# SCSI Mid-Level Multipath

*Michael Anderson, Patrick Mansfield*
IBM Linux Technology Center
andmike@us.ibm.com, patmans@us.ibm.com

## Abstract

Multipath IO is the ability to address the same storage device over multiple connections, providing improved reliability and availability. This concept is not new to Linux®. Multipath capabilities exist in the volume management layer, SCSI upper level, and in the SCSI lower level device driver. This paper examines an approach to providing multipath support in the Linux 2.5+ SCSI mid-level. An implementation at this level gives the reduced resource usage and better performance of lower level implementations, along with the device independent capabilities of upper level implementations.

The target audience is developers knowledgeable about SCSI or Linux SCSI internals that are also interested in multipath storage support.

## 1 Introduction

This section provides an overview of the characteristics of the multiple paths and multiple ports presented to the Linux kernel by the storage IO transport and by the storage device itself.

A path is the connection between the host system and the storage device. Multiple paths to a device result from the storage device having more than one port (multi-port storage device) or the host system having multiple connections into a given bus or fabric that connects to a storage device port (multi-initiated interconnect).

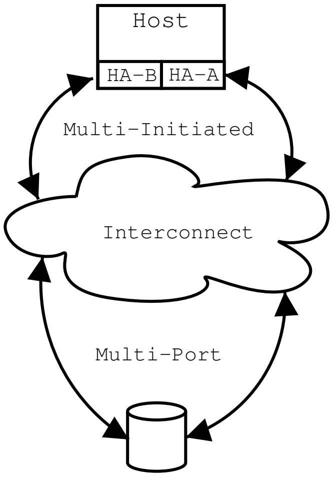

Figure 1: Multi-Initiated and Multi-ported multipath configuration

Utilization of a multipath device can increase the availability of the storage device presented to the operating system by reducing the loss of access due to a single transport problem. Multipath device support may also provide an increase in performance due to load balancing if the performance attributes of the storage device are greater than a single transport can deliver. When a system architecture like NUMA exhibits increased latency between local memory and non-local adapters multipath with NUMA aware routing can be used to route IO to adapters with the lowest latency.

Although the necessity for multipath in enterprise systems is clear, the selection of where to implement it has lead to several different approaches in the Linux kernel. This SCSI mid-level multipath solution was created to address the following:

- Implementation at a level higher in the stack than vendor unique lower level driver solutions, while not exposing device specific knowledge to layers above SCSI.
- Reduction of kernel resources while still utilizing the existing IO scheduler and interfaces of the block layer.
- Binding paths to devices without obtaining information from the devices media, allowing support for both block and character devices.
- The ability to distinguish between path and device errors.
- Selection of the optimal path for IO based on Lower Level Device Driver (LLDD) attributes, NUMA topology, and device attributes.

### 1.1 Multi-Initiated Interconnect

A multi-initiated interconnect results from attaching multiple host adapters to a single interconnect. For example, a multi-initiated SCSI bus or multi-initiated Fibre Channel.

Multi-initiated paths to a single storage device normally present equal characteristics. Some hardware platforms can create performance inequalities down separate paths to a storage device port when different latencies exist between a host adapter and the memory it is referencing. A NUMA architecture based platform can present such latencies; depending on the magnitude of the latency, platform specific routing policies can increase performance. Path selection will be discussed in detail later in the document.

### 1.2 Multi-Port Storage Device

A storage device can present multiple protocol communication ports to an IO interconnect. These ports can be accessed from the host through a single host bus adapter (single initiator) or multiple host bus adapters (multi-initiated).

While the performance characteristics of IO down multiple paths to a single port of a device are nominally equal (excluding NUMA), the performance to different ports of the device can vary greatly due to the architecture of the storage device.

These different storage device architectures can be grouped into three behavior models based on the device's differing response to IO sent to more than one port. A device may exhibit differing port behavior only on ports that cross logical unit ownership boundaries. Some storage devices can be configured to operate in more than one behavior mode.

