes, and rwsems). Whenever the lock validator subsystem detects a new locking scenario, it validates this new rule against the existing set of rules. Unfortunately, stackable file systems need to lock

(a) eCryptfs layers over a single directory. (b) Unionfs layers over multiple directories.
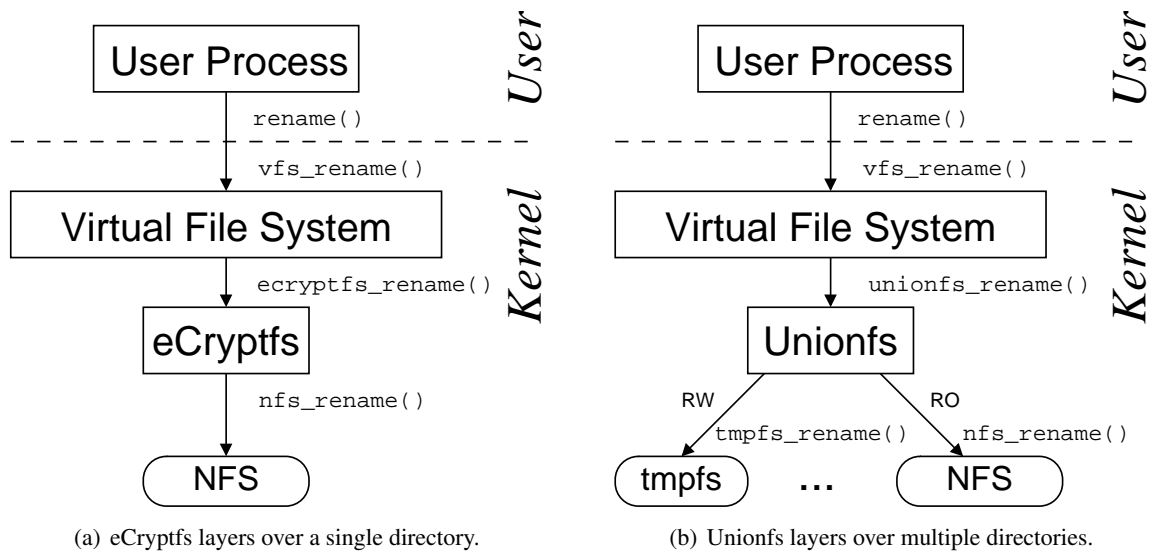
Figure 1: The user processes issue system calls, which the kernel's virtual file system (VFS) directs to stackable file systems. Stackable file systems in turn pass the calls down to lower-level file systems (e.g., tmpfs or NFS).

many of the VFS objects in a recursive manner, triggering `lockdep` warnings.

To maintain the upper to lower file system mapping of kernel objects (such as `dentrys`, `inodes`, etc.), many stackable file systems share much of the basic infrastructure. The 2.6.20 kernel introduced `fs/stack.c`, a new file that contains several helper functions useful to stackable file systems.

The rest of this paper is organized as follows. In Section 2 we discuss the cache coherency issues. In Section 3 we discuss the importance of locking order. In Section 4 we discuss `fsstack`, the emerging Linux kernel support for stacking. In Section 5, we discuss a persistent store prototype we developed for Unionfs, which can be of use to others. Finally, we conclude in Section 6.

2 Cache Coherency

There are two different cache coherency issues that stackable file systems must overcome: data and metadata.

2.1 Data Coherency

Typically, the upper file system maintains its own set of pages used for the page cache. Under ideal conditions,

all the changes to the data go through the upper file system. Therefore, either the upper file system's `write` inode operation or the `writepage` address space operation will have a chance to transform the data as necessary (e.g., eCryptfs needs to encrypt all writes) and write it to the lower file system's pages.

Data incoherency occurs when data is written to the lower pages directly, without the stacked file system's knowledge. There are two possible solutions to this problem:

Weak cache coherency – NFS also suffers from cache coherency issues as the data on the server may be changed by either another client or a local server process. NFS uses a number of assertions that are checked before cached data is used. If any of these assertions fail, the cached data is invalidated. One such assertion is a comparison of the `ctime` with what is cached, and invalidating any potentially out-of-date information.

Strong cache coherency – Another possible solution to the cache coherency problem is to modify the VFS and VM to effectively inform the stackable file systems that the data has changed on the lower file system. There are several different ways of accomplishing this, but all involve maintaining pointers from lower VFS objects to *all* upper ones. Regardless of how this is implemented, the VFS/VM

must traverse a dependency graph of VFS objects, invalidate all pages belonging to the corresponding upper addresses spaces, and `sync` all of the pages that are part of the lower address spaces.

Both approaches have benefits and drawbacks.

The major benefit of the weak consistency approach is that the VFS does not have to be modified at all. The major downside is that *every* stackable file system needs to contain a number of these checks. Even if helper functions are created, calls to these functions need to be placed throughout the stackable file systems. This leads to code duplication, which we try to address with `fsstack` (see Section 4).

The most significant benefit of the stronger coherency approach is the fact that it guarantees that the caches are always coherent. At the same time, it requires that the file system use the page cache properly, and that the file system supplies a valid address space operations vector. Some file systems do not meet these requirements. For example, GPFS (created by IBM) only has `readpage` and `writepage`, but does not have any other address space operations. If the cache coherency is maintained at the page-cache level, the semantics of using a lower file system that does not define the needed operations would be unclear.

2.2 Meta-Data Coherency

Similar to the page cache, many VFS objects, such as the cached `inode` and `dentry` objects, may become inconsistent. The meta-data contained in these caches includes the `{a,c,m}times`, file size, etc.

Just as with data consistency, either a strong or a weak cache coherency model may be used to prevent the upper and lower VFS objects from disagreeing on the file system state. The benefits and drawbacks stated previously apply here as well (e.g., weak coherency requires code duplication in most stackable file systems).

2.3 File Revalidation

The VFS currently allows for `dentry` revalidation. NFS and other network file system are the few users of this. A useful addition to this `dentry` revalidation operation would be an equivalent `file` operation. Given

a `struct file`, this would allow the file system to check for validity and repair any inconsistencies.

Unionfs works around the lack of `file` revalidation by calling its own helper function in the appropriate `struct file` operations. The reason Unionfs requires this is due to the possibility of a branch management operation changing the number or order of branches, and the lower `struct file` pointers need to be updated.

3 Locking Order

Since stackable file systems must behave as both a file system and the VFS, they need to lock many of the VFS objects in a recursive manner, triggering warnings about potential deadlocks. The in-kernel lock validator, `lockdep`, dynamically monitors the kernel's usage of locks (spinlocks, rwlocks, mutexes and rwsems) and creates rules. Whenever the lock validator subsystem detects a new locking scenario, it validates this new rule against the existing set of rules.

The `lockdep` system is aware of locking dependency chains, such as: parent→ child→ xattr→ quota. However, it does not understand that a stackable file system may cause recursion in the VFS. For example, the VFS may indirectly (but safely) call itself; `vfs_readdir` can call a stackable file system on one directory, which can in turn call `vfs_readdir` again on other lower directories. Each time `vfs_readdir` is called, the corresponding `i_mutex` is taken. This triggers a `lockdep` warning, as it considers this situation a potential place for a deadlock, and warns accordingly. In other words, `lockdep` needs to be informed of the hierarchies between stacked file systems. This, however, would require adding a "`stacked`" argument to many functions in the VFS, and passing that information to `lockdep`.

4 `fsstack`

The code duplication found in many stackable file systems (such as eCryptfs, Unionfs, and our upcoming cachefs) is another problem. The 2.6.20 kernel introduced `fs/stack.c`, a new file, which contains several useful helper functions. We are working on further abstractions to the stacking API in Linux.

Each stackable file system must maintain a set of pointers from the upper (stackable file system) objects to the lower objects. For example, each Unionfs `inode` maintains a series of lower `inode` pointers.

Currently, there are two ways to keep track of lower objects. Linear (one lower pointer) and fan-out (several lower pointers). Fan-out is the more interesting case, as linear stacking is just a special case of fan-out, with only one branch. Quite frequently, Unionfs needs to traverse all the branches. This creates the need for a `for_each_branch` macro (analogous to `for_each_node`), which would decide when to terminate.

A "reference" stackable file system, much like NullFS in many BSDs, would allow stackable file system authors to easily create new stackable file systems for Linux. This reference file system should use as many of the `fsstack` interfaces as possible. Currently, the closest thing to this is Wrapfs [7], which can be generated from FiST. Unfortunately, the generated code does not follow proper coding style and general code cleanliness.

In Section 2.1, we considered a weaker form of the cache coherency model. This model suffers from the fact that a large number of the coherency checks (e.g., checking the `{a,c,m}times`) will need to be duplicated in each stackable file system. Using `fsstack` avoids this problem by making use of generic functions to perform operations common to all stackable file systems. The code necessary to invalidate and revalidate the upper file system objects could be shared by several file systems. However, *each* file system must call these helper functions. If a bug is discovered in one stackable file system (e.g., a helper function should be called but is not), the fix may have to be ported to other file systems.

Stackable file systems must behave as a file system from the point of view of the VFS, yet they must behave as the VFS from the point of view of the file systems it is stacked on top of. Generally, the most complex code occurs in the file system lookup code. One idea, proposed at the 2007 Linux Storage and Filesystem workshop, was to divide the current VFS lookup code into two portions, and to allow the file system to override part of the functionality via a new `inode` operation. The default operation would have functionality identical to the current lookup code. The flexibility allowed by this code refactoring would simplify some of the code in more complex file systems. For example, Ext2 could use the generic lookup code provided by the VFS, while a file system requiring more complex lookup code, such as Unionfs, can provide its own lookup helper which performs the necessary operations (e.g., to perform namespace unification).

5 On Disk Format (ODF)

We have developed an On Disk Format (ODF) to help Unionfs 2.0 persistently store any meta-data it needs, such as whiteouts. ODF is a small file system on a partition or loop device. The ODF has helped us resolve most of the critical issues that Unionfs 1.x had faced, such as namespace pollution, `inode` persistence, `readdir` consistency and efficiency, and more. Since, all the meta-data is kept in a separate file system instead of in the branches themselves, Unionfs can be stacked on top of itself and have overlapping branches.

Such a format can be used by any stackable file system that needs to store meta-data persistently. For example, a versioning file system may use it to store information about the current version of a file, a caching file system may use it to store information about the status of the cached files. Unionfs benefits in many ways as well, for example there is no namespace pollution, and Unionfs can be stacked on itself.

By keeping all the meta-data together in a separate file system, simply creating an in-kernel mount can be used to easily hide it from the user, assuring that the user will not temper with the meta-data. Also, it becomes easier to backup the state of the file system by simply creating a backup of the ODF file system. If a file system which statically allocates `inode` tables is used, the user must estimate the number of `inodes` and data blocks the ODF will need before hand. Using a file system which allocates `inode` blocks dynamically (e.g., XFS) fixes this problem. This is a shortcoming of the file system, and not ODF itself.

The ODF can use another file system, such as Ext2 or XFS, to store the meta-data. Another possibility we are looking at for the future is to build an ODF file system that will have complete control of how it stores this meta-data, thus allowing us to make it more efficient, flexible and reusable.

6 Conclusion

Stackable file systems can be used today on Linux. There are some issues which should be addressed to increase their reliability and efficiency. The major issues include the data and meta-data cache coherency between the upper and lower file systems, code duplication between stackable file systems, and the recursive nature of stacking causing `lockdep` to warn about what it infers are possible deadlocks. Addressing these issues will require changes to the VFS/VM.

7 Acknowledgements

References

[1] M. Halcrow. ecryptfs: a stacked cryptographic filesystem. *Linux Journal*, (156), April 2007.

[2] M. A. Halcrow. Demands, Solutions, and Improvements for Linux Filesystem Security. In *Proceedings of the 2004 Linux Symposium*, pages 269–286, Ottawa, Canada, July 2004. Linux Symposium.

[3] M. A. Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, pages 201–218, Ottawa, Canada, July 2005. Linux Symposium.

[4] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok. UnionFS: User- and Community-oriented Development of a Unification Filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, Ottawa, Canada, July 2006.

[5] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[6] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, (128):24–29, December 2004.

[7] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.

Linux Rollout at Nortel

Ernest Szeideman

Nortel Networks Ltd.

`eszeidem@nortel.com`

Abstract

At Nortel, we have focused on delivering a "Standard Operating Environment" for our design systems whereby we maintain a common set of tools and processes in the rollout of Linux and other operating system images. There are a number of opportunities, challenges, and pitfalls with bringing this about at an enterprise level.

1 Introduction

The Linux version of the Standard Operating Environment (SOE) was born a few years ago out of an initiative to introduce a standard image configuration that would address the needs of groups who were increasingly looking to Linux for product development and testing. Since that initial SOE release, Linux has become common for desktop and server computing solutions across the corporation. The goal of every Linux SOE release is to introduce a certified and supported Enterprise Linux distribution into Nortel.

A survey of the literature reveals many articles detailing the specific implementation details and related challenges faced in creating a standardized image. Fewer articles speak of the high level design and engineering process driving the implementation, and fewer still speak specifically about lessons learned which would assist others in overcoming assumptions and processes counterproductive to such an endeavor. Ubiquitous throughout the IT industry is the concept of a Standard Operating Environment referring to a standard image configuration. However, to be useful and accepted in an enterprise, an SOE requires that the design must solve real business problems for a company, problems that vary over time, across different industries, business environments, and even different business cultures. As such, although the design and development of an SOE may vary, the lessons learned by one company should prove of benefit to others as well.

This paper will discuss what an SOE is, briefly describe how we did the design and why, and most importantly speak of the lessons we have learned and how they relate to Linux and the open source model from an enterprise perspective.

2 What is an SOE and Why

A Standard Operating Environment for Linux means a standard image configuration for both desktops and servers. The intention is that unless there is sufficient justification, all supported Linux installs within the company will use the SOE. This greatly simplifies management by ensuring consistency in the deployed image regardless of location which provides high levels of reliability and supportability.

The SOE should support a limited set of hardware that has been chosen for company-wide use for both servers and workstations. Although Linux provides perhaps the most hardware support of any operating system ever, minimizing the set of hardware reduces the matrix of testing required. This reduces hardware support costs, and reduces hardware acquisition costs due to volume purchasing.

The inclusion of a common set of Linux vendor packages on all machines, a common set of third-party packages, as well as a common set of company-developed packages, ensures consistency in the deployed image regardless of location. It also provides a vehicle whereby software can be deployed company-wide to meet ever changing business needs.

Security and network certification of the image implies security and network configuration changes (such as ensuring limited world access to init scripts or checks to ensure that IP forwarding is turned off). This helps the company in minimizing risk, taking advantage of security and network expertise, providing confidence in the

SOE, as well as ensuring that the image plays nicely within a company's environment.

Providing a standardized installation process, including appropriate storage locations for the image, reduces the net cost per Linux install by reducing complexity, ensuring standardization, and maximizing the ability to support the install via documentation and support help lines.

Lastly, having a formalized process to gather requirements, design, implement, test, and trial an SOE ensures that tasks can be adequately resourced, timelines meet business priorities, and consumers of the image can plan for the deployment and use of the image to accomplish business goals.

3   The Nortel SOE

At Nortel Networks Inc., there are over 291,000 nodes on a network with over 350 locations throughout the world containing 8,000 subnets housing a myriad of servers and desktops running many different operating systems and providing access to a number of different network services (as of May, 2006). Any opportunities at standardization will result in substantial savings to the corporation.

Level		Explanation
5	Patching	Post-SOE maintenance
4	Group specific	UML, Clearcase
3	Location specific	Postinstall, cfengine
2	Global config	Packages, security, network
1	Vendor OS	Consistent set of packages
0	Hardware	Hardware catalogue

The table above denotes the high-level design of the Linux SOE.

Level 0, or the Hardware layer, represents all activities in achieving a standard catalogue of hardware including hardware comparisons, benchmarking, vendor negotiation, and the like. Any SOE that is released will have, as a minimum, the requirement to support the catalogue hardware.

Level 1, or the Vendor operating system layer, is the inclusion of a consistent set of packages from the vendor that is supposed to achieve three things:

1. It must include a reasonable set of packages required to support the environment.

2. It must include packages that are deemed as required by the internal customers.

3. It must attempt to be consistent with previous SOE releases (i.e., it must attempt to match the functionality that was included in previous SOE releases at this layer).

In this layer, one should capitalize on and make use of the installation tools or mechanisms provided by the vendor (e.g., Anaconda/kickstart used in RHEL (Red Hat Enterprise Linux) or YaST (Yet Another Setup Tool) used in SuSE (Software und System-Entwicklung)). As Nortel is using RHEL in its SOE, this layer is accomplished with the use of kickstart where the required package groupings and packages are specified in the %packages section.

Level 2, or the Global configuration layer, is where other packages not necessarily provided by the vendor are installed. This includes security, network, and company-provided packages. A special design consideration for this layer is to keep absolute separation between whatever installation mechanisms the vendor provides and the one relied upon at this layer. At Nortel, the automated kickstart installation mechanism has a post install section that is used to automatically kick off an install script, which completes all aspects of this layer. The benefits of this are threefold:

1. It allows easy determination of where problems may exist in an install.

2. It insulates the SOE engineers from changes made to the vendor's installation tools or mechanisms.

3. It allows for changes to the underlying Linux distribution without major impact to this level or the levels above it.

Level 3, or the location-specific layer, is designed to answer the requirement of how to maintain standardization across a multitude of locations where specific infrastructure services, service names, and configuration processes differ. This is a key challenge for any large corporation. The methodology employed requires that the target node take advantage of locally dependent services

while maintaining the standardization of the SOE. For Nortel, an init (initialization) script, which allows for complete automation, handles this layer which includes support for NIS, NTP, LDAP, and cfengine as well as other services. Automation of all of these tasks is not only desirable, but is also required if one wants to ensure consistency in deployment. An additional requirement of this layer is to be able to re-implement these services in the event that a machine changes locations (for example, if a node is redeployed to another site, the employee changes locations, etc.). Making use of an init script allows for this requirement. Configuration-specific parameters are sourced from location-named files containing all data relevant to this layer for each major location at the company.

Level 4, or the group-specific layer, contains that which does not need to be installed everywhere, yet which is required by specific groups. Examples of this are the use of virtualization such as UML (User-Mode Linux) as well as Clearcase. Interestingly, group-specific software, such as Clearcase, also has location-specific dependencies (for example, which VOB (Versioned Object Base) servers to connect to may be dependent on which site you work at). The ownership of each of the capabilities relied upon at this level is provided at Nortel by specific individuals or groups who may have formal vendor relationships as required. As these people or groups have the requirement to have their code work with the SOE, they form a special community who has access to pre-releases of all SOEs. References are made to their documentation from within that provided for each SOE. In some cases, there is an automatic reinstallation of these pieces in the event a reimage is performed.

Level 5, or the patching layer, concerns itself with post-SOE maintenance. Once an image is deployed, it must still be maintained and/or kept track of for licensing. There must also be the facility to account for changing business requirements, which may include the deployment of new or updated products (for example, DST (Daylight Saving Time) fixes). At Nortel, we are currently using RHN (Red Hat Network) Satellite. We take a snapshot of the base channel minus kernel (3rd party applications are tied to kernel versions). The snapshot of the channel is tested before the patch bundle is released to ensure that the patches don't break anything in the Nortel environment. This patch bundle is produced quarterly. This allows time to deploy the bundle to all of the systems in an orderly and systematic way.

Within Nortel, rhnsd is not used. The patch window for each system is scheduled ahead of time and controlled by configuration files on the system. A generic scheduling method is employed that can be used across all the UNIX and UNIX-like operating systems. As everything is packaged as a requirement of being included in the SOE, this enables all aspects of the standardized image across all levels to be patched.

4   Lessons Learned

A number of lessons have been learned since a fully supported Linux was introduced in Nortel a few years ago. These are lessons taken from an enterprise perspective and may not apply everywhere.

1. Remote cloning of machines in an enterprise is a deployment concern that must be taken into account. Hewlett Packard's iLO (Integrated Lights-Out) or other similar remote console mechanisms are highly desirable, particularly when the system administrator or installer is located remotely from the machine being imaged. One should not assume that the installer is able to sit in front of the machine being deployed.

2. Windows interoperability solutions contained within Linux have really enhanced its value in the enterprise, particularly when compared with other proprietary UNIX operating systems. However, it has and continues to cause many challenges. VMware with a Windows guest is currently being used in Nortel with workstations to provide standardized Windows images to those running Linux. One may wish to refresh one's Linux image to the latest General Availability (GA) release, but this does not assume that one wishes to have their Windows environment upgraded as well. To accommodate this requirement, a localdisk partition is created on all default Linux installs which holds, among other things, the VMware image files. An upgrade clone is utilized which wipes all partitions, except localdisk, and thereby allows the user to get a new Linux build while keeping any VMware images they may have had.

3. One cannot assume access to an enterprise's DHCP (Dynamic Host Configuration Protocol) infrastructure to make use of PXE (Preboot eXecution En-

vironment) installs or Red Hat Network provisioning. In a large corporation, different groups are responsible for different aspects of the infrastructure. As such, it takes time to get consensus on how best to implement change. One such example is the use of PXE as it relates to the DHCP infrastructure. If this is the case, as it is at Nortel, one may need to find alternative means to accomplish remote upgrades of machines. A script, which integrates with cron (a time-based scheduling service in Linux) is currently being used to provide this functionality.

4. For security reasons, patching of one's infrastructure is necessary using internal repositories. As well, no information about the nodes being patched should leave the company network (cannot use RHN or RHN proxy; must use RHN Satellite).

5. An additional comment with regards to patching involves the potential for divergence when patching versus re-rolling an SOE using a newer update. This is particularly true if one is limited in updating kernels due to third-party reliance on the kernel (e.g. Clearcase or UML). For example, if one starts with a RHEL 4.1 machine and patches it with a patch bundle to 4.3, one may not end up with the same system as if one started with RHEL 4.2 because not all patches are included in the patch bundle. In Nortel's case, the kernel is not included in the patch bundle meaning that although both RHEL 4.1 and 4.2 machines were patched to a RHEL 4.3 level, they are not identical.

6. Users in a corporation typically do not have root access. For example, a user without root access cannot add a printer so printer configuration must be managed on a global basis. When users do not have root access, there are significant management and support implications. On the other hand, if the users do have root access, there are a whole different set of support and management implications.

7. Being able to identify an SOE machine remotely is not only desirable, but is also required from a systems management, licensing, lifecycle, and maintenance perspective.

8. Packaging all components of an SOE including in-house and third-party software in the same format as your Linux vendor's packages is important. Being able to easily upgrade if required (such as in the event of a security vulnerability), easily determining versions of software, being able to validate the authenticity of software (via digital signing), and being able to understand where files on a system came from are some of the benefits of this.

9. Communication to your deployment people as well as to those making use of the SOE is paramount to achieving success. Whether by use of WebPages, blogs, user groups, or other forms of documentation, communication gets more challenging as the size of your company grows.

10. ISV (Independent software vendor) support will probably be the single most important factor in determining which distribution your SOE is based on.

11. No matter how much you simplify an install or install process, deployment using installers without Linux experience will be a problem.

12. It is difficult to get patches from your Linux vendor fast enough (e.g., when you find a problem while producing the SOE or a patch bundle, both of which have deadlines to meet).

13. If a vendor says hardware is certified, what does that mean? Read the fine-print!

14. Despite every effort to the contrary, using third-party proprietary applications/code is a requirement that should be assumed in an enterprise (e.g. Clearcase).

15. A variety of products to choose from (KDE vs. Gnome, for example) makes standardization difficult when one must appease many palates.

16. Keep Global configuration (Level 2) and higher separate from the vendor install at all costs or you *will* be sorry!

17. Digitally sign *all* of your in-house packages. Being able to ascertain the authenticity of the packages contained within the SOE is important from a corporate and a security standpoint.

18. Have backup copies of *all* GA'd images. This will save future time and aggravation. At some point somebody will need to install an old image, either for testing or other purposes.

19. Change control is a critical component of SOE development (e.g. CVS, Clearcase, etc.).

20. Testing is important. Sadly, it will be the first thing to go when schedules are tight; having a testing matrix and test plan is paramount. A corollary to this: If someone tells you they have tested their product, but does not have a test plan, they are not telling the truth.

21. Fixes upstream are useless unless they are backported to the current SOE environment(s). (The Fix is Upstream BOF with Matthew Tippett)

22. One cannot move from proprietary UNIX's (Solaris/HP-UX) to Linux in one step, although new projects can start out quite well.

23. There have been and continue to be issues with Linux interoperating in a heterogeneous enterprise environment (e.g. assuming print servers are Linux as opposed to the non-CUPS-aware HP-UX). Do not assume your Linux vendor does any extensive testing using other operating systems your company uses.

24. Each package which you include in an SOE that does not come from the vendor needs to have someone who is responsible for it (a provider).

25. There is a large amount of proprietary thinking on the part of management that needs to be modified when using Linux and/or other open source software. An example is assuming that the Linux vendor that you are paying for support can fork some code to meet the corporation's requirements instead of the vendor waiting for the fix to be available from upstream. The Linux vendor's preferred method is to wait for the fixes to come from upstream (e.g. they would rather wait for the Evolution folks to fix the code than having to fix it themselves and then maintain the fix in subsequent upstream versions if the Evolution folks don't take the fix).

26. When creating an SOE, use of a vendor-supported Linux offering is recommended over an unsupported version. An example is the use of RHEL vs. Fedora. A supported Linux offering will have more and better ISV support. Updates tend to focus on customer problems and compatibility tends to be of higher importance than new features. And lastly, in an enterprise environment where downtime may mean the loss of substantial amounts of money, the ability to get support from a company is important from a business perspective.

5 Conclusion

The engineering and design of an SOE for an enterprise requires participation from throughout the corporation. For any large corporation, developing an SOE is worth the cost due to the many benefits it offers. The design is critical to an SOE's success and must reflect and solve real world business problems.

It is hoped that the many lessons learned in our SOE voyage at Nortel will help guide others pursuing the many benefits a Linux SOE has to offer.

6 References

[1] Office of Government Commerce, Information Technology Infrastructure Library. Retrieved April, 2007 from `http://www.itil.co.uk/`

[2] Aupek, Andrew, Architectural Design of Enterprise Wide Standard Operating Environments. Retrieved April, 2007 from
`http://www.lib.mq.edu.au/about/ conferences/Architectural%20Design%20of% 20Enterprise%20wide%20Standard% 20Operating%20Environments.pdf`

[3] Carnegie Mellon Software Engineering Institute, Capability Maturity Model Integration (CMMI). Retrieved April, 2007 from `http://www.sei.cmu.edu/cmm/`

[4] Edith Cowan University, IT Services Standard Operating Environment. Retrieved April, 2007 from `http://soe.ecu.edu.au/about/`

[5] Griffith University, Standard Operating Environment for Staff Desktop Computers. Retrieved April, 2007 from
`http://www62.gu.edu.au/policylibrary. nsf/0/1e7a27d0e03ceed44a256ee00063ed6b? opendocument`

[6] Queensland University of Technology, Standard Operating Environment. Retrieved April, 2007 from `http://www.its.qut.edu.au/soe/`

Request-based Device-mapper multipath and Dynamic load balancing

Kiyoshi Ueda
NEC Corporation
k-ueda@ct.jp.nec.com

Jun'ichi Nomura
NEC Corporation
j-nomura@ce.jp.nec.com

Mike Christie
Red Hat, Inc.
mchristi@redhat.com

Abstract

Multipath I/O is the ability to provide increased performance and fault tolerant access to a device by addressing it through more than one path. For storage devices, Linux has seen several solutions that were of two types: high-level approaches that live above the I/O scheduler (BIO mappers), and low-level subsystem specific approaches. Each type of implementation has its advantage because of the position in the storage stack in which it has been implemented. The authors focus on a solution that attempts to reap the benefits of each type of solution by moving the kernel's current multipath layer, `dm-multipath`, below the I/O scheduler and above the hardware-specific subsystem. This paper describes the block, Device-mapper, and SCSI layer changes for the solution and its effect on performance.

1 Introduction

Multipath I/O provides the capability to utilize multiple paths between the host and the storage device. Multiple paths can result from host or storage controllers having more than one port, redundancy in the fabric, or having multiple controllers or buses.

As can be seen in Figure 1(a), multipath architectures like MD Multipath or the Device-mapper (DM) layer's multipath target have taken advantage of multiple paths at a high level by creating a virtual device that is comprised of block devices created by the hardware subsystem.

In this approach, the hardware subsystem is unaware of multipath, and the lower-level block devices represent paths to the storage device which the higher-level software routes I/Os in units of BIOs over [1]. This design has the benefit that it can support any block device, and is easy to implement because it uses the same infrastructure that software RAID uses. It has the drawback that it

does not have access to detailed error information, and it resides above the I/O scheduler and I/O merging, which makes it difficult to make load-balancing decisions.

Another approach to multipath design modifies the hardware subsystem or low-level driver (LLD) to be multipath-aware. For the Linux SCSI stack (Figure 1(b)), Host Bus Adapter (HBA) manufacturers, such as Qlogic, have provided LLDs for their Fibre Channel and iSCSI cards which hide the multipath details from the OS [2]. These drivers coalesce paths into a single device which is exposed to the kernel, and route SCSI commands or driver-specific structures. There have also been multipath implementations in the SCSI layer that are able to route SCSI commands over different types of HBAs. These can be implemented above the LLD in the SCSI mid-layer or as a specialized SCSI upper-layer driver [3]. These designs benefit from being at a lower level, because they are able to quickly distinguish transport problems from device errors, can support any SCSI device including tape, and are able to make more intelligent load-balancing decisions.

This paper describes a multipath technique, ***Request***-based DM, which can be seen in Figure 1(c), that is located under the I/O scheduler and above the hardware subsystem, and routes `struct request`s over the devices created by the hardware specific subsystem. (To distinguish from the general meaning of *request*, ***request*** is used in this paper to mention the `struct request`.) As a result of working with ***request***s instead of BIOs or SCSI commands, this new technique is able to bridge the multipath layer with the lower levels because it has access to the lower-level error information, and is able to leverage the existing block-layer statistics and queue management infrastructure to provide improved load balancing.

In Section 2, we will give an overview of the Linux block and DM layer. Then Section 3 describes in more detail the differences between routing I/O at the BIO-level versus the ***request***-level, and what modifications to

File system	Direct I/O		File system	Direct I/O		File system	Direct I/O
BIO mapper (dm-multipath)							
I/O scheduler			I/O scheduler			I/O scheduler	
						Request-based DM	
Hardware subsystem (SCSI, IDE)			SCSI upper level driver disk (sd), tape (st), and CD (sr).			Hardware subsystem (SCSI, IDE)	
			SCSI mid layer				
			Low level driver (Qlogic qla2xxx, Emulex lpfc)				

(a) BIO mapper multipath   (b) SCSI multipath   (c) Request-based DM multipath

Figure 1: Multipath implementations

the block, SCSI, and DM layers are necessary to support ***Request***-based multipath. Finally, we will detail performance results and load-balancing changes in Section 4 and Section 5.

2 Overview of Linux Block I/O and DM

In this section, we look into the building blocks of multipath I/O, specifically, how the applications' read/write operations are translated into low-level I/O requests.

2.1 Block I/O

When a process, or kernel itself, reads or writes to a block device, the I/O operation is initially packed into a simple container called a BIO. (See Figure 2.) The BIO uses a vector representation pointing to an array of tuples of `<page, offset, len>` to describe the I/O buffer, and has various other fields describing I/O parameters and state that needs to be maintained for performing the I/O [4].

Upon receiving the BIO, the block layer attempts to merge it with other BIOs already packed into ***request***s and inserted in the ***request*** queue. This allows the block layer to create larger ***request***s and to collect enough of them in the queue to be able to take advantage of the sorting/merging logic in the elevator [4].
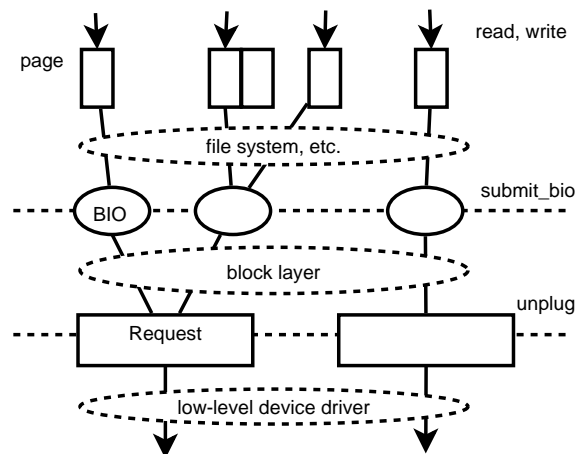
Figure 2: Relationship between page, BIO, and ***request***

This approach to collecting multiple larger ***request***s before dequeueing is called plugging. If the buffers of two BIOs are contiguous on disk, and the size of the BIO combined with the ***request*** it is to be merged with is within the LLD and hardware's I/O size limitations, then the BIO is merged with an existing ***request***; otherwise the BIO is packed into a new ***request*** and inserted into the block device's ***request*** queue. ***Request***s will continue to be merged with incoming BIOs and adjacent ***request***s until the queue is unplugged. An unplug of the queue is triggered by various events such as the

plug timer firing, the number of ***request***s exceeding a given threshold, and I/O completion.

