An Approach to Improving the Structure of Error-Handling Code in the Linux Kernel

Suman Saha

LIP6-Regal

Suman.Saha@lip6.fr

Julia Lawall

DIKU/INRIA/LIP6-Regal

julia@diku.dk

Gilles Muller

INRIA/LIP6-Regal

Gilles.Muller@lip6.fr

Abstract

The C language does not provide any abstractions for exception handling or other forms of error handling, leaving programmers to devise their own conventions for detecting and handling errors. The Linux coding style guidelines suggest placing error handling code at the end of each function, where it can be reached by `goto`s whenever an error is detected. This coding style has the advantage of putting all of the error-handling code in one place, which eases understanding and maintenance, and reduces code duplication. Nevertheless, this coding style is not always applied. In this paper, we propose an automatic program transformation that transforms error-handling code into this style. We have applied our transformation to the Linux 2.6.34 kernel source code, on which it reorganizes the error handling code of over 1800 functions, in about 25 minutes.

Categories and Subject Descriptors D.4 [*Operating Systems*]: Reliability

General Terms Reliability, Design

Keywords Linux, error handling, exception handling

1. Introduction

Reliability is essential in embedded systems. A key element of ensuring reliability is proper handling of error conditions [14]. In general, the role of error handling code is to return the system to a coherent state, typically by undoing recent operations and releasing recently allocated resources. If some of these *state-restoring* operations are omitted, the result can be deadlocks and memory leaks. If state-restoring operations are performed in the wrong order, the result can be invalid data accesses, such as null-pointer dereferences and double frees. These issues are especially critical in the case of operating systems, such as Linux, in the context of embedded systems, as an operating system manages many resources over a long period of time, increasing the amount of error conditions that can occur and state-restoring operations that are needed, and heightening the accumulated impact of any memory leaks.

Linux is implemented using the C programming language, which does not provide any built-in error handling mechanisms. In C, a typical strategy for implementing error handling code is to follow each operation that may encounter an error by a conditional that checks for an error result and, if one is found, performs the appropriate cleanup operations before returning from the function. We refer to this strategy as the *basic strategy*. The basic strategy, however, is itself error-prone, as it is easy to overlook some cleanup operations that are required, and to forget to update some existing error handling code when the function is extended with new operations that need to be undone in an error case. Furthermore, there may be substantial code duplication, as the same error handling code may be needed at many places within a function definition.

One style of programming that can somewhat alleviate these difficulties is to move the state-restoring operations from the individual error handling conditionals to a single labelled sequence of state-restoring operations at the end of the function. We refer to this style of programming as the `goto`-*based strategy*. In the `goto`-based strategy, each error-handling conditional only performs the operations that are specific to the identified error condition, such as printing a log message or recording an error indicator in a local variable. It then performs a `goto` that jumps to the correct position within the state-restoring sequence. This approach localizes all of the state-restoring operations into one easily identifiable place, at the end of the function. If the function definition is extended in a way that there are new possible error conditions, the associated error handling code only needs to jump to the right place within this sequence. If new state-changing operations are added within the function definition, the corresponding state-restoring operations only need to be added at one place within this sequence. And finally, the duplication of code is mostly limited to the introduction of the `goto`, regardless of the complexity of the error handling process.

Currently, many Linux functions use the `goto`-based strategy, and this strategy is recommended by the Linux kernel documentation.[1] Nevertheless, a large number of functions still use the basic strategy, and a number of bugs have been found in such code. For example, in the bug-fixing patches applied to Linux 2.6.20 after its release, we have found that in around a third (12/32) of those whose only effect is to add a call to `kfree` or to an unlocking function such as `spin_unlock`, the bug is in code that uses the basic strategy. Most other such bugs were not in error-handling code. We found similar results (6/20) in the set of patches contributing to Linux 2.6.34.[2] Such bugs can persist undetected for a long time, when the associated error condition only rarely occurs.

To improve the structure of error handling code in the Linux kernel, we define an algorithm to transform error handling code implemented according to the basic strategy so that it follows

---

[1] Linux-2.6.34/Documentation/CodingStyle, Chapter 7.

[2] Linux 2.6.20 patches obtained from git://git.kernel.org/pub/scm/linux/-kernel/git/stable/linux-2.6.20.y.git using the command "git log -p v2.6.20..". Linux 2.6.34 patches obtained from git://git.kernel.org/pub/-scm/linux/kernel/git/next/linux-next.git using the command "git log -p v2.6.33..v2.6.34".

the `goto`-based strategy. This algorithm merges the state-restoring code found in each conditional into a sequence of state-restoring operations at the end of the function. We have implemented this algorithm as a tool that treats the full Linux source code, including code specific to various embedded system architectures. This tool successfully converts 29% of the conditionals containing state-restoring error-handling code from the basic strategy to the `goto`-based strategy. It furthermore informs the user of anomalies that preclude this transformation, which may indicate bugs.

The rest of the paper is structured as follows. Section 2 presents some motivating examples and a quantitative analysis of the error handling strategies used in Linux kernel code. Section 3 defines our transformation of error-handling code following the basic strategy to error-handling code in the `goto` style. Section 4 evaluates our approach on the Linux 2.6.34 kernel. Finally, Section 5 presents related work and Section 6 concludes.

2. Background

In this section, we first illustrate the basic error handling strategy and the `goto`-based error handling strategy, using examples from the Linux 2.6.34 kernel source code. We then analyze the frequency of these error-handling strategies in Linux 2.6.34 and earlier versions.

2.1 Examples

Figure 1 shows a typical example of error handling code following the basic strategy. Three `if` statements are shown (lines 5, 12, and 21), each checking for a different condition. In each case, if the condition is satisfied, there is a sequence of error handling operations. In two cases (lines 12 and 21), these error handling operations begin by printing a log message specific to the error. This is followed by some new operations, which are then followed by the error handling code from the previous `if`, if any. Each `if` concludes by returning an error indicator that is specific to the error that has occurred (lines 8, 17, and 27). Overall, there is substantial duplication of code. Indeed, the final call to `DPRINT_EXIT` found in each `if` also appears at the normal exit from the function. The error-handling operations free data structures of various complexity, and omitting any of this code when constructing any new error-handling code that becomes needed as the function evolves will lead to memory leaks.