- **Failover** – When a device is operating with Failover behavior, IO to a secondary port must be preceded by control commands indicating a redirection of all IO to an alternate port. Once the storage device has transitioned, "Failed over" IO may be directed to an alternate port. This behavior model is exhibited in devices with some performance penalty in the transition of logical unit ownership. Clustered shared storage systems may use this model to keep port thrashing from degrading performance.
- **Transparent Failover** – IO to a single volume should only be sent to a single port until an availability condition arises to cause IO to be redirected to a secondary port. The storage device will transition transparently to using the secondary port

on the receipt of the first IO to this port. The storage device transition delta for this first IO is significant when compared to subsequent IOs, such that performance would degrade noticeably if port transitions occurred frequently (i.e., if round robin routing policy were used).

- **Active Load Balancing** – IO to a single storage device volume can be sent down any path without degrading performance (note: some cache warmth benefits may be achieved by using more sophisticated path selection algorithms, but this is vendor unique). These storage devices usually have a cache that can be symmetrically accessed from any input port or a single cache that all input ports feed into.

### 1.3 Linux Multipath Implementations

Multipath support to a storage device can be implemented at different levels in the Linux operating system's IO stack. This support can be provided by storage vendors, adapter vendors, and the base kernel.

- **Volume Management** – Multipath at this level is usually implemented as a modified case of existing RAID support. Multiple block devices exposed by the operating system point to the same storage device and are configured to be failover paths for IO. The "md" driver [3] and LVM multipath patch [2] are examples of support at this level.

- **Upper Level** – Support at this level involves chaining or linking the multiple block devices exposed by the operating system as failover paths. An example of this type of implementation is the T3 Multipath failover driver written by Linuxcare Inc. [5].

- **Mid-Level** – This is the level of implementation described in this document.

- **Lower Level** – An implementation at this level involves a binding of common vendor adapters exposing only one device to the operating system. On failure, the adapter driver re-drives the IO through another adapter previously paired as a failover adapter. An example of this type of implementation is the Qlogic Fibre Channel failover driver.

## 2 Data Model

### 2.1 Current Linux SCSI Device Data Model

This section provides a high level overview of the Linux 2.5 SCSI subsystem data structures with a focus on providing a background for later discussion on multipath support. General Linux SCSI information can be obtained by viewing the "The Linux 2.4 SCSI subsystem HOWTO" [1] and [4] listed in the Reference section.

Each LLDD that wishes to register with the Linux SCSI sub-system provides a SCSI host template (`Scsi_Host_Template`) data structure that describes the capabilities of the driver and interface functions.

The LLDD can register with the SCSI subsystem in two ways. One method is a Legacy interface, which is driven from the SCSI mid-level code and calls into the LLDD detect routine, which causes `scsi_register()` to be called for each adapter card detected by the driver. The other method allows the LLDD to call `scsi_register()` directly and then call `scsi_add_host()` when it is initialized and ready to be scanned. These two methods result in a `Scsi_Host` data structure being allocated for each instance of an adapter card.

If an adapter contains multiple busses or channels (not a PCI bridge of two cards), there will be only one SCSI host structure (`Scsi_Host`) created.

After a kernel boot, a insmod of a LLDD, or hotplug event, a list (`scsi_hostlist`) will contain a SCSI host structure representing each adapter detected.

During device scanning a SCSI device data structure (`scsi_device`) will be allocated for each logical unit discovered. Each SCSI device structure will be added to a linked list member of its SCSI host parent.

See figure 2 for a diagram of the data structures and their relationships.

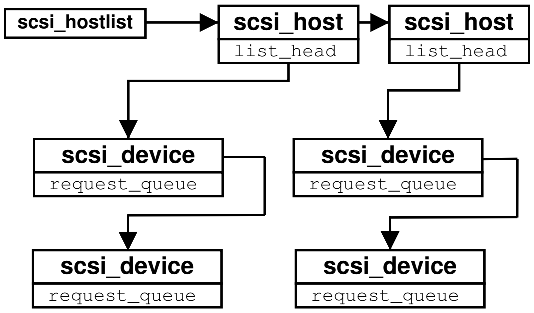

Figure 2: Linux SCSI Data Structures

Once the scanning phase is complete a `struct scsi_device` will be associated to only one `struct Scsi_Host`.

### 2.2 Mid-Level Multipath Data Model

When no multipath capabilities are enabled in the Linux SCSI subsystem, multiple paths result in a SCSI device structure being created and eventually exposed through the block or character layer for each path. This redundancy wastes system resources and creates a non-optimal presentation of structures to the block layer and user level.

The mid-level multipath implementation coalesces these extraneous SCSI device structures while still maintaining the information relating to the paths.