When the queue is unplugged the ***request***s are passed to the low-level driver's `request_fn()` to be executed. And when the I/O completes, subsystems like the SCSI and IDE layer will call `blk_complete_request()` to schedule further processing of the ***request*** from the block layer softirq handler. From that context, the subsystem will complete processing of the ***request*** by asking the block layer to requeue it, or by calling `end_request()`, or `end_that_request_first()/chunk()` and `end_that_request_last()` to pass ownership of the ***request*** back to the block layer. To notify upper layers of the completion of the I/O, the block layer will then call each BIO's `bi_end_io()` function and the ***request***'s `end_io()` function.

During this process of I/O being merged, queued and executed, the block layer and hardware specific subsystem collect a wide range of I/O statistics such as: the number of sectors read/written, the number of merges occurred and accumulated length of time ***request***s took to complete.

2.2 DM

DM is a virtual block device driver that provides a highly modular kernel framework for stacking block device filter drivers [1]. The filter drivers are called target drivers in DM terminology, and can map a BIO to multiple block devices, or can modify a BIO's data or simulate various device conditions and setups. DM performs this BIO manipulation by performing the following operations:

- Cloning

  - When a BIO is sent to the DM device, a near identical copy of that BIO is created by DM's `make_request()` function. The major difference between the two BIOs is that the clone will have a completion handler that points to DM's `clone_endio()` function.

- Mapping

  - The cloned BIO is passed to the target driver, where the clone's fields are modified. By targets that map or route I/O, the `bi_bdev` field

is set to the device it wants to send the BIO to.

  - The cloned BIO is submitted to the underlying device.

- Completion

  - DM's completion handler calls the target-specific completion handler where the BIO can be remapped or completed.

  - Original BIO is completed when all cloned BIOs are completed.

2.3 `dm-multipath`

`dm-multipath` is the DM target driver that implements multipath I/O functionality for block devices. Its map function determines which device to route the BIO to using a path selection module and by ordering paths in priority groups (Figure 3). Currently, there is only the round-robin path selector, which sends *N* number of BIOs to a path before selecting a new path.
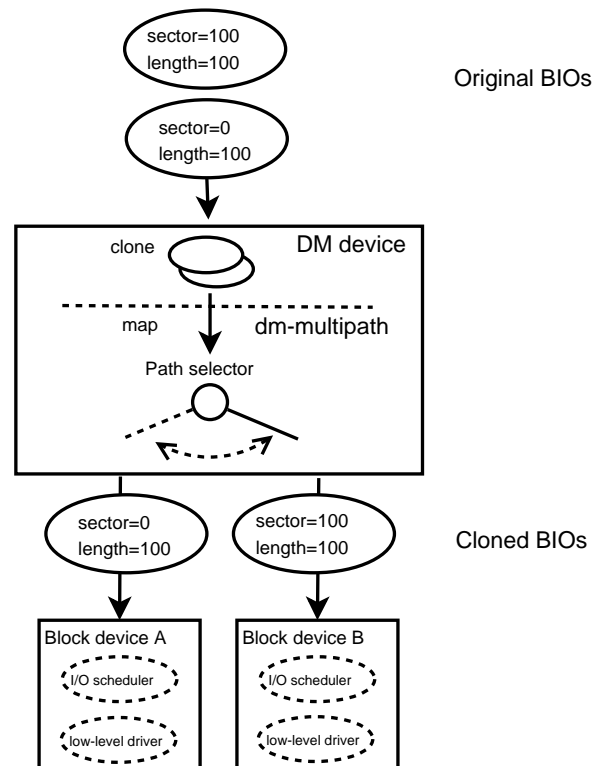

Figure 3: current (BIO-based) `dm-multipath`

3 *Request*-based DM

3.1 Limitations of BIO-based `dm-multipath`

There are three primary problems which ***Request***-based multipath attempts to overcome:

1. Path selection does not consider I/O scheduler's behavior.

2. Block layer statistics are difficult for DM to use.

3. Error severity information is not available to DM.

`dm-multipath`'s path selection modules must balance trying to plug the underlying ***request*** queue and creating large ***request***s against making sure each path is fully utilized. Setting the number of BIOs that cause a path switch to a high value will assure that most BIOs are merged. However, it can cause other paths to be underused because it concentrates I/O on a single path for too long. Setting the BIO path-switching threshold too low would cause small ***request***s to be sent down each path and would negate any benefits that plugging the queue would bring. At its current location in the storage stack, the path selector module must guess what will be merged by duplicating the block layer tests or duplicate the block layer statistics so it can attempt to make reasonable decisions based on past I/O trends.

Along with enhancing performance, multipath's other duty is better handling of disruptions. The LLD and the hardware subsystem have a detailed view of most problems. They can decode SCSI sense keys that may indicate a storage controller on the target is being maintained or is in trouble, and have access to lower level error information such as whether a connection has failed. Unfortunately, the only error information DM receives is the error code value `-EIO`.

Given that these issues are caused by the location of the dm-multipath map and completion functions, ***Request***-based DM adds new DM target drivers call outs, and modifies DM and the block layer to support mapping at the ***request***-level. Section 3.2 will explain the design and implementation of ***Request***-based DM. Section 3.3 discusses the issues that are still being worked on and problems that were discovered with the ***Request***-based approach. Finally, Section 3.4 describes user interface changes to DM.

3.2 Design and implementation

Request-based DM is based on ideas from the block layer multipath [5] patches, which were an experiment that moved the multipath layer down to the ***request***-level. The goal of ***Request***-based DM is to integrate into the DM framework so that it is able to utilize existing applications such as `multipath-tools` and `dmsetup`.

As explained in Section 2.2, DM's key I/O operations are cloning, mapping, and completion of BIOs. To allow these operations to be executed on ***request***s, DM was modified to take advantage of following block layer interfaces:

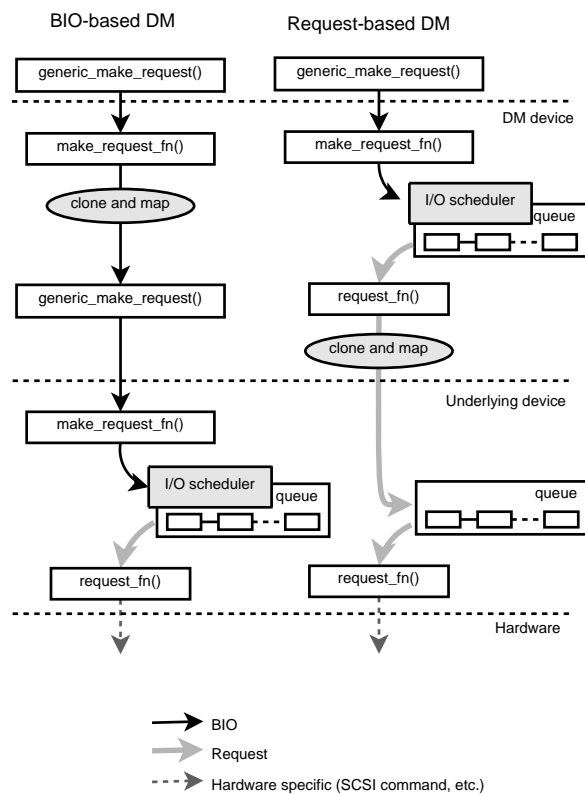| BIO	Submission	`make_request_fn`
	Completion	`bio->bi_end_io`
Request	Submission	`request_fn`
	Completion	`request->end_io`

Figure 4: Difference of I/O submission flow

As can be seen in Figure 4, with BIO-based DM, BIOs submitted by upper layers through `generic_make_request()` are sent to DM's `make_request_fn()` where they are cloned, mapped, and then sent to a device below DM. This device could be another virtual

device, or it could be the real device queue—in which case the BIO would be merged with a ***request*** or packed in a new one and queued, and later executed when the queue is unplugged.

Conversely, with ***Request***-based DM, DM no longer uses a specialized `make_request_fn()` and instead uses the default `make_request_fn()`, `__make_request()`. To perform cloning and mapping, DM now implements its `request_fn()` callback, which is called when the queue is unplugged. From this callback, the original ***request*** is cloned. The target driver is asked to map the clone. And then the clone is inserted directly into the underlying device's ***request*** queue using `__elv_add_request()`.

To handle I/O completion, the ***Request***-based DM patches allow the ***request***'s `end_io()` callback to be used to notify upper layers when an I/O is finished. More details on this implementation in Section 3.3. DM only needs to hook into the cloned ***request***'s completion handler—similar to what is done for BIO completions. (Figure 5).
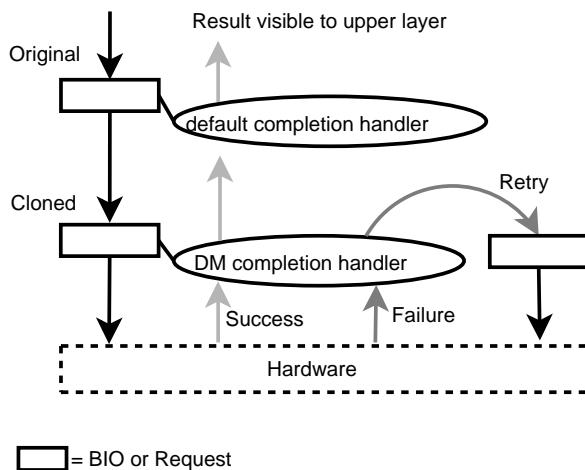
Figure 5: Completion handling of cloned I/O

3.3 ***Request***-based DM road bumps

A lot of the ***Request***-based details have been glossed over, so they could be discussed in this section. While working with ***request***s simplifies the mapping operation, it complicates the cloning path, and requires a lot of new infrastructure for the completion path.

3.3.1 ***Request*** cloning

A clone of the original ***request*** is needed, because a layer below DM may modify ***request*** fields, and the use of a clone simplifies the handling of partial completions. To allocate the clone, it would be best to use the existing ***request*** mempools [6], so DM initially attempted to use `get_request()`. Unfortunately, the block layer and some I/O schedulers assume that `get_request()` is called in the process context in which the BIO was submitted, but a ***request*** queue's `request_fn()` can be called from the completion callback which can be run in a softirq context.

Removing the need for cloning or modifying `get_request()` to be usable from any context would be the preferable solutions, but both required too many changes to the block layer for the initial release. Instead, DM currently uses a private mempool for the ***request*** clones.

3.3.2 Completion handling

`dm-multipath`'s completion handler has to do the following:

- Check completion status.
- Setup a retry if an error has occurred.
- Release its private data if a retry is not needed.
- Return a result to the upper layer.

When segments of a BIO are completed, the upper layers do not begin processing the completion until the entire operation has finished. This allows BIO mappers to hook at a single point—the BIO's `bi_end_io()` callback. Before sending a BIO to `generic_make_request()`, DM will copy the mutable fields like the size, starting sector, and segment/vector index. If an error occurs, DM can wait until the entire I/O has completed, restore the fields it copied, and then retry the BIO from the beginning.

Requests, on the other hand, are completed in two parts: `__end_that_request_first()`, which completes BIOs in the ***request*** (sometimes partially), and `end_that_request_last()`, which handles statistics accounting, releases the ***request***, and calls its `end_`
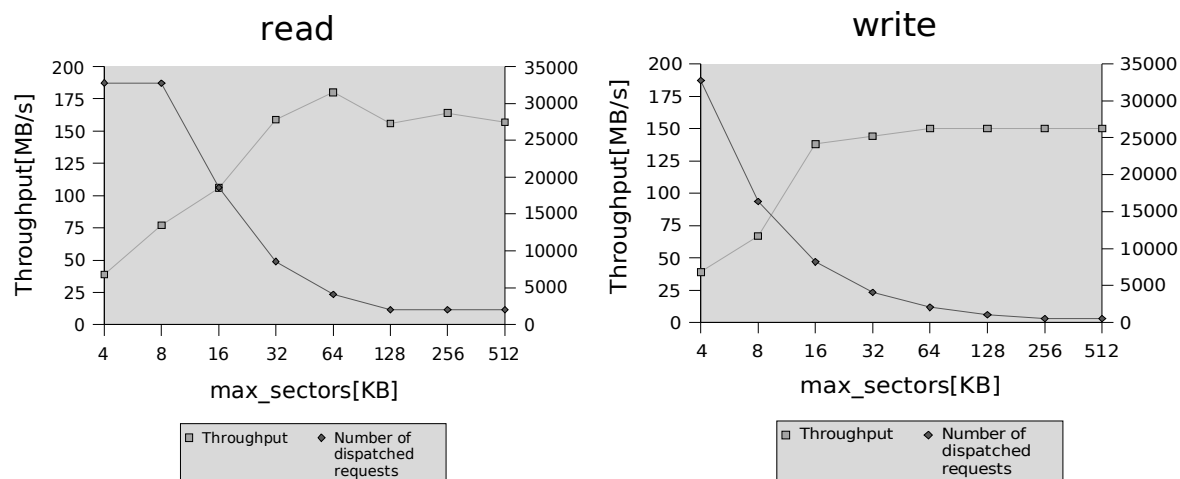
Command: `dd if=/dev/<dev|zero> of=/dev/<null|dev> bs=16777216 count=8`

Figure 6: Performance effects of I/O merging

`io()` function. This separation creates a problem for error processing, because ***Request*-based DM** does not do a deep copy of the ***request***'s BIOs. As a result, if there is an error, `__end_that_request_first()` will end up calling the BIO's `bi_end_io()` callback and returning the BIO to the upper layer, before DM has a chance to retry it.

A simple solution might be to add a new hook in `__end_that_request_first()`, where the new hook is called before a BIO is completed. DM would then be responsible for completing the BIO when it was ready. However, it just imposes additional complexity on DM because DM needs to split its completion handler into error checking and retrying as the latter still has to wait for `end_that_request_last()`. It is nothing more than a workaround for the lack of ***request*** stacking.

True *request* stacking

To solve the problem, a redesign of the block layer ***request*** completion interfaces, so that they function like BIO stacking, is necessary. To accomplish this, ***Request*-based DM** implements the following:

- Provide a generic completion handler for ***request***s which are not using their `end_io()` in the current code. `__make_request()` will set the default handler.

- Modify existing `end_io()` users to handle the new behavior.

- Provide a new helper function for device drivers to complete a ***request***. The function eventually calls the ***request***'s `end_io()` function. Device drivers have to be modified to call the new helper function for ***request*** completion. (`end_that_request_*()` are no longer called from device drivers.) If the helper function returns 0, device drivers can assume the ***request*** is completed. If the helper function returns 1, device drivers can assume the ***request*** is not completed and can take actions in response.

3.4   User-space DM interface

`dm-multipath` has a wide range of tools like `multipath-tools` and installers for major distributions. To minimize headaches caused by changing the multipath infrastructure, not only did ***Request*-based** multipath hook into DM, but it also reused the BIO-based `dm-multipath` target. As a result, the only change to the user-space DM interface is a new flag for the DM device creation `ioctl`. The existence of the flag is checked at the ioctl handler and, if it is turned on, the device will be set up for ***Request*-based DM**.
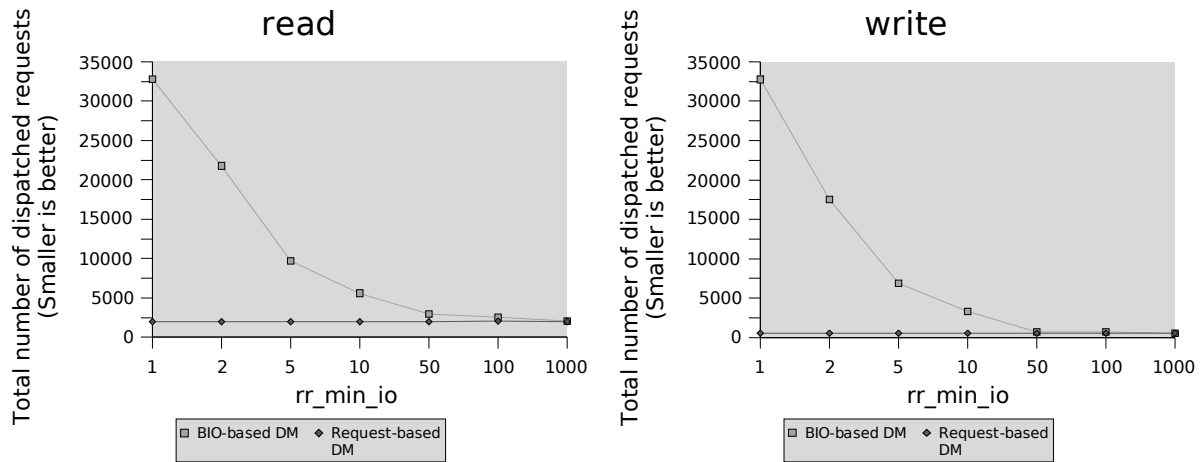
Command: `dd if=/dev/<dev|zero> of=/dev/<null|dev> bs=16777216 count=8`

Figure 7: Effects of frequent path changes on I/O merging

4 Performance testing

One of the main purposes of **_Request_**-based `dm-multipath` is to reduce the total number of **_request_**s dispatched to underlying devices even when path change occurs frequently.

In this section, performance effects of I/O merging and how **_Request_**-based `dm-multipath` reduces the number of **_request_**s under frequent path change are shown. The test environment used for the measurement is shown in Table 1.

Host	CPU	Intel Xeon 1.60[GHz]
	Memory	2[GB]
	FC HBA	Emulex LPe1150-F4 * 2
Storage	Port	4[Gbps] * 2
	Cache	4[GB]
Switch	Port	2[Gbps]

Table 1: Test environment

4.1 Effects of I/O merging

Sequential I/O results are shown in Figure 6. The throughput for a fixed amount of reads and writes on a local block device was measured using the `dd` command while changing the queue's `max_sectors` parameter.

In the test setup, the Emulex driver's max segment size and max segment count are set high enough, so that

`max_sectors` controls the **_request_** size. It is expected that when the value of `max_sectors` becomes smaller, the number of dispatched **_request_**s becomes larger. The results in Figure 6 appear to confirm this, and indicate that I/O merging, or at least larger I/Os, is important to achieve higher throughput.

4.2 Effects of **_Request_**-based `dm-multipath` on I/O merging

While the results shown in the previous section are obtained by artificially reducing the `max_sectors` parameter, such situations can happen when frequent path changes occur in BIO-based `dm-multipath`.

Testing results for the same sequential I/O pattern on a `dm-multipath` device when changing round-robin path selector's `rr_min_io` parameter which corresponds to the frequency of path change is shown in Figure 7.

This data shows that, when path change occurs frequently, the total number of **_request_**s increases with BIO-based `dm-multipath`. While under the same condition, the number of **_request_**s is low and stable with **_Request_**-based `dm-multipath`.

4.3 Throughput of **_Request_**-based `dm-multipath`

At this point in time, **_Request_**-based `dm-multipath` still cannot supersede BIO-based `dm-multipath` in sequential read performance with a simple round-robin
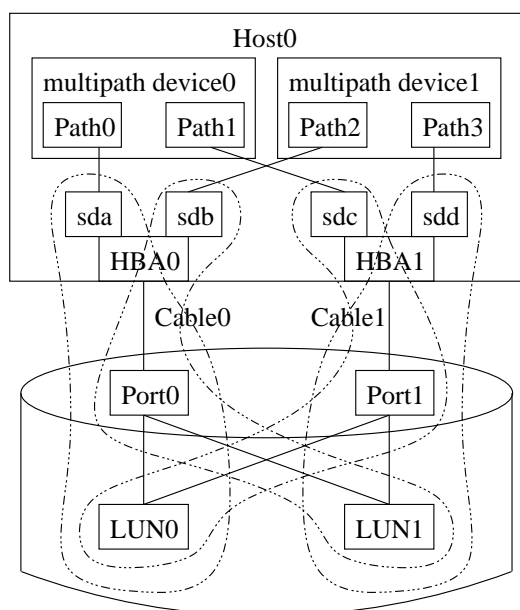
Figure 8: Examples of targets sharing cables

path selector. The performance problem is currently under investigation.

5 Dynamic load balancing

The benefit of moving multipath layer below the I/O scheduler is not only for the efficiency of I/O merging.

This section reviews the other important feature of **Request**-based `dm-multipath`, dynamic load balancing.

5.1 Needs of dynamic load balancing

There are two load balancing types in general, static and dynamic. `dm-multipath` currently supports a static balancer with weighted round-robin. It may be sufficient in an ideal environment where all paths and storage controllers are symmetric and private to the host system. But in an environment where the load of each path is not same or dynamically changes, round-robin does not work well.

For example, suppose there is a multipath configuration described in Figure 8 and `Path0(sda)` is being used heavily for `multipath device0`. It means `Cable0` is heavily loaded. In this situation, `multipath device1` should use `Path3(sdd)` to avoid heavily loaded `Cable0`. However, the round-robin path selector

does not care about that, and will select `Path2(sdb)` at next path selection for `multipath device1`. It will cause congestion on `Cable0`.

To get better performance even in such environments, a dynamic load balancer is needed.

5.2 Load parameters

It is important to define good metrics to model the load of a path. Below is an example of parameters which determine the load of a path.

- Number of in-flight I/Os;

- Block size of in-flight I/Os;

- Recent throughput;

- Recent latency; and

- Hardware specific parameters.

Using information such as the number of in-flight I/Os on a path would be the simplest way to gauge traffic. For BIO-based DM, it was described in Section 2.2 the unit of I/O is the BIO, but BIO counters do not take into account merges. **Request**-based DM, on the other hand, is better suited for this type of measurement. Its use of **request**s allows it to measure traffic in the same units of I/O that are used by lower layers, so it is possible for **Request**-based DM to take into account a lower layer's limitations like HBA, device, or transport queue depths.

Throughput or latency is another valuable metric. Many lower layer limits are not dynamic, and even if they were, could not completely account for every bottleneck in the topology. If we used throughput as the load of a path, path selection could be done with the following steps.

1. Track block size of in-flight I/Os on each path . . . *in-flight size*.

2. At path selection time, calculate recent throughput of each path by using generic diskstats (`sectors` and `io_ticks`) . . . *recent throughput*.

3. For each path, calculate the time which all in-flight I/Os will finish by using (*in-flight size*)/ (*recent throughput*).

4. Select the path of which Step 3 is the shortest time.

We plan to implement such dynamic load balancers after resolving the performance problems of Section 4.3.

6 Conclusion

Request-based multipath has some potential improvements over current `dm-multipath`. The paper focused on the I/O merging, which affects load balancing, and confirmed the code works correctly.

There is a lot of work to be done to modify the block layer so that it can efficiently and elegantly handle routing ***request***s. And there are a lot of interesting directions the path selection modules can take, because the multipath layer is now working in the same units of I/O that the storage device and LLD are.

References

[1] Edward Goggin, Linux Multipathing *Proceedings of the Linux Symposium*, 2005.

[2] Qlogic, Qlogic QL4xxx QL2xxx Driver README, `http://www.qlogic.com`.

[3] Mike Anderson, SCSI Mid-Level Multipath, *Proceedings of the Linux Symposium*, 2003.

[4] Jens Axboe, Notes on the Generic Block Layer Rewrite in Linux 2.5, *Documentation/block/ biodoc.txt*, 2007.

[5] Mike Christie, [PATCH RFC] block layer (request based) multipath, `http://lwn.net/Articles/156058/`.

[6] Jonathan Corbet, Driver porting: low-level memory allocation, `http://lwn.net/Articles/22909/`.

Short-term solution for 3G networks in Linux: umtsmon

Klaas van Gend

MontaVista Software, Inc.

`klaas.van.gend@mvista.com`

Abstract

When not at home and in need of Internet access, one can search for an open Wifi access point. But if there is none available, you'll have to set up a connection through a mobile network—GPRS, EDGE, UMTS, HSDPA, or WCDMA networks. For laptops, there are special PCMCIA cards that can do that; most mid- or high-end mobile phones can do this through USB or Bluetooth as well. To manage such a connection, one needs special software—it works differently from a regular phone dial-up, Wifi, or Ethernet connection.

`umtsmon` was created to address this need. The author wanted to have a simple tool that works for all current Linux distributions. `umtsmon` is not the final mobile network manager application—at some point, the functionality of umtsmon will have to be integrated into Network Manager and its GUI apps. But umtsmon will definitely serve as a playground to find out what users really want, so only the really used features will be implemented the right way into Network Manager. Also: umtsmon is available *now* as a simple download and run—it is usable for existing Linux users. The integrated Network Manager will only help future distributions.

In this talk, Klaas will discuss mobile networks in general, how the various brands of pccards work, and what umtsmon can and cannot do yet.

1 Short history of 3G networks

For Western European consumers, mobile communication networks started in the early 80s. Back then, most countries had their own analog standards and equipment was heavy—mainly due to the batteries and antennas. It was also rather easy to listen into conversations and it was expensive.

The organisation GSM, Groupe Spécial Mobile, was started in 1982 by the joint European telecom operators to address the issues. By the end of the eighties, the GSM standard was close to finished and was tranferred to a standardization body, ETSI. The new standard used digital transmission, performed frequency hopping, and prevented tapping of conversations.
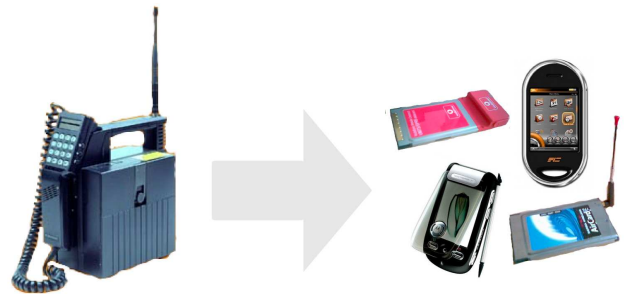
The first network to go live was in Finland in 1991; by the end of 1993, networks were operational in 48 countries with a total of one million subscribers.

The popularity of GSM surprised many—by now every person in the Netherlands (including newlyborns and the elderly) owns more than one mobile handset. This unexpected surge in users caused telecom operators to start hunting for expansion of their networks—they feared overloading of their networks.

Also, GSM features a direct tunnel from handset to local cell antenna, and users are charged for the duration of the connection. This is less useful for data connections, so an extension to the GSM standard was made: GPRS. This allowed for a packet-switching-based connection, therefore not requiring a tunnel. Charging per amount of data was possible. However, GSM/GPRS only allows for small bandwidths—typically comparable to old analog modem speeds—max 28k8 kbits/s if you are lucky.

UMTS was created to address these two needs—relieving the load of the GSM networks and addressing higher bandwidth data communication. Different radio technology required new antennas, thus requiring new

radio frequencies and equipment. Governments across Europe saw their chance and auctioned the new radio frequencies for astronomic sums of money. Several telecom operators nearly went bankrupt because they were forced to buy into expensive frequencies in several countries.

A deafening silence followed.

UMTS requires a lot of antennas. To get a decent coverage, there are far more antennas required for UMTS than there are for a normal GSM network. The year 2004 through the first half 2006 saw a heated debate on the dangers of radio transmitters in cell phones and antennas to the public health. All kind of research either suggested that one would get sick from living close to an antenna, or would get a heated brain from using a cell phone. Also some people advised against wearing a cell in one's trouser pockets as it might reduce virility (?!). This made several local municipalities refuse UMTS antennas within their territory. These "white spots" in UMTS coverage might endanger the timely roll-out of the UMTS networks...

So, telecom operators invested in frequencies, invested in equipment. And just some Internet junkies bought into it and bought PCCards for laptops or expensive phones that used the new network. At the same time, most operators added new GSM antennas to their new UMTS stations. By creating more cells, the average number of users in a cell decreased—the GSM technology was saved from overload.

PCCards and especially Blackberries were the first killer apps for UMTS—these devices use UMTS (if available) to get their users access to their e-mail. The reduction of the rates helped adoption of UMTS, but still many consider it too expensive. Nowadays, most telecom operators are on break-even for their 3G networks—when not counting in payment of the original radio licenses.

By now, telecom operators see the danger of competing technologies like WiMax and Wifi. So they are moving their UMTS networks forward to HSPA (HSDPA and HSUPA) which require yet new devices.

2 Analysis of a PCMCIA UMTS card

All telecom operators sell PCCards for laptops. However, all are OEM products, manufactured by small specialised companies like Option (Belgium), Novatel Wireless (USA), 3GSystems (Germany) and Sierra Wireless (USA).
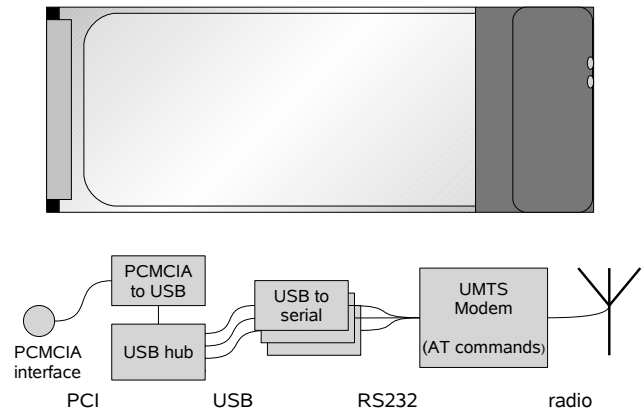
Figure 1: a PCMCIA card and its contents

The first generation of UMTS cards as pioneered by Option, have an interesting hardware architecture as depicted in Figure 1. If one inserts the card into a laptop running Linux, Linux will recognize a new USB interface. Three (or four) usbserial devices are connected (via a hub) to that USB interface. Standard Linux kernels do not recognize the VendorID/Product ID for usbserial, but if forced to load by hand, three new serial ports appear on `/dev/ttyUSBx`.

Most other vendors have cards with similar design—this is mainly due to the fact that there are very limited manufacturers of chipsets for UMTS data. In most cases, this is QualComm.

QualComm decided to go a different route for the HSDPA cards. No longer serial2usb interfaces, but specialised connections that require a specialised driver. With help from other people, Paul Hardwick from Option ported the existing Windows driver to Linux. This driver is now known as the Nozomi driver. Its path to inclusion in the Linux kernel is interesting—of course it was rejected for inclusion because it was not written the "Linux way." With help from Greg Kroah-Hartman, the driver is taking shape now.

3 Analysis of a "Zero CD" USB UMTS brick

To reduce packaging costs, some vendors now ship a technology called "ZeroCD." It was pioneered by Option in their ICON USB box (see Figure 2), but there are also PCCards available. Essential is that the device
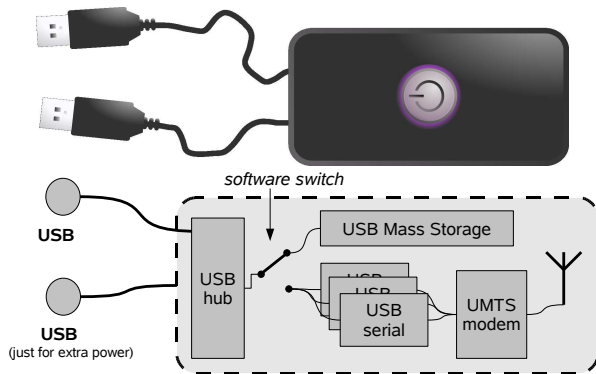
Figure 2: a "ZeroCD" USB box and its contents

boots up as a USB Mass Storage Device. For Windows users, an `autorun.inf` file will automatically take care of installing the software. Afterwards, the driver will send a magic string to the USB Hub. The USB Hub then disconnects the Mass Storage and connects the modem ports, either through USBSerial or the above mentioned Nozomi. For this type of device, vendors do not need to ship installation CDs. For users, the installation is simplified: just plugging it in is enough.

The major disadvantage of the current ZeroCD chipset is the power consumption—it requires two USB plugs because one USB plug can only carry 5W... Be careful and remove the device if your laptop runs on battery power!

4   Back to the AT commands era

Whether the card supports serial or nozomi, after loading the right driver, you will run into serial ports on `/dev/ttyUSBx`, `/dev/ttyACMx`, or `/dev/nozomiX`.

Adventurous Penguins immediately type `minicom /dev/ttyUSB0` to see what's on the serial devices. Hopefully, they are old enough to remember the good old Hayes-compatible modem era, where one could play with AT Commands and PPP settings for ages before the first connection to the outside world was successful.

Actually, they need to—because that's exactly what you get: all three interfaces are connected to an UMTS modem with an AT Command set. However, standardization committee 3GPP has standardized the AT commands for 3G network modems, so there are standardized commands to talk to the modem to enter PIN codes,

select the network of a mobile operator, send an SMS, and such. There are multiple interfaces to the modem to allow one interface to be used by PPP for the actual network connection, whereas one can use another interface to send AT commands to retrieve the status of the modem, network, or SIM card.

A few examples of new AT commands:

- `AT+COPS=?` will (after 30 seconds) return a list of all mobile networks that are available, with info on which networks you are allowed to connect to.