Figure 2 illustrates a possible reimplementation of this function, using the `goto`-based strategy. The largest sequence of error-handling operations, from the third `if`, has been moved to the end of the function. The `if` branches themselves have each been transformed to perform the operations specific to the given error, namely printing the log message and storing the error indicator in the variable `ret`. Each `if` branch then ends in a `goto` that jumps to the appropriate point in the sequence of error handling operations at the end of the function. This sequence in turn uses `goto` to jump to the original end of the function, to take advantage of the call to `DPRINT_EXIT` that is already available there. The error handling code within the function body is now limited to what is specific to each error case. The only non-local knowledge that is required to construct such code is uniformly obtained by looking at the code sequence at the end of the function.

To illustrate the full scope of the problem, we next consider an example of error-handling code that is implemented using the basic strategy and that contains a memory leak. In Figure 3, two resources are allocated: `path` and `inode` (lines 3 and 5). Both of these allocations can fail, as can the subsequent initialization of the inode index (line 9). If the allocation of `path` fails, the system simply crashes, via the operator `BUG_ON` (line 4), and no error handling code is required. On the other hand, if the allocation or initialization of `inode` fails, then the function aborts, returning an error indicator in both cases (lines 7 and 12) and additionally releasing `inode` in the latter case (line 11). The memory allocated for `path`, however, is

```
1  static int storvsc_probe(struct device *device) {
2    int ret;
3    ...
4    host_device_ctx->request_pool = kmem_cache_create(...);
5    if (!host_device_ctx->request_pool) { /* 1 */
6      scsi_host_put(host);
7      DPRINT_EXIT(STORVSC_DRV);
8      return -ENOMEM;
9    }
10   device_info.PortNumber = host->host_no;
11   ret = storvsc_drv_obj->Base.OnDeviceAdd(...);
12   if (ret != 0) { /* 2 */
13     DPRINT_ERR(STORVSC_DRV, "unable to add scsi vsc device");
14     kmem_cache_destroy(host_device_ctx->request_pool);
15     scsi_host_put(host);
16     DPRINT_EXIT(STORVSC_DRV);
17     return -1;
18   }
19   ...
20   ret = scsi_add_host(host, device);
21   if (ret != 0) { /* 3 */
22     DPRINT_ERR(STORVSC_DRV, "unable to add scsi host device");
23     storvsc_drv_obj->Base.OnDeviceRemove(device_obj);
24     kmem_cache_destroy(host_device_ctx->request_pool);
25     scsi_host_put(host);
26     DPRINT_EXIT(STORVSC_DRV);
27     return -1;
28   }
29   scsi_scan_host(host);
30   DPRINT_EXIT(STORVSC_DRV);
31   return ret;
32 }
```

Figure 1. Example of the basic error handling strategy (Linux-2.6.34/drivers/staging/hv/storvsc_drv.c)

```
1  static int storvsc_probe(struct device *device) {
2      int ret;
3      ...
4      host_device_ctx->request_pool = kmem_cache_create(...);
5      if (!host_device_ctx->request_pool) {
6        ret = -ENOMEM;
7        goto out3;
8      }
9      device_info.PortNumber = host->host_no;
10     ret = storvsc_drv_obj->Base.OnDeviceAdd(...);
11     if (ret != 0) {
12       DPRINT_ERR(STORVSC_DRV, "unable to add scsi vsc device");
13       ret = -1;
14       goto out2;
15     }
16     ...
17     ret = scsi_add_host(host, device);
18     if (ret != 0) {
19       DPRINT_ERR(STORVSC_DRV, "unable to add scsi host device");
20       ret = -1;
21       goto out;
22     }
23     scsi_scan_host(host);
24 out1: DPRINT_EXIT(STORVSC_DRV);
25     return ret;
26 out: storvsc_drv_obj->Base.OnDeviceRemove(device_obj);
27 out2: kmem_cache_destroy(host_device_ctx->request_pool);
28 out3: scsi_host_put(host);
29     goto out1;
30 }
```

Figure 2. Improved version of Figure 1

not freed in either case. We consider how this code would be written if it used the `goto`-based strategy.

```
1   static struct inode *btrfs_new_inode(...) {
2       ...
3           path = btrfs_alloc_path();
4           BUG_ON(!path);
5           inode = new_inode(root->fs_info->sb);
6           if (!inode)
7                   return ERR_PTR(-ENOMEM);
8           if (dir) {
9                   ret = btrfs_set_inode_index(dir, index);
10                  if (ret) {
11                          iput(inode);
12                          return ERR_PTR(ret);
13                  }
14          }
15      ...
16  fail:   if (dir) BTRFS_I(dir)->index_cnt--;
17          btrfs_free_path(path);
18          iput(inode);
19          return ERR_PTR(ret);
20  }
```

Figure 3. A bug in error handling code (Linux-2.6.34/fs/btrfs/-inode.c)

In Figure 3, we may observe that there is already labelled error handling code available at the end of the function that includes freeing `path` (line 17), but the programmer has not taken advantage of it. Indeed, it is not immediately exploitable, because when the allocation of `inode` fails, only `path` should be freed, and when the initialization of `inode` fails, only `inode` and `path` should be freed, while code at the `fail` label may additionally call BTRFS_I (line 16). To apply the strategy illustrated in Figure 2, we should ideally simply add `goto`s that jump to new labels within this error handling code. However, it is also necessary to invert the freeing of `path` and `inode`, so that we can create a label that only frees `path` before exiting the function. Fortunately the two freeing operations are disjoint, and so exchanging them is possible. The resulting code, shown in Figure 4 has no memory leaks and is resilient to further changes. Implementing new error handling code requires simply writing a `goto` to the correct line in this sequence. Adding a new resource allocation requires only adding the corresponding deallocation operation at the correct position in this sequence. In either case, the rest of the function remains correct automatically.