The child relationship of the SCSI device structure to the SCSI host structure is removed and a new relationship is created using the mid-level multipath structures. The SCSI multipath structure (`struct scsi_mpath`) is a container for all the paths to the storage device. The SCSI mpath structure also contains information on the path routing policy, a count of active paths, and a reference to where the last IO was routed. The `scsi_mpath` structure is associated with the SCSI device structure through a new member, `sdev_paths`.

Each path is represented by a SCSI path structure (`struct scsi_path`). This path structure contains a fast reference to the next sibling path, the state of the path, and a SCSI nexus structure (`struct scsi_nexus`).

The nexus object contains the transport specific knowledge to communicate with the storage device. In an ideal world, this nexus would be an opaque object (i.e., a handle) that was handed to the SCSI mid-level during the device scanning process.

See figure 3 for a diagram of the multipath data structures and their relationships.

#### 2.2.1 Multipath Data Model General Applicability

The data model presented for multipath has advantages even in non-multipath cases.

Because the mid-level model has separated the request queue presented to the block layer from the nexus object that is associated with the SCSI hosts, paths containing nexus objects can be added to or removed from a SCSI device

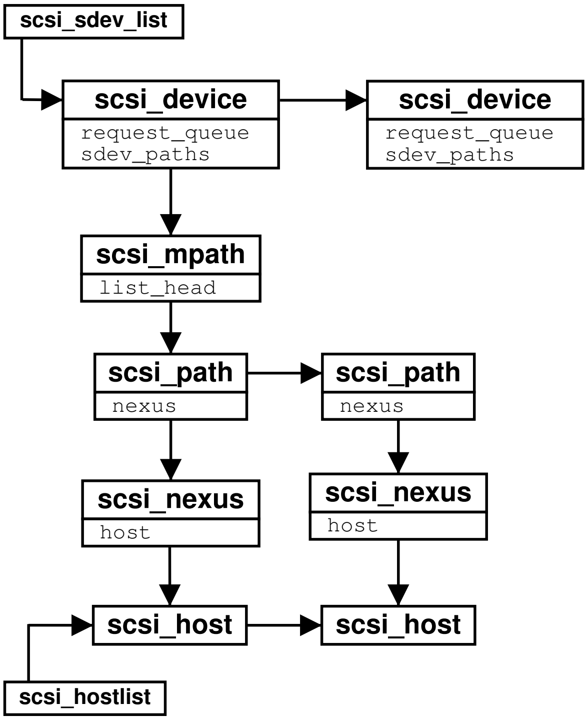

Figure 3: Mid-Level Multipath Data Structures

structure.

When all paths have failed or there are no paths to a device, a policy could be created to allow the SCSI device structure to remain in place, but suspend SCSI IO request processing. This would allow block and file system level attachments to remain established while transport connectivity is in flux.

If UUID authentication can be ensured (this would be the case for multipath devices supported by this implementation) new paths could bind to the SCSI device and IO request processing would resume. If authentication cannot be ensured, lower level resources can be released in less time than current structures allow.

Depending on the future direction of the SCSI mid layer, this separation could be used to add or remove SCSI subsystem components while IO is active.

# 3 Mid-Level Multipath

## 3.1 Linux SCSI Scanning Overview

Following is an overview of the SCSI scan algorithm for a given logical unit within `scsi_scan.c` as pertains to modifications for use with multipath (per code in linux version 2.5.68).

A call to `scsi_alloc_sdev` allocates and initializes a `scsi_device` (sdev). Note that a `sdev`, logical unit, and I_T_L nexus are all equivalent in the current linux SCSI code. The LLDD `slave_alloc()` function is called for the `sdev` [4].

The `sdev` is sent an INQUIRY. Device attributes settings are obtained by calling `scsi_get_device_flags()`.

If the logical unit responds, and it has a logical unit configured, the `sdev` is left in place, otherwise it is removed.

`scsi_load_identifier()` function is called to get a UUID (universal unique identifier) via SCSI INQUIRY VPD pages [6], and the result is stored in `sdev->name`.

`scsi_device_register()` is called, generating a hotplug event.

Last of all, the LLDD `slave_configure()` function is called for the `sdev`.

Upper level attaches are done after all scanning (on insmod or initialization of a LLDD) or the upper level attach is done after a single logical unit is scanned via /proc/scsi/scsi (in `scsi_add_device()`). These in turn can generate their own set of hotplug events as the upper level drivers (sd, st, sr, and sg) attach to each sdev.

