Dynamic Debug

Jason Baron
Red Hat
jbaron@redhat.com

Abstract

The kernel is sprinkled with debug statements that are only available by individually re-compiling the various subsystems of the kernel. In addition, each subsystem has its own rules and methods for expressing these debug statements - `dprintk()`, `DEBUGP()`, `pr_debug()`, etc. *Dynamic debug*, introduced in kernel 2.6.28, organizes these debug statements and makes them available at run-time. Statements can be enabled on an individual basis, or via higher level organizations such as per-module. *Dynamic debug* can be thought of as a verbose mode for the kernel. We explore the design, usage, and performance impact of this new feature. We also highlight issues that have been debugged with this methodology and future work.

1 Introduction

The kernel contains many debug statements. In kernel 2.6.29 there are 3058 `pr_debug()` calls, 3158 `dev_dbg()` calls, 5618 `dprintk()` calls, 206 `DEBUGP()` calls, and countless additional debugging statements. Some of these statements are activated by defining `DEBUG` in the corresponding .c files, while others are activated by enabling subsystem specific configuration parameters. Some subsystems have developed sophisticated debug statement frameworks, allowing fine grain control via bit flags and debug levels. Thus, when a problem occurs that requires additional debugging information, a kernel would typically need to be re-compiled in order to obtain this additional debugging information.

According to Linux Device Drivers, "The most common debugging technique is monitoring, which in applications programming is done by calling *printf* at suitable points. When you are debugging kernel code, you can accomplish the same goal with *printk*"[4]. Many, if not most user space programs have a verbose mode. So, why doesn't the kernel have such a mode?

Really, the kernel does already have such a mode. We currently label *printk* messages, from most severe, `KERN_EMERG`, to least severe, `KERN_DEBUG`. Thus, a *printk* of level `KERN_DEBUG` already exists. Why don't we simply convert the 10,000 or so debug statements previously mentioned, to *printk*s of level `KERN_DEBUG` and be done with it?

First, enabling all those debug statements would clutter up the logs. These debug statements are often found in high frequency code paths, and thus would make noise to signal ratio of the logs rather high. Second, even when a *printk* message doesn't make its way to the console and/or the system logs, we still 'render' every *printk* in the kernel, which is an expensive operation. In 'rendering' a *printk*, we format the messages into a buffer with locks held and irqs disabled. In fact, we saw an 86% tbench performance degradation, when enabling all the debug statements, but not logging any of the messages to the system logs or console. Thus, there is an enormous cost associated with simply turning these messages into *printk* statements of level `KERN_DEBUG`.

Dynamic debug addresses these two core concerns. The verbosity is tackled using a unique language, which allows expression for fine grain control of each debug statement, while also permitting coarser control, using for example, per-module enabling. This control language was originally developed by Greg Banks at SGI and was incorporated into *dynamic debug*[1]. The runtime performance concerns are addressed while making use of a bloom filter[2].

There has been a lot of work recently in the general area of kernel tracing. Ftrace[6], LTTng[5], and Systemtap[3] can all be used to trace the kernel. We view *dynamic debug* as a complementary technology that can be used in conjunction with these other tools. There may also be ways for *dynamic debug* to leverage and/or combine with some of these tools, which we will explore.

In the next section, we will look at the implementation of *dynamic debug*. We will then look at its usage and some examples. Next, we will analyze the impact of *dynamic debug* in terms of its size and performance. We then introduce a subtle variation on *dynamic debug* which can handle more complex debugging statements. Finally, we conclude by commenting on the organization of kernel debug statements, and suggesting areas for future work.

2 Implementation

As mentioned, the two central goals of the implementation are controlling what ends up in the system logs or filtering, and efficiency or minimal run-time cost. The initial implementation focused on two functions - `pr_debug` and `dev_dbg`. These functions are defined centrally, thus, 'overwriting' their definition was contained to two source files.