2.2 Analysis

To better understand the current state of error handling code in Linux kernel code we have analyzed the Linux 2.6.34 source code as well as several previous versions, going back to June 1996. Figure 5 shows that overall, the number of functions with error handling code is increasing, especially in the `drivers` directory. In Linux 2.0 there were only around 200 such functions in the `drivers` directory, in Linux 2.6.0 there were fewer than 4 000 such functions, and in Linux 2.6.34 there were almost 15 000. This represents an increase of almost 18 times from Linux 2.0 to Linux 2.6.0, during which time the code size increased by only 7 times,[3] and of over 4 times from Linux 2.6.0 to Linux 2.6.34, during which time the code size increased by only 2.5 times. Furthermore, in the case of `fs`, while the number of error handling functions and the code size grew at the same rate from Linux 2.0 to Linux 2.6.0, from Linux 2.6.0 to Linux 2.6.34, the number of error-handling functions grew by more than 2.5 times, while the code size grew by only 1.8 times. These figures suggest an overall increasing diligence in detecting

---

[3] The code size was measured using SLOCCount [15].

```
1   static struct inode *btrfs_new_inode(...) {
2   {
3       ...
4           path = btrfs_alloc_path();
5           BUG_ON(!path);
6           inode = new_inode(root->fs_info->sb);
7           if (!inode) {
8                   ret = -ENOMEM;
9                   goto out2;
10          }
11          if (dir) {
12                  ret = btrfs_set_inode_index(dir, index);
13                  if (ret)
14                          goto out1;
15          }
16      ...
17  fail:   if (dir) BTRFS_I(dir)->index_cnt--;
18  out1:   iput(inode);
19  out2:   btrfs_free_path(path);
20          return ERR_PTR(ret);
21  }
```

Figure 4. Improved version of Figure 3
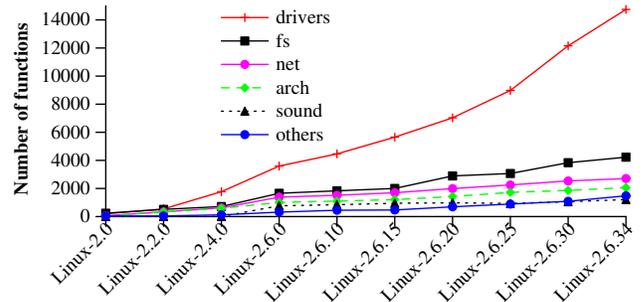
Figure 5. Number of functions with error handling code in different subdirectories of different Linux versions.

and handling error conditions, which has probably been facilitated by the increasing use of defect-finding tools [13].

For the same Linux versions, Figure 6 shows the number of functions that contain error handling code that use either the `goto`-based strategy, the basic strategy, or a mixture of both. The number of functions using the `goto`-based strategy is increasing more rapidly than the number of functions using the basic strategy. Nevertheless, the number of functions using the basic strategy or a combination of strategies is also increasing, but more slowly. Our main concern is those functions, to facilitate their conversion from the basic strategy to the `goto`-based strategy.

Finally, Figure 7 breaks down the above results by subdirectory for Linux-2.6.34. Most directories have more functions using the `goto`-based strategy than the basic strategy, although for `sound`, the numbers are essentially equal. Still, `drivers` has over 3 000 functions that use the basic strategy, while the other directories each typically have around 400-500 functions in this category.

3. Transformation Algorithm

Our goal is to merge the sequence of statements in each error-handling `if` branch into a shared sequence of statements at the end of the function, and to replace each error-handling `if` branch by a `goto` into this sequence. The algorithm considers one function at a time. The main steps are 1) identify `if` branches that correspond to the error handling code within a given function and collect some
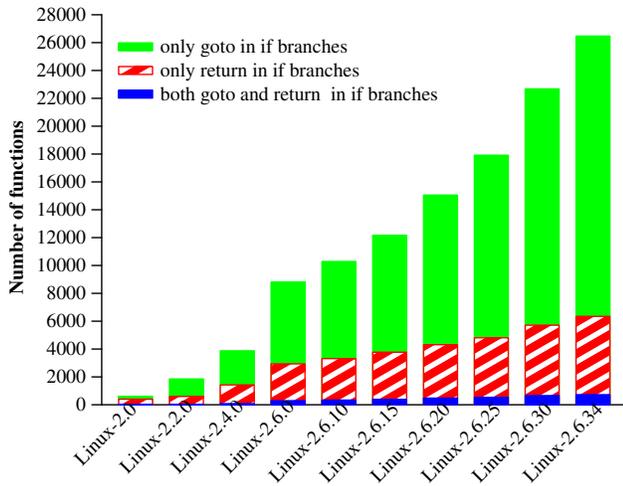
Figure 6. Number of functions using only the `goto`-based strategy, using the only the basic strategy, and using a mixture of both.
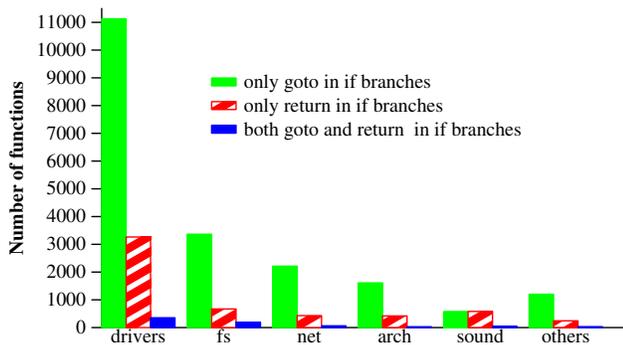
Figure 7. Number of Linux 2.6.34 functions, by directory, using only the `goto`-based strategy, using the only the basic strategy, and using a mixture of both. "Others" refers to the other Linux directories, which are much smaller than the ones listed.

other information that is useful in the transformation process, 2) identify operations in this error handling code that can be shared in a sequence at the end of the function, and 3) transform the function definition to move error handling code to the end of the function and insert appropriate `goto`s into each error handling `if` branch. These steps are described below, both formally and in terms of examples.

We describe the analysis and transformation rules with respect to a small language, defined according to the grammar shown below.

$$
\begin{array}{lll}
\text{Statement} & t & ::= exp = exp; \mid f(exp); \mid \text{if } (exp) \; [t] \; (\text{else } [t])^? \\
\text{Statements} & [t] & ::= t \ldots t \; (\text{return } exp;)^? \\
\text{Program} & P & ::= [t] \; (l: [t])^*
\end{array}
$$

The actual implementation, however, treats the full C language. A program in this language is analogous to a function body in C code. In the grammar, *exp* refers to an arbitrary expression, including function calls, *f* refers to the name of a function, and *l* refers to a label. To avoid clutter, we omit the braces around the branches of a conditional. We distinguish two sets of expressions, String and Error. String is the set of strings. Error is the set of expression that may indicate an error, as determined by common Linux coding patterns. Following Linux coding conventions, expressions in Error include

NULL, negated integers and macros, and expressions of the form `ERR_PTR`(*exp*) and `PTR_ERR`(*exp*).[4] Identifiers are also in Error, as they may be initialized to one of these values. A few Linux services, such as ACPI, define their own error conventions. The approach could be extended to take these into account, although in the long term it may be better to refactor that code to use a more standard strategy.