This means that a series of hotplug events occurs for many scsi_devices, followed by a series of hotplug events for each upper level device (such as a block device).

### 3.2 Scan Modifications for Multipath

The scanning code is changed as follows for mid-level multipath support.

The allocation of an sdev is modified to not only allocate the actual scsi_device, but to also allocate and add a single path (struct scsi_path, including a struct scsi_nexus) to the sdev. slave_alloc() is modified to take both an sdev and scsi_nexus as an argument, such that it can access both logical unit data (in the sdev) and nexus specific data.

Future plans are to supply a set of parallel slave_nnn interfaces for use with multipath, so that existing drivers not supporting the new interfaces will behave as if the multipath patch were not applied (each path to a storage device will generate a new scsi_device).

After a UUID is retrieved, all existing sdev's are searched for a match.

If no match is found, this is the first path to the device, and it is handled the same way as the current non-multipath code.

If a match is found, the new path is added to the matching sdev (the paths are coalesced), and the current sdev is freed.

slave_configure() is also modified to take both an sdev and a scsi_nexus as arguments.

### 3.3 UUID

The immutability of the UUID is key to determining if more than one nexus can access the same storage device. This is not a simple problem to deal with, as some devices return no UUID, some return a UUID that is not unique, and others require device specific methods to retrieve a truly unique UUID. Future changes (such as a UUID white list) are required to properly handle the UUID in all cases; user level scanning would simplify the problem.

Discussions were actively in progress at the time this paper was written on whether or not to keep the existing UUID retrieval code in the kernel. Depending on the outcome, and amount of time available, the multipath patch might have to carry the UUID retrieval.

The primary problem with moving UUID retrieval to user level (for use with multipath, assuming full user level scanning is beyond the scope of the current implementation) is that the current scan and upper level attach are initiated without the ability for user level intervention - all upper level devices are attached to all scsi_devices with no synchronization from user space.

Without the coalescing of paths as described above devices can show up multiple times, leading to potential resource shortages (memory as well as major/minor numbers), and potential problems for applications dependent on the hiding of duplicate paths.

Further investigation is needed to determine if it is practical to modify the scan and upper level attach to be user initiated (versus the more difficult problem of complete user level scanning). Such that all devices are scanned in kernel code, and then from user level: UUID's retrieved, coalesced, and then upper level attaches triggered.

### 3.4 Current SCSI I/O Request Flow Overview

Following is an overview of the current IO request flow as it pertains to functions modified for use with SCSI mid-level multipath IO.

A SCSI device structure request queue member (`request_queue`) is registered with the block layer at SCSI scan time for en-queuing requests. Both SCSI character and block devices utilize this queue, as do the commands issued during SCSI scan and by upper level attachment.

For block IO devices, an IO request is sent to the block layer via the `__make_request()` function. In turn it eventually calls the SCSI block request function, `scsi_request_fn()`.

For SCSI character devices, scanning, and commands sent during upper level attaches, `scsi_do_req()` or `scsi_wait_req()` are used to send SCSI commands to the `scsi_device`. These functions setup the `sr_done` function pointer, insert a request, and trigger a call to the `scsi_request_fn()` via a call to the block queue `blk_insert_request()`.

The `scsi_request_fn()` function retrieves a request and calls the `scsi_prep_fn()` via a call to the `elv_next_request()`.

`scsi_prep_fn()` allocates and initializes the `scsi_cmnd`. The `scsi_cmnd done` function is set in upper level drivers via calls to their `init_command` functions. For users of `scsi_wait_req()` or `scsi_wait_req()`, the `done` function is set to the `sr_done`.

The `scsi_cmnd` is the key data structure used to issue a request to a LLDD.

Control continues in `scsi_request_fn()`, where resource limitations and hardware limits (such as queue depth) are checked via calls to `scsi_dev_queue_ready()` and `scsi_host_queue_ready()`.

If resources are available, `scsi_dispatch_cmd()` is called, it adds a timeout, and transfers control to the LLDD by calling the `scsi_host queuecommand` function, passing the `scsi_cmnd`, and `scsi_done()`.

The LLDD is responsible for sending the command to the actual logical unit. After the request is submitted, `queuecommand` returns.