We then associate meta data with each debug statement. This meta data records the containing .c file, line number, containing module, and associated format string. This information enables the user to understand and control which statements are enabled. The size of the meta data is then proportional to information stored for each debug statement and the number of debug statements. We explore the meta data associated with each debug statement, and the run-time control, in the following two sections entitled 'data structures', and 'bloom filters' respectively.

2.1 Data Structures

We hook into the `pr_debug` macro with some simple macro magic. Prior, to the introduction of *dynamic debug* we had:

```
#ifdef DEBUG
#define pr_debug(fmt, arg...) \
    printk(KERN_DEBUG fmt, ##arg)
#else
#define pr_debug(fmt, arg...) \
    ({ if (0) printk(KERN_DEBUG fmt, ##arg); 0; })
#endif
```

We re-define this construct as follows:

```
#if defined(DEBUG)
#define pr_debug(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#elif defined(CONFIG_DYNAMIC_DEBUG)
#define pr_debug(fmt, ...) do { \
    dynamic_pr_debug(fmt, ##__VA_ARGS__); \
    } while (0)
#else
#define pr_debug(fmt, ...) \
    ({ if (0) printk(KERN_DEBUG pr_fmt(fmt),
                            ##__VA_ARGS__); 0; })
#endif
```

Thus, if `DEBUG` is defined, we continue to get an inlined *printk* statement, while no code is generated when `DEBUG` is not defined. The new case that we have introduced results from defining the new configuration parameter `CONFIG_DYNAMIC_DEBUG`. When this configuration parameter is defined, we hook into the new dynamic debug code. Note, that `DEBUG` takes precedence over `CONFIG_DYNAMIC_DEBUG`. In this way, code that defines `DEBUG` continues to work as was previously expected. The `dev_dbg()` hook is implemented in a similar manner.

The `dynamic_pr_debug()` macro expands to store debug statement data in the `struct _ddebug` data structure, which follows.

```
struct _ddebug {
    /*
     * These fields are used to drive the user
     * interface for selecting and displaying
     * debug callsites.
     */
    const char *modname;
    const char *function;
    const char *filename;
    const char *format;
    char primary_hash;
    char secondary_hash;
    unsigned int lineno:24;
    /*
     * The flags field controls the behaviour
     * at the callsite. The bits here are
     * changed dynamically when the user
     * writes commands to
     * <debugfs>/dynamic_debug/control
     */
#define _DPRINTK_FLAGS_PRINT   (1<<0)
#define _DPRINTK_FLAGS_DEFAULT 0
    unsigned int flags:8;
} __attribute__((aligned(8)));
```

The modname, function, filename and lineno fields are populated using the C code definitions `KBUILD_MODNAME`, `__func__`, `__FILE__`, and `__LINE__`, respectively. The format field is filled using the supplied in format string. The `primary_hash` and `secondary_hash` fields are explained in subsequent

bloom filter section. Finally, the flags field is used as a per-call site control variable.

Thus, the `struct _ddebug` data structure is 40 bytes in size. With a combined 6,000 call sites for `pr_debug` and `dev_dbg`, that amounts to 23k(6000*40) in addition to the size of associated strings. We share results of the kernel image size increases in the results section.

When the kernel boots or as new modules are inserted and removed we create a new `struct ddebug_table` structure for each logically module, which follows.

```
struct ddebug_table {
        struct list_head link;
        char *mod_name;
        unsigned int num_ddebugs;
        unsigned int num_enabled;
        struct _ddebug *ddebugs;
};
```

The pointer to the `_ddebugs` structures which are associated with a particular module are then assigned to the `_ddebug` field of the associated `ddebug_table` data structure. The `ddebug_table` structures are then linked together in a linked list. Thus, using this list we can easily look up entries and display them. A linked list works fine since looking up these entries when setting values, is not a hot path.