- `AT+CPIN="1234"` to enter a PIN code for the smart card.

- `AT+CSQ` returns the signal strength of the mobile connection.

So we're stuck with AT interfaces. Let's re-install ppp and go back to the old days. Or install umtsmon—which will interact with the modem and do all the AT commands for you.

5   Single Serial Port devices

The most popular cards in Europe all have multiple interfaces. This means that we can, for example, use `/dev/ttyUSB0` to connect PPP to, whilst we can simultaneously send AT commands to `/dev/ttyUSB2`. However, some cards (like the Sony Ericsson GC79, most Sierra Wireless cards, and some of the Novatels) only have a single serial port. This means that AT commands and PPP data need to share the same port. Technically, this is not a problem—software like `kppp` implements this functionality already. However, the Open-Moko project also spawned a new project called the GSM Daemon that also can do this. This might be an interesting road for the future—at the cost of another external dependency.

6   umtsmon system design—dependencies

As stated before, umtsmon was designed to work on as wide a range of Linuxes as possible. This is why we attempt to make as few assumptions about the operating system as possible. Yet we have to rely on a few packages to be present:

- PPP
  We need PPP to make the actual network connection and change routing and such.

- QT3
  The QT library is needed for the UI. `umtsmon` cannot run without it. QT in itself also has a few dependencies, like X.

For versions of umtsmon beyond 0.6, we probably will add a few more requirements:

- pcmcia-utils
  This one obviously is not necessary for the ICON and other USB-only devices. We want to use pcmcia-utils to enable users to reset the card (`pccard eject` and `pccard insert`) and/or to simplify the autodetection code.

- libusb
  At this moment, a lot of the autodetection is done by browsing through the `/sys` filesystem. This is rather complex to code consistently for all Linux kernel revisions. Using libusb should solve that problem for us and again simplify the autodetection code. It might be wise to move the libusb-dependent code into a shared library that is `dlopen()`ed at runtime to prevent umtsmon from trying to run if libusb isn't present or if it is too old to work.

- icon_switch
  This is a small utility that switches the ICON box from mass storage to modem operation. Unfortunately, it is rather unreliable and it needs extensions for the other ZeroCD devices that have different USB IDs and may require different code sequences. `umtsmon` at this moment requires PPP to be SUID—umtsmon calls pppd directly with arguments controlling the connection. `umtsmon` will complain to the user if PPP is not set SUID and ask the user if it is allowed to fix it. This is a security hole: in theory people could start writing malicious dialers dialing expensive foreign numbers. This should be addressed in a future revision by having umtsmon create profiles in the `/etc/ppp/peers/` directory.

7 umtsmon software design

Internally, umtsmon is written to follow the MVC design pattern. MVC means Model View Controller. Basically, this comes down to a separation of concerns—a class should only contain code that represents data (=model), changes data (=controller), or displays it (=view).

Central in the umtsmon 0.6 design are the classes Query, SerialPort, ConnectionInfo, and PPPConnection. Any AT Command sequence that is to be sent to the card is represented as a Query instance. The Query class also contains rudimentary parsing and will strip off echos and such. Query connects to a SerialPort instance for the actual communication.

The following paragraphs discuss some details of the design of umtsmon.

7.1 PPPConnection

PPPConnection is the beating heart of the application. As can be seen in Figure 3, the main GUI class MainWindow subscribes one of its attributes, the MainWindowPPPObserver, to receive any state changes of the PPP daemon. If someone outside umtsmon or umtsmon itself then starts the ppp daemon to make a connection, the PPPConnection class with call all its attached Observers to notify the state changes. MainWindow responds to that by enabling/disabling buttons and menu items.

At this moment, only MainWindow is subscribed to receive the PPP state changes. This will change in the near future when we start talking about the NetworkManager integration—that will require another Observer to the PPP state.

7.2 (Inhibiting) ConnectionInfo

ConnectionInfo regularly polls the card to ask for the mobile operator, signal strength, and such. On some occasions, ConnectionInfo must be prevented from sending out Queries, like during the PPP connection setup or whilst `AT+COPS=?` (see Section 4) is running. In such cases, the PPPConnection or NetworkChanger class just creates a ConnectionInfoInhibitor instance. Creation of the Inhibitor instance will increase a counter inside
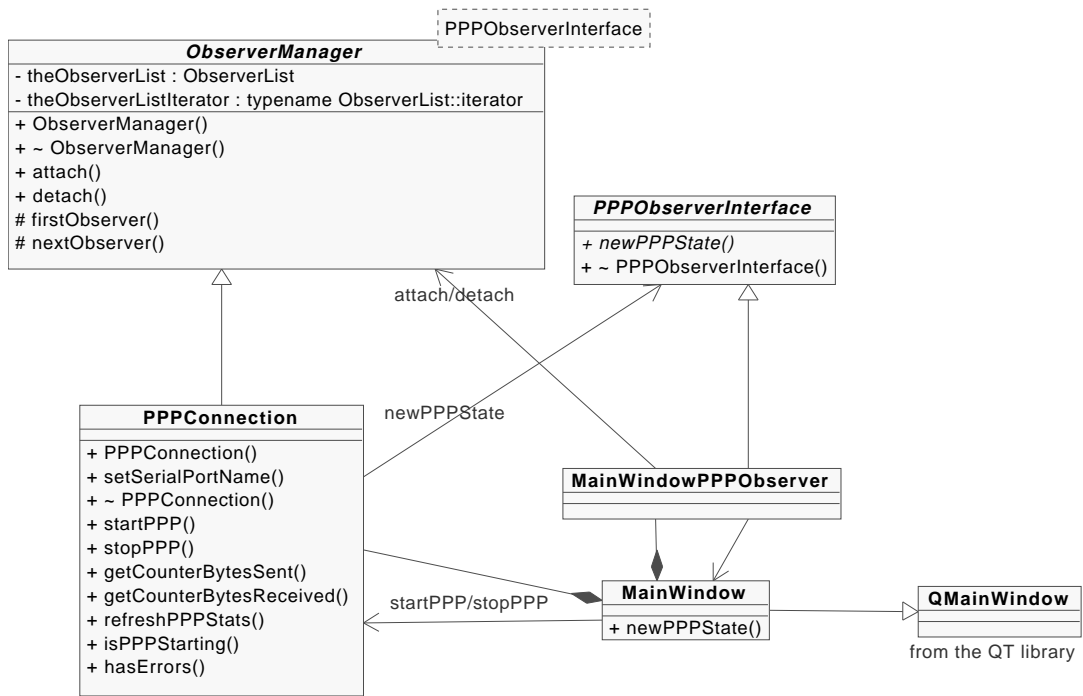
Figure 3: Class Diagram of PPPConnection and interacting classes
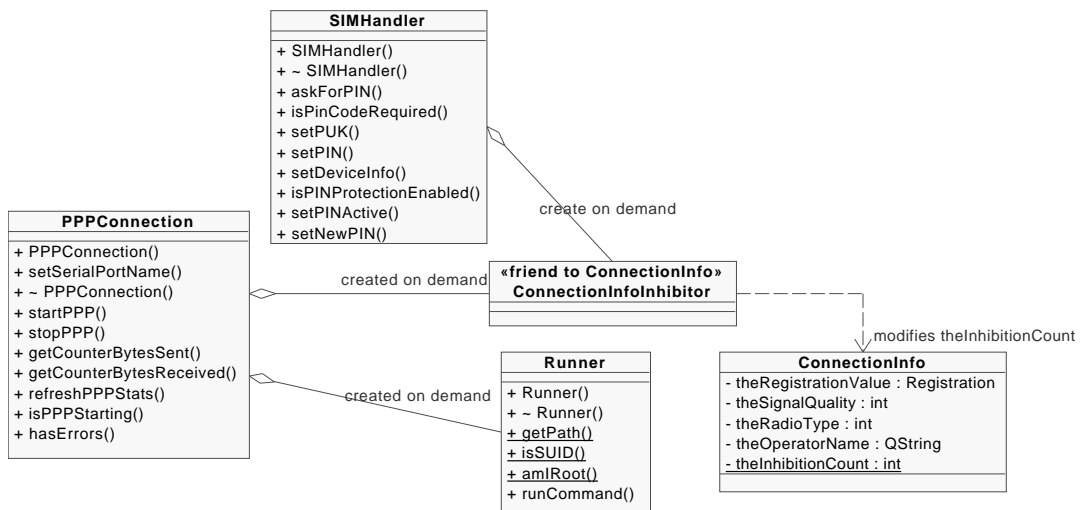
Figure 4: Class Diagram of ConnectionInfo and interacting classes
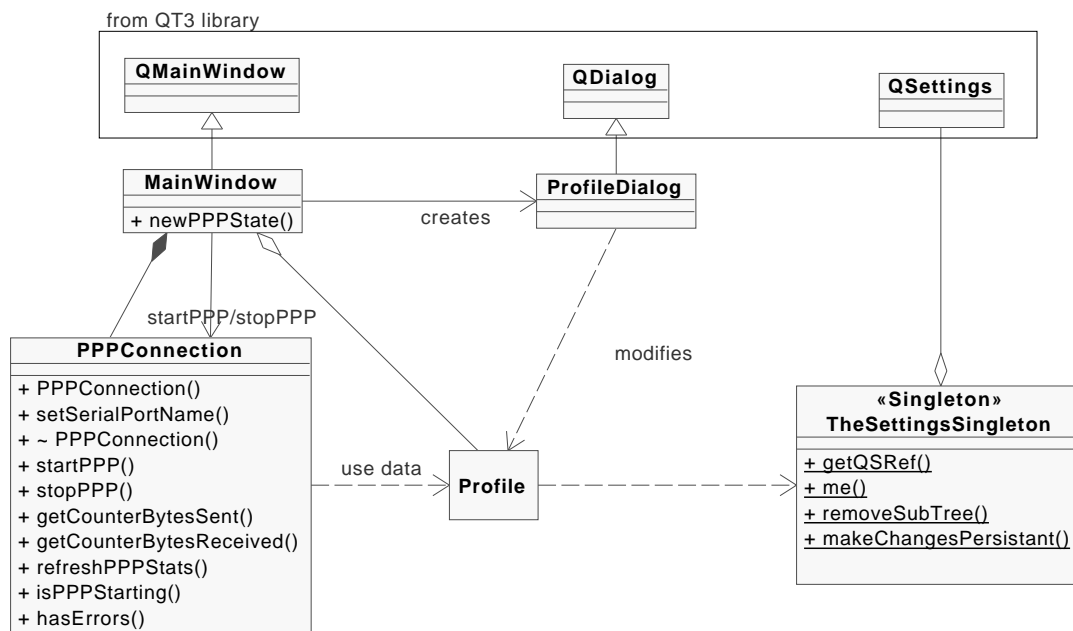
Figure 5: Class Diagram of Profile and interacting classes

ConnectionInfo; upon destruction, the counter will automatically be decreased. PPPConnection uses an instance of the Runner class to manage the execution of `/usr/sbin/pppd`. This is shown in Figure 4. In the case of single serial port cards, the AT commands and the PPP data stream need to be sent over the same serial port. In that case, ConnectionInfo never can run when a PPP connection exists.

7.3 Profile Management

Refer to Figure 5 for a class diagram. Once the user clicks on the connect button in MainWindow, PPPConnection gets called with a reference to a Profile. PPPConnection will use the info inside the Profile class to setup the connection. To change a profile, the user selects Profile Management in the menu in the GUI. The ProfileDialog will be instantiated and filled with the data from the active Profile. Users then can choose to create another profile, change the current one, etc. The data is stored to disk in a `key=data` formatted `~/.umtsmon/umtsmonrc` file. The QSettings class takes care of that. Because settings need to be accessed throughout the program, possibly even before `main()` is started, settings are always retrieved through a Singleton pattern class. This also solves the issue that a QSettings class only saves its data upon destruction. Call-

ing `makeChangesPersistent()` will thus cause the QSettings instance to be destroyed.

8 NetworkManager integration and/or takeover

High on the wish list is to integrate at least a little with NetworkManager. The big annoyance is that at the moment, even if a UMTS connection is made, software like Gaim and Firefox will refuse to connect because according to them there is no connection. Apparently they use libnm to ask NetworkManager if the system is on-line or not. As stated before, umtsmon should run, regardless of NetworkManager's presence. However, this is only loose coupling—we're not discussing adding UMTS support to Network Manager yet. In the end, UMTS connections should be just another item that is implemented in NetworkManager and its GUIs. We currently view umtsmon as a playing ground for that. What features are actually used? How to implement stuff? Where to put security constraints? It all is easier to do in a small program than in the collection of binaries that makes up NetworkManager. Yet NetworkManager is the future—it is the most convenient way for users to manage yet another networking connection. It remains to be seen if the current umtsmon team will actually do the NetworkManager implementation.

Brand	model	type	remarks
Sony Ericsson	GC79	single-port serial	GPRS and EDGE only
Option	GT GPRS EDGE		
Option	3G Quad	three of four port usb2serial	need kernel module usbserial.ko with parameters or the specialised option kernel module.
Huawei	E612		
Option		nozomi	need nozomi kernel module
Option	ICON	external usb box	ZeroCD chipset, need switching
3GSystems	XSPlug3		
Sierra Wireless	7xx series	PCMCIA serial modem	single serial port
Novatel	U630/U530		
Novatel	XU870	dual serial port, only first usable	first Expresscard to be supported
Kyocera	KPC650	dual serial port	serial ports don't communicate.
various mobile phones	various	connect through either serial, USB or Bluetooth	handled as a single serial port card

Table 1: Hardware support of umtsmon

9 Device, Commercial and Distribution support

At this moment, umtsmon supports a wide variety of hardware. The 0.6 release will support devices from Novatel, 3G Systems, Sony Ericsson, Option, Sierra Wireless, and Huawei. Also mobile phones that have a laptop connection through USB, Bluetooth, or serial can be supported, with a few limitations. See Table 1 for a more complete list.

None of the device vendors is cooperating with the development, however. The development team is currently investigating creating a fund to buy all available hardware and distribute it amongst the developers to ensure that all hardware is supported and remains operational.

Network operator T-Mobile Germany sponsored the development by providing a laptop and devices to one of the developers, who happened to also have done an internship on UMTS on Linux for T-Mobile. The internship resulted in a lot of improvements to umtsmon, thanks Christofer!

At the moment of writing this paper, umtsmon is available as a standard package in OpenSuse (starting umtsmon 0.3 in OpenSuse 10.2) and Gentoo (starting with umtsmon 0.5, currently in ~amd64 and ~x86 only).

10 Conclusions

- Support for UMTS cards is in the same position as hardware enablement projects like ALSA were a few years ago. Several devices work—mostly the devices owned by the developers. Manufacturers don't see the need to cooperate yet, nor to fund development.

- All known 3G mobile devices implement serial interfaces, either directly or through drivers.

- UMTS is just another radio technology reusing existing communication standards: AT commands.

- umtsmon was created to handle the AT commands exchange for the user and start the PPP daemon.

- umtsmon is not really integrated into Linux—it's a standalone application.

- NetworkManager integration should start from scratch, possibly inheriting the GSM multiplexing daemon from OpenMoko to solve the single serial port problem correctly.

- Writing a paper for OLS is a good stimulus for coders to finally write down some parts of their software design.

11 Links

The umtsmon website:
`http://umtsmon.sourceforge.net/`

PharScape (HOWTOs and support forum for all Option
based cards and the Nozomi drivers):
`http://www.pharscape.org/`

The 3GPP approved AT command set:
`http://www.3gpp.org/ftp/Specs/latest/`
`Rel-7/27_series/`

The GFS2 Filesystem

Steven Whitehouse

Red Hat, Inc.

swhiteho@redhat.com

Abstract

The GFS2 filesystem is a symmetric cluster filesystem designed to provide a high performance means of sharing a filesystem between nodes. This paper will give an overview of GFS2's make subsystems, features and differences from GFS1 before considering more recent developments in GFS2 such as the new on-disk layout of journaled files, the GFS2 metadata filesystem, and what can be done with it, fast & fuzzy statfs, optimisations of `readdir/getdents64` and optimisations of glocks (cluster locking). Finally, some possible future developments will be outlined.

To get the most from this talk you will need a good background in the basics of Linux filesystem internals and clustering concepts such as quorum and distributed locking.

1 Introduction

The GFS2 filesystem is a 64bit, symmetric cluster filesystem which is derived from the earlier GFS filesystem. It is primarily designed for Storage Area Network (SAN) applications in which each node in a GFS2 cluster has equal access to the storage. In GFS and GFS2 there is no such concept as a metadata server, all nodes run identical software and any node can potentially perform the same functions as any other node in the cluster.

In order to limit access to areas of the storage to maintain filesystem integrity, a lock manager is used. In GFS2 this is a distributed lock manager (DLM) [1] based upon the VAX DLM API. The Red Hat Cluster Suite provides the underlying cluster services (quorum, fencing) upon which the DLM and GFS2 depend.

It is also possible to use GFS2 as a local filesystem with the `lock_nolock` lock manager instead of the DLM. The locking subsystem is modular and is thus easily substituted in case of a future need of a more specialised lock manager.

2 Historical Detail

The original GFS [6] filesystem was developed by Matt O'Keefe's research group in the University of Minnesota. It used SCSI reservations to control access to the storage and ran on SGI's IRIX.

Later versions of GFS [5] were ported to Linux, mainly because the group found there was considerable advantage during development due to the easy availability of the source code. The locking subsystem was developed to give finer grained locking, initially by the use of special firmware in the disk drives (and eventually, also RAID controllers) which was intended to become a SCSI standard called dmep. There was also a network based version of dmep called memexp. Both of these standards worked on the basis of atomically updated areas of memory based upon a "compare and exchange" operation.

Later when it was found that most people preferred the network based locking manager, the Grand Unified Locking Manager, gulm, was created improving the performance over the original memexp based locking. This was the default locking manager for GFS until the DLM (see [1]) was written by Patrick Caulfield and Dave Teigland.

Sistina Software Inc, was set up by Matt O'Keefe and began to exploit GFS commercially in late 1999/early 2000. Ken Preslan was the chief architect of that version of GFS (see [5]) as well as the version which forms Red Hat's current product. Red Hat acquired Sistina Software Inc in late 2003 and integrated the GFS filesystem into its existing product lines.

During the development and subsequent deployment of the GFS filesystem, a number of lessons were learned about where the performance and administrative problems occur. As a result, in early 2005 the GFS2 filesystem was designed and written, initially by Ken Preslan

and more recently by the author, to improve upon the original design of GFS.

The GFS2 filesystem was submitted for inclusion in Linus' kernel and after a lengthy period of code review and modification, was accepted into 2.6.16.

3   The on-disk format

The on-disk format of GFS2 has, intentionally, stayed very much the same as that of GFS. The filesystem is big-endian on disk and most of the major structures have stayed compatible in terms of offsets of the fields common to both versions, which is most of them, in fact.

It is thus possible to perform an in-place upgrade of GFS to GFS2. When a few extra blocks are required for some of the *per node* files (see the metafs filesystem, Subsection 3.5) these can be found by shrinking the areas of the disk originally allocated to journals in GFS. As a result, even a full GFS filesystem can be upgraded to GFS2 without needing the addition of further storage.

3.1   The superblock

GFS2's superblock is offset from the start of the disk by 64k of unused space. The reason for this is entirely historical in that in the dim and distant past, Linux used to read the first few sectors of the disk in the VFS mount code before control had passed to a filesystem. As a result, this data was being cached by the Linux buffer cache without any cluster locking. More recent versions of GFS were able to get around this by invalidating these sectors at mount time, and more recently still, the need for this gap has gone away entirely. It is retained only for backward compatibility reasons.

3.2   Resource groups

Following the superblock are a number of resource groups. These are similar to ext2/3 block groups in that their intent is to divide the disk into areas which helps to group together similar allocations. Additionally in GFS2, the resource groups allow parallel allocation from different nodes simultaneously as the locking granularity is one lock per resource group.

On-disk, each resource group consists of a header block with some summary information followed by a number

Bit Pattern	Block State
00	Free
01	Allocated non-inode block
10	Unlinked (still allocated) inode
11	Allocated inode

Table 1: GFS2 Resource Group bitmap states

of blocks containing the allocation bitmaps. There are two bits in the bitmap for each block in the resource group. This is followed by the blocks for which the resource group controls the allocation.

The two bits are nominally *allocated/free* and *data (non-inode)/inode* with the exception that the *free inode* state is used to indicate inodes which are unlinked, but still open.

In GFS2 all metadata blocks start with a common header which includes fields indicating the type of the metadata block for ease of parsing and these are also used extensively in checking for run-time errors.

Each resource group has a set of flags associated with it which are intended to be used in the future as part of a system to allow in-place upgrade of the filesystem. It is possible to mark resource groups such that they will no longer be used for allocations. This is the first part of a plan that will allow migration of the content of a resource group to eventually allow filesystem shrink and similar features.

3.3   Inodes

GFS2's inodes have retained a very similar form to those of GFS in that each one spans an entire filesystem block with the remainder of the block being filled either with data (a "stuffed" inode) or with the first set of pointers in the metadata tree.

GFS2 has also inherited GFS's equal height metadata tree. This was designed to provide constant time access to the different areas of the file. Filesystems such as ext3, for example, have different depths of indirect pointers according to the file offset whereas in GFS2, the tree is constant in depth no matter what the file offset is.

Initially the tree is formed by the pointers which can be fitted into the spare space in the inode block, and is then

grown by adding another layer to the tree whenever the current tree size proves to be insufficient.

Like all the other metadata blocks in GFS2, the indirect pointer blocks also have the common metadata header. This unfortunately also means that the number of pointers they contain is no longer an integer power of two. This, again, was to keep compatibility with GFS and in the future we eventually intend to move to an extent based system rather than change the number of pointers in the indirect blocks.

3.3.1 Attributes

GFS2 supports the standard get/change attributes `ioctl()` used by ext2/3 and many other Linux filesystems. This allows setting or querying the attributes listed in Table 2.

As a result GFS2 is directly supported by the `lsattr(1)` and `chattr(1)` commands. The hashed directory flag, `I`, indicates whether a directory is hashed or not. All directories which have grown beyond a certain size are hashed and section 3.4 gives further details.

3.3.2 Extended Attributes & ACLs

GFS2 supports extended attribute types *user*, *system* and *security*. It is therefore possible to run selinux on a GFS2 filesystem.

GFS2 also supports POSIX ACLs.

3.4 Directories

GFS2's directories are based upon the paper "Extendible Hashing" by Fagin [3]. Using this scheme GFS2 has a fast directory lookup time for individual file names which scales to very large directories. Before ext3 gained hashed directories, it was the single most common reason for using GFS as a single node filesystem.

When a new GFS2 directory is created, it is "stuffed," in other words the directory entries are pushed into the same disk block as the inode. Each entry is similar to an ext3 directory entry in that it consists of a fixed length part followed by a variable length part containing the file name. The fixed length part contains fields to indicate the total length of the entry and the offset to the next entry.

Once enough entries have been added that it's no longer possible to fit them all in the directory block itself, the directory is turned into a hashed directory. In this case, the hash table takes the place of the directory entries in the directory block and the entries are moved into a directory "leaf" block.

In the first instance, the hash table size is chosen to be half the size of the inode disk block. This allows it to coexist with the inode in that block. Each entry in the hash table is a pointer to a leaf block which contains a number of directory entries. Initially, all the pointers in the hash table point to the same leaf block. When that leaf block fills up, half the pointers are changed to point to a new block and the existing directory entries moved to the new leaf block, or left in the existing one according to their respective hash values.

Eventually, all the pointers will point to different blocks, assuming that the hash function (in this case a CRC-32) has resulted in a reasonably even distribution of directory entries. At this point the directory hash table is removed from the inode block and written into what would be the data blocks of a regular file. This allows the doubling in size of the hash table which then occurs each time all the pointers are exhausted.

Eventually when the directory hash table hash reached a maximum size, further entries are added by chaining leaf blocks to the existing directory leaf blocks.

As a result, for all but the largest directories, a single hash lookup results in reading the directory block which contains the required entry.

Things are a bit more complicated when it comes to the `readdir` function, as this requires that the entries in each hash chain are sorted according to their hash value (which is also used as the file position for `lseek`) in order to avoid the problem of seeing entries twice, or missing them entirely in case a directory is expanded during a set of repeated calls to `readdir`. This is discussed further in the section on future developments.

3.5 The metadata filesystem

There are a number of special files created by `mkfs.gfs2` which are used to store additional metadata related to the filesystem. These are accessible by

Attribute	Symbol	Get or Set
Append Only	a	Get and set on regular inodes
Immutable	i	Get and set on regular inodes
Journaling	j	Set on regular files, get on all inodes
No atime	A	Get and set on all inodes
Sync Updates	S	Get and set on regular files
Hashed dir	I	Get on directories only

Table 2: GFS2 Attributes

mounting the `gfs2meta` filesystem specifying a suitable gfs2 filesystem. Normally users would not do this operation directly since it is done by the GFS2 tools as and when required.

Under the root directory of the metadata filesystem (called the master directory in order that it is not confused with the real root directory) are a number of files and directories. The most important of these is the resource index (rindex) whose fixed-size entries list the disk locations of the resource groups.

3.5.1 Journals

Below the master directory there is a subdirectory which contains all the journals belonging to the different nodes of a GFS2 filesystem. The maximum number of nodes which can mount the filesystem simultaneously is set by the number of journals in this subdirectory. New journals can be created simply by adding a suitably initialised file to this directory. This is done (along with the other adjustments required) by the `gfs2_jadd` tool.

3.5.2 Quota file

The quota file contains the system wide summary of all the quota information. This information is synced periodically and also based on how close each user is to their actual quota allocation. This means that although it is possible for a user to exceed their allocated quota (by a maximum of two times) this is in practise extremely unlikely to occur. The time period over which syncs of quota take place are adjustable via sysfs.

3.5.3 statfs

The statfs files (there is a master one, and one in each `per_node` subdirectory) contain the information required to give a fast (although not 100% accurate) result for the `statfs` system call. For large filesystems mounted on a number of nodes, the conventional approach to `statfs` (i.e., iterating through all the resource groups) requires a lot of CPU time and can trigger a lot of I/O making it rather inefficient. To avoid this, GFS2 by default uses these files to keep an approximation of the true figure which is periodically synced back up to the master file.

There is a sysfs interface to allow adjustment of the sync period or alternatively turn off the fast & fuzzy `statfs` and go back to the original 100% correct, but slower implementation.

3.5.4 inum

These files are used to allocate the `no_formal_ino` part of GFS2's `struct gfs2_inum` structure. This is effectively a version number which is mostly used by NFS, although it is also present in the directory entry structure as well. The aim is to give each inode an additional number to make it unique over time. The master inum file is used to allocate ranges to each node, which are then replenished when they've been used up.

4 Locking

Whereas most filesystems define an on-disk format which has to be largely invariant and are then free to change their internal implementation as needs arise, GFS2 also has to specify its locking with the same degree of care as for the on-disk format to ensure future compatibility.

Lock type	Use
Non-disk	mount/umount/recovery
Meta	The superblock
Inode	Inode metadata & data
Iopen	Inode last closer detection
Rgrp	Resource group metadata
Trans	Transaction lock
Flock	`flock(2)` syscall
Quota	Quota operations
Journal	Journal mutex

Table 3: GFS2 lock types

GFS2 internally divides its cluster locks (known as glocks) into several types, and within each type a 64 bit lock number identifies individual locks. A lock name is the concatenation of the glock type and glock number and this is converted into an ASCII string to be passed to the DLM. The DLM refers to these locks as resources. Each resource is associated with a lock value block (LVB) which is a small area of memory which may be used to hold a few bytes of data relevant to that resource. Lock requests are sent to the DLM by GFS2 for each resource which GFS2 wants to acquire a lock upon.

All holders of DLM locks may potentially receive callbacks from other intending holders of locks should the DLM receive a request for a lock on a particular resource with a conflicting mode. This is used to trigger an action such as writing back dirty data and/or invalidating pages in the page cache when an inode's lock is being requested by another node.

GFS2 uses three lock modes internally, *exclusive*, *shared* and *deferred*. The *deferred* lock mode is effectively another shared lock mode which is incompatible with the normal *shared* lock mode. It is used to ensure that direct I/O is cluster coherent by forcing any cached pages for an inode to be disposed of on all nodes in the cluster before direct I/O commences. These are mapped to the DLMs lock modes (only three of the six modes are used) as shown in table 4.

The DLM's `DLM_LOCK_NL` (*Null*) lock mode is used as a reference count on the resource to maintain the value of the LVB for that resource. Locks for which GFS2 doesn't maintain a reference count in this way (or are unlocked) may have the content of their LVBs set to zero upon the next use of that particular lock.

5 NFS

The GFS2 interface to NFS has been carefully designed to allow failover from one GFS2/NFS server to another, even if those GFS2/NFS servers have CPUs of a different endianness. In order to allow this, the filehandles must be constructed using the `fsid=` method. GFS2 will automatically convert endianness during the decoding of the filehandles.

6 Application writers' notes

In order to ensure the best possible performance of an application on GFS2, there are some basic principles which need to be followed. The advice given in this section can be considered a FAQ for application writers and system administrators of GFS2 filesystems.

There are two simple rules to follow:

- Make maximum use of caching

- Watch out for lock contention

When GFS2 performs an operation on an inode, it first has to gain the necessary locks, and since this potentially requires a journal flush and/or page cache invalidate on a remote node, this can be an expensive operation. As a result for best performance in a cluster scenario it is vitally important to ensure that applications do not contend for locks for the same set of files wherever possible.

GFS2 uses one lock per inode, so that directories may become points of contention in case of large numbers of inserts and deletes occurring in the same directory from multiple nodes. This can rapidly degrade performance.

The single most common question asked relating to GFS2 performance is how to run an smtp/imap email server in an efficient manner. Ideally the spool directory is broken up into a number of subdirectories each of which can be cached separately resulting in fewer locks being bounced from node to node and less data being flushed when it does happen. It is also useful if the locality of the nodes to a particular set of directories can be enhanced using other methods (e.g. DNS) in the case of an email server which serves multiple virtual hosts.

GFS2 Lock Mode	DLM Lock Mode
LM_ST_EXCLUSIVE	DLM_LOCK_EX (exclusive)
LM_ST_SHARED	DLM_LOCK_PR (protected read)
LM_ST_DEFERRED	DLM_LOCK_CW (concurrent write)

Table 4: GFS2/DLM Lock modes

6.1 `fcntl(2)` caveat

When using the `fcntl(2)` command `F_GETLK` note that although the PID of the process will be returned in the `l_pid` field of the `struct flock`, the process blocking the lock may not be on the local node. There is currently no way to find out which node the lock blocking process is actually running on, unless the application defines its own method.

The various `fcntl(2)` operations are provided via the userspace `gfs2_controld` which relies upon openais for its communications layer rather than using the DLM. This system keeps on each node a complete copy of the `fcntl(2)` lock state, with new lock requests being passed around the cluster using a token passing protocol which is part of openais. This protocol ensures that each node will see the lock requests in the same order as every other node.

It is faster (for whole file locking) for applications to use `flock(2)` locks which do use the DLM. In addition it is possible to disable the cluster `fcntl(2)` locks and make them local to each node, even in a cluster configuration for higher performance. This is useful if you know that the application will only need to lock against processes local to the node.

6.2 Using the DLM from an application

The DLM is available through a userland interface in order that applications can take advantage of its cluster locking facility. Applications can open and use lockspaces which are independent of those used by GFS2.

7 Future Development

7.1 `readdir`

Currently we have already completed some work relating to speeding up `readdir` and also considered the

way in which `readdir` is used in combination with other syscalls, such as `stat`.

There has also been some discussion (and more recently in a thread on lkml [2]) relating to the `readdir` interface to userspace (currently via the `getdents64` syscall) and the other two interfaces to NFS via the `struct export_operations`. At the time of writing, there are no firm proposals to change any of these, but there are a number of issues with the current interface which might be solved with a suitable new interface. Such things include:

- Eliminating the sorting in GFS2's `readdir` for the NFS `getname` operation where ordering is irrelevant.

- Boosting performance by returning more entries at once.

- Optionally returning `stat` information at the same time as the directory entry (or at least indicating the intent to call `stat` soon).

- Reducing the problem of `lseek` in directories with insert and delete of entries (does it result in seeing entries twice or not at all?).

7.2 inotify & dnotify

GFS2 does not support inotify nor do we have any plans to support this feature. We would like to support dnotify if we are able to design a scheme which is both scalable and cluster coherent.

7.3 Performance

There are a number of ongoing investigations into various aspects of GFS2's performance with a view to gaining greater insight into where there is scope for further improvement. Currently we are focusing upon increasing the speed of file creations via `open(2)`.

8   Resources

GFS2 is included in the Fedora Core 6 kernel (and above). To use GFS2 in Fedora Core 6, install the `gfs2-utils` and `cman` packages. The `cman` package is not required to use GFS2 as a local filesystem.