Upon completion of the IO request, the LLDD calls the `scsi_cmnd scsi_done()` function.

`scsi_done()` puts the completed command onto a per-CPU queue, and raises the SCSI_SOFTIRQ.

`scsi_softirq()` determines the completion status of each `scsi_cmnd` via calls to `scsi_decide_disposition()`.

`scsi_decide_disposition()` classifies the completion status of the IO (the scsi_cmnd) and returns the following values to `scsi_done()`, that lead to further actions:

`SUCCESS`: the IO has completed without error, the command is completed by calling `scsi_finish_command()`. This includes an IO completion with failures (for example, an IO went to a disk, but had media errors).

`ADD_TO_MLQUEUE`: the IO completed with a SCSI QUEUE FULL status. The command is re-queued for a retry via `scsi_queue_insert()`, effectively resending the command.

`NEEDS_RETRY`: the IO had a temporary or other condition such that it can be immediately resent, resend the IO (without re-queuing it) by calling `scsi_retry_command()`.

`FAILURE` or any other value: a device or transport error occurred. Call `scsi_eh_scmd_add()` to queue the failed command for error handling; when the host adapter has no more IO outstanding (when the `active_count` equals the `host_failed` count) the error handler wakes up and handles all failed commands.

`scsi_finish_command()` is the main path for IO completion, it calls the `scsi_cmnd done` function, either the upper level completion function (for sd, `sd_rw_intr`) or the function specified in `scsi_wait_req()` or `scsi_wait_done()`.

### 3.5 Modifications for IO Path Selection and Path Failures

The SCSI code is modified as follows for mid-level multipath.

A path (including nexus) is selected via a call to `scsi_get_best_path()` from `scsi_request_fn()`.

Path selection is affected by the number of paths, path state, path policy, and NUMA topology.

A list of all available paths and all active paths to a device are kept. In addition, for NUMA systems, there is a list of paths local to a given node.

If no active paths are available, the `scsi_request_fn()` function fails the IO request.

Currently, path selection policy is controlled via the global variable `scsi_path_dflt_path_policy`. This is set via the kernel config and can be modified at boot time, with future plans to allow setting this both per device and via sysfs.

Setting `scsi_path_dflt_path_policy` to SCSI_PATH_POLICY_LPU (1) sets the path selection policy to last path used. This means that another path will only be used on failure.

Setting `scsi_path_dflt_path_policy` to SCSI_PATH_POLICY_ROUND_ROBIN (2) sets the path selection policy to round robin. This means that paths will be rotated across all available paths on every request sent to the device.

A last-path used policy is safest for general purpose use (for example with a device using a transparent failover model). Future plans are to add device specific attribute hooks and code to fully support a transparent failover model, so that round-robin path selection can be done across a subset (lowest path weight) of all paths, not just a single path.

NUMA path selection where possible picks a path local to the node containing the memory to be used for the IO operation. If no local path is available, all paths are valid for selection. So, for round-robin path selection, path selection is either round-robin with respect to all paths local to a given node, or for all paths to a device.

Current NUMA multipath support is limited to a one-to-one mapping of path to node. Future plans are to support multiple nodes connected to the same path (the same bus). Changes are required in the current kernel NUMA topology in order to support topologies that have varied or unequal distances (that is, where the inter-node distances can vary), or for cases where a node contains no CPU's.

For multipath, the `scsi_cmnd device` is changed from a `struct scsi_device` pointer to a struct scsi_nexus pointer; the scsi_nexus retains the same fields as those used by the LLDD in order to determine the nexus (that is, the scsi_nexus contains a scsi_host pointer, channel, id and lun).

Then as part of the path selection, the scsi_cmnd device pointer is set to point to the selected path's nexus. (Renaming the scsi_cmnd device to nexus would be appropriate.)

The LLDD queuecommand function is called, passing the scsi_cmnd that includes a pointer to a scsi_nexus. Existing LLDD code can then be used, with no changes required to the core of the LLDD.

Upon IO completion, scsi_path_decide _disposition() categorizes failures as transport (path failure) or device specific (device failure). Path specific failures cause a path to fail, and the IO can be retried on any remaining paths. Device specific failures generally offline the device, and do not allow an IO to be retried.

scsi_check_paths() is then called with an indication as to whether a path failure has occurred or not, it updates the path state, and returns a result specifying the action to take: the standard SUCCESS (meaning the IO has completed, not that it has completed successfully), FAILURE, or a new REQUEUE value signaling that the IO should be re-queued. NEEDS_RETRY is no longer returned.

