Reprinted from the

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.*

relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space

Tom Zanussi zanussi@us.ibm.com
Karim Yaghmour karim@opersys.com
Robert Wisniewski bob@watson.ibm.com
Richard Moore richardj_moore@uk.ibm.com
Michel Dagenais michel.dagenais@polymtl.ca

Abstract

Linux has several mechanisms for relaying information about the system and applications to the user. Some examples include printk and other syslog events, evlog, ltt, oprofile, etc. Each subsystem has its own method for relaying information from the kernel to user space. Some of these mechanisms have difficulties, e.g. logging of printk messages is unreliable. In addition to selected difficulties, the replication of code and maintenance is undesirable. In this paper we describe a high-speed data relay filesystem that satisfies the buffering requirements of the above subsystems while providing a unified, efficient, and reliable relay mechanism. relayfs allows subsystems to log data efficiently and safely using lockless technology that is designed to scale well on multiprocessor systems. relayfs includes the flexibility to be expanded should other subsystems need additional services, but has a simple design intended to meet the needs of currently available subsystems. In this paper we discuss the architecture, implementation, and usage of relayfs. relayfs uses channels that allow data to be directed to a suitable buffer or buffers for the subsystems that allocated the channel. We describe the kernel API and file naming conventions, address init-time issues, and discuss performance trade-offs available using relayfs. Finally we demonstrate how existing subsystems use relayfs to log their data.

1 Introduction

Sharing data between the kernel and user-space applications requires buffering. For relatively small transfers, such as passing variables or small data arrays, the normal system call API is sufficient; the overhead required for safely transferring data across the kernel boundary is acceptable. For larger data transfers, the subsystem generating the data is responsible for providing a buffering and transfer mechanism to deliver the data to user space. With increasing system speed and growing demand for more information regarding the kernel's operation, many of the conventional buffering mechanisms have reached their limits. Further, the buffering and transfer code for each of the subsystems is replicated and needs to be independently maintained.

To address these challenges, we designed and implemented relayfs. relayfs is a unified, reliable, efficient, and simple mechanism for transferring large amounts of data between the kernel and user space. The algorithms used for relayfs have been designed to handle both high frequency and large data applications such as kernel tracing. To satisfy these demand-

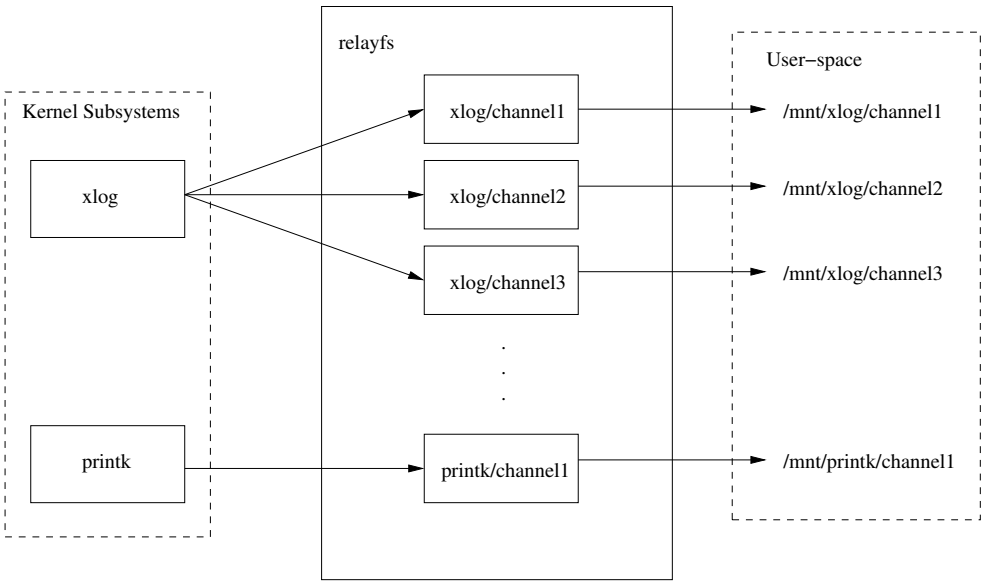

Figure 1: relayfs architecture

ing clients, relayfs is a high-speed, low-impact data relay filesystem. Data logged by a subsystem using relayfs appears under the directory where the filesystem is mounted. For example, if relayfs is mounted under /mnt, printk data may appear in /mnt/printk/data. Although this paper focuses on the use of relayfs for kernel subsystems, user-space clients can also be served by relayfs.