2.2 Bloom Filter

We associate two hash values with each instrumented debug statement. Both hashes are in the range 0-64. These are stored in the `primary_hash` and `secondary_hash` fields of the `_ddebug` structure. We use the djb2 and the r5 hash algorithms. The input to the hash function is the Linux source code directory and the module name. Thus, modules of the same name that are located in different source directories likely have different hash values. Thus, hash bits 'n' and 'm' are associated with each debug statement.

Two global variables of type `long long` are also introduced. They are `dynamic_debug_enabled` and `dynamic_debug_enabled2`. When we wish to enable a debug statement, we set bit 'n' and 'm' in the global variables `dynamic_debug_enabled` and `dynamic_debug_enabled2` respectively. Thus, each debug statement is conditioned on having bits 'n' and 'm' set in the global variables `dynamic_debug_enabled` and `dynamic_debug_enabled2` respectively. This is a variation on a bloom filter[2]. There can be false positives, and thus we use the `flags` field of the `struct _ddebug` field as a third, and definitive check before calling into the associated *printk* statement.

Thus, the pseudo-code for the above described case is:

```
bit1 = hash1(kernel path + module name)
bit2 = hash2(kernel path + module name)
if (bit1 & dynamic_debug_enabled &&
    bit2 & dynamic_debug_enabled &&
    _ddebug field is set)
    (do the printk)
```

While there may be more complex implementation which involve live code patching, such as the immediate variable work, we find this implementation to be a good trade off between complexity and speed. Notice we have no locking or synchronization of any kind. Thus in the off case we expect to execute only a couple of additional instructions, and because we are relying only two global variables, we expect the code to exhibit good caching property. We can also further tweak the properties of the bloom filter by creating additional levels of hashing. Notice we could also fold the hashing into one global variable as well.

Next we turn to hash collisions. In building the v2.6.29 kernel with these options, when all module are disabled, both global variables are set to 0, and thus we have no false positives. When we turn all debugging on, we set both of the globals to all 1s, and thus we have no false positives in this case either. When one module is enabled, we also have no false positives. This is not necessarily true, but is true for v2.6.29 kernel that was tested. This kernel produced eighty separate modules, and thus eighty unique sets hashes. We did not compute three way collisions.

Thus, when no, one, or all modules are enabled we have no false positives. Even in the case where we do have a false positive, we do not call through to a function we simply check the unique variable associated with the corresponding debug statement.

3 Usage and Examples

3.1 Controlling Dynamic Debug Behavior

The behaviour of `pr_debug()` and `dev_debug()` are controlled by writing to a control file in the debugfs filesystem. Thus, you must first mount the debugfs filesystem, in order to make use of this feature. Subsequently, we refer to the control file as: `<debugfs>/dynamic_debug/control`. For example, if you want to enable printing from source file svcscok.c, line 1603 you simply do:

```
# echo 'file svcsock.c line 1603 +p' >
<debugfs>/dynamic_debug/control
```

If you make a mistake with the syntax, the write will fail thus:

```
# echo 'file svcsock.c blah 1 +p' >
<debugfs>/dynamic_debug/control
-bash: echo: write error: Invalid argument
```

3.2 Viewing Dynamic Debug Behavior

Viewing the current configuration is done with a simple read. See Figure 1

3.3 Command Language Reference

At the lexical level, a command comprises a sequence of words separated by whitespace characters. Note that newlines are treated as word separators and do not end a command or allow multiple commands to be done together. So these are all equivalent:

```
# echo -c 'file aio.c line 1603 +p' >
<debugfs>/dynamic_debug/control
# echo -c ' file aio.c     line  1603 +p  ' >
<debugfs>/dynamic_debug/control
# echo -c 'file aio.c\nline 1603 +p' >
<debugfs>/dynamic_debug/control
# echo -n 'file aio.c line 1603 +p' >
<debugfs>/dynamic_debug/control
```