In scsi_softirq(), when a REQUEUE result is returned from scsi_decide_disposition(), the IO is re-queued via scsi_queue_insert().

So, on a path failure, the IO is re-queued, and in scsi_request_fn() the IO can be retried on any of the remaining paths.

### 3.6 User Space Interface

#### 3.6.1 Procfs

The mid-level multipath code provides a procfs interface for viewing and setting attributes related to paths. The path to the procfs file is /proc/scsi/scsi_path/paths. The file supports both read and write operations, and displays attributes about the paths. The table below provides a description of each of the columns in the output.

| Column | Description |
|---|---|
| 1 | UUID |
| 2 | Host Number |
| 3 | Host Channel |
| 4 | Target ID |
| 5 | Lun |
| 6 | State |
| 7 | Failures |
| 8 | Weight |

Table 1: Procfs Columns

An edited output to show only one device of a read is shown as follows:

```
#cat /proc/scsi/scsi_path/paths
...
2000002037171f24 3 0 1 0 1 0 0
2000002037171f24 4 0 1 0 1 0 0
...
```

Writing to the file allows path attributes to be modified. Currently the only meaningful write operation is to modify path state. A path state may be modified from dead to good or good to dead. A good path state has a value of "1" and a dead path has a value of "3".

An example of failing a path is shown as follows:

```
echo ‘2000002037171f24 3 0 1 0 3 0 0‘
```

```
> /proc/scsi/scsi_path/paths
```

In the near term the procfs interface will be replaced with a sysfs interface. At the time of this writing, the interface was not complete and is discussed in the Future Work section.

# 4 Future Work

## 4.1 Multipath Device Personality

The use of device-neutral information through standard SCSI interfaces limits the set of multipath capabilities that can be supported on a given device to a minimal set known to be safe for all devices. To utilize the extended capabilities of some storage device's, the device attributes or a device's "personality" needs to be exposed.

The interfaces provided for obtaining this personality knowledge will not be restricted to kernel space. Some data can be set from user space, but other operations will need to be kernel resident to avoid deadlock. Further direction toward user level scanning will affect these interfaces.

The single kernel config time path policy can be enhanced with device attribute information allowing support for device specific path policies.

Path weighting values related to a device's attributes would allow proper primary and secondary paths to be determined. The ability to determine preferred paths to assist in the balancing of load across a storage device's port can also be determined.

## 4.2 SCSI Reservations

The support of SCSI Reserve/Release affects the type of path selection policy that can be selected for a storage device restricting it to the last path used. Support of SCSI persistent reserve requires an interface to accept a reservation key and special IO operations before paths can used for normal IO. Future work is trying to meet the requirements of SCSI reservation with a general purpose path preparation capability.

## 4.3 Sysfs Interface

As mentioned in a previous section, the procfs interface to mid-level multipath is being migrated to a sysfs-based interface. The migration to a sysfs interface allows for the linkage to the device tree for increased path topology information and the utilization of common kernel code infrastructure, reducing code duplication.

# 5 Availability

The SCSI Mid-Level Multipath project page is located at:

http://www-124.ibm.com/storageio/multipath/scsi-multipath/

# Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

# References

[1] Douglas Gilbert, "The Linux 2.4 SCSI subsystem HOWTO" http://www.linuxdoc.org/HOWTO/SCSI-2.4-HOWTO/index.html

[2] “LVM multipath support”,
`http://oss.software.ibm.com/linux390/useful_add-ons_lvm.shtml`

[3] “MD Multiple Devices driver”,
`drivers/md/*`

[4] “SCSI mid_level - lower_level driver interface”,
`Documentation/scsi/scsi_mid_low_api.txt`

[5] “Linux T3 Driver”,
`http://open-projects.linuxcare.com/t3/`

[6] “SCSI Primary Commands - 3 (SPC-3)”,
`ftp://ftp.t10.org/t10/drafts/sam3/sam3r06.pdf`

[7] “SCSI Architecture Model - 3 (SAM-3)”,
`ftp://ftp.t10.org/t10/drafts/sam3/sam3r06.pdf`

# Proceedings of the
# Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

## Conference Organizers

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

## Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*