There are two GFS2 git trees available at kernel.org. Generally the one to look at is the `-nmw` (next merge window) tree [4] as that contains all the latest developments. This tree is also included in Andrew Morton's `-mm` tree. The `-fixes` git tree is used to send occasional fixes to Linus between merge windows and may not always be up-to-date.

The user tools are available from Red Hat's CVS at: `http://sources.redhat.com/cgi-bin/ cvsweb.cgi/cluster/?cvsroot=cluster`

References

[1] "DLM—Kernel Distributed Lock Manager,"
    Patrick Caulfield, *Minneapolis Cluster Summit
    2004*, `http://sources.redhat.com/
    cluster/events/summit2004/
    presentations.html#mozTocId443696`

[2] Linux Kernel Mailing List. Thread "If not
    readdir() then what?" started by Ulrich Drepper
    on Sat, 7 Apr 2007.

[3] "Extendible Hashing," Fagin, et al., *ACM
    Transactions on Database Systems*, Sept., 1979.

[4] The GFS2 git tree:
    `git://git.kernel.org/pub/scm/
    linux/git/steve/gfs2-2.6-nmw.git`
    (next merge window)

[5] "64-bit, Shared Disk Filesystem for Linux,"
    Kenneth W. Preslan, et al., *Proceedings of the
    Seventh NASA Goddard Conference on Mass
    Storage*, San Diego, CA, March, 1999.

[6] "The Global File System," S. Soltis, T. Ruwart,
    and M. O'Keefe, *Fifth NASA Goddard Conference
    on Mass Storage Systems and Technologies*,
    College Park, MD, September, 1996.

Driver Tracing Interface

David J. Wilder
IBM Linux Technology Center
dwilder@us.ibm.com

Michael Holzheu
IBM Linux on zSeries Development
holzheu@de.ibm.com

Thomas R. Zanussi
IBM Linux Technology Center
zanussi@us.ibm.com

Abstract

This paper proposes a driver-tracing interface (DTI) that builds on the existing Relay tool and the proven Debug Feature model used by IBM™ zSeries Linux. Users of this infrastructure are provided with individual, manageable channels for capturing or passing debug data into user space. Separate channels are created by each subsystem or driver. Data is stored in kernel ring buffers providing *flight recorder* type functionality. Unwanted or unconsumed data is simply discarded where pertinent data can be saved for future analysis. In the instance of a system crash, all unconsumed tracing data is automatically saved in crash dumps. With support from crash analysis tools like crash or lcrash, trace data can be extracted directly from a crash dump, providing an exact trace of the events leading up to the crash.

Developers of Linux™ device drivers will be interested in DTI as a tool to aid in the troubleshooting of their drivers. Service engineers and support personnel who are tasked with isolating driver bugs will learn how to capture DTI data on a live system and extract DTI data from a crash dump.

1 Introduction

Webster defines a trace as "the track left by the passage of a person, animal, or object." Applied to computer systems, we can adapt this definition to mean the track left by the execution of a program. A typical program doesn't normally leave tracks, other than the expected side effect of the program. To cause a program to create tracks so that its passage can meaningfully be tracked, code that explicitly leaves those tracks must be added into the execution path of the program. We call the individual tracks we've inserted *tracepoints*. Two types of tracepoints can be used.

- **Static:** Tracks that are added to the source code and compiled with it.

- **Dynamic:** Tracks that are added to the execution stream at run time.

Tracing is the act of causing special-purpose code associated with a program to report something specific about what the program is doing at a given point. The information can be simple or complex, high or low-frequency, binary or text-based, time-based or unsequenced and so on. The resulting data stream can be continuously persisted to long-term storage, sent to a destination over a network connection, or it can be endlessly cycled around a constant-sized buffer. The buffer will only be read when an event of interest occurs and a user needs details about the sequence of events that led up to that event, for example a system crash or a failed assertion in the normal program flow.

1.1 Why is tracing needed

Tracing is needed because, in many situations, only a detailed, sequenced, or timestamped history of program execution can explain the behavior of a program or the pathology of a problem. In many cases, coarse-grained statistical or summary information can show the general area of a problem, but only trace data can show the true source of the problem.

The detailed data from a complete trace can be post-processed and summary information or aggregated statistics can be calculated based on it. The converse however is not true. Detailed information cannot be extracted from statistics or summaries, because that information is lost in the process. Keeping in mind practical considerations such as storage costs, it is always better

to have the detailed trace information instead of only the statistics, because the appropriate trace data allows for more varied and flexible analysis.

2 Current solutions for static tracing

Currently, there are three kernel APIs available to do static tracing using kernel memory buffers:

- printk

- relay

- s390 debug feature (s390dbf)

2.1 Printk

Printk often is misused for tracing purposes, since there is no other standard way for device drivers to log debug information.[1] There are multiple printk levels defined, which indicate the importance of a kernel message. The kernel messages are written in one global printk buffer, which can be accessed from the user space with the `syslog()` system call or via the */proc/kmesg* file. The dmesg tool prints the content of the message buffer to the screen, and then the kernel log daemon klogd reads kernel messages and redirects them either to the syslogd or into a file. The printk message buffer can also be accessed from system dumps through the lcrash or other crash dump analysis tools.

2.2 Relay

Relay provides a basic low-level interface that has a variety of uses, including tracing. In order to define a kernel trace buffer, the `relay_open()` function is used. This function creates a relay channel. Relay channels are organized as wrap around buffers in memory. There are two mechanisms to write trace data into a channel:

- `relay_write(chan, data, length)` is used to place data in the global buffer.

- `relay_reserve(chan, length)` is used to reserve a slot in a channel buffer which can be written to later.

---

[1]This paper is not purposing a replacement for printk. The DTI is purposed as an additional tool that should be used in place of printk only when true tracing is needed.

Relay buffers are represented as files in a host file system such as debugfs; data previously written into a relay channel can be retrieved by read(2)ing or mmap(2)ing these files.

2.3 The s390 debug feature

The s390 debug feature (s390dbf) is a tracing API, which is used by most of the s390 specific device drivers. Each device driver creates its own debug feature in order to log trace records into memory areas, which are organized as wrap around ring buffers. The s390dbf uses its own ring buffer implementation. The main purpose of the debug feature is to inspect the trace logs after a system crash. Dump analysis tools like crash or lcrash can be used to find out the cause of the crash. If the system still runs but only a subcomponent which uses dbf fails, it is possible to look at the debug logs on a live system via the Linux debugfs file system.

Device drivers can register themselves to the debug feature with the `debug_register()` function. This function initializes an s390dbf for the caller. For each s390dbf there are a number of debug areas where exactly one is active at one time. Each debug area consists of a set of several linked pages in memory. In the debug areas, there are stored debug entries (trace records) which are written by event and exception calls.

An event call writes the specified debug entry to the active debug area and updates the log pointer for the active area. If the end of the active debug area is reached, a wrap around in the ring buffer is done and the next debug entry will be written at the beginning of the active debug area.

An exception call writes the specified debug entry to the log and switches to the next debug area. This is done in order to guarantee that the records that describe the origin of the exception are not overwritten when a wrap around for the current area occurs.

The debug areas themselves are also ordered in the form of a ring buffer. When an exception is thrown in the last debug area, the next debug entries are then written again in the very first area.

Each debug entry contains the following data:

- Timestamp

- Cpu-Number of calling task

- Level of debug entry (0...6)

- Return Address to caller

- Flag that indicate whether an entry is an exception or not

The trace logs can be inspected in a live system through entries in the debugfs file system. Files in the debugfs that were created by s390dbf represent different views to the debug log. The purpose of s390dbf views is to format the trace records in a human-readable way. Predefined views for hex/ascii, sprintf and raw binary data are provided. It is also possible to define other component specific views. The content of a view can be seen by reading the corresponding debugfs file. The standard views are also available in the dump analysis tools lcrash and crash. Figure 1 shows an example of an s390dbf sprintf view.

3   Driver Tracing Interface

This section describes the proposed Driver Tracing Interface (DTI) for the Linux kernel. It starts by examining the project goals and usage models that were considered in its design. Also provided is a brief description of two existing subsystems that DTI depends on, the DebugFS and relay subsystems.

3.1   Design goals

DTI will add supportability to drivers and subsystems that adopt it. However, from the support community viewpoint the deployment of DTI must be propagated into a significant number of subsystems, drivers, and system architectures before it usefulness is proven. Developing support tools and process around one off solutions is costly and unproductive to support organizations, therefore a key aspect of our project goals is to provide a feature that can easily be adopted by the Linux development community, thus ensuring its wide use.

- The DTI's API should be as simple and easy to implement as possible.

- DTI should be architecture independent.

Adding code into a driver or subsystem that is not contributing to the core functionality might be seen as unnecessary. This concern must be addressed by DTI, ensuring that the added benefit is balanced with low overhead of the feature.

- DTI should have as little performance impact as possible.

- DTI should reuse existing code in the Linux kernel.

- DTI must implement per-CPU buffering.

- DTI should minimize the in-kernel processing of trace data.

The remaining goals specify functionality requirements.

- DTI's API should be usable in both user context and interrupt context.

- Trace data must be buffered so that it can be retrieved from a crash dump.

- Trace data must be viewable from a live system.

- DTI must allow for a rich set of tools to process trace data.

3.2   Usage models

Two primary usage models were examined when designing the DTI API.

Usage Model 1: **Isolating a driver problem from a post-mortem cash dump analysis.** In this scenario, the system has crashed and a crashed system image (crash dump) has been obtained. By analyzing the crash dump the user suspects there is a problem in the XYZ driver. Using the DTI commands integrated into the crash dump analysis tool, the user can extract the DTI trace buffer from the crash dump and examine the records. The entire trace buffer containing the last trace records recorded by the XYZ driver is available. Using this data the user can obtain a trace showing what the driver was doing when the crash dump was taken. This allows the user to learn more about the cause of the crash.

Usage Model 2: **Troubleshooting a driver on a running system.** In this scenario, the user has encountered

```
00 01173807785:527586 0 — 02 00000000001eed86   Subchannel 0.0.4e20 reports non—I/O sc type 0001
00 01173807786:095834 2 — 02 00000000001f0962   reprobe done (rc=0, need_reprobe=0)
00 01173884042:004944 2 — 03 00000000001f7344   SenseID : UC on dev 0.0.1700, lpum 80, cnt 00
```

Figure 1: Example of the s390dbf sprintf view

a problem that is suspected to be related to one or more specific drivers. The user would like to examine the trace data just after the problem has occurred. To do so the following steps are taken:

1. Set the trace level to an appropriate level to produce the interested trace records.

2. Wait for the problem to occur, or reproduce the problem if possible.

3. Switch off tracing on the affected driver. Trace records produced just before, during, and after the problem occurred will remain in the DTI buffer.

4. Collect the trace data using a user level trace formatting tool.

5. Switch tracing back on.

By integrating DTI with systemtap additional usage modules can be realized.[2]

3.3 Debugfs

Debugfs is a minimalistic pseudo-filesystem existing mainly to provide a *namespace* that kernel facilities can hang special-purpose files of off, which in turn provides file-based access to kernel data. It provides a simple API for creating files and directories, as well as a set of ready made file operations which make it easy to create and use files that read and write primitive data types such as integers. For more complex data, it provides a means for facilities to associate custom file operations with debugfs files; in the case of relay, the exported `relay_file_operations` are associated with the debugfs files created to represent relay buffers. Despite the name, debugfs is not meant to be used only for debugging applications; it's enabled by default in many Linux distributions, and its use is encouraged especially for things that don't obviously belong in other pseudo-filesystems such as procfs or sysfs.

3.4 Relay

Relay[3] is a kernel facility designed for 'relaying' potentially large quantities of data from the kernel to the user space. Its overriding goal is to provide the shortest and cheapest possible path for a byte of data from the point it's generated in the kernel to the point it's usable by a program in user space. To accomplish this goal, it allocates a set of pages in the kernel, strings them together into a contiguous kernel address range via vmap(), and provides a set of functions designed to efficiently write data into the resulting relay buffer.

Each relay buffer is represented to user space as a file. This relay file is the abstraction used by the user space programs to retrieve the data contained in the relay buffer. The standard set of file operations allows for both read(2) and mmap(2) access to the data. These relay file operations are exported by the kernel, which allows them to be created in a pseudo-filesystem such as debugfs or procfs. In fact relay files **must be** created in one of these pseudo-filesystems in order for them to be accessible to user space programs (older versions of relay did actually include a file system called relayfs but the fs portion of the code was later subsumed by almost identical code in debugfs, and thus removed).

Relay buffers are logically subdivided into a number of equally sized *sub-buffers*. The purpose of sub-buffers is to provide the same benefits as double-buffering, but with more granularity. As data is written, it fills up sub-buffers, which are then considered ready for consumption by the user space. At the same time data is being written by the kernel into these unfinished sub-buffers, user space can be reading and releasing other finished sub-buffers. Relay channels can be configured to do double-buffereing or single-buffering if desired, or they can be configured to use large numbers of sub-buffers. A sub-buffer isn't considered readable until it's full and the next sub-buffer is entered (a sub-buffer switch). The latency between when a given event is written and the

---

[2]These advanced usage models will be explored in more detail in **Section 9, Integration With Other Tools**.

[3]See Documentation/filesystems/relay.txt for complete details.

time it's available to the user increases with the sub-buffer size. If sub-buffers are small, the latency is small and the amount of data that would be lost if the machine were to loses power is small. However, using small sub-buffers results in more time spent in the sub-buffer switching code (the slow path) instead of the main logging path (the fast path). Relay was designed with some reasonable middle ground in mind, efficiently buffering data implies some nontrivial amount of latency. If an application requires more immediacy, another mechanism should be considered. The assumption is that any mechanism that offers more immediacy by definition also creates more tracing overhead. The implied goal of relay is to cause as little disruption to a running system as possible.

By default, a relay buffer is created for each CPU; the combination of all per-CPU relay buffers along with associated meta-information is called a relay channel. Most of the relay API deals with the relay channels, and almost every aspect of a relay channel are configurable through the relay API.

4   Proposal for the Driver Tracing Interface

This section describes the purposed DTI architecture. The kernel API is introduced, trace record formatting and buffering are also discussed.

4.1   The DTI kernel API

The proposed DTI API can be broken in the following four major operations:

- Creating a trace handle and binding it to a relay channel.

- Writing trace records.

- Closing the channel.

- Setting the trace level.

The prototype of the API to the DTI is shown in Figure 2.

4.2   Creating the trace channel

The creation and binding of the trace handle are performed by calling `dti_register()`. When called, `dti_register()` creates a relay channel, associated data files and two additional control files in the debug file system. Upon successful completion, a `struct dti_info` pointer is returned. The caller will pass this pointer to all subsequent calls to the API. The format of `struct dti_info` is:

```
struct dti_info {
    struct dentry* root;
    struct rchan* chan;
    int    level;
    struct dentry *reset_consumed_ctrl;
    struct dentry *level_ctrl;
};
```

4.3   Writing trace records

Trace records are passed to the user using a variable length record as described in the `struct dti_event`.

```
struct dti_event {
        __u16 len;
        __u64 time;
        char data[0];
} __attribute__ ((packed));
```

Trace record suppliers only need to supply a pointer to the data buffer containing the raw data and the length of the data. DTI places no restriction on the format of the data supplied.

4.4   Trace level

The trace level is used to control what trace records should be placed in the buffer. Suppliers of trace data provide a trace level value each time a trace record is written. The trace level value is compared to the current trace level found in `dti_info->level`. Trace records are only placed in the buffer if the supplied trace level is less than or equal to the current trace level.

The current trace level is set using `dti_set_level()` or by the user writing a new trace level to the level control file. Trace levels are defined as an integer between −1 and `DTI_LEVEL_MAX`. A value of −1 means no tracing is to be done. When a trace channel is first registered the current trace level is set to `DTI_LEVEL_DEFAULT`.

```
/**
 * dti_register: create new trace
 * @name: name of trace
 * @size_in_k: size of subbuffer in KB
 *
 * returns trace handle or NULL, if register failed.
 */
struct dti_info *dti_register(const char *name,
                  int size_in_k);


/**
 * dti_unregister: unregister trace
 * @trace: trace handle
 */
void dti_unregister(struct dti_info *trace);


/**
 * dti_printk_raw: Write formatted string to trace
 * @trace: trace handle
 * @fmt: format string
 * @...: parameters
 *
 * returns 0, if event is written. Otherwise -1.
 */
int dti_printk_raw(struct dti_info *trace, int prio, const char* fmt, ...);


/**
 * dti_event_raw: Write buffer to trace
 * @trace: trace handle
 * @prio: priority of event (the lower, the higher the priority)
 * @buf: buffer to write
 * @len: length of buffer
 *
 * returns 0, if event is written. Otherwise -1.
 */
int dti_event_raw(struct dti_info *trace, int prio, char* buf, size_t len);


/**
 * dti_set_level: set trace level
 * @trace: trace handle
 */
void dti_set_level(struct dti_info *trace, int new_level);
```

Figure 2: Prototype of the DTI API

- Three sub-buffers are shown.

- The numbers represent the trace records.

- The 1st and the 2nd trace records have already been overwritten.

Sub-buffer 1 | 13 | 14 | 03 | 04 |

Sub-buffer 2 | 05 | 06 | 07 | 08 |

Sub-buffer 3 | 09 | 10 | 11 | 12 |
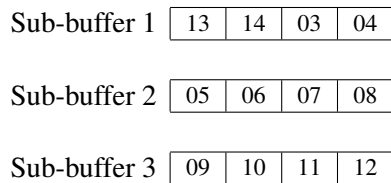
When relay data is read, the following records are returned:

| 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 |

Figure 3: An example of reading relay buffers

4.5   Data buffering

DTI depends on relay to handle the data buffering. Relay arranges trace records into a fixed number of sub-buffers arraigned in a ring. Each sub-buffer may contain one or more trace records or may be unused. Records are never split across sub-buffers. As sub-buffers are filled, new records are placed in the next sub-buffer in the ring. If no unconsumed sub-buffers are available, the sub-buffer containing the oldest data is overwritten.

When trace records are read by the user using the relay read interface, the oldest complete sub-buffer returned first, then the second oldest and so on. Therefore, the trace records are returned in the exact the same order they were written. For the current (newest) sub-buffer, the trace records up to the latest written trace record is returned. We lose the rest of the trace records (the oldest ones) of the current sub-buffer. This is acceptable, if enough sub-buffers are used. This process is illustrated in Figure 3.

4.6   Picking a buffer size

When the DTI trace is registered, you supply a size-in-k value, which is the total size of each relay channel buffer:

```
dti_register(name, size_in_k)
```

DTI automatically divides size-in-k by 8 and calls the exported `__dti_register()` function:

```
__dti_register(name, subbuf_size,
n_subbufs)
```

Users normally use the `dti_register()` version which does the calculation on behalf of the user. The user is not required to understand buffer internals, however if the user wants more control over the internal sub-buffer sizes, the `__dti_register()` version is available.[4]

4.7   Record time stamps

The time field in the `struct dti_event` is generated by the DTI. Its purpose is to provide both a time reference for when the trace record was written and a tool to sequence the records chronologically. The order of the trace records in a buffer of a single CPU is guaranteed to be in chronological order. DTI creates one relay buffer for each CPU. Therefore, the user must fully read each per-CPU buffer, then order the records correctly. It is possible for records read from different per-CPU buffers to contain the same time stamp. The choice of a sufficiently high resolution timer reduces the possibility of duplicate time-stamps; if the possibility is small, it might be acceptable.

5   DTI handle API

This section describes an extension to the basic DTI API called DTI handles. The DTI handle API simplifies writing kernel code that utilizes DTI. The features provided by the DTI handle API are:

- Auto-registration

- Support for early boot time tracing

Auto-registration eliminates the need to explicitly call `dti_register()`. Both modules and built-in drivers are supported. Registration of the DTI handle is automatically performed the first time trace data is written.

Early boot time tracing allows built-in drivers to log trace data before kmalloc memory is available. Static buffers are used to hold DTI events until it is safe to setup the relay channels. The DTI handle code creates

---

[4]See **Section-3.4 Relay** for a summary of sub-buffer size trade-offs to consider when choosing buffer/sub-buffer sizes.

```
#include <linux/dti.h>

static struct dti_handle my_handle

#ifdef MODULE
/* On the first event, channel will be auto-registered. */
DEFINE_DTI_HANDLE(my_handle, DRV_NAME, 4096 * 32, DTI_LEVEL_DEBUG,NULL);
#endif

#ifdef KERNEL
/*
 * Built-in drivers can optionally provide a static buffer used for
 * early tracing.
 */
static char my_buf[4096 * 4] __initdata;
DEFINE_DTI_HANDLE(my_handle, DRV_NAME, 4096 * 32, DTI_LEVEL_DEBUG,my_buf);
#endif

static int __init init_mydriver(void)
{
    ....
    INIT_DTI_HANDLE(my_handle);
    ....
}

my_driver_body(..)
{
    ....
     /* trace some events */
    dti_printk(my_handle, DTI_LEVEL_DEFAULT, format, _fmt, ## _args);
    ....
}

void cleanup_mydriver(void)
{
    ....
    CLEANUP_DTI_HANDLE(my_handle);
    ....
}

module_init(init_testdriver);
module_exit(cleanup_testdriver);
```

Figure 4: Example of using DTI handles

a `postcore_initcall` to switch tracing from static buffers to relay channels. All trace records written into static buffers are made available to the user interface after the `postcore_initcall` has run.

An example of using DTI handles is shown in Figure 4.

6 User interface

The section covers how trace data is read by the user on a running system and how tracing is controlled by the user.

6.1 File structure and control files

When a trace handle is bound, the following files are created in the traces directory of the root of the mounted debug file system.

```
dti/
    driver-name/
    data0 ... data[max-cpus]
        level
        reset_consumed
```

6.2 Retrieving trace data

One data file per CPU is created for each registered DTI trace provider. Trace records (`struct dti_event`) are read from the data files using a user supplied trace formatting tool. A trace formatting tool should read each per-CPU data file for a specified trace provider then arrange records according to the time stamp field of the `struct dti_event`.

The sequence of events normally followed when reading trace data is:

1. Switch off tracing by writing a –1 into the level file.

2. Read each of the per-CPU data files.

3. Switch tracing back on.

6.3 Trace level

The level file is used to inform the DTI provider of the level of trace records that should be placed in the data buffer. Reading the level file will return an ASCII value indicating the current level of tracing. The level can be changed by writing the ASCII value of the desired tracing level into the level file.

6.4 Reset consumed

When trace records are read, the records are marked as consumed by the relay subsystem as they are read. Therefore, subsequent reads will only return unread or new records in the buffers. If the desired behavior of a trace formatting tool is to return all records in the buffer each time the tools executed, the tool must reset the consumed value after reading all records currently in the buffer. Resetting the consumed value is performed by writing any value into the reset-consumed file.

7 Dump analysis tool support

Analysis of trace data from a crashed system is one of the most important use-cases for DTI. As such, support will be added to the crash and lcrash dump analysis tools enabling those tools to extract and make use of the DTI trace buffers and related trace information from a crashed system image.

7.1 Retrieving trace data from a crash dump

When a DTI trace is registered through `dti_register()`, a text name is specified as one of the parameters. This string not only identifies the trace to the user, but is also effectively concatenated with a prefix string, unlikely to be used by a user, in order to make the complete identifying string easily locatable in a memory dump for example if a trace name is given as "mydriver" by the user, the string the dump tool would use to find the corresponding `dti_info` struct would be `__DTI__mydriver`. When this string is located, it's a straightforward exercise to locate the associated relay channel and its buffers. For example, assuming a simplified `dti_info` struct:

```
struct dti_info
{
  struct rchan *chan;
  char dti[7] = "__DTI__";
  char name[DTI_TRACENAME_LEN];
  .
  .
  .
}
```

The dump tool would locate the beginning of the dti[] array, and subtracting the size of a pointer would find the pointer to the struct rchan:

```
struct rchan
{
  size_t subbuf_size;
  size_t n_subbufs;
  struct rchan_buf[NR_CPUS];
  .
  .
  .
}
```

From the pointer to the rchan, you can locate each of the buffers that make up the channel and its size.

```
struct rchan_buf
{
  /* start of buffer */
  void *start;
  /* start of current sub-buffer */
  void *data;
  /* write offset into the current
     sub-buffer */
  size_t offset;
  /* number of sub-buffers
     consumed */
  size_t consumed;
  .
  .
  .
}
```

From this information, you can extract all the available data from each buffer and send it to the post-processing tool to be combined and sorted as usual.

8   Sample implementations

8.1   Port of the S390 dbf

Most of the s390 device drivers currently use the s390dbf. Examples are:

- ZFCP: SCSI host adapter device driver

- QETH: Ethernet network device driver

- DASD: s390 harddisk driver

- TAPE: Driver for 3480/90 and 3590/92 channel attached tape devices

s390dbf API	DTI API
debug_register()	dti_ register()
debug_unregister()	dti_ unregister()
debug_event()	dti_event_ raw()
debug_sprintf_ event()	dti_printk_ raw()
debug_set_level()	dti_set_ level()

Table 1: Mapping of the s390dbf API to the DTI API

All the users of the s390dbf API can be quite easily converted to use the new DTI API. Table 1 shows functions that can be mapped directly.

There is no DTI equivalent for the s390dbf exception calls. Those must be replaced by the appropriate DTI event functions. The exception functionality used to switch debug areas is not frequently used by the s390 device drivers. Therefore it is acceptable to remove this functionality in order to keep the API small and simple.

The functions `debug_register_view()` and `debug_unregister_view()` are not needed any more, since formatting of DTI traces is done in the user space.

Currently the s390 ZFCP device driver uses non-default self-defined s390dbf views. For that driver, it is necessary to implement a user space tool with the same formatting functionality as the ZFCP specific s390dbf view.

Some details of the port have not yet been resolved.

- How should the number of pages value used by `debug_register` be mapped to the buffer size used in `dti_register()`?

- Is an equivalent for `debug_stop_all()` needed?

8.2   Port of some other drivers

There are currently a large number of custom logging APIs in the kernel, each mainly restricted to logging formatted debugging string data related to a particular driver or subsystem. Most of them are a variation on `#define DPRINTK(x...) printk(x...)`.

These can easy be ported to DTI using the DTI handle API. However, DTI's strength is focused on continuous tracing to a buffer which can be retrieved when necessary rather than the continuous logging implemented by these special-purpose facilities.

There are, however, a handful of continuous tracing facilities similar to DTI in the kernel, varying from very minimalistic to fairly full-featured. Included in this category is the current s390dbf facility, which is presently the most advanced. Briefly examined are some of the others that tracing facilities that would be candidates for porting to the proposed DTI API.

- `drivers/scsi/mesh.c` provides the `dlog()` function for logging formatted records. It keeps data in a ring of structs. The dump function printk's the whole buffer.

- `drivers/net/wan/lmc/lmc_debug.h` provides `LMC_EVENT_LOG()` which logs two u32 args along with an event number and jiffies.

- `drivers/char/ip2` provides `ip2trace()` which is used to trace any number of longs into a buffer containing 1000 longs. It provides a read(2) interface to read the data.

- `drivers/isdn/hardware/eicon/debug.c` provides an extensive API and set of functions used to log and maintain a queue of debug messages.

- `fs/xfs/support/ktrace.c` provides a simple yet extensive API for tracing the xfs file system. The main logging function is `ktrace_enter()`, which allows up to 16 values to be logged per entry into a buffer containing 64 ktrace entries.

9 Integration of other tools

DTI tracing is of course, useful in its own right. Combined with a trace analysis tool such as SystemTap[5] it can become an even more powerful tool. A DTI trace can be used to see with great detail exactly what happens within a particular driver over a given time interval, but it doesn't have any other context associated with it, such as system calls or interrupt activity that may have triggered activity within the driver. In many cases, it would

be extremely useful to have such associated data available for analysis. DTI's strength is its data-gathering functionality. Using Systemtap, DTI's functionality can be extended by adding context to the data, perform time analysis of the data being gathered or maintain a summary information about the trace. For example, if a certain pattern of interest only occurs intermittently, the user could detect it and either halt the trace, preserving the events that led up to it, or alert the user of the condition. In addition, SystemTap can be used to gather aggregated summaries of the data over long periods of time (longer than the limited size of a relay buffer would allow) to get an overall picture of activity with respect to the driver and associated context.

The DTI tapset is being developed to allow SystemTap to put kprobes on the high-level DTI tracing functions, which makes all data passing through them accessible to SystemTap. Note that doing this in no way affects the normal flow of DTI events; the only additional effect is the probe effect, which means that each event recorded in this way incurs a penalty equal to the time required to fire the probe and run the SystemTap handler.

Systemtap can also be used to dynamically place new probe points into a driver. To do this Systemtap places a kprobe in the driver where a trace point is to be added and all data available at the probe point is fed back into the DTI data stream. The DTI post-processing tools can then be used to format and display this external data along with the normal DTI trace output. To do this, users make use of the 'DTI-control' tapset, which allows SystemTap scripts to control and log data to DTI via the standard DTI kernel interface.

10 Conclusion

Customers of s390 systems demand very high reliability and quick turnaround time for bug fixes. Since its introduction early in the history of Linux on the s390, the s390 Debug Facility has proven itself as an invaluable tool for meeting customers' reliability expectations. The ability to analyze trace data in a crash dump and perform first fault analysis is key to the s390dfb's success. The question has been posed, "Why reinvent functionality that already exists?" The answer is simple: DTI intends to exploit the s390db model and bring this technology to all Linux platforms. Service organizations gain not only by the additions of DTI functionality but by the uniformity of a tracing infrastructure between all platforms. In

---

[5]`http://sourceware.org/systemtap`

addition, DTI utilizes the existing relay subsystem that did not exist when s390dbf was written. Therefore, DTI can be implemented with a much smaller footprint than the s390dbf.

s390 drivers provide the perfect sandbox for porting drivers to the DTI and testing its implementation. We plan to pursue this effort as well as encouraging other driver developers to adopt DTI.

Legal statement

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM, IBM (logo), e-business (logo), pSeries, e (logo) server, and xSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

Linux readahead: less tricks for more

Fengguang Wu   Hongsheng Xi   Jun Li
University of Science and Technology of China
`wfg@ustc.edu, {xihs,Ljun}@ustc.edu.cn`

Nanhai Zou
Intel Corporation
`nanhai.zou@intel.com`

Abstract

The Linux 2.6 readahead has grown into an elaborate work that is hard to understand and extend. It is confronted with many subtle situations. This paper highlights these situations and proposes alternative solutions to them. By adopting new calling conventions and data structures, we demonstrate that readahead can be made more clean, flexible and reliable, which opens the door for more opportunities.

1 Introduction

1.1 Background

Readahead is a widely deployed technique to bridge the huge gap between the characteristics of disk drives and the inefficient usage by applications. At one end, disk drives are good at large sequential accesses and bad at seeks. At the other, applications tend to do a lot of tiny reads. To make the two ends meet, modern kernels and disk drives do readahead: to bring in the data before it is needed and try to do so in big chunks.

Readahead brings three major benefits. Firstly, I/O delays are effectively hidden from the applications. When an application requests a page, it has been prefetched and is ready to use. Secondly, disks are better utilized by large readahead requests. Lastly, it helps amortize request processing overheads.

Readahead typically involves actively detecting the access pattern of all read streams and maintaining information about them. Predictions based on on where and how much data will be needed in the near future are made. Finally, carefully selected data is read in before being requested by the application.

There exist APIs (`posix_fadvise(2)`, `madvise(2)`) for the user-space to inform the kernel about its access pattern or more precise actions about data, but few applications bother to take advantage of them. They are mostly doing sequential or random reads, and expecting the kernel to serve these common and simple cases right.

So the kernel has to guess. When done right, readahead can greatly improve I/O throughput and reduce application visible I/O delays. However, a readahead miss can waste bandwidth and memory, and eventually hurt performance.

1.2 A brief history

Linux 2.6 implements a generic readahead heuristic in its VFS layer.

The unified readahead framework was first introduced [1] in the early 2.5 time by Andrew Morton. It features the current/ahead windows data structure; readahead/read-around heuristics; protection against readahead thrashing, aggressive cache hits [2] and congested queues [3]. The mmap read-around logic was later taken out by Linus Torvalds[4]. The separation of read-ahead and read-around yields better solutions for both. The rather chaotic mmap reads from executables can now be prefetched aggressively, and the readahead logic can concentrate on detecting sequential reads from random ones.

Handling sequential and random reads right, however, turned out to be a surprisingly hard mission. One big

challenge comes from some mostly random database workloads. In three years, various efforts were made to better support random cases [5, 6, 7, 8, 9]. Finally, Steven Pratt and Ram Pai settled it down by passing the read request size into `page_cache_readahead()` [10, 11]. The addition of read size also helped another known issue: when two threads are doing simultaneous `pread()`s, they will be overwriting each other's readahead states. The readahead logic may see lots of 1-page reads, instead of the true `pread()` sizes. The old solution, in the day when `page_cache_readahead()` was called once per page, was to take a local copy of readahead state in `do_generic_mapping_read` [12, 13].