Commands are bounded by a write() system call. If you want to do multiple commands you need to do a separate "echo" for each:

```
# echo 'file aio.c line 1603 +p' >
       <debugfs>/dynamic_debug/control;\
> echo 'file svcsock.c line 1563 +p' >
       <debugfs>/dynamic_debug/control
```

or even:

```
# (
> echo 'file svcsock.c line 1603 +p' ;\
> echo 'file svcsock.c line 1563 +p' ;\
> ) > <debugfs>/dynamic_debug/control
```

At the syntactical level, a command comprises a sequence of match specifications, followed by a flags change specification.

```
command ::= match-spec* flags-spec
```

The match-spec's are used to choose a subset of the known debug statements to which to apply the flags-spec. Think of them as a query with implicit ANDs between each pair. Note that an empty list of match-specs is possible, but is not very useful because it will not match any debug statement call sites.

A match specification comprises a keyword, which controls the attribute of the call site to be compared, and a value to compare against. Possible keywords are:

- *match-spec* ::= 'func' string | 'file' string | 'module' string | 'format' string | 'line' line-range
- *line-range* ::= lineno | '-'lineno | lineno'-' | lineno'-'lineno // Note: line-range cannot contain space, e.g. // "1-30" is valid range but "1 - 30" is not.
- *lineno* ::= unsigned-int

The meanings of each keyword are:

- *func.* The given string is compared against the function name of each callsite. Example:

  func `svc_tcp_accept`

- *file.* The given string is compared against either the full pathname or the basename of the source file of each callsite. Examples:

  file svcsock.c
  file sched.c

You can view the currently configured behaviour of all the debug statements via:

```
# cat <debugfs>/dynamic_debug/control
# filename:lineno [module]function flags format
fs/sysfs/file.c:147 [file]sysfs_read_file - "%s: count = %zd, ppos = %lld, buf = %s\012"
fs/sysfs/dir.c:788 [dir]__sysfs_remove_dir - "sysfs %s: removing dir\012"
fs/sysfs/bin.c:110 [bin]read - "offs = %lld, *off = %lld, count = %d\012"
fs/debugfs/inode.c:217 [debugfs]debugfs_create_file - "debugfs: creating file '%s'\012"
```

You can also apply standard Unix text manipulation filters to this data, e.g.:

```
# grep -i aio <debugfs>/dynamic_debug/control  | wc -l
10
# grep -i security <debugfs>/dynamic_debug/control | wc -l
163
```

Note in particular that the third column shows the enabled behaviour flags for each debug statement call site. The default value, no extra behaviour enabled, is "-". So you can view all the debug statement call sites with any non-default flags:

```
[root@mets dynamic_debug]# awk '$3 != "-"' control
# filename:lineno [module]function flags format
fs/aio.c:77 [aio]aio_setup p "aio_setup: sizeof(struct page) = %d\012"
fs/aio.c:222 [aio]__put_ioctx p "__put_ioctx: freeing %p\012"
fs/aio.c:1788 [aio]sys_io_cancel p "calling cancel\012"
fs/aio.c:1698 [aio]sys_io_submit p "EINVAL: io_submit: invalid context id\012"
fs/aio.c:1604 [aio]io_submit_one p "EINVAL: io_submit: overflow check\012"
fs/aio.c:1594 [aio]io_submit_one p "EINVAL: io_submit: reserve field set\012"
fs/aio.c:1335 [aio]sys_io_destroy p "EINVAL: io_destroy: invalid context id\012"
fs/aio.c:1303 [aio]sys_io_setup p "EINVAL: io_setup: ctx %lu nr_events %u\012"
fs/aio.c:248 [aio]ioctx_alloc p "ENOMEM: nr_events too high\012"
fs/aio.c:1022 [aio]aio_complete p "added to ring %p at [%lu]\012"
```