Selecting If Branches (step 1a) The first step is to select the `if` branches that should potentially be converted from the basic strategy to the `goto`-based strategy. Such `if` branches must at a minimum represent error-handling code. We identify error-handling code as an `if` branch that ends by returning an expression in the set Error. The `if` branch must also not contain other conditionals and must contain at least one function call that is not debugging code, *i.e.*, that does not have a string as an argument. The latter constraint avoids introducing a jump to a jump, and follows Linux coding style.

Figure 8 shows a function having multiple `if` branches. The branches labeled 2, 3, 5, and 6 (lines 6, 11, 19, and 25) meet the above conditions and are thus selected for further consideration. On the other hand, the branch labeled 1 (line 4) is not selected because it does not contain any function calls other than the Error call, the branch labeled 4 (line 17) is not selected because it contains another conditional, and the branch labeled 7 (line 31) is not selected because it does not end with a return.

```
1   static int acct_on(char *name) {
2       ...
3       if (IS_ERR(file)) /* 1, not selected */
4           return PTR_ERR(file);
5
6       if (!S_ISREG(file->f_path.dentry->d_inode->i_mode)) { /* 2 */
7           filp_close(file, NULL);
8           return -EACCES;
9       }
10
11      if (!file->f_op->write) { /* 3 */
12          filp_close(file, NULL);
13          return -EIO;
14      }
15
16      ...
17      if (ns->bacct == NULL) { /* 4, not selected */
18          acct = kzalloc(sizeof(struct bsd_acct_struct), GFP_KERNEL);
19          if (acct == NULL) { /* 5 */
20              filp_close(file, NULL);
21              return -ENOMEM;
22          }
23      }
24      ...
25      if (error) { /* 6 */
26          kfree(acct);
27          filp_close(file, NULL);
28          return error;
29      }
30      ...
31      if (ns->bacct == NULL) { /* 7, not selected */
32          ns->bacct = acct;
33          acct = NULL;
34      }
35      return 0;
36  }
```

Figure 8. `If`s that do and do not represent error-handling code (Linux-2.6.34/kernel/acct.c)

---

[4] `ERR_PTR`(*exp*) and `PTR_ERR`(*exp*) coerce an integer error indicator to and from a pointer type, respectively.

The rule for selecting `if` branches from a program P is formalized as follows. The notation $e_1 \in e_2$, for any terms e_1 and e_2, means that e_1 is a subterm of e_2. The notation "$\ldots$" means an arbitrary sequence of statements. The result $\mathcal{S}$ of this rule is a set of pairs of a line number and an `if` branch, where the line number is that of the `if` containing the collected branch, obtained using the operator startline. We refer to this set $\mathcal{S}$ as the *branch list*.

$$
\begin{aligned}
\mathcal{S} = \{\langle ln, [t]\rangle \mid \\
\texttt{if } (exp) \ [t] \in P \ \wedge \\
[t] \equiv \ldots f(exp') \ldots \texttt{return } exp''; \ \wedge \\
exp' \notin String \wedge exp'' \in Error \ \wedge \\
\forall exp''', [t]_1, [t]_2 : \texttt{if } (exp''') \ [t]_1 \ (\texttt{else } [t]_2)^? \notin [t] \ \wedge \\
ln = \texttt{startline}(\texttt{if } (exp) \ [t])\}
\end{aligned}
$$

Storing the error number in a variable (step 1b) An `if` branch implementing the basic strategy can return an error indicator directly or it can store this value in some variable, either before or within the `if` branch. In Linux 2.6.34 among error-handling `if` branches following the basic strategy, the error indicator is returned directly 63% of the time. When different `if` branches return different error indicators, this approach prevents merging the error-handling code. To make this merging possible, our algorithm transforms `if` branches that directly return an error indicator as follows: 1) a new statement is added at the beginning of the `if` branch that stores the current return value in a variable that is common to all selected `if` branches, which we refer to as the *return variable*, and 2) the `return` statement at the end of the `if` branch is transformed to return the value of the return variable.

To carry out this transformation, the algorithm first searches for a variable that is used to return a result somewhere in the function, which can be used as the return variable. Using an existing return variable enables merging return statements and improves readability. If there is no such variable, the algorithm creates a fresh variable for this purpose. If there is more than one such variable, then the first one is chosen. For example, in Figure 8, the variable `error` is already used to return the error indicator in the `if` labeled 6 (line 28). Our algorithm thus uses that variable as the return variable. On the other hand, in Figure 9 there is no such variable, so the algorithm creates a fresh one.

Formally, the choice of the return variable is determined by the function rv, defined below. This function takes as arguments the complete program P and the branch list $\mathcal{S}$, defined previously. It returns a pair of the return variable and a new version of the program, which may be augmented with the declaration of the return variable if no suitable existing variable can be found.[5]

$$
\begin{aligned}
\texttt{rv}(P, \mathcal{S}) = \\
\langle x, P\rangle \qquad & \text{if } \texttt{return } x; \in P \ \wedge \\
& \forall \langle ln, [t]\rangle \in \mathcal{S} : \texttt{return } x; \in [t] \vee x \notin [t] \\
\langle x, \texttt{int } x; @P\rangle \ & \text{where } x \notin P, \text{ otherwise}
\end{aligned}
$$

After choosing a return variable, the next step is to transform each selected `if` branch that does not already use that variable in its `return` statement. An assignment of the current return value to the return variable is added at the beginning of each such branch, and the `return` statement is modified to return the return variable at the end. This transformation is safe because the return variable has been chosen such that it is not already used anywhere in the branch. The transformation is performed on elements of the branch list $\mathcal{S}$, producing an extended version of the branch list, $\mathcal{S}_{\text{rv}}$, as follows, where $\langle x, P'\rangle = \texttt{rv}(P, \mathcal{S})$:

$$
\begin{aligned}
\mathcal{S}_{\text{rv}} = \{\langle ln, [t]'\rangle \mid \langle ln, ifcode@\texttt{return } e;\rangle \in \mathcal{S} \ \wedge \\
[t]' \equiv ifcode@\texttt{return } e; \ \text{if } e \equiv x \ \wedge \\
[t]' \equiv x \ \texttt{= } e; @ifcode@\texttt{return } x; \text{ otherwise}\}
\end{aligned}
$$