relayfs provides the abstraction of a *channel* to the subsystem using its services. Channels and files have a 1-to-1 mapping. A given subsystem may log data to multiple channels. For example, a tracing subsystem may log the majority of its data to one channel and create a control channel for less frequent but informative events such as new process creation. printk could, for example, use different channels to log information from devices versus the memory system. Alternatively, a kernel developer could create a separate channel for just the events in newly written (and currently being debugged) code, so as to view only those.

relayfs has a simple design and mechanisms intended to meet the demands of current subsystems while affording flexibility in meeting new requirements of future subsystems. relayfs provides the option of using locking or lockless data logging. The lockless option is very efficient but introduces potential difficulties as discussed later. relayfs provides the choice of using per-processor buffers or a single buffer shared across the machine. It also provides choice between block or packet delivery of events. The trade-offs of these options are discussed later as well.

In addition to these options, relayfs reduces code replication among subsystems by providing mechanisms common to the subsystems. These include handling overflow issues, providing efficient timestamping on events, providing efficient delivery to user space, and handling dynamic allocation and de-allocation of memory needed to provide the buffering. Tests conducted using relayfs have shown it can handle significant amounts of data with very low overhead.

The rest of the paper is structured as follows.

Section 2 describes the channel architecture and the lockless algorithm. Section 3 describes how the code is structured and where to find the implementation files. Section 4 describes the interface to relayfs, the different options available, and how it handles overflows. Section 5 describes the impact and performance of relayfs. Section 6 describes how to modify some example subsystems to use relayfs. Section 7 concludes.

2 Architecture

Figure 1 presents the relayfs architecture. Kernel subsystems create and use different relayfs channels to log their data, while user-space applications see those same channels as files located in the mounted relayfs filesystem.

2.1 Channels

The main building block of relayfs is a channel. To relay data to user-space applications, kernel subsystems allocate channels to transfer their data. Multiple channels can be used to implement any data multiplexing desired, including using one channel per CPU to implement per-CPU buffering.

The buffering implemented by relayfs is transparent to the client subsystem. Writing to the channel requires that the following be specified: channel ID, pointer to the data, and size of data. relayfs does not parse the data being relayed; it just transfers bytes. Because some relayfs clients may implement a particular data protocol (for example, special markings on buffer ends), relayfs provides callback functions for its clients when significant changes in the channel buffers occur. The callbacks are optional.

From user space, channels are accessed as files. relayfs implements the standard operations required for normal file manipulation. For example, to gain access to a relayfs file, applications can perform read() operation on the file specifying a number of bytes. Alternatively the application can mmap() in the file and reference the contents of the file via memory pointer.

2.2 Lockless Event Logging

An important feature of relayfs is its ability to write data to buffers without requiring locks. Previous lockless logging schemes[2] used fixed-length events with valid bits. There are several advantages to using variable-length events (assuming the random access problem is solved, see Section 4.2). The algorithm integrated into relayfs allows variable-length events to be logged without locking.

Each process attempts to *reserve* enough space in the buffer immediately after the current index for the event it wants to log. Once the process makes a successful reservation, it may proceed to log its data. To reserve space, each process attempts to atomically increment the current index using a `compare and store`. The process that successfully increments (as determined by the return value of the `compare and store` operation) the index has the right to proceed to log data into the buffer; failing processes retry. Figure 2 shows on the left, in step 1, two processes, A and B, attempting to log events of different lengths after the current index from the initial configuration in step 0. Each process attempts to increment the current index by the size of the event being logged. The process that succeeds, in this case B, will log the event immediately following the old current index (see step 2). This will be followed by process A's data, assuming no other competing processes attempt to log more data (see step 3). Because it is important to guarantee monotonically increasing timestamps, processes must re-determine the timestamp during each attempt to atomically