1.3 Moving on

Linux 2.6 is now capable of serving common random/sequential access patterns right. That may be sufficient for the majority, but it sure can do better.

So comes the adaptive readahead patch [14], an effort to bring new readahead capabilities to Linux. It aims to detect semi-sequential access patterns by querying the page-cache context. It also measures the read speed of individual files relative to the page-cache aging speed. That enables it to be free from readahead thrashing, and to manage the readahead cache in an economical way.

The lack of functionality is not an issue with regard to readahead, in fact the complexity of the code actually prevents further innovation. It can be valuable to summarize the experiences we learned in the past years, to analyze and reduce the source of the complexity is what motivated us to do the work of required for this paper.

We propose a readahead algorithm that aims to be clean. It is called on demand, which helps free readahead heuristics from most of the chores that the current algorithm suffers from. The data structure is also revised for ease of use, and to provide exact timing information.

1.4 Overview of the rest of the paper

In Section 2, we first discuss the readahead algorithm as in Linux 2.6.20, then proceed to discuss and propose new solutions to the three major aspects (data structure, call scheme, and guideline) of readahead. In Section 3, we analyze how the old/new readahead algorithms work out in various situations. Finally, Section 4 gives benchmark numbers on their overheads and performance.

2 Readahead Algorithms

2.1 Principles of 2.6 readahead

Figure 1 is a brief presentation of the readahead algorithm in Linux 2.6.20. The heuristics can be summarized in four aspects:

sequential detection If the first read at the start of a file, or a read that continues from where the previous one ends, assume a sequential access. Otherwise it is taken as a random read.

The interface demands that it be informed of every read requests. `prev_page` is maintained to be the last page it saw and handled. An oversize read (`read_size > max_readahead`) will be broken into chunks of no more than `max_readahead` and fed to the readahead algorithm progressively.

readahead size There are three phases in a typical readahead sequence:

initial When there exists no `current_window` or `ahead_window`, the size of initial readahead is mainly inferred from the size of current read request. Normally `readahead_size` will be 4 or 2 times `read_size`.

ramp-up When there is a previous readahead, the size is doubled or x4.

full-up When reaching `max_readahead`.

It is possible to jump directly into full-up phase, if the request size is large enough (e.g. `sendfile(10M)`).

readahead pipelining To maximally overlap application processing time and disk I/O time, it maintains two readahead windows: `current_window` is where the application expected to be working on; `ahead_window` is where asynchronous I/O happens. `ahead_window` will be opened/renewed in advance, whenever it sees a sequential request that

- is oversize
- has only `current_window`
- crossed into `ahead_window`

```
 1  do_generic_mapping_read:
 2      call page_cache_readahead
 3      for each page
 4          if is prev_page + 1
 5              call page_cache_readahead
 6          if page not cached
 7              report cache miss
 8              leave cache hit mode
 9
10  page_cache_readahead:
11      handle unaligned read
12      set prev_page to current page index
13      if in cache hit mode
14          return
15      shift prev_page to the last requested page,
16          but no more than max_readahead pages
17      if is sequential read and no current_window
18          make current_window
19          call blockable_page_cache_readahead
20          if is oversize read
21              call make_ahead_window
22      elif is random read
23          clear readahead windows
24          limit size to max_readahead
25          call blockable_page_cache_readahead
26      elif no ahead_window
27          call make_ahead_window
28      elif read request crossed into ahead_window
29          advance current_window to ahead_window
30          call make_ahead_window
31      ensure prev_page do not overrun ahead_window
32
33  make_ahead_window:
34      if have seen cache miss
35          clear cache miss status
36          decrease readahead size by 2
37      else
38          x4 or x2 readahead size
39          limit size to max_readahead
40      call blockable_page_cache_readahead
41
42  blockable_page_cache_readahead:
43      if is blockable and queue congested
44          return
45      submit readahead io
46      if too many continuous cache hits
47          clear readahead windows
48          enter cache hit mode
```

Figure 1: readahead in 2.6.20

cache hit/miss A generic *cache hit/miss* happens when a page to be accessed is found to be cached/missing. However, the terms have specific meanings in 2.6 readahead.

A *readahead cache hit* happens when a page to be readahead is found to be cached already. A long run of readahead cache hits indicates an already cached file. When the threshold `VM_MAX_CACHE_HIT`(=256) is reached, readahead will be turned off to avoid unnecessary lookups of the page-cache.

A *readahead cache miss* happens when a page that was brought in by readahead is found to be lost on time of read. The page may be relaimed prematurely, which indicates readahead thrashing. The readahead size can be too large, so decrease it by 2 for the next readahead.

2.2 Readahead windows

The 2.6 readahead adopts dual windows to achieve readahead pipelining: while the application is walking in the `current_window`, I/O is underway in the `ahead_window`. Although it looks straightforward, the implementation of this concept is not as simple as one would think.

For the purpose of pipelining, we want to issue I/O for the next readahead before the not-yet-consumed readahead pages fall under a threshold, *lookahead size*. A value of `lookahead_size = 0` disables pipelining, whereas `lookahead_size = readahead_size` opens full pipelining.

The current/ahead windows scheme is one obvious way to do readahead pipelining. It implies `lookahead_size` to be `readahead_size - read_size`[1] for the initial readahead. It then ranges from `readahead_size` to `readahead_size + read_size` for the following ones. It is a vague range due to the fact that 2.6.20 readahead pushes forward the windows as early as it sees the read request (instead of one realtime page read) crossing into the `ahead_window`.

However, the scheme leads to obscure information and complicated code. The lookahead size is implicitly coded and cannot be freely tuned. The timing information for the previous readahead may be too vague to be

---

[1]assume `read_size <= max_readahead`

useful. The two windows bring three possible combinations of on/off states. The code has to probe for the existence of `current_window` and/or `ahead_window` before it can do any query or action on them. The heuristics also have to explicitly open `ahead_window` to start readahead pipelining.

Now let's make a study of the information necessary for a sequential readahead:

1. To work out the position of next readahead, that of the previous one will be sufficient: We normally apply a simple size ramp up rule: `(offset, size) => (offset+size, size*2)`.

2. Optionally, in case the previous readahead pages are lost, the timing information of their first enqueue to `inactive_list` would be helpful. Assume the reader is now at `offset`, and was at page `la_index` when the lost pages were first brought in, then `thrashing_threshold = offset - la_index`.

3. To achieve pipelining, indicating a *lookahead page* would be sufficient: on reading of which it should be invoked to do readahead in advance.

We revised the data structure (Figure 2) to focus on the previous readahead, and to provide the exact timing information. The changes are illustrated in Figure 3 and compared in Table 1.

2.3 Readahead on demand

The 2.6 readahead works by inspecting *all* read requests and trying to discover sequential patterns from them. In theory, it is a sound way. In practice, it makes a lot of fuss. Why should we call excessively into `page_cache_readahead()` only to do nothing? Why handle cache hits/misses in convoluted feedback loops?

In fact, there are only two cases that qualify for a readahead:

sync readahead on cache miss A cache miss occurred, the application is going to do I/O anyway. So try readahead and check if some more pages should be piggybacked.
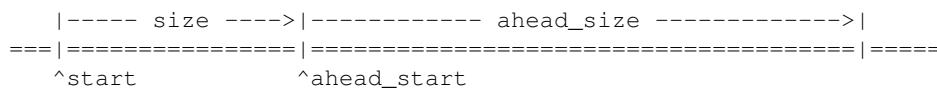
```
 struct file_ra_state {
-    unsigned long start;            /* Current window */
-    unsigned long size;
-    unsigned long ahead_start;      /* Ahead window */
-    unsigned long ahead_size;
-    unsigned long cache_hit;        /* cache hit count */
-    unsigned long flags;            /* RA_FLAG_MISS | RA_FLAG_INCACHE */
+    pgoff_t la_index;               /* enqueue time */
+    pgoff_t ra_index;               /* begin offset */
+    pgoff_t lookahead_index;        /* time to do next readahead */
+    pgoff_t readahead_index;        /* end offset */
     unsigned long prev_page;        /* Cache last read() position */
     unsigned long ra_pages;         /* Maximum readahead window */
 };
```

Figure 2: revising readahead data structure

```
(a) 2.6.20 readahead

    |----- size ---->|------------ ahead_size ------------->|
===|================|=====================================|=====
    ^start           ^ahead_start

(b) on-demand readahead

                    |----------------- readahead size -->|
         |---- reader on the way --->|-- lookahead size -->|
===|=====#==========|===============#====================|=====
         ^          ^               ^                     ^
     la_index   ra_index  lookahead_index        readahead_index
```
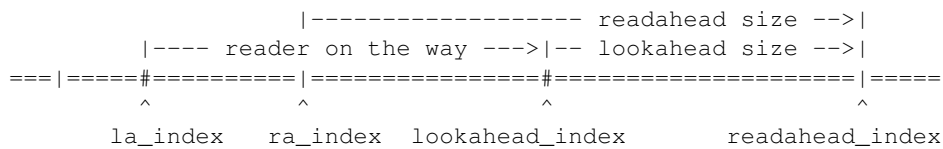
Figure 3: tracking readahead windows

question	on-demand readahead	2.6.20 readahead
when to do next read-ahead	`lookahead_index`	around `ahead_start` or `start`
where to do next read-ahead	`readahead_index`	`start + size` or `ahead_start + ahead_size`
the time of previous readahead	`la_index`	maybe/roughly `start`
the size of previous readahead	`readahead_index - ra_index`	`size` or `ahead_size`

Table 1: deciding the next readahead

async readahead on lookahead page The application is walking onto a readahead page with flag `PG_readahead`, or a *lookahead mark*. It indicates that the readahead pages in the front are dropping to `lookahead_size`, the threshold for pipelining. So do readahead in advance to reduce application stalls.

When called on demand, the readahead heuristics can be liberated from a bunch of special cases. The details about them will be covered in Section 3.

2.4   The new algorithm

Figure 4 shows the proposed on-demand readahead algorithm. It is composed of a list of condition-action blocks. Each condition tests for a specific case (Table 2), and most actions merely fill the readahead state with proper values (Table 3).

random A small, stand-alone read. Take it as a random read, and read as is.

lookahead It is lookahead time indicated by the readahead state, so ramp up the size quickly and do the next readahead.

readahead It is readahead time indicated by the readahead state. We can reach here if the lookahead mark was somehow ignored (queue congestion) or skipped (sparse read). Do the same readahead as in lookahead time.

initial First read on start of file. It may be accessing the whole file, so start readahead.

oversize An oversize read. It cannot be submitted in one huge I/O, so do it progressively as a readahead sequence.

miss A sequential cache miss. Start readahead.

interleaved A lookahead hit without a supporting readahead state. It can be some interleaved sequential streams that keep invalidating each other's readahead state. The lookahead page indicates that the new readahead will be at least the second one in the readahead sequence. So get the initial readahead size and ramp it up once.

The new algorithm inherits many important behaviors from the current one, such as random reads, and the size ramp up rule on sequential readahead. There are also some notable changes:

1. A new parameter `page` is passed into `ondemand_readahead()`. It tells whether the current page is present. A value of `NULL` indicates a synchronous readahead, otherwise an asynchronous one.

2. A new parameter `begin_offset` is introduced to indicate where the current read request begins. `prev_page` now simply tracks the last accessed page of previous request. Hence the new sequential access indicator becomes: `sequential = (begin_offset - prev_page <= 1)`.

3. I/O for overlapped random reads may not be submitted as early. Suppose a 8-page random read, whose first 4 pages are overlapped with a previous read. The 2.6 readahead will emit request for all of the 8 pages before accessing the first page. While the on-demand readahead will ask for the remaining 4 pages on accessing the 5th page. Hence it avoids some unnecessary page-cache lookups, at the cost of not being able to overlap transfer of the leading cached pages with I/O for the following ones.

4. Linux 2.6.20 only does readahead for sequential read requests. In the new design, we loosen the criteria a bit: the lookahead hit alone can trigger the next readahead. It enables detection of interleaved reads.

It is not as safe to ignore sequentialness, but the risk is pretty low. Although we can create a set of random reads to trigger a long run of readahead sequences, it is very unlikely in reality. One possible candidate may be stride reads. But it cannot even cheat the algorithm through the size ramp-up phase, where the lookahead pages distribute in a non-uniform way.

The support of interleaved reads is minimal. It makes no extra efforts to detect interleaved reads. So the chances of discovering them is still low. Interleaved sequential reads may or may not be readahead, or may be served intermittently.

```
1   do_generic_mapping_read:
2       for each page
3           if page not cached
4               call ondemand_readahead
5           if page has lookahead mark
6               call ondemand_readahead
7       set prev_page to last accessed page
8
9   ondemand_readahead:
10      if is asynchronous readahead and queue congested
11          return
12      if at start of file
13          set initial sizes
14      elif is small random read in wild
15          read as is
16          return
17      elif at lookahead_index or readahead_index
18          ramp up sizes
19      else
20          set initial sizes
21          if has lookahead mark
22              ramp up size
23      fill readahead state
24      submit readahead io
25      set lookahead mark on the new page at new lookahead_index
```

Figure 4: on-demand readahead algorithm

case	description	condition
initial	read on start of file	`!offset`
oversize	random oversize read	`!page && !sequential && size > max`
random	random read	`!page && !sequential`
lookahead	lookahead hit	`offset == ra->lookahead_index`
readahead	readahead hit	`offset == ra->readahead_index`
miss	sequential cache miss	`!page`
interleaved	lookahead hit with no context	`page`

Table 2: detecting access patterns

case	ra_index	ra_size	la_size
random	`offset`	`size`	`0`
lookahead,readahead	`ra->readahead_index`	`get_next_ra_size(ra)`	
initial,oversize,miss	`offset`	`get_init_ra_size(size,max)`	`1`
interleaved	`offset + 1`	`get_init_ra_size(...) * 4`	`1`

Table 3: deciding readahead parameters

3 Case Studies

In this section, we investigate special situations the readahead algorithm has to confront:

1. Sequential reads do not necessary translate into incremental page indexes: multi-threaded reads, retried reads, unaligned reads, and sub-page-size reads.

2. Readahead should not always be performed on sequential reads: cache hits, queue congestion.

3. Readahead may not always succeed: out of memory, queue full.

4. Readahead pages may be reclaimed before being read: readahead thrashing.

3.1 Cache hits

Ideally, no readahead should ever be performed on cached files. If a readahead is done on a cached file, then this can cost many pointless page cache lookups. In a typical system, reads are mostly performed on cached pages. Cache hits can far outweigh cache misses.

The 2.6 readahead detects excessive cache hits via `cache_hit`. It counts the continuous run of readahead pages that are found to be already cached. Whenever it goes up to `VM_MAX_CACHE_HIT(=256)`, the flag `RA_FLAG_INCACHE` will be set. It disables further readahead, until a cache miss happens, which indicates that the application have walked out of the cached segment.

In summary,

1. Always call `page_cache_readahead()`;

2. Disable readahead after 256 cache hits; and

3. Enable readahead on cache miss.

That scheme works, but is not satisfactory.

1. It only works for large files. If a file is fully cached but smaller than 1*MB*, it won't be able to see the light of `RA_FLAG_INCACHE`, which can be a common case. Imagine a web server that caches a lot of small to medium `html/png` files and desktop systems.

2. Pretend that it happily enters cache-hit-no-readahead mode for a `sendfile(100M)` and avoids page-cache lookups. Now another overhead arises: `page_cache_readahead()` that used to be called once every `max_readahead` pages will be called on each page to ensure in time restarting of readahead after the first cache miss.

The above issues are addressed in on-demand readahead by the following rules:

1. Call `ondemand_readahead()` on cache miss;

2. Call `ondemand_readahead()` on lookahead mark; and

3. Only set lookahead mark on a newly allocated readahead page.

Table 4 compares the two algorithms' behavior on various cache hit situations. It is still possible to apply the threshold of `VM_MAX_CACHE_HIT` in the new algorithm, but we'd prefer to keep it simple. If a random cached page happens to disable one lookahead mark, let it be. It would be too rare and non-destructive to ask for attention. As for real-time applications, they need a different policy—to persist on cache hits.

3.2 Queue Congestion

When the I/O subsystem is loaded, it becomes questionable to do readahead. In Linux 2.6, the load of each disk drive is indicated by its request queue. A typical request queue can hold up to `BLKDEV_MAX_RQ(=128)` requests. When a queue is 7/8 full, it is flagged as *congested*; When it is completely full, arriving new requests are simply dropped. So in the case of a congested queue, doing readahead risks wasting the CPU/memory resources: to scan through the page-cache, allocate a bunch of readahead pages, get rejected by the request system, and finally free up all the pages—a lot of fuss about nothing.

Canceling readahead requests on high I/O pressure can help a bit for the time being. However, if it's only about breaking large requests into smaller ones, the disks will be serving the same amount of data with much more seeks. In the long run, we hurt both I/O throughput and latency.

	ondemand_readahead()	page_cache_readahead()
large cached chunk		called on *every* page to recheck
full cached small files	not called	
prefetched chunks		called to do readahead
small cached chunk	may or may not be called	
cache miss	called and do readahead *now*	restart readahead *after* first miss

Table 4: readahead on cache hits

So the preferred way is to defer readahead on a congested queue. The on-demand readahead will do so for asynchronous readaheads. One deferred asynchronous readahead will return some time later as a synchronous one, which will always be served. The process helps smooth out the variation of load, and will not contribute more seeks to the already loaded disk system.

The current readahead basically employs the same policy. Only that the decisions on whether to force a readahead are littered throughout the code, which makes it less obvious.

3.3 Readahead thrashing

In a loaded server, the page-cache can rotate pages quickly. The readahead pages may be shifted out of the LRU queue and reclaimed, before a slow reader is able to access them in time.

Readahead thrashing can be easily detected. If a cache miss occurs inside the readahead windows, readahead thrashing happened. In this case, the current readahead decreases the next readahead size by 2. By doing so it hopes to adapt to the thrashing threshold. Unfortunately, the algorithm does not remember it. Once it steps slowly off to the thrashing threshold, the thrashings stop. It then immediately reverts back to the normal behavior of ramping up the window size by 2 or 4. Which starts a new round of thrashings. On average, about half of the readahead pages can be thrashed.

It would be even more destructive for disk throughput. Suppose that `current_window` is thrashed when an application is walking in the middle of it. The 2.6 readahead algorithm will be notified via `handle_ra_miss()`. But it merely sets a flag `RA_FLAG_MISS`, and takes no action to recover the `current_window` pages to be accessed. `do_generic_mapping_read()` then starts to fault in them one by one, generating a lot of disk seeks. Overall, up to half pages may be faulted in this crude way.

The on-demand readahead takes no special action against readahead thrashing. Once thrashed, an initial readahead will be started from the current position. It does not cut down the number of thrashed pages, but does avoid the catastrophic seeks. Hence it performs much better on thrashing.

3.4 Unaligned reads

File operations work on byte ranges, while the readahead routine works on page offsets. When an application issues 10000B sized reads, which do not align perfectly to the 4K page boundary, the readahead code will see an $offset + size$ flow of $0 + 3, 2 + 2, 4 + 3, 7 + 2, 9 + 3, 12 + 2, 14 + 3, 17 + 2, 19 + 3, \ldots$. Note that some requests overlap for one page. It's no longer an obvious sequential pattern.

Unaligned reads are taken care of by allowing `offset == prev_page` [15] in 2.6 readahead and on-demand readahead.

3.5 Retried reads

Sometimes the readahead code will receive an interesting series of requests[16] that looks like: $0 + 1000, 10 + 990, 20 + 980, 30 + 970, \ldots$. They are one normal read followed by some retried ones. They may be issued by the retry-based AIO kernel infrastructure, or retries from the user space for unfinished `sendfile()`s.

This pattern can confuse the 2.6.20 readahead. Explicit coding is needed to ignore the return of reads that have already been served.

The on-demand readahead is not bothered by this issue. Because it is called on the page to be accessed now, instead of the read request.

case	2.6.20	on-demand	overheads	reasoning
sequential re-read in $4KB$	20.30	20.05	−1.2%	no readahead invocation
sequential re-read in $1MB$	37.68	36.48	−3.2%	
small files re-read (`tar /lib`)	49.13	48.47	−1.3%	no page-cache lookup
random reading sparse file	81.17	80.44	+0.9%	one extra page-cache lookup per cache miss
sequential reading sparse file	389.26	387.97	−0.3%	less readahead invocations

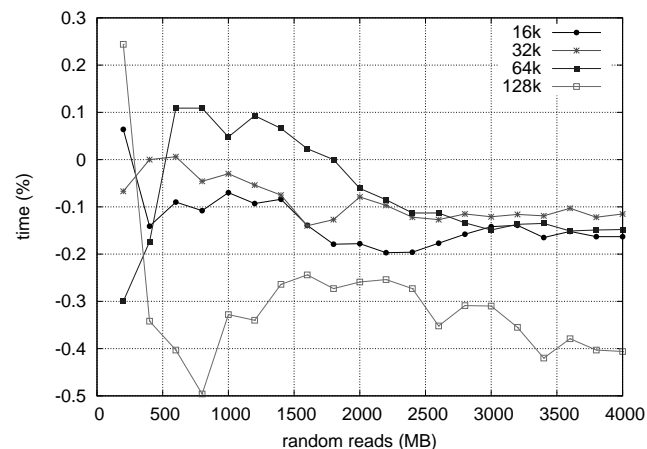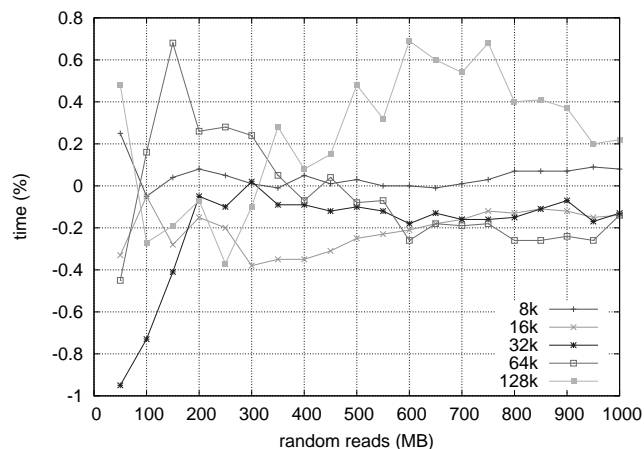Table 5: measuring readahead overheads

Figure 5: timing overlapped random reads

3.6 Interleaved reads

When multiple threads are reading on the same file descriptor, the individual sequential reads get interleaved and look like random ones to the readahead heuristics. Multimedia files that contain separated audio/video sections may also lead to interleaved access patterns.

Interleaved reads will totally confuse the current readahead, and are also beyond the mission of on-demand readahead. However, it does offer minimal support that may help some interleaved cases. Take, for example, the 1-page requests consisting of two streams: $0, 100, 1, 101, 2, 102, \ldots$. The stream starting from 0 will get readahead service, while the stream from 100 will still be regarded as random reads. The trick is that the read on page 0 triggers a readahead (the initial case), which will produce a lookahead mark. The following reads will then hit the lookahead mark, make further readahead calls and push forward the lookahead mark (the lookahead case).

4 Performance

4.1 Benchmark environment

The benchmarks are performed on a Linux 2.6.20 that is patched with the on-demand readahead. The basic setup is

- $1MB$ max readahead size

- 2.9GHz Intel Core 2 CPU

- $2GB$ memory

- 160G/8M Hitachi SATA II 7200 RPM disk

4.2 Overheads

Table 5 shows the max possible overheads for both algorithms. Each test is repeated sufficient times to get the stable result. When finished, the seconds are summed up and compared.

Cache hot sequential reads on a huge file are now faster by 1.2% for 1-page reads and by 3.2% for 256-page

reads. Cache hot reads on small files (`tar /lib`) see a 1.3% speed up.

We also measured the maximum possible overheads on random/sequential reads. The scenario is to do 1-page sized reads on a huge sparse file. It is 0.9% worse for random reads, and 0.3% better for sequential ones. But don't take the two numbers seriously. They will be lost in the background noise when doing large sized reads, and doing it on snail-paced disks.

4.3 Overlapped random reads

We benchmarked $8/16/32/64/128KB$ random reads on a $500/2000MB$ file. The requests are aligned to small $4KB$ boundaries and therefore can be overlapping with each other. On every $50/200MB$ read, the total seconds elapsed are recorded and compared. Figure 5 demonstrates the difference of time in a progressive way. It shows that the $128KB$ case is unstable, while others converge to the range $(-0.2\%, 0.1\%)$, which are trivial variations.

4.4 iozone throughput

We ran the iozone benchmark with the command `iozone -c -t1 -s 4096m -r 64k`. That's doing $64KB$ non-overlapping reads on a $4GB$ file. The throughput numbers in Table 6 show that on-demand readahead keeps roughly the same performance.

| access pattern	2.6.20	on-demand	gain
Read	62085.61	62196.38	+0.2%
Re-read	62253.49	62224.99	−0.0%
Reverse Read	50001.21	50277.75	+0.6%
Stride read	8656.21	8645.63	−0.1%
Random read	13907.86	13924.07	+0.1%
Mixed workload	19055.29	19062.68	+0.0%
Pread	62217.53	62265.27	+0.1%

Table 6: iozone throughput benchmark (KB/s)

4.5 Readahead thrashing

We boot the kernel with `mem=128m single`, and start a $100KB/s$ stream on every second. Various statistics are collected and showed in Figure 6. The thrashing begins at $20sec$. The 2.6 readahead starts to overload the disk at $40sec$, and eventually achieved $5MB/s$ maximum network throughput. The on-demand readahead throughput keeps growing, and the trend is going up to $15MB/s$. That's three times better.
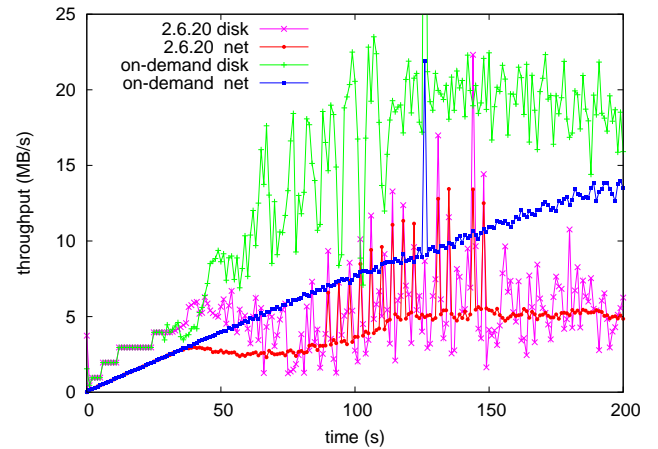
(a) disk/net throughput on loaded disk
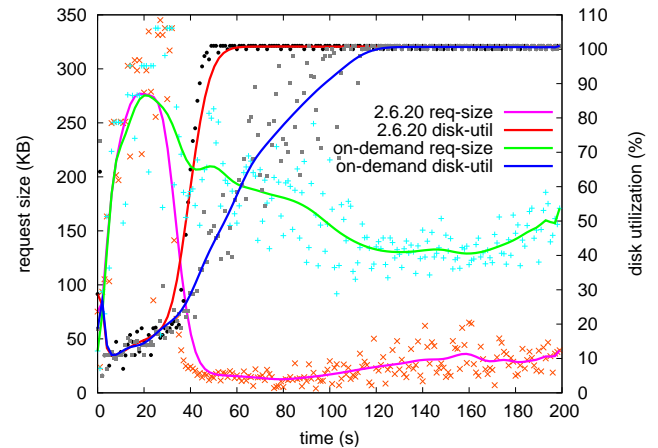
(b) average I/O size and disk utilization

Figure 6: performance on readahead thrashing

5 Conclusion

This work greatly simplified Linux 2.6 readahead algorithm. We successfully eliminated the complexity of dual windows, cache hit/miss, unaligned reads, and retried reads. The resulting code is more clean and should be easier to work with. It maintains roughly the same behavior and performance for common sequential/random access patterns. Performance on readahead thrashing and cache hits are improved noticeably.

6 Future Work

The algorithm is still young and imperfect. It needs more benchmarks, real-world tests, and fine tuning. Look-ahead size may be a bit smaller, especially for the initial readahead. It does not impose strict sequential

checks, which may or may not be good. The overlapped random reads may also be improved.

Then we can embrace all the fancy features that were missed for so long time. To name a few: interleaved reads from multimedia/multi-threaded applications; clustered random reads and chaotic semi-sequential reads from some databases; backward reads and stride reads in scientific arenas; real thrashing prevention and efficient use of readahead cache for file servers. Sure there are more. They can be packed into an optional kernel module for ease of use and maintenance.

References

[1] Andrew Morton, *[PATCH] readahead*, git commit
b546b96d0969de0ff55a3942c71413392cf86d2a [2]

[2] Andrew Morton, *[PATCH] readahead optimisations*, git commit
213f035476c932921d6281e4d5d39585f214a2eb

[3] Andrew Morton, *[PATCH] rework readahead for congested queues*, git commit
10d05dd588a3879f9b40725a9073bc97fcd44776

[4] Linus Torvalds, *[PATCH] Simplify and speed up mmap read-around handling*, git commit
d5cfe1b35c4e81f4c4dc5139bd446f04870ebf90

[5] Andrew Morton, *[PATCH] Allow VFS readahead to fall to zero*, git commit
71ddf2489c68cf145fb3f11cba6152de49e02793

[6] Ram Pai, *[PATCH] readahead: multiple performance fixes*, git commit
2bb300733b3647462bddb9b993a6f32d6cbcdbbc

[7] Ram Pai, *[PATCH] speed up readahead for seeky loads*, git commit
ef12b3c1abce83e8e25d27bdaab6380238e792ff

[8] Suparna Bhattacharya, Ram Pai, *[PATCH] adaptive lazy readahead*, git commit
87698a351b86822dabbd8c1a34c8a6d3e62e5a77

[9] Ram Pai, Badari Pulavarty, Mingming Cao, *Linux 2.6 performance improvement through readahead optimization*, http://www.linuxsymposium.org/proceedings/reprints/Reprint-Pai-OLS2004.pdf

[10] *Simplified Readahead*, http://groups.google.com/group/linux.kernel/browse_thread/thread/e5f475d4a11759ba/697d85a0d86458d3?&hl=en#697d85a0d86458d3

[11] Steven Pratt, Ram Pai, *[PATCH] Simplified readahead*, git commit
e8eb956c01529eccc6d7407ab9529ccc6522600f

[12] *Linux: Random File I/O Regressions In 2.6*, http://kerneltrap.org/node/3039

[13] Andrew Morton, *[PATCH] readahead: keep file->f_ra sane*, git commit
2ea7dd3fc9bc35ad0c3c17485949519cb691c097

[14] *Linux: Adaptive Readahead*, http://kerneltrap.org/node/6642

[15] Oleg Nesterov, *[PATCH] readahead: improve sequential read detection*, git commit
03a554d2325ef5f3160514359330965fd7640e81

[16] Suparna Bhattacharya, John Tran, Mike Sullivan, Chris Mason, *Linux AIO Performance and Robustness for Enterprise Workloads*, http://www.linuxsymposium.org/proceedings/reprints/Reprint-Bhattacharya-OLS2004.pdf

---

[2]all git commits are accessible from http://git.kernel.org/?p=linux/kernel/git/torvalds/old-2.6-bkcvs.git

Regression Test Framework and Kernel Execution Coverage

Hiro Yoshioka
Miracle Linux Corporation
hyoshiok@miraclelinux.com

Abstract

We have developed a Linux kernel regression test framework ("crackerjack") and a branch coverage test tool (hereinafter "btrax") to capture kernel regression. Crackerjack is a harness program and a set of test programs. It runs test programs, records the results, and then compares the expected results. Therefore, if a particular system call failed in a release and was then fixed in a later release, crackerjack records this as non-favorable because of incompatibility. The btrax is an integrated component of crackerjack and is a tool to assess test programs' effectiveness. It uses Intel processor's branch trace capability and records how much code was traced by a test program. Crackerjack is initially designed for Linux kernel system call testing, but care has been taken to allow future expansion to other types of software.

1 Introduction

1.1 Test Early, Test Often

The Linux kernel development process does not explicitly define an automated mechanism for maintenance of compatibility. Unintended introduction of incompatibilities as the result of a bugfix and/or an upgrade do not get detected in an automated way. The basis of testing in the Linux kernel development community is predicated on frequent releases, with feedback and review by a large number of contributors. Therefore, end-users and middleware developers can find incompatibility problems only after the release of a kernel.

Developers may introduce new functionality which, intentionally or unintentionally, introduced incompatibility but there are few cases which describe the incompatibility explicitly. For example, you may write in the change log, "Function XXX is added" but you may not write "There is an incompatibility YYY because of introducing function XXX."

The cost of validating such incompatibility for middleware developers is increasing, therefore we need some mechanism to find such incompatibility.

If you find the incompatibility early, then analysing the issue and fixing it is very easy.

However, if you find the issue very late, some applications may already make use of this incompatible behavior and therefore you can not change or fix the behavior, even the fix is trivial.

Therefore, finding incompatibility very early has practical benefit not only for Linux kernel developers but also for middleware developers.