---

[5] @ is used to concatenate code fragments.

```
1   int dvb_register_device(struct dvb_adapter *adap, ...) {
2   ...
3     if ((id = dvbdev_get_free_id (adap, type)) < 0) {
4       mutex_unlock(&dvbdev_register_lock);
5       *pdvbdev = NULL;
6       printk(KERN_ERR "%s: couldn't find free
7                       device id\n",%__func__);
8       return -ENFILE;
9     }
10  ...
11    if (!dvbdev) {
12      mutex_unlock(&dvbdev_register_lock);
13      return -ENOMEM;
14    }
15  ...
16    if (!dvbdevfops) {
17      kfree (dvbdev);
18      mutex_unlock(&dvbdev_register_lock);
19      return -ENOMEM;
20    }
21  ...
22    if (minor == MAX_DVB_MINORS) {
23      kfree(dvbdevfops);
24      kfree(dvbdev);
25      mutex_unlock(&dvbdev_register_lock);
26      return -EINVAL;
27    }
28  ...
29    if (IS_ERR(clsdev)) {
30      printk(KERN_ERR "%s: failed to create
31              device dvb%d.%s%d %(%ld)\n",
32              __func__, adap->num, dnames[type], id,
33              PTR_ERR(clsdev));
34      return PTR_ERR(clsdev);
35    }
36  ...
37    return 0;
38  }
```

Figure 9. A function that does not have any variable for storing the error indicator. (Linux-2.6.34/drivers/media/dvb/dvb-core/dvbdev.c)

Creating the label environment (step 1c) The algorithm next creates a label environment that maps each label to all of the code that can be executed when jumping to that label. This label environment is used subsequently to determine whether the state-restoring code of a given `if` branch matches the code following any existing label. Two kinds of judgements are used. For a program P, the judgement, $\vdash P \rightarrow lblenv$ indicates that the final label environment is $lblenv$. For a sequence of labeled statements $(l' : [t])^*$, a judgement of the form $\vdash (l' : [t])^* \rightarrow \langle [t]', lblenv\rangle$, indicates that $[t]'$ is the sequence of statements at the beginning of $(l' : [t])^*$ that preceding code may fall through to, and $lblenv$ is the label environment derived from $(l' : [t])^*$. In this definition, for conciseness, we follow the convention that the first rule that matches is the one that applies. ε is an empty sequence of statements.

$$
\frac{\vdash (l : [t]')^* \rightarrow \langle [t]'', lblenv\rangle}{\vdash [t](l : [t]')^* \rightarrow lblenv} \qquad \vdash \varepsilon \rightarrow \langle \varepsilon, \emptyset\rangle
$$

$$
\frac{\vdash (l' : [t]')^* \rightarrow \langle [t]'', lblenv\rangle \wedge [t]''' = [t] \ \texttt{return } e;}{\vdash l : [t] \ \texttt{return } e; \ (l' : [t]')^* \rightarrow \langle [t]''', \{\langle l, [t]'''\rangle\} \cup lblenv\rangle}
$$

$$
\frac{\vdash (l' : [t]')^* \rightarrow \langle [t]'', lblenv\rangle \wedge [t]''' = [t]@[t]''}{\vdash l : [t] \ (l' : [t]')^* \rightarrow \langle [t]''', \{\langle l, [t]'''\rangle\} \cup lblenv\rangle}
$$

Partition (step 2a) The next step is to partition each branch in the branch list to separate the code that is specific to the given error condition, which we refer to as *if code*, from the potentially sharable state-restoring code, which we refer to as *label code*. In particular, `return` statements are label code, as is any non-debugging function

call, *i.e.*, a function call that does not have a string argument. For example, in the first branch of Figure 9, the call to `printk` and the assignment statement are considered to be specific to the given error condition, and are thus if code.

The transformation performed in step 3 will leave the if code in the `if` branch and move the label code to the end of the function. If label code were to appear before if code, then the transformation process would change the order of the operations. To prevent this, the label code is defined to be the largest suffix of an `if` branch that satisfies the above properties. Given the $\mathcal{S}_{rv}$ computed above, the result of the partitioning process is a refined branch list:

$$\mathcal{S}_{part} = \{\langle ln, ifcode, lblcode\rangle \mid \langle ln, [t]\rangle \in \mathcal{S}_{rv} \wedge \vdash [t] \rightarrow \langle ifcode, lblcode\rangle\}$$

where the judgement $\vdash [t] \rightarrow \langle ifcode, lblcode\rangle$ is defined below. We again follow the convention that the first rule that matches is the one that applies. There is no need for a rule for an `if` statement because an element of the branch list contains no nested conditionals.

$$\frac{\vdash [t] \rightarrow \langle ifcode, lblcode\rangle}{\vdash [t] \; \texttt{return} \; e; \; \rightarrow \langle ifcode, lblcode@\texttt{return} \; e;\rangle}$$

$$\frac{\vdash [t] \rightarrow \langle ifcode, lblcode\rangle}{\vdash exp_1 \; \texttt{=} \; exp_2; [t] \rightarrow \langle exp_1 \; \texttt{=} \; exp_2;@ifcode, lblcode\rangle}$$

$$\frac{\vdash [t] \rightarrow \langle ifcode, lblcode\rangle \wedge (exp \in \text{String} \vee ifcode \neq \varepsilon)}{\vdash f(exp); [t] \rightarrow \langle f(exp);@ifcode, lblcode\rangle}$$

$$\frac{\vdash [t] \rightarrow \langle \varepsilon, lblcode\rangle}{\vdash f(exp); [t] \rightarrow \langle \varepsilon, f(exp);@lblcode\rangle} \qquad \vdash \varepsilon \rightarrow \langle \varepsilon, \varepsilon\rangle$$