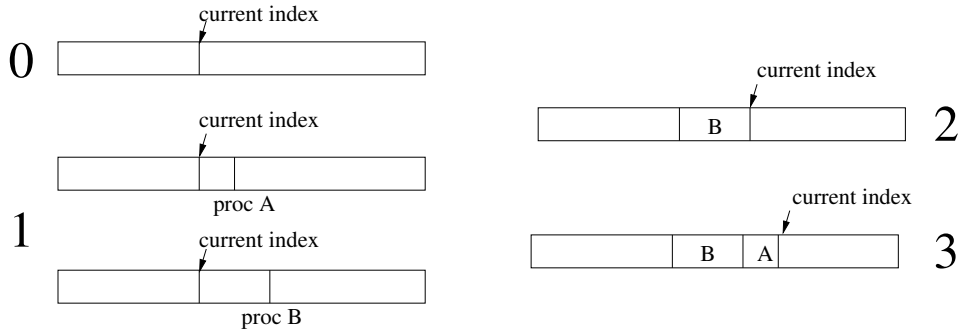

Figure 2: Illustration of Lockless Event Logging

```
eventReserve(length, *indexPtr, *timestampPtr)
  integer: oldIndex, newIndex
  EvtCtl: *evtCtlPtr
    update evtCtlPtr
    do
      oldIndex = evtCtlPtr->index
      newIndex = oldIndex + length
      if (newIndex >= buffer end)
        eventReserveSlow(length, indexPtr, timestampPtr)
          // generates filler event, sets timestamp, moves to new buffer
        return
      *timestampPtr = getTimestamp()
    while (!CompareAndStore(&(evtCtlPtr->index), oldIndex, newIndex))
    *indexPtr = oldIndex & INDEXMASK  // confine index to buffer bounds

eventLog(majorID, minorID, data)
  integer: index, timestamp, length
    length = length of data
    eventReserve(length, &index, &timestamp)
    evtArray[index] = logEvtHeader(timestamp, length, majorID, minorID)
    evtArray[index+1 ... index+length] = data
    eventCommit(index, length) // optional, see explanation in text
```

Figure 3: Pseudo code for lockless event logging

increment the index.

The memory used for logging is logically divided into buffers. Once a buffer is full, the logging facility proceeds to the subsequent buffer and the previous buffer is available to be written out. The pseudo code appears in Figure 3 and complete C code can be obtained by downloading relayfs from the relayfs web site[1].

Despite having good performance, complications can arise from using the lockless algorithm. A process's execution may be interrupted after it has reserved space to log an event, but before it actually performs the log. The interruption can occur because the process is preempted, blocks for a long time, or is killed. Depending on where (in the sequence of code in Figure 3) the process is interrupted, different problems occur. If the process has had a chance to write the event header, but not the data, then only the data will be unrecoverable. If, however, the process has not yet logged the event header then it is possible the

rest of the buffer will be uninterpretable. Only by locking, making the kernel perform the log, and disabling interrupts can this problem be prevented (in practice there are low-level kernel events that would still exhibit the problem). There are methods that avoid these difficulties.

If the reason the process's execution was interrupted was due to preemption, then it is likely the process will run again soon and finish filling in the event before another entity notices, thus posing no real problem. If the reason for the process's interruption was because the process was killed, then the data will never finish being logged. The last line of pseudo-code detects this situation. The `eventCommit` function updates a per-buffer count of the amount of data that has been logged to that buffer. The count is zeroed during the `start new buffer` code. When the code responsible for writing the data (to a network stream, file, etc.) writes this buffer, it can compare the amount of data logged to this buffer with the buffer's size and report an anomaly if they do not match. If the reason the process was interrupted was because of a long blocking operation, it is possible that both the current buffer will not have enough data logged, and that the same buffer, when reused in the future, will have too much (because the long-blocked process was unblocked and logged data into a recycled buffer). Again the per-buffer counts can detect this situation.

Besides a per-buffer count there are other possible ways to detect or minimize the occurrence of corrupted data. For example, it is possible to use a flag in a per-process data structure to indicate to the kernel that a process should not be killed while the flag is set. Other possibilities include zero-filling a buffer before use, or keeping a side array of valid bits for the header data. In practice the probability of corrupting a buffer, and the ease with which tools can handle the situation, reduces the issue's importance except perhaps in setups where recreating the situation generating the logged data is very difficult. In those cases the locking version of the event logging may be the best option.