Figure 1: Viewing current dynamic debug status

- *module.* The given string is compared against the module name of each callsite. The module name is the string as seen in "lsmod", i.e. without the directory or the .ko suffix and with '-' changed to '_'. Examples:

  module sunrpc
  module nfsd

- *format.* The given string is searched for in the dynamic debug format string. Note that the string does not need to match the entire format, only some part. Whitespace and other special characters can be escaped using C octal character escape notation, e.g. the space character is
  040. Alternatively, the string can be enclosed in double quote. Examples:

  format svcrdma: // many of the NFS/RDMA server dprintks
  format readahead // some dprintks in the readahead cache
  format nfsd:
  040SETATTR // one way to match a format with whitespace
  format "nfsd: SETATTR" // a neater way to match a format with whitespace
  format 'nfsd: SETATTR' // yet another way to match a format with whitespace

- *line.* The given line number or range of line numbers is compared against the line number of each debug statement call site. A single line number matches the call site line number exactly. A range of line numbers matches any call site between the first and last line number inclusive. An empty first number means the first line in the file, an empty line number means the last number in the file. Examples:

  line 1603 // exactly line 1603
  line 1600-1605 // the six lines from line 1600 to line 1605
  line -1605 // the 1605 lines from line 1 to line 1605
  line 1600- // all lines from line 1600 to the end of the file

The flags specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

- - remove the given flags
- + add the given flags
- = set the flags to the given flags

The flags are:

- *p* Causes a `printk()` message to be emitted to dmesg

Note the regexp `^[-+=][p]+$` matches a flags specification. Note also that there is no convenient syntax to remove all the flags at once, you need to use "-p".

3.4 Examples

In the figure 2 below we show two examples, to give a flavor of the output. The first example shows enabling all messages. The second example shows enabling kobject module output while the cifs module is loaded.

4 Size and Performance

The kernel used for testing was v2.6.28 compiled for x86_64. An Intel quad core machine running at 1.6 GHz with 2GB of RAM was used for all tests.

test case	tbench throughput
`CONFIG_DYNAMIC_DEBUG` disabled	773.054 MB/sec
`CONFIG_DYNAMIC_DEBUG` enabled	773.913 MB/sec
`CONFIG_DYNAMIC_DEBUG` enabled and all debug statements enabled	79.664 MB/sec

Table 1: performance results

Thus, the run-time cost of having `CONFIG_DYNAMIC_DEBUG` enabled, but none of the debug statements printing, is negligible. However, when we enable all of the debugging statements, the system throughput drops quite dramatically. Thus, simply converting all of these high frequency debug statements to *printk* at `KERN_DEBUG` level is not viable. This also suggests that alternate methods for 'rendering' the format strings might be worth investigating. We discuss this further in the future work section.

In terms of kernel code size growth, the kernel increased 2% when enabling `CONFIG_DYNAMIC_DEBUG`.

5 More Complex Debugging Statements

Thus far, we've looked at debug statements that are binary - they are either enabled or disabled. However, several kernel subsystems have developed more complex debugging facilities based on 'levels' or 'flags'. The 'levels' model is employed by the CPU frequency subsystem, where messages above a configurable level *n* are emitted. Currently, the level is set via module parameters. Thus, to change the level, one would need to unload and re-load the CPU frequency modules. The NFS filesystem uses a 'flags' style of debugging, where each 'flag' or bit in an integer toggles on or off a set of debugging statements.

If we extend the *dynamic debug* construct somewhat, we can accommodate both the 'level' and 'flags' debugging style. For 'flags' we can create the following general function (pseudo-code):

```
#define debug_enabled_flag
        (flag_bit, subsys_set_bits)
if (normal dynamic debug checks) {
    if (flag_bit & subsys_set_bits) {
        return 1;
   }
}
return 0;
```

The 'flag_bit' refers to the bit associated with this particular debug statement. The 'subsys_set_bits' refers to the global integer which is associated with this subsystem. The 'normal dynamic debug checks' have the associated hash bits set corresponding module if any of the flag bits are set. We can then design a subsystem specific macro for any subsystem as follows:

```
#define subsystem_foo_level_debug
        (flag_bit, fmt, va_args)
if (debug_enabled_flag(flag_bit,
                       subsys_set_bits)) {
    print(fmt, va_args);
}
```