Filtering (step 2b) Moving label code to the end of the function can only reduce the code size if part of the label code, including at least one state-restoring operation, is shared with the label code of some other `if` branch or some other existing code at the end of the function. If there is no such shared code, then we consider that the benefit of transforming the code does not outweigh the cost of introducing a `goto` and remove the entry from the branch list $\mathcal{S}_{part}$. This filtering process is defined as follows, using the program P and the *lblenv* computed above, to produce a filtered branch list $\mathcal{S}_{filter}$:

$$\mathcal{S}_{filter} = \{\langle l, ifcode, lblcode\rangle \mid \\ \langle l, ifcode, lblcode\rangle \in \mathcal{S}_{part} \wedge \\ lblcode \equiv \ldots f(exp_1); \; \texttt{return} \; (exp_2); \wedge \\ (\exists \langle l', ifcode', \ldots f(exp_1); \; \texttt{return} \; (exp_2);\rangle \in \mathcal{S}_{part} : l \neq l' \vee \\ \exists \langle l', \ldots f(exp_1); \; \texttt{return} \; (exp_2);\rangle \in lblenv \vee \\ P \equiv \ldots f(exp_1); \; \texttt{return} \; (exp_2);)\}$$

Figure 10 shows the number of functions that use the basic strategy or a mixture of the `goto`-based strategy and the basic strategy (red/leftmost bars), the number of these functions that survive the filtering process (green/middle bars), implying that there is some shared state-restoring code, and the number of functions that are not considered for transformation (blue/rightmost bars), indicating that there is no shared state-restoring code. The figure showing that the filtering eliminates two thirds of the selected functions in most directories. However, almost half of the selected functions in the `sound` directory are transformed.
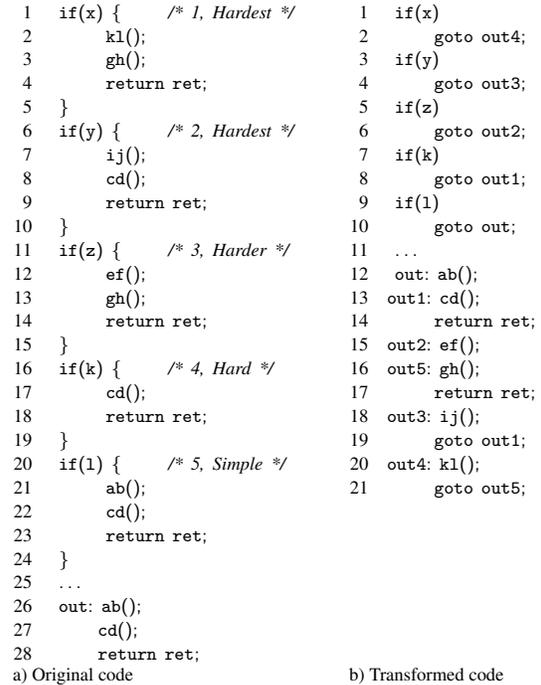
Classification and transformation (step 3) This step classifies the remaining elements of the branch list, $\mathcal{S}_{filter}$, according to how difficult they are to transform. On the basis of difficulty, the algorithm chooses the appropriate transformation. We classify the `if` branches into four categories: Simple, Hard, Harder and Hardest. The classification and transformation process iterates over the elements of $\mathcal{S}_{filter}$ starting with the one with the largest line number. This element typically contains the longest sequence of state-restoring code (*cf* Figure 1), undoing all of the operations that have been performed in the function, and thus offers the most opportunity for sharing. We explain the classification and transformation process
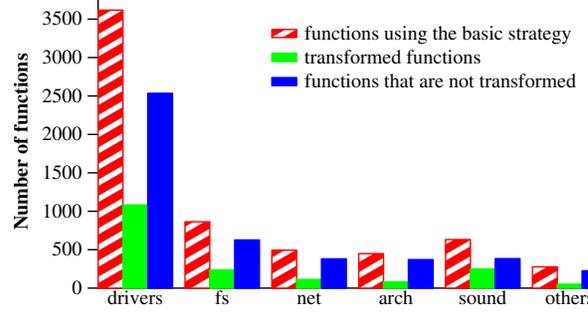
Figure 10. The red/leftmost bars show the number of functions collected in the `if` selection step. The green/middle bars show the number that the filtering step keeps for transformation. The blue/rightmost bars show the number that the filtering step discards.

in terms of an artificial example that illustrates all of the possible cases. This example is more complex than typical functions.

For a given `if` branch, if the label code is the same as the code already associated with some label in the label environment, then the `if` branch is classified as *Simple*, because no code has to be moved. Instead, it is sufficient to remove the label code from the branch and replace it with a `goto` statement with that label name. An example is the branch labeled 5 in Figure 11a (line 20). In this case, the label code is exactly same as the code at the label `out`.

```
1   if(x) {       /* 1, Hardest */        1   if(x)
2       kl();                              2       goto out4;
3       gh();                              3   if(y)
4       return ret;                        4       goto out3;
5   }                                      5   if(z)
6   if(y) {       /* 2, Hardest */        6       goto out2;
7       ij();                              7   if(k)
8       cd();                              8       goto out1;
9       return ret;                        9   if(l)
10  }                                      10      goto out;
11  if(z) {       /* 3, Harder */         11  ...
12      ef();                              12  out: ab();
13      gh();                              13  out1: cd();
14      return ret;                        14      return ret;
15  }                                      15  out2: ef();
16  if(k) {       /* 4, Hard */           16  out5: gh();
17      cd();                              17      return ret;
18      return ret;                        18  out3: ij();
19  }                                      19      goto out1;
20  if(l) {       /* 5, Simple */         20  out4: kl();
21      ab();                              21      goto out5;
22      cd();
23      return ret;
24  }
25  ...
26  out: ab();
27      cd();
28      return ret;
a) Original code                          b) Transformed code
```

Figure 11. Simple, Hard, Harder, Hardest branches

On the other hand, if the label code is not exactly same as the code at any label, but is the same as a suffix of some existing label's code or is the same as a suffix of the entire function, then the `if` branch is classified as *Hard*. In this case it is also not necessary to move any code, but the algorithm has to identify a position for a new label. The label code of branch 4 in Figure 11a (line 16) matches a suffix of the code at the label `out`. The algorithm thus creates a

new label just before the call to `cd` at the end of the function, and replaces branch 4 by a `goto` to the new label (line 8, Figure 11b).

In the third case, an `if` branch's label code does not match a suffix of any existing label's code or the code at the original end of the function. Such a branch is classified as *Harder*. For such a branch, the algorithm creates a new label and places it at the end of the function, along with the branch's label code. In the case of a `void` function (outside the scope of our small language), there may be no `return` at the end of the original function. In this case, the algorithm additionally adds `return;` before the new label. For example, after transforming branches 5 and 4 in Figure 11a, the code in branch 3 is not a suffix of any existing label's code. So, the algorithm inserts a new label with this code after the `return` statement of the `out` label.