3 Implementation

The relayfs code is structured as follows: the public API and common relay code are contained in fs/relayfs/relay.c, with the scheme-specific code in fs/relayfs/relay_lockless.c and fs/relayfs/relay_locking.c. The file fs/relayfs/inode.c implements the VFS layer on top of the relay channel code.

relayfs can be compiled either directly into the kernel or as a kernel module.

4 Interface and Use

relayfs is used both as a temporary repository for logged data and as a filesystem from which user-space clients may retrieve logged data. The first of these is supported by set of kernel-space APIs. For the second, relayfs is mounted as a filesystem:

```
mount -t relayfs relayfs /mnt/relay
```

Kernel subsystems (also referred to as kernel clients) create and write to channels via the kernel API described below. The contents of these channels are available to user-space programs via a standard file abstraction that can be read using mmap() or read(). relayfs provides automatic support for locking or lockless logging, overflow handling, data delivery, and timestamping.

4.1 Interface

This section describes the basic usage of the kernel and user APIs. Com-

plete details can be found in Documentation/filesystems/relayfs.txt. To initiate data logging, a kernel client creates a channel relative to the mountpoint of the relayfs filesystem via relay_open():

```
int channel_id =
      relay_open("file", ...);
```

This would cause the creation of a relayfs file named /mnt/relay/file, assuming relayfs was mounted at /mnt/relay.

relayfs does not impose any namespace conventions; clients may choose names as they wish. We recommend the adoption of the convention whereby a client specifies a top-level directory name that is closely associated with the corresponding subsystem. For example, the Linux Trace Toolkit would manage the namespace under /mnt/relay/trace, printk would use /mnt/relay/printk, driver debugging channels might use /mnt/relay/debug/drivers/mydriver, etc.

A kernel client can then log a variable-length data item to the channel via relay_write(), given the channel id.

```
relay_write(channel_id, data,
            count,...);
```

In user space, a program can open the relayfs file and wait in a read() loop waiting for data.

```
fd = open("/mnt/relay/file", ...);

while(1) {
  n = read(fd, buf, sizeof(buf));
  if(n <= 0) {
     close(fd);
     break;
  }
}
```

Alternatively, the file can be mmap()'ed and directly accessed via a pointer to the mapped buffer when data is ready.

```
fd = open("/mnt/relay/file", ...);
char *map = mmap(..., fd, ...);

void on_ready(count) {
   write(diskfile, map, count);
}
```

There are five callbacks that can optionally be registered by the kernel client when a channel is opened. These are used to notify the client when significant events occur (buffer start, buffer end, event deliver, buffers full, buffer resize) and are described below.

4.2 Channel and data management schemes

The channel data for a given channel is internally managed via one of two schemes defined at channel creation. They are *lockless* or *locking*. The lockless scheme is described in Section 2. The locking scheme is a simple two-buffer ping-pong scheme. One of the buffers is the current write buffer, into which events are written, and the other is the current read buffer, from which events are read (by, for instance, a user daemon). When the current write buffer is filled, it becomes the current read buffer and the current read buffer becomes the new write buffer. The key feature of the locking scheme is that the channel is locked (the exact semantics are described below) while space is allocated for the event and for the duration of the write.

The reason two schemes exist is that while it would be ideal to have all channels managed by the lockless scheme, the availability of the lockless scheme depends on the availability of a cmpxchg instruction or a generic equivalent, which does not exist on all Linux platforms. Thus, the locking scheme is available as a fallback scheme for those platforms that cannot support the lockless scheme.

relayfs channels are implemented as circular buffers divided into a number of sub-buffers. If a scheme has not been explicitly specified, the channel creation code will choose lockless. The number and size of these sub-buffers

are specified at channel creation (or with certain restrictions can be dynamically sized afterward). For efficiency reasons, in the lockless code, both of these are a power of 2. Having each buffer size be a power of 2 allows a cheap logical and comparison to determine if the end of a buffer has been reached. Having the number of buffers be a power of 2 allows a cheaper operation to be performed to ensure the index remains within the memory allocated for the buffers. Both of these operations occur on the fast path of every event log and thus are critical. The index value could be interpreted as a straight index into a single circular buffer, but for other reasons multiple buffers are preferable. For example, breaking up the buffers into multiple sub-buffers rather than a single large buffer allows flexibility in buffer processing, allows more timely delivery of events to user space, and minimizes the impact of potential data corruption.