Subsystem can thus pass in any subsystem specific print information. Also, by designing this interface in this manner, subsystems could easily perform any additional checks that they wish where the 'print' statement is located. Thus, we can imagine *dynamic debug* being used for more than just printing information. We've

```
# cut -f2 -d"[" control | cut -f1 -d"]" | xargs -i echo 'module {} +p' > control

Apr 14 15:17:49 mets kernel: [ 3883.017536] nf_conntrack:tcp_in_window: START
Apr 14 15:17:49 mets kernel: [ 3883.017539] nf_conntrack:tcp_in_window: <7>nf_conntrack:seq=376950469
ack=3577053373 sack=3577053373 win=1803 end=376950469
Apr 14 15:17:49 mets kernel: [ 3883.017549] nf_conntrack:tcp_in_window: sender end=376950469
maxend=376964293 maxwin=115392 scale=6 receiver end=3577053373 maxend=3577168717 maxwin=13824 scale=7
Apr 14 15:17:49 mets kernel: [ 3883.017555] nf_conntrack:tcp_in_window: <7>nf_conntrack:seq=376950469
ack=3577053373 sack =3577053373 win=1803 end=376950469
Apr 14 15:17:49 mets kernel: [ 3883.017565] nf_conntrack:tcp_in_window: sender end=376950469
maxend=376964293 maxwin=115392 scale=6 receiver end=3577053373 maxend=3577168717 maxwin=13824 scale=7
Apr 14 15:17:49 mets kernel: [ 3883.017571] nf_conntrack:tcp_in_window: I=1 II=1 III=1 IV=1
Apr 14 15:17:49 mets kernel: [ 3883.017577] nf_conntrack:tcp_in_window: res=1 sender end=376950469
maxend=376964293 maxwin=115392 receiver end=3577053373 maxend=3577168765 maxwin=13824
Apr 14 15:17:49 mets kernel: [ 3883.017582] nf_conntrack:tcp_conntracks: <7>nf_conntrack:syn=0 ack=1
fin=0 rst=0 old=3 new=3
Apr 14 15:17:50 mets kernel: [ 3883.110062] file:sysfs_read_file: count = 4096, ppos = 0,
buf = 00000000,0000000f
Apr 14 15:17:50 mets kernel: [ 3883.110116] file:sysfs_read_file: count = 4096, ppos = 0,
buf = 00000000,00000001
Apr 14 15:17:50 mets kernel: [ 3883.110164] file:sysfs_read_file: count = 4096, ppos = 0,
buf = 00000000,00000005
Apr 14 15:17:50 mets kernel: [ 3883.110204] file:sysfs_read_file: count = 1, ppos = 0,
buf = 1

# echo 'module kobject +p' > /mnt/debugfs/dynamic_debug/control
# /sbin/modprobe cifs

Apr 14 15:54:45 mets kernel: [  184.968002] kobject:kobject: 'cifs' (ffffffffa007e0f0):
kobject_add_internal: parent: 'module', set: 'module'
Apr 14 15:54:45 mets kernel: [  184.970204] kobject:kobject: 'holders' (ffff880073c34580):
kobject_add_internal: parent: 'cifs', set: '<NULL>'
Apr 14 15:54:45 mets kernel: [  184.970225] kobject:kobject: 'cifs' (ffffffffa007e0f0):
fill_kobj_path: path = '/module/cifs'
Apr 14 15:54:45 mets kernel: [  184.970267] kobject:kobject: 'notes' (ffff880073c34440):
kobject_add_internal: parent: 'cifs', set: '<NULL>'
Apr 14 15:54:45 mets kernel: [  184.970761] kobject:kobject: 'cifs_inode_cache' (ffff880075d230a8):
kobject_add_internal: parent: 'slab', set: 'slab'
Apr 14 15:54:45 mets kernel: [  184.970807] kobject:kobject: 'cifs_inode_cache' (ffff880075d230a8):
fill_kobj_path: path = '/kernel/slab/cifs_inode_cache'
Apr 14 15:54:45 mets kernel: [  184.970862] kobject:kobject: ':0016512' (ffff880075d214a8):
kobject_add_internal: parent: 'slab', set: 'slab'
```