The final category is *Hardest*. In this category, the complete label code is not a suffix of any existing label's code, however a suffix of the label code is the same as a suffix of some existing label's code. This results in two parts of the label code; one that does not match any existing label's code and the other that is a suffix of some existing label's code. The unmatched part can be treated as Harder and the matched part can be treated as though it is Simple or Hard. Branches 2 and 1 of Figure 11a (lines 6 and 1) are in the Hardest category. In each case, the first statement of the label code, `ij()` or `kl()`, respectively, is not found in any branch. So these statements are treated as Harder. In each case, the code in the remainder of the branch is identical to or a proper suffix of the code at an existing label. For branch 2 (line 6), the call to `cd` and the `return` are identical to the code at the label introduced for treating branch 4, so this code is treated as Simple. For branch 1 (line 1), the call to `gh` and the `return` match a suffix of the code at the label introduced when treating branch 3, so this code is treated as Hard.

The complete transformation process selects the element of S_{filter} with the largest line number, classifies it according to the rules below, transforms it according to the result of the classification, and then repeats on the next element of S_{filter}, until all elements have been considered. The label environment *lblenv* is recomputed after each transformation step, according to the rules described in step 1c.

The rules for classifying branches as Simple, Hard, Harder, and Hardest are formalized as shown below for an element of S_{filter}, $\langle ln, ifcode, lblcode \rangle$. In these rules, createlbl() creates a new label and suffix(a, b) is satisfied if a is a suffix of b. A judgement has the form $\vdash_c \langle ln, ifcode, lblcode \rangle \rightarrow \langle ln, ifcode, status \rangle$ where *status* is defined as follows:

$$status \quad = \quad \begin{aligned} &\mathsf{Simple}(label) \\ | \quad &\mathsf{Hard}(label, fresh\ label, lblcode) \\ | \quad &\mathsf{Harder}(fresh\ label, lblcode) \\ | \quad &\mathsf{Hardest}(fresh\ label, lblcode, status) \end{aligned}$$

The classification rules are as follows. In each case, the first rule that matches is the one that is applied.

$$\frac{\exists l : \langle l, lblcode \rangle \in lblenv}{\vdash_c \langle ln, ifcode, lblcode \rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ l;, \mathsf{Simple}(l) \rangle}$$

$$\frac{\exists \langle l, [t] \rangle \in lblenv : \mathsf{suffix}(lblcode, [t]) \wedge nl = \mathsf{createlbl}()}{\vdash_c \langle ln, ifcode, lblcode \rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ nl;, \mathsf{Hard}(l, nl, lblcode) \rangle}$$

$$\frac{\begin{array}{c} lblcode \equiv \ldots\ f(exp);\ \texttt{return}\ e; \wedge nl = \mathsf{createlbl}() \wedge \\ (\forall \langle l, [t] \rangle \in lblenv \wedge \mathsf{not}(\mathsf{suffix}(f(exp);\ \texttt{return}\ e;, [t]))) \end{array}}{\vdash_c \langle ln, ifcode, lblcode \rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ nl;, \mathsf{Harder}(nl, lblcode) \rangle}$$

$$\frac{\begin{array}{c} (\exists [t]_1, [t]_2 : lblcode \equiv [t]_1@[t]_2 \wedge \\ (\forall \langle l, [t] \rangle \in lblenv : \forall [t]_1' \neq \varepsilon : \\ \mathsf{suffix}([t]_1', [t]_1) \Rightarrow \mathsf{not}(\mathsf{suffix}([t]_1'@[t]_2, [t]))) \wedge \\ nl = \mathsf{createlbl}() \wedge \\ \vdash_c \langle -1, [t]_1, [t]_2 \rangle \rightarrow \langle -1, [t]_1'', status \rangle \end{array}}{\vdash_c \langle ln, ifcode, lblcode \rangle \rightarrow \langle ln, ifcode@\texttt{goto}\ nl;, \mathsf{Hardest}(nl, [t]_1, status) \rangle}$$

The transformation is then in two parts, considering first the label code and then the if code. Given a classified triple $\langle ln, ifcode, status \rangle$,

we first transform the program P to reposition the label code as indicated by *status*. This part of the transformation uses judgements of the form $status \vdash_{\text{lbl}} P \rightarrow P'$, producing a new program P'.

$$\mathsf{Simple}(l) \vdash_{\text{lbl}} P \rightarrow P$$

$$\frac{}{\mathsf{Hard}(l, nl, lblcode) \vdash_{\text{lbl}} \ldots l : [t]\ @\ lblcode \ldots \rightarrow}$$
$$\ldots l : [t]\ @\ nl : lblcode \ldots$$

$$\mathsf{Harder}(nl, lblcode) \vdash_{\text{lbl}} \ldots \texttt{return}\ e; \rightarrow \ldots \texttt{return}\ e;\ nl : lblcode$$

$$\mathsf{Hardest}(nl, [t]_1, \mathsf{Simple}(l)) \vdash_{\text{lbl}} \ldots \texttt{return}\ e;\ l : [t]_2 \ldots \rightarrow$$
$$\ldots \texttt{return}\ e;\ nl : [t]_1\ @\ l : [t]_2 \ldots$$

$$\mathsf{Hardest}(nl, [t], \mathsf{Simple}(l)) \vdash_{\text{lbl}} \ldots \texttt{return}\ e; \rightarrow$$
$$\ldots \texttt{return}\ e;\ nl : [t]\ @\ \texttt{goto}\ l$$

$$\mathsf{Hardest}(nl, [t]_1, \mathsf{Hard}(l, nl', [t]_2)) \vdash_{\text{lbl}} \ldots\ l : [t]@[t]_2 \ldots \texttt{return}\ e; \rightarrow$$
$$\ldots\ l : [t]@nl' : [t]_2 \ldots \texttt{return}\ e;\ nl : [t]_1\ @\ \texttt{goto}\ l$$

Finally, given a classified triple $\langle ln, ifcode, status \rangle$ and the program P' produced by the above rules, we adjust the corresponding `if` statement in the program to use the new if code. This part of the transformation uses judgements of the form $status \vdash_{\text{if}} P \rightarrow P'$, producing a new program P'. This new program P' is then used on the next iteration, to treat the next element of S_{filter}.

$$\frac{\mathsf{startline}(\texttt{if}\ (exp)\ [t]) = ln}{\langle ln, ifcode, status \rangle \vdash_{\text{if}} \ldots\ \texttt{if}\ (exp)\ [t]\ \ldots \rightarrow}$$
$$\ldots\ \texttt{if}\ (exp)\ ifcode \ldots$$

4. Evaluation

The algorithm, excluding the parser, has been implemented as 1300 lines of OCaml code. For the parser, we have reused the parser developed for the program transformation system Coccinelle [11, 12]. This parser does not require first executing the C preprocessor, and thus all Linux code, for all possible architectures and configurations, can be treated. In this section, we present the results of applying our tool to the source code of the Linux 2.6.34 kernel, which was released in May 2010. Linux 2.6.34 contains over 8 million lines of C code, as calculated using SLOCCount [15], and processing this code using our tool takes approximately 25 minutes on one core of an 8-code 3GHz machine with 16GB memory.