One aspect of using the lockless scheme is the use of variable-length events. There are tradeoffs between using fixed-length or variable-length events. Fixed-size events allow for simpler logging and reading out as the consumer of events always knows the starting point of an event. This allows validity bits to be used, and allows invalid events to be skipped. Fixed-length events allow easy random access to the data stream, aiding reading and displaying large files. The disadvantages of fixed-length events are that they waste space, they take longer to write (to disk or network) because extra data needs to be written for short events, and they make it complicated to log data that is larger than the fixed size. The lockless scheme obtains the benefits of each by ensuring that events never cross medium-scale alignment boundaries. We insert filler events as necessary to align the event stream. Data analysis tools can skip to any of the alignment points in a large event buffer and can begin interpreting events from that point. This

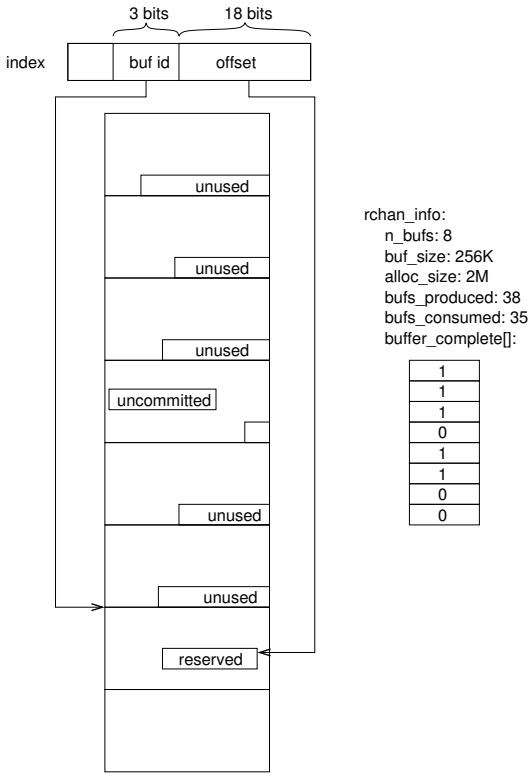

Figure 4: Buffer layout.

technique provides the advantages of variable-length events and still allows fast access to all parts of a large file. A filler event is a header with a length equal to the remainder of the current buffer; no data need be logged. For the clients we have studied this alignment wastes little space. Other applications that have few large events and whose events frequently end on buffer boundaries will exhibit similar behavior. This technique does not provide completely random access, but is a close enough approximation that it allows post-processing tools to make it appear to the end user that the stream is completely random access.

Figure 4 shows a 2M buffer divided into 8 sub-buffers of 256K (in this case, 256K would be the alignment boundary described above). This splits the index logically into a 3-bit portion containing the buffer id and 18 bits containing the current offset within the buffer. The rchan_info struct, retrieved via relay_info(),

contains a snapshot of the current state of the channel. The data in Figure 4 shows that 35 sub-buffers have been consumed and 38 have been produced. The buffer_complete array shows the completeness state of all of the sub-buffers not including the current one. As Figure 4 shows, sub-buffer 3 is not yet complete as there is still a pending write. This is also the reason buffers_consumed has not caught up with buffers_produced. The user-space client suspends processing the sub-buffers when it encounters the incomplete buffer until such time as the buffer becomes complete, or is "lapped", in which case the buffer contents would be lost.

4.3 Buffer start/end callbacks

Two of the five channel callbacks registered by the kernel client exist to allow the client to be notified when sub-buffer boundaries are crossed, i.e. when buffer switches occur. These callbacks are invoked in the slow path of event logging, which is executed when an event write would overflow the current sub-buffer. The buffer_end() callback is invoked to allow the kernel subsystem the opportunity to perform end-of-buffer processing on the just-filled buffer. To allow space for data even when a buffer is exactly filled, there needs to be space reserved at the end of the buffer into which the client can write an unused count. This is the purpose of the end_reserve parameter to relay_open. It specifies the number of bytes at the end of each sub-buffer that should be left alone by the logging algorithm and left available for the client to write data. Similarly, the buffer_start() callback is invoked to give the client an opportunity to write header data at the beginning of a sub-buffer. The start_reserve parameter to relay_open() allows the client to specify a value for this purpose. One of the parameters of the buffer_start() callback is the buffer id, which has a value of zero for the very first buffer in the channel. Clients can check for this value and optionally write channel header data in the position reserved for it by the channel_start_reserve parameter of relay_open().