1.2 Regression Testing

1.2.1 Necessities of Regression Testing

The word *regression* has a negative connotation; *degrade* also has a bad image. Incompatibility is not always a bad thing. We have to have some incompatibility to introduce new features, bug fixes, and performance improvements.

Although the word *regression* has negative connotation, we will use the word in this paper, because the term "regression test" is commonly used in the testing community.

Regression testing is a mechanism to find different (diff) behavior of implementation. Maintaining compatibility is very important. But it is not always possible, we have to have some tradeoff.

1.2.2   Automatic Test

Testing is a boring process but we'd like to change so it is a fun and easy process.

We need a systematic way to find a regression of the Linux Kernel and OSS.

Regression testing is an automatic process which runs a test and compares the expected results, then validates the compatibility of the software. Regression testing is a common practice in commercial software developments but not well done in the open source software.

As such, we have developed a compatibility testing tool for the Linux kernel interface, the goal of which is to avoid unintended introduction of incompatibilities, and to effectively reduce application development cost by detecting such incompatibilities upfront. This tool is different from standard certification tests such as Linux Standard Base, in that the tool can detect incompatibilities between a particular version of a kernel and its revised versions.

The test tool includes following features.

- Automatically assess kernel behaviors and storing the results.

- Detect differences between stored results and point out incompatibilities.

- Manage (register, modify, remove) test results and expected test results.

We will promote the development of this test tool by communicating with the Linux kernel community, the test community, the North-east Asia OSS Promotion Forum, etc. from an early stage, through the so-called bazaar model.

1.2.3   Expected Usage

Linux kernel developer

Bugfix of an existing kernel: Verify that the bugfix does not destroy previous features in an incompatible way. Add a test program for the bug in question and verify that the existing kernel includes the bug and the bugfix version does not.

Development of new features: Verify that the new feature in question does not destroy previous features in an incompatible way. If the feature extension was an incompatible extension, verify the extent of such incompatibility. Add test program for the new feature to maintain compatibility in the future.

Middleware developer

These developers should execute a regression test against a new version of kernel, and if incompatibilities are found, specifically understand the extent of the incompatiblity. It will be easy to find kernel regression, if middleware tests are prepared and run.

1.2.4   Non Goals

The following are non goals.

- Certification Tests e.g. POSIX, LSB

- Performance Tests, e.g. Benchmarks

We don't build certification tests nor performance tests. They are non goals.

1.3   Summary of theme

Develop a regression test framework, the regression tests, and the test coverage measurement tool, as shown in figure 1. Regression test framework is a framework that executes each of the test sets, and is a basis for the framework of this Linux kernel compatibility testing tool.

2   Regression Test Framework - crackerjack

Crackerjack is a regression test framework which provides 1) execution of test sets, 2) reporting based on results of test set execution, and 3) management of test programs, expected results, test sets, and test results.

It is implemented using Ruby on Rails. Ruby makes it easy to modify, ensuring a low maintenance cost.

Regression testing is defined as testing two builds of software, with emphasis on detection of regression (incompatibility of functionality).

In Figure 1, the result (`Rn`) of a test executed on a certain version of software (`Vn`) is compared against an expected result (`En`). The initial expectation (`E1`) is typically identical to `R1`.

```
        V1----- V2----- V3----- V4
        R1----- R2----- R3----- R4
        ^ ----- ^ ----- ^ ----- ^
compare | ~~~~~ | ~~~~~ | ~~~~~ |
        v ----- v ----- v ----- v
        E1----> E2----> E3----> E4
```

Figure 1: Comparison between test results (Rn) and expected results (En)

To detect a regression, the results from two separate builds have to be compared. If a result from a system call can be statically determined to be correct, such a comparison would return a valid result (OK) and or not (NG). We can add a comparison routine to determine the results.

```
pid = getpid ()
if (pid > 5000 && pid < 66536) {
        printf ("OK\n");
} else {
        printf ("NG\n");
}
return EXIT_SUCCESS
```

Figure 2: Example – Exact Match

```
pid = get_pid
printf ("%d\n", pid)
return EXIT_SUCCESS
```

Figure 3: Example - Range

In the first example above (see Figure 2), the test program statically determines whether the function call returned a valid result ('OK') or not ('NG'). In the second example (see Figure 3), the test program only outputs the function call result, for later comparison.

The comparison method may vary between test programs. Therefore, it is the test program developer's responsibility to define a comparison method for that test program.

The objective of the crackerjack system is to perform regression tests. crackerjack consists of (1) framework, (2) a collection of test programs, and (3) branch tracer

(btrax). It is outside the scope of crackerjack project, to measure performance or certify compatibility.

2.1 Software Functionality

The framework provides the following functions.

- Run both GUI (Graphical User Interface) mode and CUI (Character User Interface) mode.

- Execute test programs.

- Compare test execution results and expected results.

- Report based on results of test set execution.

- Manage test programs, expected results, test sets, test target, and test results.

- Define compare programs.

The expected end-users of crackerjack are kernel developers, middleware developers, and test program developers.

For more information, please refer to the Appendix.

3 Branch Tracer for Linux – btrax

3.1 btrax

We have integrated a branch tracer for Linux (known as btrax hereinafter) in the crackerjack to find the execution coverage of regression tests.

This program (btrax) traces the branch executions of the target program, analyzes the trace log file, and displays coverage information and execution path. It's possible to trace it for an application program, a library, a kernel module, and the kernel.

The btrax consists of the following commands.

1. Collect the branch trace log via `bt_collect_log`,

2. Report the coverage information via `bt_coverage`,

3. Report the execution path via `bt_execpath`, and

4. Split the branch trace log by each process via `bt_split`.

The btrax collects a branch trace information using Intel's last branch record capability. `bt_collect_log` stores the branch trace log, `bt_coverage` and `bt_execpath` report the branch coverage and the execution path information respectively.

3.2   Coverage

The coverage information gathered includes function, branch, and state coverage. Usually, it would be displayed in address value order.

The branch tracer (`bt_collect_log`) collects the last branch information using Intel's last branch record capability.

`bt_coverage` analyzes the ELF files such as kernel, module, etc. and gets the branch and function-call information. Then, it analyzes the traced log(s) with this information, and displays the coverage information.

Function coverage displays how many functions were executed among total functions. Displayed functions are almost compatible with objdump's output. Note that it would not show the "function call coverage." (See Figure 12.)

Branch coverage displays how many conditional branches (i.e. both branch and fall-through) were executed among total ones. (See Figure 13.)

State (basic block) coverage shows how many states (basic block) were executed among total states. State means straight-line piece of code without any jumps or jump targets in the middle (a.k.a. basic block). In the previous "switch case" example, if the codes of the "case 0" and "case 2" were both executed three times, then each state and coverage would be as follows. (See Figure 15.)

A chain of function calls is displayed, for example, function FA calls function FB, and function FB calls function FC, and so on. If you would like to limit the coverage target to the functions which are included in function call tree, -I option can be used. In this case, the function call tree would be displayed with other coverage information (such as function/branch/state coverage). Example of the function call tree is shown below. (See Figure 16.)

4   Linux Kernel System Call Test Coverage

We made test programs which use Linux system calls and used the crackerjack with the btrax. We measured the function coverage, the branch coverage, and the state coverage of the each test program execution (Table 1).

We selected 50 system call test programs of LTP and measured the execution coverages of them as our benchmark. (Table 2)

We know the LTP is comprehensive test suite but the execution coverage is not large enough.

The average function coverage, branch coverage, and state coverage are 41.39%, 23.1%, and 30.94% respectively. (Note: 38%, 10%, and 12% of the test programs exceeded 50% of function, branch, and state coverage.)

It is very difficult to increase the execution coverage. The following are the reasons.

- exception/error condition: Making an exception program is not easy. Making an Error condition is not easy. Sometimes it is not feasible.

- asynchronous processing: For example, making spin/lock, spin/wait condition is not easy.

- excluding functions from coverage measures: For example, a common routine like printk has a lot of execution path. If a given system call uses printk, it contains a lot of execution path which are not covered the test programs.

We need to exclude some functions from the coverage measurement but it is hard to find such functions. Pruning unnecessary code path is difficult. If a given system call has more than three digits of function calls, it implies insufficient pruning.

It is almost impossible to run codepaths for low memory situations. It is even questionable that such test could even run.

System call	Func coverage %	branch coverage %	state coverage %
13 time	100.0	70.0	88.24
20 getpid	0.00	100.00	0.00
20 getpid (wb)	100.00	100.00	100.00
25 stime	69.23	41.67	50.68
27 alarm	14.19	2.50	5.65
30 utim (wb)	30.77	10.79	16.73
30 utime	23.56	8.81	13.38
42 pipe	67.74	31.91	49.79
42 pipe (wb)	82.26	47.04	65.60
45 brk	23.88	8.96	12.74
45 brk (wb)	23.13	9.12	12.53
66 setsid	29.41	12.08	18.10
78 gettimeofday	87.50	52.00	70.31
79 settimeofday	77.78	45.19	57.04
82 old_select	3.96	1.78	2.67
87 swapon	19.12	9.05	12.21
90 old_mmap	7.14	2.91	3.54
91 munmap	13.74	5.78	7.66
92 truncate	4.48	1.06	1.77
93 ftruncate	0.88	0.11	0.20
101 ioperm	19.78	8.84	10.91
103 syslog	5.45	1.67	2.44
104 setitimer	3.79	1.44	2.02
105 getitimer	1.35	0.45	0.61
110 iopl	100.00	50.00	81.82
115 swapoff	12.28	4.12	6.18
116 sysinfo	15.11	3.31	5.79
120 clone	19.30	6.79	9.92
121 setdomainname	7.46	1.88	3.15
124 adjtimex	5.67	1.05	1.79
125 mprotect	7.82	3.90	5.28
142 select	3.96	1.78	2.67
144 msync	1.52	0.42	0.62
147 getsid	72.73	46.67	61.70
150 mlock	2.08	0.37	0.72
151 munlock	1.66	0.31	0.56
152 mlockall	24.12	9.29	12.62
153 munlockall	1.66	0.21	0.44
162 nanosleep	31.76	13.41	18.67
163 mremap	13.83	6.21	8.71
168 poll	2.30	0.95	1.47
187 sendfile	1.01	0.11	0.19
193 truncate64	4.48	1.06	1.77
194 ftruncate64	0.88	0.11	0.2
203 setreuid	1.27	0.23	0.38
204 setregid	100.00	20.00	46.43
205 getgroups	1.54	0.25	0.41
206 setgroups	1.26	0.10	0.23
208 setresuid	1.27	0.26	0.41
209 getresuid	66.67	33.33	60.00
210 setresgid	100.00	18.42	45.16
211 getresgid	66.67	33.33	60.00
218 mincore	0.62	0.06	0.10
219 madvise	1.60	0.50	0.73

Table 1: Summary of coverage of tests %

System call	Func coverage %	branch coverage %	state coverage %
13 time	75.00	60.00	82.35
20 getpid	100.00	100.00	100.00
25 stime	61.54	35.42	47.95
27 alarm	53.45	15.58	25.03
30 utime	28.85	9.83	15.27
42 pipe	77.42	44.08	61.32
45 brk	13.43	6.76	8.88
66 setsid	23.53	7.92	12.76
78 gettimeofday	87.50	44.00	67.19
79 settimeofday	66.67	32.69	45.77
82 old_select	No exist	No exist	No exist
87 swapon	29.08	11.40	16.80
90 old_mmap	17.41	11.25	13.11
91 munmap	21.15	9.45	13.24
92 truncate	19.77	6.21	9.73
93 ftruncate	15.48	4.38	7.22
101 ioperm	4.27	1.56	2.02
103 syslog	26.32	13.64	14.13
104 setitimer	48.44	11.01	21.05
105 getitimer	67.86	24.65	40.64
110 iopl	100.00	100.00	84.62
115 swapoff	20.61	5.90	9.75
116 sysinfo	86.96	40.35	59.06
120 clone	27.52	11.48	16.80
121 setdomainname	100.00	50.00	71.43
124 adjtimex	57.14	16.18	26.49
125 mprotect	11.53	5.70	7.57
142 select	10.88	4.72	6.61
144 msync	5.50	2.45	3.25
147 getsid	72.73	26.67	48.94
150 mlock	20.60	8.39	12.23
151 munlock	5.32	3.12	3.76
152 mlockall	17.82	7.08	9.88
153 munlockall	2.31	0.39	0.69
162 nanosleep	58.82	15.16	25.32
163 mremap	17.57	7.68	11.04
168 poll	6.12	1.86	3.13
187 sendfile	12.77	3.72	6.05
193 truncate64	No exist	No exist	No exist
194 ftruncate64	No exist	No exist	No exist
203 setreuid	11.76	4.08	5.69
204 setregid	100.00	96.43	96.43
205 getgroups	83.33	40.00	60.78
206 setgroups	9.43	2.95	4.55
208 setresuid	10.92	4.14	5.68
209 getresuid	66.67	50.00	80.00
210 setresgid	100.00	68.42	96.88
211 getresgid	66.67	50.00	80.00
218 mincore	13.57	4.85	7.04
219 madvise	11.43	4.03	6.15

Table 2: Summary of coverage of tests (LTP)%

In spin lock situations, program code normally runs through locked side path. It is hard to prepare conditions for competing locks.

Making race condition is also hard to test.

There is a common function called from gettimeofday / settimeofday. In this common function, paths are uniquely determined by get / set. So other execution paths have never been executed.

5 Discussion

5.1 Kernel Tests – Writing a Good Test is Very Hard

5.1.1 Test Coverage

Writing good kernel tests is very difficult. Our data shows that the execution coverage is low. As discussed above, it is very hard to increase the execution coverage.

5.1.2 man Pages are Not Enough

Some man pages do not have enough detail information. For example, utime does not have a description of error return values.

If we don't have clear definition, we can not determine if the test result is OK or NG.

5.1.3 Behavior of 2.4 vs. 2.6

2.6 introduced more strict parameter checking. We discovered a condition where 2.4 did not report errors but 2.6 did. (We discovered implementation-dependent incompatibility.)

For example, settimeofday second had loose error checking, but 2.6 introduced more strict error checking.

We think fixing a bug is important but changing behavior may introduce other incompatibility. So we need to know as early as possible to assess it.

5.2 Finding Incompatibility

For portability, middleware developers should preferably not be using implementation-dependent and undefined functions. However there are cases when such functions are used without intention.

There is a large hidden cost to avoid unintended introduction of incompatibility. It is said that the most costly activity in development of commercial software is the maintenance of backward compatibility.

Crackerjack detects not only unintended incompatibilities, but also intended incompatibilities. Both are important for notification to middleware developers. It is important to notify the changes in behavior, in a timely manner. We can detect behavior diffs across versions.

For example, the memory range acquired from `brk()` is static but the memory range is randomized by turning on the exec shield. Crackerjack detects such a specification change.

We can write user-defined compare program not only simple OK/NG but allows some statistical allowance.

6 Related Work

6.1 LTP

LTP (Linux Test Project) is a set of Linux test programs which includes the Linux kernel test, stress tests, benchmark tests, and so on. LTP is a comprehensive test suite but it is not intended to be a regression test suite.

The Linux kernel test validates the POSIX definition of the kernel therefore it does not cover features like implementation defined, implementation dependent, and or undefined functionality.

On the other hand, crackerjack is a test harness and tries to capture all execution behavior and difference between each version. Such behavior includes not only standard features but also implementation defined, implementation dependent, and undefined by the standard.

Crackerjack finds the diff of implementation behavior.

We can integrate crackerjack with the LTP. For example, adding some regression tests to LTP and invoke them from crackerjack.

6.2 gcov/lcob

The gcov/lcob is a coverage tool. There is a kernel patch to measure the test coverage of the Linux kernel. The gcov uses gcc to get the coverage data but you need to patch the kernel.

The btrax uses Intel's last branch record capability and you don't need any patch nor rebuild kernel. So you can measure your standard kernel without rebuilding kernel.

6.3 autotest

The autotest is a harness program of invoking several tests program including LTP, benchmark programs and so on.

We can plug crackerjack into autotest.

7 Future Work

7.1 Kernel Development Process

We believe finding incompatibility in an early stage is very important and all of us can get much benefit from it.

It is good thing to run regression tests on every kernel release. Adding this practice into the kernel development process is our big challenge. We need to show our benefit to the kernel community and convince them to use it.

7.2 Expanding the Area

The concept of regression testing is very simple, just find the diff of the implementation between releases. Crackerjack can be used on other types of software interfaces, for example, the `/proc` file system, glibc, and so on.

The system calls are very stable and we don't see much incompatibility in them. However, /proc file system is much more flexible, so we may easily find some incompatibility.

7.3 Crackerjack

A richer set of default compare programs for crackerjack is needed. Today we have to add compare programs if the default does not match your needs.

We need methodology to relieve the test program developers. It might be a test pattern, convention or environment.

Crackerjack has to record system information, environment, and reproducible information.

Current development focuses on regression test framework and test coverage tool. Extending the test to all of Linux kernel functions, and sustained execution of regression tests, would wait until the next project.

7.4 Community Activity

Our development is supported by a working group of Japan OSS promotion forum which consists of private sector and public sector. There is a collaboration with China and Korea groups too.

We'd like to expand this activity with the Linux and test community.

8 Acknowledgements

References

[1] Linux Test Project
    http://ltp.sourceforge.net/

[2] autotest
    http://test.kernel.org/autotest/

[3] ABAT http://test.kernel.org

[4] Test Tools Wiki(OSS Testing Summit)
    http://developer.osdl.org/dev/test_
    tools/index.php/Main_Page

[Eric Raymond] The Cathedral and the Bazaar
    http://www.catb.org/~esr/writings/
    cathedral-bazaar/cathedral-bazaar

[5] Ruby on Rails
    http://www.rubyonrails.org/

Appendix:

A crackerjack – Getting Started

A.1 Get the Code and make

The latest source code is always available at https://crackerjack.svn.sourceforge.net/svnroot/crackerjack/

You can get it by Subversion. For example, see the Figure 4 "Getting Source code and make."

```
$ svn co \
  https://crackerjack.svn.sourceforge.net/svnroot/crackerjack/
A    crackerjack/trunk
A    crackerjack/trunk/crackerjack
(...snip...)
Checked out revision 477.

$ make
(...snip...)
```

Figure 4: Getting Source Code and make

If a test program has compiler errors, then try `make -k`. Some system calls are not available in old Linux kernels, for example, 2.6.9 does not have mkdirat, mknodat etc.

```
$ su -
Password:
# cd /usr/src/crackerjack/trunk/crackerjack/
# ./crackerjack
crackerjack>h
 number push test program to stack
 d      Delete(pop) stack
 e      register test program result
        on stack as Expected result
 h      help
 l      List the test programs
 p      Print stack
 x      eXecute test program on stack
```

Figure 5: Invoke crackerjack

```
crackerjack>l
0000) access
0001) adjtimex
0002) adjtimex/whitebox
0003) alarm
...
```

Figure 6: List the Tests Command

A.2 Invoke crackerjack

Become the root user and run crackerjack.

Read help with 'h' command. (See Figure 5)

List the test programs with 'l' command. (See Figure 6)

Push the test programs to the stack with "number" which corresponds to the test program you want. You can look the contents of stack with `p` command, and pop the test programs from stack with `d` command.

Execute the test program on stack with the 'x' command. (See Figure 7.)

Quit the crackerjack with the 'q' command. (See Figure 8.)

A.3 Run in Non-interactive way

You can execute test programs from a file which indicates the order of tests (order file). (See Figure 9.)

You can create an order-file using the 'l' command similar to the following then execute the test. (See Figure 10.)

```
crackerjack>1
1
crackerjack>2
1 2
crackerjack>3
1 2 3
crackerjack>4
1 2 3 4
crackerjack>5
1 2 3 4 5
crackerjack>x
 Action   SystemCallName       Id
 X        adjtimex             20070412133111
 X        adjtimex/whitebox    20070412133111
 X        alarm                20070412133111
 X        alarm/whitebox       20070412133111
 X        brk/basic            20070412133111
```

Figure 7: Execute Tests

```
crackerjack>q
#
```

Figure 8: Quit with the 'q' Command

A.4   GUI mode

Invoke the GUI server similar to the following then use a web browser to access localhost:3000. (See Figure 11.)

You may need to install the Ruby on Rails.

B   btrax – Getting Started

B.1   Basic Concept

This program (btrax) traces the branch executions of the target program, analyzes the trace log file, and displays coverage information and execution path. It's possible to trace it for an application program, a library, a kernel module, and the kernel.

The btrax consists of the following commands.

```
# ./crackerjack -h
USGE: crackerjack [option] [FILE]
-l         list the test programs
-x FILE    execute the test program on reading order from file
-e FILE    register the current result as the expected result
-c result_kerv,result_id,expected_kerv,expected_id
           compare the current result to the expected result
-b         execute with btrax, avaiable with -e option
-h         show this help
-v         show version
```

Figure 9: crackerjack Non-interactive Mode

```
# ./crackerjack -l > m.order
# ./crackrjack -x m.order
```

Figure 10: crackerjack Non-interactive with 'l' and 'x' Commands

```
# ./crackerjack-gui-server
```

Figure 11: crackerjack GUI Mode

1. Collect the branch trace log via `bt_collect_log`,

2. Report the coverage information via `bt_coverage`,

3. Report the execution path via `bt_execpath`,

4. Split the branch trace log by each process via `bt_split`.

The btrax collects a branch trace information using Intel's last branch record capability. `bt_collect_log` stores the branch trace log, `bt_coverage` and `bt_execpath` report the branch coverage and the execution path information respectively.

B.2   Coverage

The coverage information includes function coverage, branch coverage and state coverage. Usually, it would be displayed in address value order.

B.2.1   bt_coverage

`bt_coverage` analyzes the ELF files such as kernel, module, etc. and get the branch and function-call information. Then, it analyzes the traced log(s) with this information, and displays the coverage information.

`bt_coverage` tries to show the source information such as source file name and line number, but if there is no debug information in the ELF file, it shows only the address value.

You can check whether the source code was executed or not by using html output (for this, it needs debug information in the ELF file and the source code files). Note that inline functions and macros would not be colorized

correctly in the html files. It is because they were expanded to the other functions and could not be found in the ELF file.

You can also compare the two log files' kernel coverage by generating html files. Even if the log files were generated on the different kernels, `bt_coverage` can still compare them.

B.2.2 Function Coverage

It displays how many functions were executed among total functions. Displayed functions are almost compatible with objdump's output. Note that it would not show the "function call coverage." For example, refer to the below chart ('//' means comment). (See Figure 12.)

```
(example source code)
funcA(); // executed once
funcB(); // executed once
...
funcB();
...
funcB();
funcC();

(then, coverage output would be...)
---- function coverage (2/3=66.67%) ----
(OK) <funcA> (1) // funcA executed once
(OK) <funcB> (1) // funcB executed once
(NT) <funcC> (0) // funcC not executed
```

Figure 12: Function Coverage

B.2.3 Branch Coverage

It displays how many conditional branches (i.e. both branch and fall-through) were executed among total ones. For example, branch coverage counting for each coverage case is as follows. (See Figure 13.)

There is a case that the branch address would be determined by indirect addressing such as "switch case" code. In this case, it is impossible to know the branch addresses and number of these by analyzing the ELF file. (See Figure 14.)

We call this kind of branch as "unknown branch." Unknown branches are counted whether the branch was executed or not, and are counted separate from normal branches. In this example, if all of the switch case codes were not executed, branch coverage would be as follows.

```
(example source code)
 Address   C              Assembler
    1:    if (xxx)                jxx LABEL
    2:      aaa;                   aaa
    3:    bbb;         LABEL:      bbb
(coverage counting for each case)
```

branch (1->3)	fall-through (1->2)	coverage counting	symbols
not executed	not executed	0/2=0.00%	NT
not executed	executed	1/2=50.00%	HT
executed	not executed	1/2=50.00%	HT
executed	executed	2/2=100.00%	OK

(if branch was executed 3 times, and fall-through was executed 1 time, then coverage output would be...)

```
---- branch coverage (OK:1,HT:0,NT:0/2=100.00% UK:0/0=100.00%)

(OK) 1 [3/1] 3:2 // 1->3 3 times, and 1->2 1 time executed
```

Figure 13: Branch Coverage

```
(example source code)
    Address  C                        Assembler
    -------  ------------------       -------------------
       1: switch (xxx) {              jmp *(%eax)
       2: case 0: aaa;                aaa
       3:        break;               jmp LABEL
       4: case 1: bbb;                bbb
       5:        break;               jmp LABEL
       6: case 2: ccc;                ccc
       7:        break;       LABEL:
       8: }
```

Figure 14: Branch Coverage

```
---- branch coverage (OK:0,HT:0,NT:0/0=100.00% UK:0/1=0.00%)
(UN) 1 [0/x] ----------:xxxxxxxxxxx      // 1 jumps nowhere
```

Or, if the codes of the "case 0" and "case 2" were both executed 3 times, then branch coverage would be as follows.

```
---- branch coverage (OK:0,HT:0,NT:0/0=100.00% UK:1/1=100.00%)
(UT) 1 [3/x] 2:xxxxxxxxxxx           // 1->2 executed 3 times
(UT) 1 [3/x] 6:xxxxxxxxxxx           // 1->6 executed 3 times
```

In the coverage output, you cannot check that how many branches were executed for the unknown branches. But each case block's execution information would be showed in state coverage. Although you can check that in html output.

B.2.4 State (Basic Block) Coverage

It shows how many states (basic block) were executed among total states. State means straight-line piece of code without any jumps or jump targets in the middle *a.k.a. basicblock*. In the previous 'switch case' example, if the codes of the "case 0" and "case 2" were both executed three times, then each state and coverage would be as follows. (See Figure 15.)

```
(example source code)
Address  C                    Assembler
-------  ----------------     ------------------
   1: switch (xxx) {                jmp *(%eax)
      - - - - - - - - - - - -// state border
   2: case 0: aaa;                   aaa
   3:       break;                   jmp LABEL
      - - - - - - - - - - - - - - -
   4: case 1: bbb;                   bbb
   5:       break;                   jmp LABEL
      - - - - - - - - - - - - - - -
   6: case 2: ccc;                   ccc
      - - - - - - - - - - - - - -
   7:       break;        LABEL:
   8: }

(coverage output would be...)
------ state coverage (4/5=80.00%) ------
(OK) 1    // state from address 1 was executed
(OK) 2
(NT) 4    // state from address 4 was not executed
(OK) 6
(OK) 7
```

Figure 15: State (basic block) Coverage

B.2.5 Function Call Tree

It means the chain of function call, for example, function FA calls function FB, and function FB calls function FC, and so on. If you would like to limit the coverage target to the functions which are included in function call tree, -I option can be used. In this case, function call tree would be displayed with other coverage information (such as function/branch/state coverage). Example of the function call tree is shown below. (See Figure 16.)

```
==== includes: sys_open ======
==== excludes: schedule,printk,dump_stack,panic,show_mem ====
==== function tree (59/498=11.85%) ======
(OK) <sys_open>:fs/open.c,1101      (3, F:498)
(OK) +-<do_sys_open>:fs/open.c,1079      (3, F:497)
(OK) +-+-<do_filp_open>:fs/open.c,874    (3, F:196)
(OK) +-+-+-<nameidata_to_filp>:fs/open.c,942 (3, F:4)
(OK) +-+-+-+-<__dentry_open>:fs/open.c,799   (3, F:3)
(NT) +-+-+-+-<wake_up_process>:kernel/sched.c,1521 (0, F:1)
//... snip ...
(UT) <generic_file_open>:fs/open.c,1216 (2)
(UT) <dummy_inode_follow_link>:security/dummy.c,324   (1)
//... snip ...
```

Figure 16: Function Call Tree

In this example, we can see that the sys_open was executed (it is seen as 'OK') 3 times and it contains 498 functions including itself (it is seen as (3, F:498)). If the function was already displayed, then it is displayed with (--) symbol instead of (OK) or (NT).

A function call tree may contain unnecessary functions for the user. In this case, it could be excluded by using -E option. You can also check that if it would be excluded, then how many functions would be decreased by using "F:N" information (we call it "function excluding guidance") in the function call tree.

Function excluding guidance is also displayed in the

"function coverage," and each function's information is sorted by this value.

There is the function call whose address would be determined by indirect addressing such as "function call by using function pointer." In this case, it is impossible to know the function address by analyzing the ELF file. So, this kind of functions would not be displayed in the function tree. Example of this kind of function call is shown below.

```
(example source code)
Address  C                               Assembler
-------  ------------------------------  ----------------
   1: int call_function(void (*func),,, )
   2: {
   3:       func();                        call *0x84(%edx)
   4: }
```

If you would like to include this kind of function to the coverage target, you must check the source code if there is no information. To improve this situation, (bt_coverage) also analyzes the trace log(s) and shows the functions which were executed by indirect addressing. This kind of function is displayed with (UT) mark (we call it "function including guidance"). Read the man page for more information.

B.3 Get the Code, Build and Install

The project home page is the following, Download the tarball from the project page.

```
http://sourceforge.net/projects/
btrax/
```

Read the README and follow the instruction. (See Figure 17.)

B.4 Checking System Call Coverage

In order to determine the correctness of our regression test, we measure the execution branch coverage of tests. The following subsections show the steps to take to measure the branch coverage using the btrax.

B.4.1 Create a program calling system call

Create a regression test which uses a system call(s). (You can use LTP as an example.)

```
Build and Install.

 $ cd $(WHERE_YOUR_WORK_DIRECTORY)
 $ tar jxvf btrax-XXX.tar.bz2
 $ cd btrax-XXX

Install the command.

 $ make
 $ su (input super-user password here)
 # make install

Create relayfs mount point.

 # mkdir /mnt/relay
```

Figure 17: Build and Install

B.4.2   Start Branch Trace

Start branch trace.   (See Figure 18.)   Note that
`syscall_name` must be defined in the kernel's `sys_`
`call_table`.

```
# bt_collect_log --syscall \
  $(syscall_name) -d $(ODIR)\
  -c $(program)
```

Figure 18: Start branch trace

B.4.3   Checking Coverage

To check the system call coverage summary, do the next
command. (See Figure 19.)

```
# cd $(ODIR)
# bt_coverage --ker -f \
  'echo $(ls cpu*)|sed 's/\s\+/,/g'' \
  -I $(syscall_name) -s
```

Figure 19: Checking coverage

To check the system call coverage detail, do the next
command. (See Figure 20.)

If you want to exclude the function(s) from coverage
result, use the -E option. (See Figure 21.) To check
whether the code in that system call was executed or
not, do the next. (See Figure 22.) Note that html files are
using the Javascript. If you have some trouble browsing,
check the Javascript setting.

```
# bt_coverage --ker -f \
  'echo $(ls cpu*)|sed 's/\s\+/,/g'' \
  -I $(syscall_name)
```

Figure 20: Checking Call coverage

```
# bt_coverage --ker -f \
  'echo $(ls cpu*)|sed 's/\s\+/,/g'' \
  -I $(syscall_name) \
  -E schedule,dump_stack,printk
```

Figure 21: Excluding functions

```
# bt_coverage --ker -f \
  'echo $(ls cpu*)|sed 's/\s\+/,/g'' \
  -I $(syscall_name) \
  -E schedule,dump_stack,printk \
  -o $(HTML_OUT_DIR) -S $(KERN_SRC_DIR)
# (mozilla) file://$(HTML_OUT_DIR)/top.html
```

Figure 22: Excluding functions

B.4.4   Compare System Call Coverage

```
# bt_coverage --ker \
  -I $(syscall_name)\
  -E schedule,dump_stack,printk \
  -o $(HTML_OUT_DIR)\
  -S $(KERN_SRC_DIR)\
  -f 'echo $(log1/cpu*)|sed 's/\s\+/,/g'' \
  --f2 'echo $(log2/cpu*)|sed 's/\s\+/,/g''
# (mozilla) file://$(HTML_OUT_DIR)/top.html
```

Figure 23: Comparing the system call coverage

To compare same kernel's system call coverage, do the
next command line which uses the -S, -f and –f2 op-
tions. In this example, each log directory is log1 and
log2. (See Figure 23.)

```
# bt_coverage --ker\
 -I $(syscall_name)\
 -E schedule,dump_stack,printk \
 -o $(HTML_OUT_DIR)\
 -u uname_r1 --u2 uname_r2\
 -S src1 --S2 src2 \
 -f 'echo $(log1/cpu*)|sed 's/\s\+/,/g''\
 --f2 'echo $(log2/cpu*)|sed 's/\s\+/,/g''
# (mozilla) file://$(HTML_OUT_DIR)/top.html
```

Figure 24: Comparing the system call coverage

If you had traced different kernel's system calls, you can
also compare these logs. To compare the different ker-
nel's system call coverage, use the next command line
which also employs the −u option. In this example, each
log directory is `log1` and `log2`, each kernel version is
`uname_r1` and `uname_r2`, and each kernel source di-
rectory is `src1` and `src2`. (See Figure 24.)