Figure 2: Dynamic debug output examples

recently re-named this work to *dynamic debug* from the original *dynamic printk*, to make this clear. The above `debug_enabled_flag()` macro could easily accommodate the 'level' style debugging, by replacing the bit check with a greater than check. Prototypes have already implemented and will be proposed in the near future.

There are a number of modules that set module debugging levels using module parameters Thus, we would propose system wide 'standard' module name parameters that *dynamic debug* can implement. For example, as dynamic_debug_level=n, dynamic_debug_flag=0101, and dynamic_debug=enabled/disabled.

6 Debug Statement Organization

As mentioned at the outset of the work, there are a myriad of styles and macros for debug printing. We propose that the various subsystems make use of the 'core' debugging functions to the extent that they suit their needs. For example, if you are just printing out text use `pr_debug()`, or `dev_dbg()` if you are in a driver. If you are doing 'flag' or 'level' style debugging use the corresponding *dynamic debug* macros. For example, in kernel/module.c, `DEBUGP()` is used. We should convert it to `pr_debug()` so that it can tie into the *dynamic debug* infrastructure.

The question also becomes when should one use `pr_debug()` and when should `printk(KERN_DEBUG)`

be used? Obviously, given the test results posted one can not simply sprinkle `printk(KERN_DEBUG)` everywhere. Thus, for frequently used codepaths need to use `pr_debug()`. Additionally, now that `pr_debug()` can be compiled in, `pr_devel()` can be used for those cases where you wouldn't want *dynamic debug* to pick up the debug statement.

7 Future Work

Clearly, converting more kernel code to use the standard debug statements of `pr_debug()` and `dev_dbg()` is desired. Further, along these lines would be converting subsystems that have more complex debugging styles to the proposed framework. Also, adding command line control parameters, module parameters, and a simple enable and disable all debug statements control mode would be desirable.

As mentioned in the results section, when all the debug statements are enabled, the system performance drops significantly. Although, this is not the 'hot path', it might be nice to improve this case. By simply attaching the backend of these patches to the new ring buffer code we should drastically speed things up. Perhaps, its an option as I think getting the information out via the normal printk path is important as well. There might also be a chance for further integration with Ftrace code specifically the event code[7].

8 Conclusion

'Dynamic Debug' has successfully made high frequency debug statements available at run-time in a manner that does not degrade performance. It has already been used by SGI to help resolve NFS problems, and it is planned to be incorporated into upcoming enterprise kernel releases. We hope that this paper will further understanding of this new feature, in hopes that it can be further adopted and expanded.

9 Acknowledgements

SGI independently developed a very similar debugging system which tied into *dprintk()* and has been used for a number of years to help diagnose customer issues[1]. Greg Banks submitted this work upstream shortly after I submitted the *dynamic debug* work. *Dynamic debug* owes its control language to this work. Section 3, Usage and Examples, is largely taken from the documentation that Greg Banks wrote for the *dprintk()* work.

References

[1] Greg Banks. activate & deactivate dprintks individually and severally. `http://marc.info/?l=linux-kernel&m=123241522202638&w=2`.

[2] Pei Cao. Bloom filters - the math. `http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html`.

[3] Frank Ch. Eigler. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[4] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 2005. ISBN 0-596-00590-3.

[5] Michel Dagenais Mathieu Desnoyers. Lttng: Tracing across execution layers, from the hypervisor to user-space. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[6] Steve Rostedt. ftrace tracing inftrastructure. `http://lwn.net/Articles/270971/`.

[7] Steven Rostedt. event tracer. `http://marc.info/?l=linux-kernel&m=123550413414913&w=2`.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

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

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*