4.4 Channel attributes

The characteristics of a given channel are derived from a set of channel attributes specified when the channel is opened. These are:

- **scheme**: lockless or locking. Indicates which of the logging algorithms to use. If 'any' is specified, relayfs attempts to use the lockless scheme; if unavailable it reverts to the locking scheme.
- **SMP usage**: global or SMP. Applies only if the locking scheme is being used. Global indicates that the channel is being shared across multiple processors. In this case lock acquisition involves a spinlock_irqsave/restore. The SMP option indicates per-processor channels and thus lock acquisition can use the cheaper local_irqsave/restore.
- **delivery mode**: bulk or packet. If bulk delivery mode is specified, the kernel client is notified via the delivery callback when complete buffers become available. If packet delivery mode is specified, the delivery callback is invoked after each write. Bulk delivery is suited for clients logging large volumes of data. They would be noticeably affected by callbacks after each event, but still may be interested in the beginnings and ends of buffers. Less demanding clients may require notification for each event logged. Clients specify callbacks to, for example, signal a user-level program indicating data is ready. This is a typical mode of operation for a bulk data client and is somewhat less typical for packet client as its user-space pro-

gram is often polling for the next piece of data. Either type of client may be interested in filtering the data and/or dispatching events to other channels or subsystems. Typically, packet clients would be more interested in data filtering due to the more manageable volume of events.

- **timestamping**: TSC or gettimeofday() deltas. This attribute allows the client to choose the granularity and cost of timestamping. Timestamping is optional and timestamps are not written to a channel unless explicitly requested. Events are timestamped by relay_write() timestamp events using either the efficient TSC (or equivalent cheap clock on other architectures), or a slower but globally consistent gettimeofday() time delta method. gettimeofday() is also a fallback in the cases where the TSC, or an equivalent clock counter, is unavailable on a given architecture. In brief, part of the task of writing an event involves obtaining the current TSC, or the current gettimeofday() value. If the gettimeofday() option is chosen each timestamp is the difference between the current time and the time when the buffer was started. The value logged is either the TSC value or this difference and is written at the offset within the event slot specified by the td_offset parameter of relay_write() (if the parameter value is negative, the timestamp is not written). The start and end buffer gettimeofday() and TSC (if applicable) values are available as parameters to the buffer_start() and buffer_end() kernel client callbacks. If a client is interested in timestamping, it can write these values into the reserved space for later inter-buffer correlation.

4.5 Channel overflow handling

As with any buffering scheme, data may be written into a channel faster than the channel's clients can read it out. relayfs channel clients have three options for dealing with an overflow:

- do nothing, writers overwrite old data (flight recorder mode)
- suspend writing into the channel, causing loss of new events
- resize the channel, making more space for writers

The first two options are controlled by the return value of the buffers_full() callback. This callback provides the client with the option of what action to take if the consumer has not kept pace with the logging. A value of 0 directs relayfs to continue logging events, overwriting the oldest data. A value of 1 directs relayfs to discard subsequent events until the overflow situation has been resolved. In this case, an events_lost count is kept and is available via relay_info(). Once the consumer (usually the user-space daemon reading from the channel) has caught up, the relayfs client can call relay_resume() to allow the channel to continue logging events. To implement this, the client needs to keep the channel informed of how many buffers it has read. It increments the count of buffers consumed by calling relay_buffers_consumed(n_buffers). This value is compared with the channel's count of buffers produced (tracked on the buffer-switch slow path) to determine whether a buffers-full condition exists. If the difference is greater than or equal to the number of sub-buffers in the channel, the buffers are considered full and the callback is invoked.