Enable PCI Express Advanced Error Reporting in the Kernel

Yanmin Zhang and T. Long Nguyen
Intel Corporation
`yanmin.zhang@intel.com, tom.l.nguyen@intel.com`

Abstract

PCI Express is a high-performance, general-purpose I/O Interconnect. It introduces AER (Advanced Error Reporting) concepts, which provide significantly higher reliability at a lower cost than the previous PCI and PCI-X standards. The AER driver of the Linux kernel provides a clean, generic, and architecture-independent solution. As long as a platform supports PCI Express, the AER driver shall gather and manage all occurred PCI Express errors and incorporate with PCI Express device drivers to perform error-recovery actions.

This paper is targeted toward kernel developers interested in the details of enabling PCI Express device drivers, and it provides insight into the scope of implementing the PCI Express AER driver and the AER conformation usage model.

1 Introduction

Current machines need higher reliability than before and need to recover from failure quickly. As one of failure causes, peripheral devices might run into errors, or go crazy completely. If one device is crazy, device driver might get bad information and cause a kernel panic: the system might crash unexpectedly.

As a matter of fact, IBM engineers (Linas Vepstas and others) created a framework to support PCI error recovery procedures in-kernel because IBM Power4 and Power5-based pSeries provide specific PCI device error recovery functions in platforms [4]. However, this model lacks the ability to support platform independence and is not easy for individual developers to get a Power machine for testing these functions. The PCI Express introduces the AER, which is a world standard. The PCI Express AER driver is developed to support the PCI Express AER. First, any platform which supports the PCI Express could use the PCI Express AER driver to process device errors and handle error recovery accordingly. Second, as lots of platforms support the PCI Express, it is far easier for individual developers to get such a machine and add error recovery code into specific device drivers.

2 PCI Express Advanced Error Reporting Driver

2.1 PCI Express Advanced Error Reporting Topology

To understand the PCI Express Advanced Error Reporting Driver architecture, it helps to begin with the basics of PCI Express Port topology. Figure 1 illustrates two types of PCI Express Port devices: the Root Port and the Switch Port. The Root Port originates a PCI Express Link from a PCI Express Root Complex. The Switch Port, which has its secondary bus representing switch internal routing logic, is called the Switch Upstream Port. The Switch Port which is bridging from switch internal routing buses to the bus representing the downstream PCI Express Link is called the Switch Downstream Port. Each PCI Express Port device can be implemented to support up to four distinct services: native hot plug (HP), power management event (PME), advanced error reporting (AER), virtual channels (VC).

The AER driver development is based on the service driver framework of the PCI Express Port Bus Driver design model [3]. As illustrated in Figure 2, the PCI Express AER driver serves as a Root Port AER service driver attached to the PCI Express Port Bus driver.

2.2 PCI Express Advanced Error Reporting Driver Architecture

PCI Express error signaling can occur on the PCI Express link itself or on behalf of transactions initiated on the link. PCI Express defines the AER capability, which is implemented with the PCI Express AER Extended
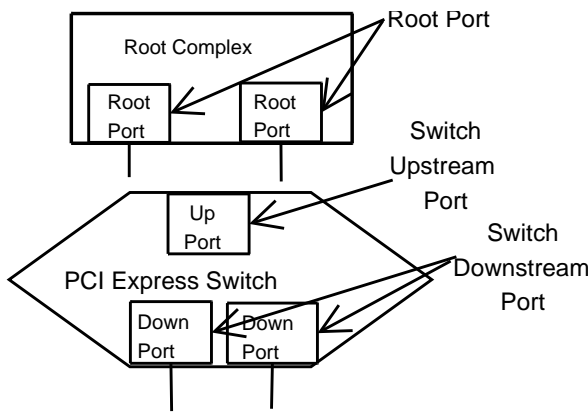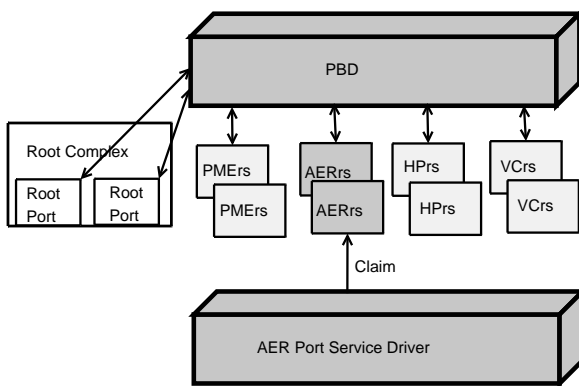
Figure 1: PCI Express Port Topology

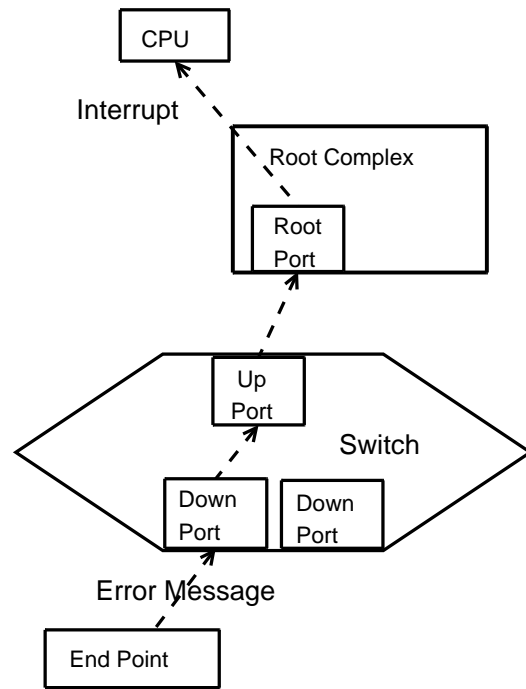Figure 2: AER Root Port Service Driver

Figure 3: PCI Express Error Reporting procedures

When errors happen, the PCI Express AER driver could provide such infrastructure with three basic functions:

- Gathers the comprehensive error information if errors occurred.

- Performs error recovery actions.

- Reports error to the users.

2.2.1 PCI Express Error Introduction

Traditional PCI devices provide simple error reporting approaches, PERR# and SERR#. PERR# is parity error, while SERR# is system error. All non-PERR# errors are SERR#. PCI uses two independent signal lines to represent PERR# and SERR#, which are platform chipset-specific. As for how software is notified about the errors, it totally depends on the specific platforms.

To support traditional error handling, PCI Express provides baseline error reporting, which defines the basic error reporting mechanism. All PCI Express devices have to implement this baseline capability and must map required PCI Express error support to the PCI-related error registers, which include enabling error reporting

Capability Structure, to allow a PCI Express component (agent) to send an error reporting message to the Root Port. The Root Port, a host receiver of all error messages associated with its hierarchy, decodes an error message into an error type and an agent ID and then logs these into its PCI Express AER Extended Capability Structure. Depending on whether an error reporting message is enabled in the Root Error Command Register, the Root Port device generates an interrupt if an error is detected. The PCI Express AER service driver is implemented to service AER interrupts generated by the Root Ports. Figure 3 illustrates the error report procedures.

Once the PCI Express AER service driver is loaded, it claims all AERrs service devices in a system device hierarchy, as shown in Figure 2. For each AERrs service device, the advanced error reporting service driver configures its service device to generate an interrupt when an error is detected [3].

and setting status bits that can be read by PCI-compliant software. But the baseline error reporting doesn't define how platforms notify system software about the errors.

PCI Express errors consist of two types, correctable errors and uncorrectable errors. Correctable errors include those error conditions where the PCI Express protocol can recover without any loss of information. A correctable error, if one occurs, can be corrected by the hardware without requiring any software intervention. Although the hardware has an ability to correct and reduce the correctable errors, correctable errors may have impacts on system performance.

Uncorrectable errors are those error conditions that impact functionality of the interface. To provide more robust error handling to system software, PCI Express further classifies uncorrectable errors as fatal and non-fatal. Fatal errors might cause corresponding PCI Express links and hardware to become unreliable. System software needs to reset the links and corresponding devices in a hierarchy where a fatal error occurred. Non-fatal errors wouldn't cause PCI Express link to become unreliable, but might cause transaction failure. System software needs to coordinate with a device agent, which generates a non-fatal error, to retry any failed transactions.

PCI Express AER provides more reliable error reporting infrastructure. Besides the baseline error reporting, PCI Express AER defines more fine-grained error types and provides log capability. Devices have a header log register to capture the header for the TLP corresponding to a detected error.

Correctable errors consist of receiver errors, bad TLP, bad DLLP, REPLAY_NUM rollover, and replay timer time-out. When a correctable error occurs, the corresponding bit within the advanced correctable error status register is set. These bits are automatically set by hardware and are cleared by software when writing a "1" to the bit position. In addition, through the Advanced Correctable Error Mask Register (which has the similar bitmap like advanced correctable error status register), a specific correctable error could be masked and not be reported to root port. Although the errors are not reported with the mask configuration, the corresponding bit in advanced correctable error status register will still be set.

Uncorrectable errors consist of Training Errors, Data Link Protocol Errors, Poisoned TLP Errors, Flow Con-

trol Protocol Errors, Completion Time-out Errors, Completer Abort Errors, Unexpected Completion Errors, Receiver Overflow Errors, Malformed TLPs, ECRC Errors, and Unsupported Request Errors. When an uncorrectable error occurs, the corresponding bit within the Advanced Uncorrectable Error Status register is set automatically by hardware and is cleared by software when writing a "1" to the bit position. Advanced error handling permits software to select the severity of each error within the Advanced Uncorrectable Error Severity register. This gives software the opportunity to treat errors as fatal or non-fatal, according to the severity associated with a given application. Software could use the Advanced Uncorrectable Mask register to mask specific errors.

2.2.2 PCI Express AER Driver Designed To Handle PCI Express Errors

Before kernel 2.6.18, the Linux kernel had no root port AER service driver. Usually, the BIOS provides basic error mechanism, but it couldn't coordinate corresponding devices to get more detailed error information and perform recovery actions. As a result, the AER driver has been developed to support PCI Express AER enabling for the Linux kernel.

2.2.2.1 AER Initialization Procedures

When a machine is booting, the system allocates interrupt vector(s) for every PCI Express root port. To service the PCI Express AER interrupt at a PCI Express root port, the PCI Express AER driver registers its interrupt service handler with Linux kernel. Once a PCI Express root port receives an error reported from the downstream device, that PCI Express root port sends an interrupt to the CPU, from which the Linux kernel will call the PCI Express AER interrupt service handler.

Most of AER processing work should be done under a process context. The PCI Express AER driver creates one worker per PCI Express AER root port virtual device. Depending on where an AER interrupt occurs in a system hierarchy, the corresponding worker will be scheduled.

Most BIOS vendors provide a non-standard error processing mechanism. To avoid conflict with BIOS while handling PCI Express errors, the PCI Express AER

driver must request the BIOS for ownership of the PCI Express AER via the ACPI _OSC method, as specified in PCI Express Specification and ACPI Specification. If the BIOS doesn't support the ACPI _OSC method, or the ACPI _OSC method returns errors, the PCI Express AER driver's probe function will fail (refer to Section 3 for a workaround if the BIOS vendor does not support the ACPI _OSC method).

Once the PCI Express AER driver takes over, the BIOS must stop its activities on PCI Express error processing. The Express AER driver then configures PCI Express AER capability registers of the PCI Express root port and specific devices to support PCI Express native AER.

2.2.2.2 Handle PCI Express Correctable Errors

Because a correctable error can be corrected by the hardware without requiring any software intervention, if one occurs, the PCI Express AER driver first decodes an error message received at PCI Express root port into an error type and an agent ID. Second, the PCI Express AER driver uses decoded error information to read the PCI Express AER capability of the agent device to obtain more details about an error. Third, the PCI Express AER driver clears the corresponding bit in the correctable error status register of both PCI Express root port and the agent device. Figure 4 illustrates the procedure to process correctable errors. Last but not least, the details about an error will be formatted and output to the system console as shown below:

```
+—— PCI-Express Device Error ——+
Error Severity : Corrected
PCIE Bus Error type : Physical Layer
Receiver Error : Multiple
Receiver ID : 0020
VendorID=8086h, DeviceID=3597h, Bus=00h, Device=04h,
Function=00h
```

The Requester ID is the ID of the device which reports the error. Based on such information, an administrator could find the bad device easily.

2.2.2.3 Handle PCI Express Non-Fatal Errors

If an agent device reports non-fatal errors, the PCI Express AER driver uses the same mechanism as described in Section 2.2.2 to obtain more details about an error from an agent device and output error information to the system console. Figure 5 illustrates the procedure to process non-fatal errors.
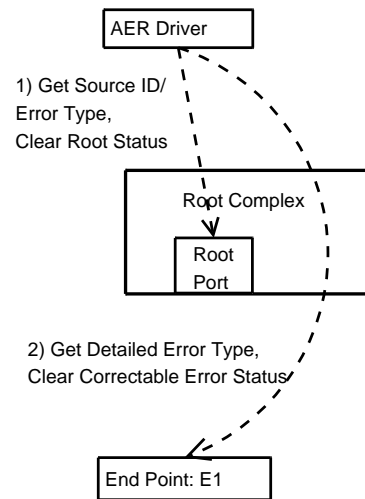

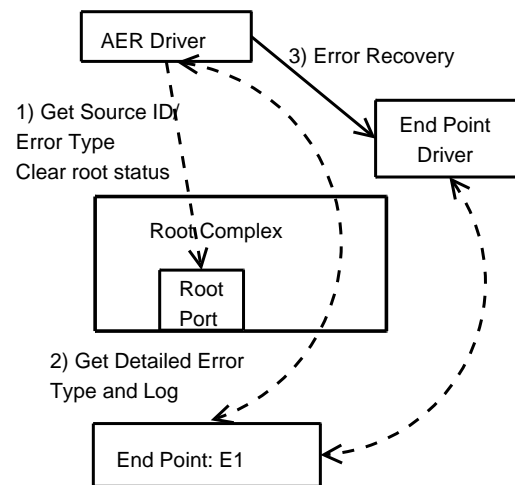Figure 4: Procedure to Process Correctable Errors

Figure 5: Procedures to Process Non-Fatal Errors

The first two steps are like the ones to process correctable errors. During Step 2, the AER driver need to retrieve the packet header log from the agent if the error is TLP-related.

Below is an example of non-fatal error output to the system console.
```
+—— PCI-Express Device Error ——+
Error Severity : Uncorrected (Non-Fatal)
PCIE Bus Error type : Transaction Layer
Completion Timeout : Multiple
Requester ID : 0018
VendorID=8086h, DeviceID=3596h, Bus=00h, Device=03h,
Function=00h
```

Unlike correctable errors, non-fatal errors might cause

some transaction failures. To help an agent device driver to retry any failed transactions, the PCI Express AER driver must perform a non-fatal error recovery procedure, which depends on where a non-fatal error occurs in a system hierarchy. As illustrated in Figure 6, for example, there are two PCI Express switches. If endpoint device E2 reports a non-fatal error, the PCI Express AER driver will try to perform an error recovery procedure only on this device. Other devices won't take part in this error recovery procedure. If downstream port P1 of switch 1 reports a non-fatal error, the PCI Express AER driver will do error recovery procedure on all devices under port P1, including all ports of switch 2, end point E1, and E2.
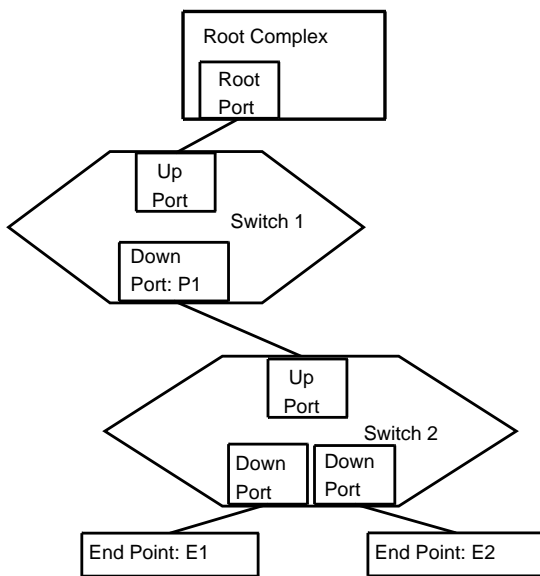

Figure 6: Non-Fatal Error Recovery Example

To take part in the error recovery procedure, specific device drivers need to implement error callbacks as described in Section 4.1.

When an uncorrectable non-fatal error happens, the AER error recovery procedure first calls the `error_detected` routine of all relevant drivers to notify their devices run into errors by the deep-first sequence. In the callback `error_detected`, the driver shouldn't operate the devices, i.e., do not perform any I/O on the devices. Mostly, `error_detected` might cancel all pending requests or put the requests into a queue.

If the return values from all relevant `error_detected` routines are `PCI_ERS_RESULT_CAN_RECOVER`, the AER recovery procedure calls all resume callbacks of the relevant drivers. In the resume functions, drivers could resume operations to the devices.

If an `error_detected` callback returns `PCI_ERS_RESULT_NEED_RESET`, the recovery procedure will call all `slot_reset` callbacks of relevant drivers. If all `slot_reset` functions return `PCI_ERS_RESULT_CAN_RECOVER`, the resume callback will be called to finish the recovery. Currently, some device drivers provide `err_handler` callbacks. For example, Intel's E100 and E1000 network card driver and IBM's POWER RAID driver.

The PCI Express AER driver outputs some information about non-fatal error recovery steps and results. Below is an example.

```
+—— PCI-Express Device Error ——+
Error Severity : Uncorrected (Non-Fatal)
PCIE Bus Error type : Transaction Layer
Unsupported Request : First
Requester ID : 0500
VendorID=14e4h, DeviceID=1659h, Bus=05h, Device=00h,
Function=00h
TLB Header:
04000001 0020060f 05010008 00000000
Broadcast error_detected message
Broadcast slot_reset message
Broadcast resume message
tg3: eth3: Link is down.
AER driver successfully recovered
```

2.2.2.4 Handle PCI Express Fatal Errors

When processing fatal errors, the PCI Express AER driver also collects detailed error information from the reporter in the same manner as described in Sections 2.2.2.2 and 2.2.2.3. Below is an example of non-fatal error output to the system console:

```
+—— PCI-Express Device Error ——+
Error Severity : Uncorrected (Fatal)
PCIE Bus Error type : Transaction Layer
Unsupported Request : First
Requester ID : 0200
VendorID=8086h, DeviceID=0329h, Bus=02h, Device=00h,
Function=00h
TLB Header:
04000001 00180003 02040000 00020400
```

When performing the error recovery procedure, the major difference between non-fatal and fatal is whether

the PCI Express link will be reset. If the return values from all relevant `error_detected` routines are `PCI_ERS_RESULT_CAN_RECOVER`, the AER recovery procedure resets the PCI Express link based on whether the agent is a bridge. Figure 7 illustrates an example.
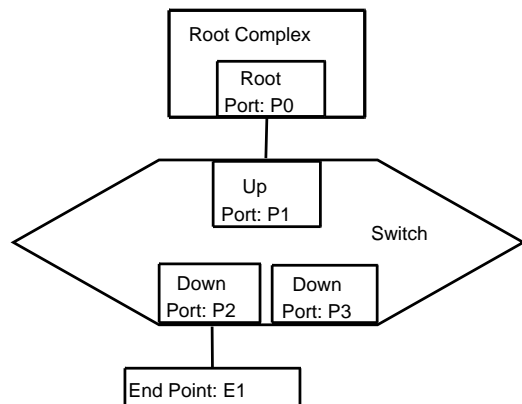

Figure 7: Reset PCI Express Link Example

In Figure 7, if root port P0 (a kind of bridge) reports a fatal error to itself, the PCI Express AER driver chooses to reset the upstream link between root port P0 and upstream port P1. If end-point device E1 reports a fatal error, the PCI Express AER driver chooses to reset the upstream link of E1, i.e., the link between P2 and E1.

The reset is executed by the port. If the agent is a port, the port will execute reset. If the agent is an end-point device, for example, E1 in Figure 7, the port of the upstream link of E1, i.e., port P2 will execute reset.

The reset method depends on the port type. As for root port and downstream port, the PCI Express Specification defines an approach to reset their downstream link. In Figure 7, if port P0, P2, P3, and end point E1 report fatal errors, the method defined in PCI Express Specification will be used. The PCI Express AER driver implements the standard method as default reset function.

There is no standard way to reset the downstream link under the upstream port because different switches might implement different reset approaches. To facilitate the link reset approach, the PCI Express AER driver adds `reset_link`, a new function pointer, in the data structure `pcie_port_service_driver`.

```
struct pcie_port_service_driver {
    ...
    /* Link Reset Capability - AER service
    driver specific */
    pci_ers_result_t (*reset_link) (struct
        pci_dev *dev);
    ...
};
```

If a port uses a vendor-specific approach to reset link, its AER port service driver has to provide a `reset_link` function. If a root port driver or downstream port service driver doesn't provide a `reset_link` function, the default `reset_link` function will be called. If an upstream port service driver doesn't implement a `reset_link` function, the error recovery will fail.

Below is the system console output example printed by the PCI Express AER driver when doing fatal error recovery.

```
+—— PCI-Express Device Error ——+
Error Severity : Uncorrected (Fatal)
PCIE Bus Error type : (Unaccessible)
Unaccessible Received : First
Unregistered Agent ID : 0500
Broadcast error_detected message
Complete link reset at Root[0000:00:04.0]
Broadcast slot_reset message
Broadcast resume message
tg3: eth3: Link is down.
AER driver successfully recovered
```

2.3 Including PCI Express Advanced Error Reporting Driver Into the Kernel

The PCI Express AER Root driver is a Root Port service driver attached to the PCI Express Port Bus driver. Its service must be registered with the PCI Express Port Bus driver and users are required to include the PCI Express Port Bus driver in the kernel [5]. Once the kernel configuration option `CONFIG_PCIEPORTBUS` is included, the PCI Express AER Root driver is automatically included as a kernel driver by default (`CONFIG_PCIEAER = Y`).

3 Impact to PCI Express BIOS Vendor

Currently, most BIOSes don't follow PCI FW 3.0 to support the ACPI _OSC handler. As a result, the PCI Express AER driver will fail when calling the ACPI

control method _OSC. The PCI Express AER driver provides a current workaround for the lack of ACPI BIOS _OSC support by implementing a boot parameter, `forceload=y/n`. When the kernel boots with parameter `aerdriver.forceload=y`, the PCI Express AER driver still binds to all root ports, which implements the AER capability.

4 Impact to PCI Express Device Driver

4.1 Device driver requirements

To conform to AER driver infrastructure, PCI Express device drivers need support AER capability.

First, when a driver initiates a device, it needs to enable the device's error reporting capability. By default, device error reporting is turned off, so the device won't send error messages to root port when it captures an error.

Secondly, to take part in the error recovery procedure, a device driver needs to implement error callbacks as described in the `pci_error_handlers` data structure as shown below.

```
struct pci_error_handlers {
  /* PCI bus error detected on this device */
  pci_ers_result_t (*error_detected)(struct
    pci_dev *dev, enum pci_channel_state error);
  /* MMIO has been re-enabled, but not DMA */
  pci_ers_result_t (*mmio_enabled)(struct
    pci_dev *dev);
  /* PCI slot has been reset */
  pci_ers_result_t (*slot_reset)(struct
    pci_dev *dev);
  /* Device driver may resume
    normal operations */
  void (*resume)(struct pci_dev *dev);
};
```

In data structure `pci_driver`, add `err_handler` as a new pointer to point to the `pci_error_handlers`. In kernel 2.6.14, the definition of `pci_error_handlers` had already been added to support PCI device error recovery [4]. To be compatible with PCI device error recovery, PCI Express device error recovery also uses the same definition and follows a similar rule. One of our starting points is that we try to keep the recovery callback interfaces as simple as we can. If the interfaces are complicated, there will be no driver developers who will be happy to add error recovery callbacks into device drivers.

4.2 Device driver helper functions

To communicate with device AER capabilities, drivers need to access AER registers in configuration space. It's easy to write incorrect code because they must access/change the bits of registers. To facilitate driver programming and reduce coding errors, the AER driver provides a couple of helper functions which could be used by device drivers.

4.2.1 int pci_find_aer_capability (struct pci_dev *dev);

`pci_find_aer_capability` locates the PCI Express AER capability in the device configuration space. Since offset 0x100 in configuration space, PCI Express devices could provide a couple of optional capabilities and they link each other in a chain. AER is one of them. To locate AER registers, software needs to go through the chain. This function returns the AER offset in the device configuration space.

4.2.2 int pci_enable_pcie_error_reporting (struct pci_dev *dev);

`pci_enable_pcie_error_reporting` enables the device to send error messages to the root port when an error is detected. If the device doesn't support PCI-Express capability, the function returns 0. When a device driver initiates a device (mostly, in its probe function), it should call `pci_enable_pcie_error_reporting`.

4.2.3 int pci_disable_pcie_error_reporting (struct pci_dev *dev);

`pci_disable_pcie_error_reporting` disables the device from sending error messages to the root port. Sometimes, device drivers want to process errors by themselves instead of using the AER driver. It's not encouraged, but we provide this capability.

4.2.4 int pci_cleanup_aer_uncorrect_error_status (struct pci_dev *dev);

`pci_cleanup_aer_uncorrect_error_status` cleans up the uncorrectable error status

register. The AER driver only clears correctable error status register when processing errors. As for uncorrectable errors, specific device drivers should do so because they might do more specific processing. Usually, a driver should call this function in its `slot_reset` or `resume` callbacks.

4.3 Testing PCI Express AER On Device Driver

It's hard to test device driver AER capabilities. By lots of experiments, we have found that UR (Unsupported Request) can be used to test device drivers. We triggered UR error messages by probing a non-existent device function. For example, if a PCI Express device only has one function, when kernel reads the ClassID from the configuration space of the second function of the device, the device might send an Unsupported Request error message to the root port and set the bit in uncorrectable error status register. By setting different values in the corresponding bit in uncorrectable error mask register, we could test both non-fatal and fatal errors.

5 Conclusion

The PCI Express AER driver creates a generic infrastructure to support PCI Express AER. This infrastructure provides the Linux kernel with an ability to capture PCI Express device errors and perform error recovery where in a hierarchy an agent device reports. Last but not least the system administrators could get formatted, useful error information to debug device errors.

Linux kernel 2.6.19 has accepted the PCI Express AER patches. Future work includes enabling PCI Express AER for every PCI Express device by default, blocking I/O when an error happens, and so on.

6 Acknowledgement

Special thanks to Steven Carbonari for his contributions to the architecture design of PCI Express AER driver, Rajesh Shah for his contributions to code review, and the Linux community for providing great input.

Legal Statement

This paper is copyright © 2007 by Intel Corporation. Permission to redistribute in accordance with Linux Symposium submission guidelines is granted; all other rights are reserved.

References

[1] PCI Express Base Specification Revision 1.1. March 28, 2005. `http://www.pcisig.com`

[2] PCI Firmware Specification Revision 3.0, `http://www.pcisig.com`

[3] Tom Long Nguyen, Dely L. Sy, & Steven Carbonari. "PCI Express Port Bus Driver Support for Linux." *Proceedings of the Linux Symposium*, Vol. 2, Ottawa, Ontario, 2005. `http://www.linuxsymposium.org/ 2005/linuxsymposium_procv2.pdf`

[4] pci-error-recovery.txt. Available from: 2.6.20/Documentation.

[5] PCIEBUS-HOWTO.txt. Available from: 2.6.20/Documentation.

[6] pcieaer-howto.txt. Available from: 2.6.20/Documentation.

Enabling Linux* Network Support of Hardware Multiqueue Devices

Zhu Yi

Intel Corp.

yi.zhu@intel.com

Peter P. Waskiewicz, Jr.

Intel Corp.

peter.p.waskiewicz.jr@intel.com

Abstract

In the Linux kernel network subsystem, the Tx/Rx SoftIRQ and Qdisc are the connectors between the network stack and the net devices. A design limitation is that they assume there is only a single entry point for each Tx and Rx in the underlying hardware. Although they work well today, they won't in the future. Modern network devices (for example, E1000 [8] and IPW2200 [6]) equip two or more hardware Tx queues to enable transmission parallelization or MAC-level QoS. These hardware features cannot be supported easily with the current network subsystem.

This paper describes the design and implementation for the network multiqueue patches submitted to *netdev* [2] and *LKML* [1] mailing lists early this year, which involved the changes for the network scheduler, Qdisc, and generic network core APIs. It will also discuss breaking the `netdev->queue_lock` with fine-grained per-queue locks in the future. At the end of the paper, it takes the IPW2200 and E1000 drivers as an example to illustrate how the new network multiqueue features will be used by the network drivers.

1 A Brief Introduction for the Linux Network Subsystem

When a packet is passed from the user space into the kernel space through a socket, an *skb* (socket kernel buffer) is created to represent that packet in the kernel. The *skb* is passed through various layers of the network stack before it is handed to the device driver for transmission. Inside the Linux kernel, each network device is represented by a `struct net_device` structure. All the `struct net_device` instances are linked into a doubly linked list with a single pointer list head (called `hlist`); the list head is named `dev_base`. The `struct net_device` contains all information and function pointers for the device. Among them there is a `qdisc` item. Qdisc stands for queuing discipline. It defines how a packet is selected on the transmission path. A Qdisc normally contains one or more queues (`struct sk_buff_head`) and a set of operations (`struct Qdisc_ops`). The standard `.enqueue` and `.dequeue` operations are used to put and get packets from the queues by the core network layer. When a packet first arrives to the network stack, an attempt to enqueue occurs. The `.enqueue` routine can be a simple FIFO, or a complex traffic classification algorithm. It all depends on the type of Qdisc the system administrator has chosen and configured for the system. Once the enqueue is completed, the packet scheduler is invoked to pull an *skb* off a queue somewhere for transmission. This is the `.dequeue` operation. The *skb* is returned to the stack, and is then sent to the device driver for transmission on the wire. The network packets transmission can be started by either `dev_queue_xmit()` or the TX SoftIRQ (`net_tx_action()`) depending on whether the packet can be transmitted immediately or not. But both routines finally call `qdisc_run()` to dequeue an *skb* from the root Qdisc of the *netdev* and send it out by calling the `netdev->hard_start_xmit()` method.

When the hardware Tx queue of a network device is full (this can be caused by various reasons—e.g., carrier congestion, hardware errors, etc.), the driver should call `netif_stop_queue()` to indicate to the network scheduler that the device is currently unusable. So the `qdisk_restart()` function of the network scheduler won't try to transmit the packet (with the device's `.hard_start_xmit()` method) until the driver explicitly calls `netif_start_queue()` or `netif_wake_queue()` to indicate the network scheduler its hardware queue is available again. The `netdev->hard_start_xmit()` method is responsible for checking the hardware queue states. If the device hardware queue is full, it should call `netif_stop_queue()` and returns `NETDEV_TX_BUSY`. On the other side, when the network scheduler receives the

NETDEV_TX_BUSY as the return value for `netdev->hard_start_xmit()`, it will reschedule. Note that if a driver returns NETDEV_TX_BUSY without calling `netif_stop_queue()` in the `hard_start_xmit()` method when the hardware transmit queue is full, it will chew tons of CPU.

For a normal network device driver, the general rules to deal with the hardware Tx queue are:

- Driver detects queue is full and calls `netif_stop_queue()`;

- Network scheduler will not try to send more packets through the card any more;

- Even in some rare conditions (`dev->hard_start_xmit()` is still called), calls into the driver from top network layer always get back a NETDEV_TX_BUSY;

- EOT interrupt happens and driver cleans up the TX hardware path to make more space so that the core network layer can send more packets (driver calls `netif_start_queue()`)

- Subsequent packets get queued to the hardware.

In this way, the network drivers use `netif_stop_queue()` and `netif_start_queue()` to provide feedback to the network scheduler so that neither packet starvation nor a CPU busy loop occurs.

2 What's the Problem if the Device has Multiple Hardware Queues?

The problem happens when the underlying hardware has multiple hardware queues. Multiple hardware queues provide QoS support from the hardware level. Wireless network adapters such as the Intel® PRO/Wireless 3945ABG, Intel® PRO/Wireless 2915ABG, and Intel® PRO/Wireless 2200BG Network Connections have already provided this feature in hardware. Other high speed ethernet adapters (i.e., e1000) also provide multiple hardware queues for better packet throughput by parallelizing the Tx and Rx paths. But the current Qdisc interface isn't multiple-hardware-queue aware. That is, the `.dequeue` method is not able to dequeue the correct *skb* according to the device hardware queue states. Take a device containing two hardware Tx queues for

an example: if the high-priority queue is full while the low one is not, the Qdisc will still keep dequeueing the high priority *skb*. But it will always fail to transmit in the high-priority queue because the corresponding hardware queue is full. To make the situation even worse, `netif_stop_queue()` and friends are also ignorant of multiple hardware queues. There is no way to schedule Tx SoftIRQ based on hardware queues (vs. based on the global *netdev*). For example, if the low-priority hardware queue is full, should the driver call `netif_stop_queue()` or not? If the driver does, the high priority *skb*s will also be blocked (because the netdev Tx queue is stopped). If the driver doesn't, the high CPU usage problem we mentioned in Section 1 will happen when low priority *skb*s remain in the Qdisc.