The third option, resizing the channel, is available to clients that have specified non-zero values for the resize_min and resize_max parameters to relay_open() when the channel was created. If (during the buffer-switch slow path)

relayfs detects that the channel is almost full (if 3/4 of the sub-buffers remain unread, by default), the needs_resize() callback is invoked with parameter values containing a suggested new sub-buffer size and/or sub-buffer count, which can be used to expand the buffer space. The client can use these values (or ignore them and supply its own) to allocate a new buffer for the channel via the API function relay_resize_channel(). This function can block, so it should not be called with spinlocks held. If called from user context, it directly allocates the new buffer, which is available upon return. If called from within interrupt context, the allocation is put onto a work queue, and the client is notified upon completion via another call to the needs_resize() callback. Once the new buffer is allocated, the client can call relay_replace_buffer() to replace the channel's buffer. This function can be called from any context. Clients call it when they can guarantee the replacement does not interfere with other channel activity, such as outstanding writes. Reducing the buffer size follows a similar path. When relayfs detects that the "almost-full" condition has not existed for a period of time (1 minute by default), the needs_resize() callback is invoked with the new suggested (smaller) sub-buffer values. Buffer reduction is handled by the client in a similar manner to buffer expansion. Clients can choose to ignore the details of buffer resizing. To do so, they specify a non-zero value for the autoresize parameter to relay_open() causing buffer re-allocation requests to be placed onto a work queue. The resizing strategy reflects empirical observations that channel traffic tends to be bursty in nature with sudden activity creating immediate short-term need for increased buffer capacity, which after a short period is no longer needed.

Config.	avg time per run (in seconds)	difference
1	756.1	
2	766.7	+1.40%
3	771.3	+2.01%

Figure 5: LTT on relayfs test results.

5 Testing

To test the efficiency of relayfs, we ran LTT, a very demanding client of relayfs, while performing 10 kernel compiles under each of the following conditions:

1. not tracing anything (baseline)
2. tracing everything, daemon not writing to disk
3. tracing everything, daemon writing to disk

Testing was performed on a 4-way 700MHz Pentium III system. The tests generated large amounts of data for relayfs to process. Approximately 200 million events comprising about 2 gigabytes were generated during each 10-compile run. As can be seen from the results in Figure 5, the overhead of relayfs was at most 1.4 percent (a portion of this overhead is in fact due to tracing code), demonstrating the efficiency of relayfs.

6 Example client subsystems

In this section we examine some practical uses of relayfs. In particular, we discuss how relayfs can be used as an engine for printk and LTT, how it can be used for driver debugging, and how it could become a replacement for rvmalloc and rvfree.

6.1 printk

We have developed a version of printk that replaces the static printk buffer with a dynamically resizeable relayfs channel. This solves the lost printk problem by providing reliable printk logging services. A dynamically resizeable channel prevents lost printk messages in normal usage, but it can also prevent the loss of init-time printk messages. At init-time (before free_initmem()), the contents of the static printk buffer are copied into the relayfs channel created for printk. The static printk buffer used at init-time is marked as __initdata and is subsequently discarded. A benefit of this scheme is that the printk kernel buffer can be made relatively large. In fact, it can be large enough so that it does not overflow even when copious boot messages are printed on a large system. The relayfs version of printk modifies the syslog(3) system call to read from the printk channel instead of from the static kernel buffer. Since /proc/kmsg also uses syslog(3) to retrieve data from the kernel buffer, user-space programs that read from the printk buffer do not need to be modified to use the relayfs version of printk. The relayfs version of printk adds commands to syslog(3) allowing user-mode clients to manually resize the printk channel; these of course must be coded for. See the relayfs web page[1] for code and status.

6.2 Linux Trace Toolkit

LTT creates a separate bulk-delivery channel for each processor. The LTT kernel client replicates the inner workings of relay_write by using the relayfs low-level API described briefly in Documentation/filesystems/relayfs.txt. More detail can be found by examining trace() in kernel/trace.c and the relay_write() implementation. It uses the low-level API to obtain maximum performance from relayfs, because system tracing is one of the most demanding clients. We would expect most subsystems to function acceptably using the described APIs, but the low-level API is available for those requiring maximum performance. The low-level API allows a client to write directly into the reserved slot in the channel, rather than passing the address of a buffer to relay_write for it to copy. By doing so, LTT avoids the overhead of forcing trace() to collate its fields into a separate buffer before passing it to relay_write, and avoids putting the event data on the stack (in reality this would not be possible because some events can be 8K in length). This is more convenient than having trace() manage a staging buffer between potentially multiple writers.

```
rchan = rchan_get(channel_handle);
if (rchan == NULL)
   return -ENODEV;

/* this is a nop for lockless */
relay_lock_channel(rchan, flags);

reserved =
   relay_reserve(rchan,
        data_size, &time_stamp,
        &time_delta, &reserve_code,
        &interrupting);

if (reserved == NULL)
   goto check_buffer_switch_signal;

bytes_written +=
   relay_write_direct(reserved,
       &event_id, sizeof(event_id));
bytes_written +=
   relay_write_direct(reserved,
         &data, sizeof(data));

relay_commit(rchan, reserved,
      bytes_written, reserve_code,
      interrupting);

relay_unlock_channel(rchan, flags);
rchan_put(rchan);
```

In the above code, the first few tasks are bookkeeping tasks. These involve getting the channel structure backing the channel handle, and locking the channel (if applicable—for the

lockless scheme, this is effectively a no-op). We then reserve a slot, and using the special relay_write_direct() low-level method (essentially a memcpy), write fields directly into the channel buffer. When we are done writing the fields, we indicate to the channel that the data is ready by 'committing' the slot, and then finish up with a couple more bookkeeping tasks.

See the relayfs web page[1] for code and status.

6.3 Driver debugging

It is straightforward to create and use a relayfs channel for driver tracing/debugging. Open a channel with the desired_attributes:

```
channel = relay_open("channel", ...);
```

and write to it from within your driver:

```
relay_write(channel, ...)
```

Then write a simple user-space program that loops around read():

```
fd = open("/mnt/relay/channel", ...);

for(;;)
   read(fd, ...);
```

6.4 rvmalloc/rvfree replacement

At last count there were 9 drivers with separate implementations of rvmalloc/rvfree. There have been discussions in the past of exporting a single 'blessed' instance of rvmalloc/rvfree, but so far that has not happened. Since relayfs channels are based on rvmalloc/rvfree, relayfs provides the equivalent of a public rvmalloc/rvfree by providing clients a means to create rvmalloc'ed buffers using degenerate values to relay_open() e.g. a large frame buffer could be allocated via relay_open with most parameters 0/NULL:

```
int channel_id =
  relay_open(
    "framebuffer",
    bufsize, /* size of sub-buffer */
    1, /* one sub-buffer only */
    0, /* no flags in this case */,
    NULL, /* no callbacks */
    0, /* no reserve */
    0, /* no reserve */
    0, /* no reserve */
    0, /* no resize_min */
    0, /* no resize_max */
    0, /* don't autoresize */
    NULL); /* no special file ops */
```

The above opens a relay channel with a filename of /mnt/relay/framebuffer, assuming relayfs is mounted at /mnt/relay.

Information about the channel can be retrieved using relay_info().

```
relay_info(channel_id, &rchan_info);
```

The data contained in rchan_info includes the virtual address of the buffer allocated for the channel via rvmalloc. This can be directly used to write into the buffer memory from the kernel side.

The file created for the channel can then subsequently be mmap()'ed into user space:

```
int framebuf_file =
  open("/mnt/relay/framebuffer",
       ...);
char * framebuf = mmap(NULL, bufsize,
  PROT_READ|PROT_WRITE, MAP_PRIVATE,
  framebuf_file);
```

Note that if a channel is mmap()'ed, it cannot be resized, but a client wishing to resize the channel can unmap the file, resize it, then remap it.

Finally, closing the channel frees the buffer via rvfree():

```
relay_close(channel_id);
```

7 Conclusion

We have presented relayfs, an efficient and unified mechanism for transferring data from kernel to user space. relayfs addresses the need

to have a reliable mechanism that can be used across various kernel subsystems, without having to replicate and separately maintain the functionality for each subsystem. The lockless and per-processor techniques allow efficient logging on multiprocessor systems, and the channels provide flexibility to the clients. Test results show that relayfs performs well even under demanding workloads. We presented several example kernel subsystems that use relayfs including printk, LTT, and driver debugging. As relayfs becomes more widely accepted, other kernel subsystems will be able to use it, reducing code replication and improving reliability of the kernel subsystems.

References

[1] relayfs home page, `http://www.opersys.com/relayfs`.

[2] Robert W. Wisniewski and Luis F. Stevens. A model and tools for supporting parallel real-time applications in unix environments. In *Proceedings of The 12th IEEE Real-Time Technology and Applications Symposium*, pages 126–133, Chicago Illinois, May 15-17 1995.