3 How to Solve the Problem?

Since the root cause for this problem is that the network scheduler and Qdisc do not expect the network devices to have multiple hardware queues, the obvious fix is to add the support for multi-queue features to them.

4 The Design Considerations

The main goal of implementing support for multiple queues is to prevent one flow of traffic from interfering with another traffic flow. Therefore, if a hardware queue is full, the driver will need to stop the queue. With multiqueue, the driver should be able to stop an individual queue, and the network stack in the OS should know how to check individual queue states. If a queue is stopped, the network stack should be able to pull packets from another queue and send them to the driver for transmission.

One main consideration with this approach is how the stack will handle traffic from multiqueue devices and non-multiqueue devices in the same system. This must be transparent to devices, while maintaining little to no additional overhead.

5 The Implementation Details

The following details of implementation are under discussion and consideration in the Linux community at the writing of this paper. The concepts should remain the same, even if certain implementation details are changed to meet requests and suggestions from the community.

The first part of implementation is how to represent queues on the device. Today, there is a single queue state and lock in the `struct net_device`. Since we need to manage the queue's state, we will need a state for each queue. The queues need to be accessible from the `struct net_device` so they can be visible to both the stack and the driver. This is added to `include/linux/netdevice.h`:

Inside `struct net_device`:

```
struct net_device
{

 ...

 struct net_device_subqueue
   *egress_subqueue;

 unsigned long
   egress_subqueue_count;

 ...
}
```

Here, the *netdev* has the knowledge of the queues, and how many queues are supported by the device. For all non-multiqueue devices, there will be one queue allocated, with an `egress_subqueue_count` of 1. This is to help the stack run both non-multiqueue devices and multiqueue devices simultaneously. The details of this will be discussed later.

When a network driver is loaded, it needs to allocate a `struct net_device` and a structure representing itself. In the case of ethernet devices, the function `alloc_etherdev()` is called. A change to this API provides `alloc_etherdev_mq()`, which allows a driver to tell the kernel how many queues it wants to allocate for the device. `alloc_etherdev()` is now a macro that calls `alloc_etherdev_mq()` with a queue count of 1. This allows non-multiqueue drivers to transparently operate in the multiqueue stack without any changes to the driver.

Ultimately, `alloc_etherdev_mq()` calls the new `alloc_netdev_mq()`, which actually handles the `kzalloc()`. In here, `egress_subqueue` is assigned to the newly allocated memory, and `egress_subqueue_count` is assigned to the number of allocated queues.

On the deallocation side of things, `free_netdev()` now destroys the memory that was allocated for each queue.

The final part of the multiqueue solution comes with the driver being able to manipulate each queue's state. If one hardware queue runs out of descriptors for whatever reason, the driver will shut down that queue. This operation should not prevent other queues from transmitting traffic. To achieve this, the network stack should check the state for the global queue state, as well as the individual queue states, before deciding to transmit traffic on that queue.

Queue mapping in this implementation happens in the Qdisc, namely PRIO. The mapping is a combination of which PRIO band an *skb* is assigned to (based on TC filters and/or IP TOS to priority mapping), and which hardware queue is assigned to which band. Once the *skb* has been classified in `prio_classify()`, a lookup to the `band2queue` mapping is done, and assigned to a new field in the *skb*, namely `skb->queue_mapping`. The calls to `netif_subqueue_stopped()` will pass this queue mapping to know if the hardware queue to verify is running or not. In order to help performance and avoid unnecessary requeues, this check is done in the `prio_dequeue()` routine, prior to pulling an *skb* from the band. The check will be done again before the call to `hard_start_xmit()`, just as it is done today on the global queue. The *skb* is then passed to the device driver, which will need to look at the value of `skb->queue_mapping` to determine which Tx ring to place the *skb* on in the hardware. At this point, multiqueue flows have been established.

The network driver will need to manage the queues using the new `netif_{start|stop|wake}_subqueue()` APIs. This way full independance between queues can be established.

6 Using the Multiqueue Features

6.1 Intel® PRO/Wireless 2200BG Network Connection Driver

This adapter supports the IEEE 802.11e standard [3] for Quality of Service (QoS) on the Medium Access Control (MAC) layer. Figure 1 shows the hardware implementation. Before a MSDU (MAC Service Data Unit) is passed to the hardware, the network stack maps the
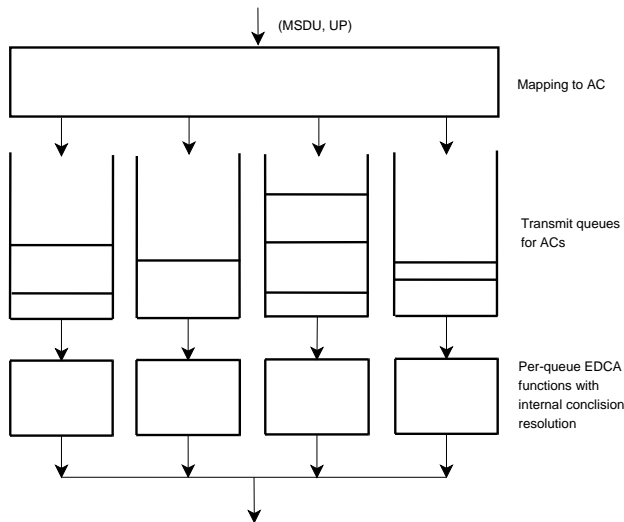
Figure 1: MAC level QoS implementation

Figure 2: Mapping CPU cores to multiple Tx queues on SMP system

frame type or User Priority (UP) to Access Category (AC). The NIC driver then pushes the frame to the corresponding one of the four transmit queues according to the AC. There are four independent EDCA (enhanced distributed channel access) functions in the hardware, one for each queue. The EDCA function resolves internal collisions and determines when a frame in the transmit queue is permitted to be transmitted via the wireless medium.

With the multiqueue devices supported by the network subsystem, such hardware-level packet scheduling is easy to enable. First, a specific PRIO Qdisc queue mapping for all the IEEE 802.11 wireless devices is created. It maps the frame type or UP to AC according to the mapping algorithm defined in [3]). With this specific queue mapping, the IEEE 802.11 frames with higher priority are always guaranteed to be scheduled before the lower priority ones by the network scheduler when both transmit queues are active at the time. In other cases, for example, the higher priority transmit queue is inactive while the lower priority transmit queue is active, and the lower priority frame is scheduled (dequeued). This is the intention because the hardware is not going to transmit any higher priority frames at this time. When the `dev->hard_start_xmit()` is invoked by the network scheduler, the `skb->queue_mapping` is already set to the corresponding transmit queue index for the *skb* (by the Qdisc `.dequeue` method). The driver then just needs to read this value and move the *skb* to the target transmit queue accordingly. In the normal
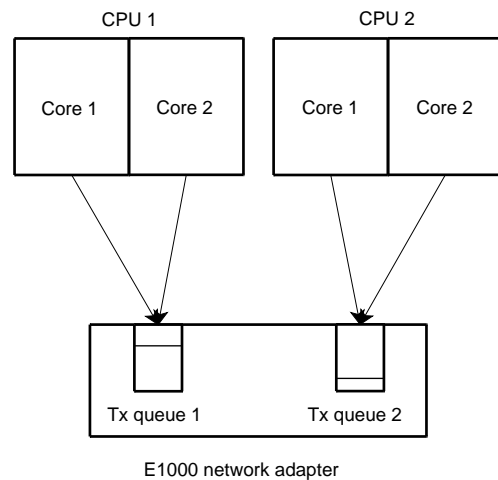
cases, the queue will still remain active since the network scheduler has just checked its state in the Qdisc `.dequeue` method. But in some rare cases, a race condition would still happen during this period to make the queue state inconsistent. This is the case where the Qdisc `.requeue` method is invoked by the network scheduler.

6.2 Intel® PRO/ 1000 Adapter Driver

This adapter with MAC types of 82571 and higher supports multiple Tx and Rx queues in hardware. A big advantage with these multiple hardware queues is to achieve packets transmission and reception parallelization. This is especially useful on SMP and multi-core systems with a lot of processes running network loads on different CPUs or cores. Figure 2 shows an e1000 network adapter with two hardware Tx queues on a multi-core SMP system.

With the multiqueue devices supported by the network subsystem, the e1000 driver can export all its Tx queues and bind them to different CPU cores. Figure 3 illustrates how this is done in the e1000 multiqueue patch [4]. The per-CPU variable `adapter->cpu_tx_ring` points to the mapped Tx queue for the current CPU. After the e1000 queue mapping has been setup, the access for the Tx queues should always be referenced by the `adapter->cpu_tx_ring` instead of manipulating the `adapter->tx_ring` array directly. With spreading CPUs on multiple hardware Tx

```
netdev = alloc_etherdev_mq(sizeof(struct e1000_adapter), 2);
netdev->features |= NETIF_IF_MULTI_QUEUE;

...

adapter->cpu_tx_ring = alloc_percpu(struct e1000_tx_ring *);

lock_cpu_hotplug();
i = 0;
for_each_online_cpu(cpu) {
  *per_cpu_ptr(adapter->cpu_tx_ring, cpu) =
              &adapter->tx_ring[i % adapter->num_tx_queues];
  i++;
}
unlock_cpu_hotplug();
```

Figure 3: E1000 driver binds CPUs to multiple hardware Tx queues

queues, transmission parallelization is achieved. Since the CPUs mapped to different Tx queues don't contend for the same lock for packet transmission, `LLTX` lock contention is also reduced. With breaking the `netdev->queue_lock` into per-queue locks in the future, this usage model will perform and scale even better. The same parallelization is also true for packet reception.

Comparing with the multiqueue usage model for the hardware QoS scheduler by the wireless devices, the e1000 usage model doesn't require a special frame type to queue mapping algorithm in the Qdisc. So any type of multiqueue-aware Qdiscs can be configured on top of e1000 hardware by the system administrator with command *tc*, which is part of the `iproute2` [7] package. For example, to add the PRIO Qdisc to your network device, assuming the device is called eth0, run the following command:

```
# tc qdisc add dev eth0 root \
        handle 1: prio
```

As of the writing of this paper, there are already patches in the linux-netdev mailing list [5] to enable the multiqueue features for the `pfifo_fast` and `PRIO` Qdiscs.

7 Future Work

In order to extend flexibility of multiqueue network device support, work on the Qdisc APIs can be done. This is needed to remove serialization of access to the Qdisc itself. Today, the Qdisc may only have one transmitter inside it, governed by the `__LINK_STATE_QDISC_RUNNING` bit set on the global queue state. This bit will need to be set per queue, not per device. Implications with Qdisc statistics will need to be resolved, such as the number of packets sent by the Qdisc, etc.

Per-queue locking may also need to be implemented in the future. This is dependent on performance of higher speed network adapters becoming throttled by the single device queue lock. If this is determined to be a source of contention, the stack will need to change to know how to independently lock and unlock each queue during a transmit.

8 Conclusion

The multiqueue devices support for network scheduler and Qdisc enables modern network interface controllers to provide advanced features like hardware-level packet scheduling and Tx/Rx parallelization. As processor packages ship with more and more cores nowadays, there is also a trend that the network adapter hardware may equip more and more Tx and Rx queues in the future. The multiqueue patch provides the fundamental features for the network core to enable these devices.

Legal

References

[1] Linux kernel mailing list.
`linux-kernel@vger.kernel.org`.

[2] Linux network development mailing list.
`linux-netdev@vger.kernel.org`.

[3] IEEE Computer Society. LAN/MAN Committee.
*IEEE Standard for Information technology –
Telecommunications and information exchange
between systems – Local and metropolitan area
networks – Specific requirements Part 11: Wireless
LAN Medium Access Control (MAC) and Physical
Layer (PHY) specifications Amendment 8: Medium
Access Control (MAC) Quality of Service
Enhancements*. 3 Park Avenue New York, NY
10016-5997, USA, November 2005.

[4] Peter P Waskiewicz Jr. E1000 example
implementation of multiqueue network device api.
`http:`
`//marc.info/?l=linux-netdev&m=`
`117642254323203&w=2`.

[5] Peter P Waskiewicz Jr. Multiqueue network device
support implementation. `http:`
`//marc.info/?l=linux-netdev&m=`
`117642230823028&w=2`.

[6] James Ketrenos and the ipw2200 developers.
Intel® PRO/Wireless 2200BG Driver for Linux.
`http://ipw2200.sf.net`.

[7] Alexey Kuznetsov and Stephen Hemminger.
*Iproute2: a collection of utilities for controlling
TCP / IP networking and Traffic Control in Linux*.
`http://linux-net.osdl.org/index.`
`php/Iproute2`.

[8] John Ronciak, Auke Kok, and the e1000
developers. *The Intel® PRO/10/100/1000/10GbE
Drivers*. `http://e1000.sf.net`.

Concurrent Pagecache

Peter Zijlstra
Red Hat
`pzijlstr@redhat.com`

Abstract

In this paper we present a concurrent pagecache for Linux, which is a continuation of the existing lockless pagecache work [5].

Currently the pagecache is protected by a reader/writer lock. The lockless pagecache work focuses on removing the reader lock, however, this paper presents a method to break the write side of the lock. Firstly, since the pagecache is centered around the radix tree, it is necessary to alter the radix tree in order to support concurrent modifications of the data structure. Secondly, we adapt the pagecache, by removing all non-radix tree consumers of the lock's protection, and extend the pageflag, introduced by the lockless pagecache, into a second per page lock. Then we can fully utilize the radix tree's new functionality to obtain a concurrent pagecache. Finally, we analyze the improvements in performance and scalability.

1   Introduction

In order to provide some background for this work, we will give a quick sketch of the Linux memory management. For a more in depth treatment see Mel Gorman's excellent book on the subject [2].

1.1   Pagecache

As we know, in Linux, the pagecache caches on-disk data in memory per file. So the Linux pagecache stores and retrieves disk pages based on the `(inode, offset)`-tuple. This per inode page-index is implemented with a radix tree [3].

The typical consumer of this functionality is the VFS, Virtual File-System. Some system-interfaces, such as `read(2)` and `write(2)`, as well as `mmap(2)`, operate on the pagecache. These instantiate pages for the pagecache when needed, after which, the newly allocated pages are inserted into the pagecache, and possibly filled with data read from disk.

1.1.1   Page Frames

Each physical page of memory is managed by a `struct page`. This structure contains the minimum required information to identify a piece of data, such as a pointer to the inode, and the offset therein. It also contains some management state, a reference counter for instance, to control the page's life-time, as well as various bits to indicate status, such as `PG_uptodate` and `PG_dirty`. It is these page structures which are indexed in the radix tree.

1.1.2   Page Reclaim

In a situation when a free page of memory is requested, and the free pages are exhausted, a used page needs to be reclaimed. However the pagecache memory can only be reclaimed when it is clean, that is, if the in-memory content corresponds to the on-disk version.

When the page is not clean, it is called dirty, and requires data to be written back to disk. The problem of finding all the dirty pages in a file is solved by *tags*, which is a unique addition to the Linux radix tree (see Section 2.1).

1.2   Motivation

The lockless pagecache work by Nick Piggin [5] shows that the SMP scalability of the pagecache is greatly improved by reducing the dependency on high-level locks. However, his work focuses on lookups.

While lookups are the most frequent operation performed, other operations on the pagecache can still form a significant amount of the total operations performed.

Therefore, it is necessary to investigate the other operations as well. They are:

- *insert* an item into the tree;

- *update* an existing slot to point at a new item;

- *remove* an item from the tree;

- *set a tag* on an existing item;

- *clear a tag* on an existing item.

2 Radix Tree

The radix tree deserves a thorough discussion, as it is the primary data structure of the pagecache.

The radix tree is a common dictionary-style data structure, also know as Patricia Trie or crit bit tree. The Linux kernel uses a version which operates on fixed-length input, namely an `unsigned long`. Each level represents a fixed number of bits of this input space (usually 6 bits per tree level - which gives a maximum tree height of $\lceil 64/6 \rceil = 11$ on 64-bit machines). See Figure 1 for a representation of a radix tree with nodes of order 2, mapping an 8-bit value.
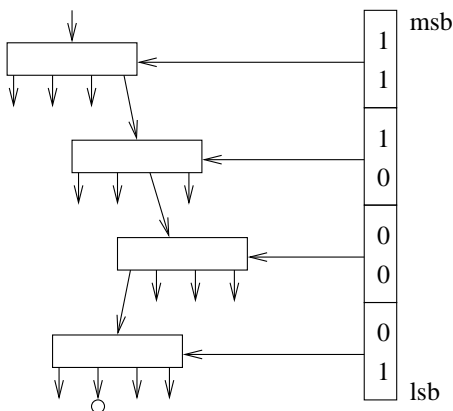

Figure 1: 8-bit radix tree

2.1 Tags

A unique feature of the Linux radix tree is its *tags*-extension. Each tag is basically a bitmap index on top of the radix tree. Tags can be used in conjunction with gang lookups to find all pages which have a given tag set within a given range.

The tags are maintained in per-node bitmaps, so that on each given level we can determine whether or not the next level has at least a single tag set. Figure 2 shows the same tree as before, but now with two tags (denoted by an open and closed bullet respectively).
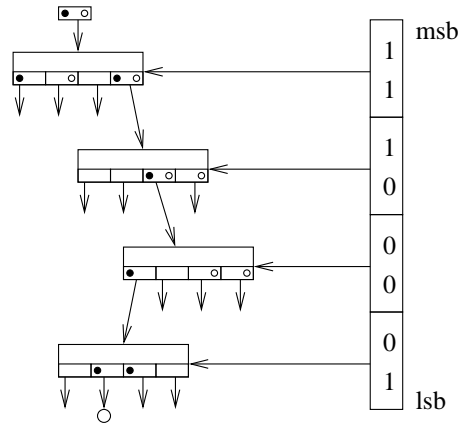

Figure 2: 8-bit radix tree with 2 bitmap indices

2.2 Concurrency Control

Traditionally, the radix tree is locked as a whole, and does not support concurrent operations. Lock contention of this high-level lock is where the scalability problems come from.

Linux currently uses a reader/writer lock, called `tree_lock`, to protect the tree. However, on large SMP machines it still does not scale properly due to cache line bouncing; the lock also fully serialises all modifications.

2.3 RCU Radix Tree

The RCU radix tree enables fully concurrent lookups (cf. [4]), which is done by exploiting the Read-Copy-Update technique [1].

RCU basically requires us to atomically flip pointers from one version of a (partial) data structure to a new version, while it will keep the old one around until the system passes through a quiescent state. At this point it is guaranteed that there are no more users of the old structure, and therefore it can be freed safely.

However, the radix tree structure is sufficiently static, so that often modifications are nothing but a single atomic change to the node. In this case we can just keep using the same node.

The radix-tree modifications still need to be serialised with respect to each other. The lookups, however, are no longer serialised with respect to modifications.

This allows us to replace the reader/writer lock with a regular one, and thus reduce the cache-line bouncing by not requiring an exclusive access to the cache line for the lookups.

2.4 Concurrent Radix Tree

With lookups fully concurrent, modifying operations become a limiting factor. The main idea is to 'break' the tree lock into many small locks.[1]

The obvious next candidate for locking would be the nodes.

When we study the operations in detail, we see that they fall into two categories:

- uni-directional;

- bi-directional.

The most simple of the two is the uni-directional operations; they perform only a single traversal of the tree: from the root to a leaf node. These include: *insert*, *update* and *set tag*.

The bi-directional operations are more complex, since they tend to go back up the tree after reaching the leaf node. These are the remaining operations: *remove* and *clear tag*.

2.4.1 Ladder Locking aka Lock-Coupling

This technique, which is frequently used in the database world, allows us to walk a node-locked tree in a single direction (bi-directional traffic would generate deadlocks).

If all modifiers alter the tree top-to-bottom, and hold a lock on the node which is being modified, then walking down is as simple as taking a lock on a child, while

---

[1]Ideally we reduce the locks so far that we end up with single atomic operations. However tags and deletion (the back tracking operations) seem to make this impossible; this is still being researched.

holding the node locked. We release the node lock as soon as the child is locked.

In this situation concurrency is possible, since another operation can start its descent as soon as the root node is unlocked. If their paths do not have another node in common, it might even finish before the operation which started earlier. The worst case, however, is pipelined operation, which is still a lot better than a fully serialised one.

2.4.2 Path Locking

Obviously, this model breaks down when we need to walk back up the tree again, for it will introduce cyclic lock dependencies. This implies that we cannot release the locks as we go down, which will seriously hinder concurrent modifications.

An inspection of the relevant operations shows that the upwards traversal has distinct termination conditions. If these conditions were reversed, so that we could positively identify the termination points during the downward traversal, then we could release all locks upwards of these points.

In the worst case, we will hold the root node lock, yet for non-degenerate trees the average case allows for good concurrency due to availability of termination points.

clear tag   Clearing a tag in the radix tree involves walking down the tree, locating the item and clearing its tag. Then we go back up the tree clearing tags, as long as the child node has no tagged items.

Thus, the termination condition states:

> we terminate the upward traversal if we encounter a node, which still has one or more entries with the tag set after clearing one.

Changing this condition, in order to identify the termination points during downwards traversal, gives:

> the upwards traversal will terminate at nodes which have more tagged items than the one we are potentially clearing.

So, whenever we encounter such a node, it is clear that we will never pass it on our way back up the tree, therefore we can drop all the locks above it.

remove Element removal is a little more involved: we need to remove all the tags for a designated item, as well as remove unused nodes. Its termination condition captures both of these aspects.

The following termination condition needs to be satisfied when walking back up the tree:

> the upward traversal is terminated when we encounter a node which is not empty, and none of the tags are unused.

The condition, identifying such point during the downward traversal, is given by:

> we terminate upwards traversal when a node that has more than two children is encountered, and for each tag it has more items than the ones we are potentially clearing.

Again, this condition identifies points that will never be crossed on the traversal back up the tree.

So, with these details worked out, we see that a node-locked tree can achieve adequate concurrency for most operations.

2.4.3 API

Concurrent modifications require multiple locking contexts and a way to track them.

The operation that has the richest semantics is `radix_tree_lookup_slot()`. It is used for speculative lookup, since that requires `rcu_dereference()` to be applied to the obtained slot. The update operation is performed using the same radix tree function, but by applying `rcu_assign_pointer()` to the resulting slot.

When used for update, the encompassing node should still be locked after return of `radix_tree_lookup_slot()`. Hence, clearly, we cannot hide the locking in the `radix_tree_*()` function calls.

Thus we need an API which elegantly captures both cases; lookup and modification.

We also prefer to retain as much of the old API as possible, in order to leave the other radix tree users undisturbed.

Finally, we would like a `CONFIG` option to disable the per-node locking for those environments where the increased memory footprint of the nodes is prohibitive.

We take the current pattern for lookups as an example:

```
struct page **slot, *page;

rcu_read_lock();
slot = radix_tree_lookup_slot(
        &mapping->page_tree, index);
page = rcu_dereference(*slot);
rcu_read_unlock();
```

Contrary to lookups, which only have global state (the RCU quiescent state), the modifications need to keep track of which locks are held. As explained before, this state must be external to the operations, thus we will need to instantiate a local context to track these locks.

```
struct page **slot;
DEFINE_RADIX_TREE_CONTEXT(ctx,
        &mapping->page_tree);

radix_tree_lock(&ctx);
slot = radix_tree_lookup_slot(
        ctx.tree, index);
rcu_assign_pointer(*slot, new_page);
radix_tree_unlock(&ctx);
```

As can be seen above, `radix_tree_lock()` operation locks the root node. By giving `ctx.tree` as the tree root instead of `&mapping->page_tree`, we pass the local context on, in order to track the held locks. This is done by using the lower bit of the pointer as a type field.

Then we adapt the modifying operations, in order to move the lock downwards:

```
void **radix_tree_lookup_slot(
        struct radix_tree *root,
        unsigned long index)
{
    ...
    RADIX_TREE_CONTEXT(context, root);
    ...
    do {
        ...
        /* move the lock down the tree */
        radix_ladder_lock(context, node);
        ...
    } while (height > 0);
    ...
}
```

The `RADIX_TREE_CONTEXT()` macro extracts the context and the actual root pointer.

Note that unmodified operations will be fully exclusive because they do not move the lock downwards.

This scheme captures all the requirements mentioned at the beginning of this section. The old API is retained by making the new parts fully optional; and by moving most of the locking specific functionality into a few macros and functions, it is possible to disable the fine grained locking at compile time using a `CONFIG` option.

3 Pagecache Synchronisation

The lockless pagecache paper [5] discusses the pagecache synchronisation in terms of guarantees provided by the read and write side of the pagecache lock. The read side provides the following guarantees (by excluding modifications):

- the *existence* guarantee;

- the *accuracy* guarantee.

The write side provides one additional guarantee (by being fully exclusive), namely:

- the *no-new-reference* guarantee.

The existence guarantee ensures that an object will exist during a given period. That is, the `struct page` found must remain valid. The read lock trivially guarantees this by excluding all modifications.

The accuracy guarantee adds to this by ensuring that not only will the object stay valid, but it will also stay in the pagecache. The existence only avoids deallocation, while the accuracy ensures that it keeps referring to the same `(inode, offset)`-tuple during the entire time. Once again, this guarantee is provided by the read lock by excluding all modifications.

The no-new-reference guarantee captures the fully exclusive state of the write lock. It excludes lookups from obtaining a reference. This is especially relevant for element removal.

3.1 Lockless Pagecache

The lockless pagecache focuses on providing the guarantees, introduced above, in view of the full concurrency of RCU lookups. That is, RCU lookups are not excluded by holding the tree lock.

3.1.1 Existence

The existence guarantee is trivially satisfied by observing that the page structures have a static relation with the actual pages to which they refer. Therefore they are never deallocated.

A free page still has an associated `struct page`, which is used by the page allocator to manage the free page. A free page's reference count is 0 by definition.

3.1.2 Accuracy

The accuracy guarantee is satisfied by using a *speculative-get* operation, which tries to get a reference on the page returned by the RCU lookup. If we did obtain a reference, we must verify that it is indeed the page requested. If either the speculative get, or the verification fails, e.g. the page was freed and possibly reused already, then we retry the whole sequence.

The important detail here is the *try-to-get-a-reference* operation, since we need to close a race with freeing pages, i.e. we need to avoid free pages from temporarily having a non-zero reference count. The reference count is modified by using atomic operations, and to close the race we need an `atomic_inc_not_zero()` operation, which will fail to increment when the counter is zero.

3.1.3 No New Reference

The no-new-reference guarantee is met by introducing a new page flag, `PG_nonewrefs`, which is used to synchronise lookups with modifying operations. That is, the *speculative get* should not return until this flag is clear. This allows atomic removal of elements which have a non-zero reference count (e.g. the pagecache itself might still have a reference).

3.1.4 Tree Lock

When we re-implement all lookup operations to take advantage of the *speculative get*, and re-implement the modifying operations to use `PG_nonewrefs`, then the read-side of the `tree_lock` will have no users left. Hence we can change it into a regular spinlock.

3.2 Concurrent Pagecache

The lockless pagecache leaves us with a single big lock serialising all modifications to the radix tree. However, with the adaptations to the radix tree, discussed in Section 2.4, the serialisation, required to meet the synchronisation guarantees of Section 3, is per page.

3.2.1 PG_nonewrefs vs. PG_locked

Since we need to set/clear `PG_nonewrefs` around most modifying operations, we might as well do it around all modifying operations, and change `PG_nonewrefs` into an exclusion primitive, which serialises each individual pagecache page modification.

We can't reuse `PG_locked` for this because, they have a different place in the locking hierarchy.

```
inode->i_mutex
  inode->i_alloc_sem
  mm->mmap_sem
    PG_locked
      mapping->i_mmap_lock
        anon_vma->lock
          mm->page_table_lock or pte_lock
            zone->lru_lock
            swap_lock
              mmlist_lock
              mapping->private_lock
              inode_lock
                sb_lock
                mapping->tree_lock
```

Figure 3: mm locking hierarchy

Figure 3 represents the locking hierarchy. As we can see, `PG_locked` is an upper level lock, whereas the `tree_lock` (now to be replaced by `PG_nonewrefs`) is at the bottom.

Also, `PG_locked` is a sleeping lock, whereas `tree_lock` must be a spinning lock.

3.2.2 Tree-Lock Users

Before we can fully remove the `tree_lock`, we need to make sure that there are no other users left.

A close scrutiny reveals that `nr_pages` is also serialised by the `tree_lock`. This counter needs to provide its own serialisation, for we take no lock covering the whole inode. Changing it to an `atomic_long_t` is the easiest way to achieve this.

Another unrelated user of the tree lock is architecture specific dcache flushing. However, since its use of the tree lock is a pure lock overload, it does not depend on any other uses of the lock. We preserve this usage and rename the lock to `priv_lock`.

3.2.3 No New Reference

The lockless pagecache sets and clears `PG_nonewrefs` around insertion operations, in order to avoid half inserted pages to be exposed to readers. However, the insertion could be done without `PG_nonewrefs` by properly ordering the operations.

On the other hand, the deletion fully relies on `PG_nonewrefs`. It is used to hold off the return of the *speculative get* until the page is fully removed. Then the accuracy check, after obtaining the speculative reference, will find that the page is not the one we requested, and will release the reference and re-try the operation. We cannot rely on `atomic_inc_not_zero()` failing in this case, because the pagecache itself still has a reference on the page.

By changing `PG_nonewrefs` into a bit-spinlock and using it around all modifying operations, thus serialising the pagecache on page level, we satisfy the requirements of both the lockless and the concurrent pagecache.

4 Performance

In order to benchmark the concurrent radix tree, a new kernel module is made. This kernel module exercises the modifying operations concurrently.

This module spawns a number of kernel threads, each of which applies a radix tree operation on a number of indices. Two range modes were tested: interleaved and

sequential. The interleaved mode makes each thread iterate over the whole range, and pick only those elements which match *i mod nr_threads = nr_thread*. The sequential mode divides the full range into *nr_thread* separate sub ranges.

These two patterns should be able to highlight the impact of cache-line bouncing. The interleaved pattern has a maximal cache-line overlap, whereas the sequential pattern has a minimal cache-line overlap.

4.1 Results

Here we present results obtained by running the new kernel module, mentioned above, on a 2-way x86-64 machine, over a range of 16777216 items. The numbers represent the runtime (in seconds).

The interleaved mode gives:

operation	serial	concurrent	gain
insert	16.006	19.485	-22%
tag	14.989	15.538	-4%
untag	17.515	16.982	3%
remove	14.213	16.506	-16%

The sequential mode gives:

operation	serial	concurrent	gain
insert	15.768	14.792	6%
tag	15.110	14.581	4%
untag	18.138	15.027	17%
remove	14.607	16.250	-11%

As we see from the results, the lock induced cache-line bouncing is a real problem, even on small SMP systems. The locking overhead is not prohibitive however.

5 Optimistic Locking

Now that the pagecache is page-locked, and the basic concurrency control algorithms are in place, effort can be put into investigating more optimistic locking rules for the radix tree.

For example, for the insertion we can do an RCU lookup of the lowest possible matching node, then take its lock and verify that the node is still valid. After this, we continue the operation in a regular locked fashion. By doing this we would avoid locking the upper nodes in many cases, and thereby significantly reduce cache-line bouncing.

Something similar can be done for the item removal: find the lowest termination point during an RCU traversal, lock it and verify its validity. Then continue as a regular path-locked operation.

In each case, when the validation fails, the operation restarts as a fully locked operation.

Since these two examples cover both the ladder-locking model in Section 2.4.1, and the path-locking model in Section 2.4.2, they can be generalised to cover all other modifying operations.

5.1 Results

Rerunning the kernel module, in order to test the concurrent radix tree performance with this new optimistic locking model, yields much better results.

The interleaved mode gives:

operation	serial	optimistic	gain
insert	16.006	12.034	25%
tag	14.989	7.417	51%
untag	17.515	4.135	76%
remove	14.213	6.529	54%

The sequential mode gives:

operation	serial	optimistic	gain
insert	15.768	3.446	78%
tag	15.110	5.359	65%
untag	18.138	4.126	77%
remove	14.607	6.488	56%

These results are quite promising for larger SMP machines.

Now we see that, during the interleaved test, the threads slowly drifted apart, thus naturally avoiding cache-line bouncing.

6 Availability

This work resulted in a patch-set for the Linux kernel, and is available at:

```
http://programming.kicks-ass.net/
kernel-patches/concurrent-pagecache/
```

References

[1] Wikipedia, *Read-Copy-Update*
`http://en.wikipedia.org/wiki/RCU`

[2] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004.

[3] Wikipedia, *Radix Tree*, `http://en.wikipedia.org/wiki/Radix_tree`

[4] N. Piggin, *RCU Radix Tree*,
`http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16-rc5/radix-intro.pdf`

[5] N. Piggin *A Lockless Pagecache in Linux - Introduction, Progress, Performance*, Proceedings of the Ottawa Linux Symposium 2006, pp. 241–254.