An example from the Linux kernel Figure 12 shows an extract of code from Linux 2.6.34 using the basic strategy, and Figure 13 shows the result of the transformation. The transformation starts with the branch labelled 4 (line 34). This branch has no code in common with the only label that is available, and so it is classified as Harder. Its code is moved to the end of the function, with the label `out` (Figure 13, line 37). Next, the branch labelled 3 (line 26) is considered. This branch contains a superset of the operations at the label `out`, and so it is classified as Hardest. Because the label `out` is at this point immediately preceded by `return 0` and the code at the label `out` is a suffix of the code in branch 3, the new label `out1` can be placed just before `out`. Next, we consider the branch labelled 2 (line 18). This branch has the same state-restoring operations as the branch labelled 3, and thus the branch labelled 2 is considered to Simple and reuses the label `out1`. Finally, the branch labelled 1 (line 10) contains a suffix of this code, implying that it is classified as Hard. The label `out2` is introduced in transforming this branch. Overall, all of the `if`s of the function are transformed, and most are reduced to a `goto` and possibly a debugging statement.

The impact of filtering As was shown in Figure 10, for many functions that use the basic strategy, all of the branches are filtered, and thus the function is not transformed by our algorithm. Furthermore, due to the filtering, only a subset of the error handling code within a function may be transformed. Table 1 shows the number of functions

		Functions that are not transformed				Functions that are transformed		
Directory	Return	Strict	Single if	Unordered	No sharing	Full processing	Partial processing	Total
drivers	3617	376(10%)	2043(56%)	102(3%)	14(0.5%)	999(28%)	83(2%)	1082(30%)
fs	863	81(9%)	514(60%)	24(3%)	8(1%)	213(25%)	23(3%)	236(27.5%)
net	493	21(4%)	337(68%)	23(5%)	0(0%)	100(20%)	12(2%)	112(22.5%)
arch	449	26(6%)	326(73%)	1(0.5%)	18(4%)	72(16%)	6(1%)	78(17%)
sound	632	27(4%)	337(53%)	16(3%)	4(1%)	225(36%)	23(4%)	248(39.5%)
others	277	13(5%)	202(73%)	6(2%)	5(2%)	48(17%)	3(1%)	51(18%)
total	6331	544(9%)	3759(59%)	172(3%)	49(1%)	1657(27%)	150(2%)	1807(29%)

Table 1. Number of functions of Linux-2.6.34 that cannot and can be transformed.

```
1   static int download_fw(struct edgeport_serial *serial) {
2     ...
3     if (serial−>product_info.TiMode == TI_MODE_DOWNLOAD) {
4        ...
5     }
6     else if ((start_address = get_descriptor_addr(serial,
7        I2C_DESC_TYPE_FIRMWARE_BLANK, rom_desc)) != 0) {
8        ...
9        if (!vheader) { /* 1 */
10          dev_err(dev, "%s - out of memory.\n", __func__);
11          kfree(header);
12          kfree(rom_desc);
13          kfree(ti_manuf_desc);
14          return −ENOMEM;
15       }
16       ...
17       if (status) { /* 2 */
18          kfree(vheader);
19          kfree(header);
20          kfree(rom_desc);
21          kfree(ti_manuf_desc);
22          return status;
23       }
24       ...
25       if (status) { /* 3 */
26          dbg("%s - can't read header back", __func__);
27          kfree(vheader);
28          kfree(header);
29          kfree(rom_desc);
30          kfree(ti_manuf_desc);
31          return status;
32       }
33       ...
34       if (status) { /* 4 */
35          dev_err(dev,"%s - UMPC_COPY_DNLD_TO_I2C failed\n",...);
36          kfree(rom_desc);
37          kfree(ti_manuf_desc);
38          return status;
39       }
40    }
41    ...
42    stayinbootmode:
43       dbg("%s - STAYING IN BOOT MODE", __func__);
44       serial−>product_info.TiMode = TI_MODE_BOOT;
45       return 0;
46 }
```

Figure 12. Example for transformation. (Linux-2.6.34/drivers/usb/-serial/io_ti.c)

```
1   static int download_fw(struct edgeport_serial *serial) {
2     ...
3     if (serial−>product_info.TiMode == TI_MODE_DOWNLOAD) {
4        ...
5     }
6     else if ((start_address = get_descriptor_addr(serial,
7        I2C_DESC_TYPE_FIRMWARE_BLANK, rom_desc)) != 0) {
8        ...
9        if (!vheader) {
10          status = −ENOMEM;
11          dev_err(dev, "%s - out of memory.\n", __func__);
12          goto out2;
13       }
14       ...
15       if (status)
16          goto out1;
17       ...
18       if (status) {
19          dbg("%s - can't read header back", __func__);
20          goto out1;
21       }
22       ...
23       if (status) {
24          dev_err(dev,"%s - UMPC_COPY_DNLD_TO_I2C failed\n",...);
25          goto out;
26       }
27    }
28    ...
29    stayinbootmode:
30       dbg("%s - STAYING IN BOOT MODE", __func__);
31       serial−>product_info.TiMode = TI_MODE_BOOT;
32       return 0;
33    out1:
34       kfree(vheader);
35    out2:
36       kfree(header);
37    out:
38       kfree(rom_desc);
39       kfree(ti_manuf_desc);
40       return status;
41 }
```

Figure 13. Transformed version of Figure 12.

using the basic strategy, quantifies the reasons why functions are not transformed, and indicates the number of functions that are partially affected and unaffected by the filtering process. Overall 29% of the functions are fully or partially transformed.

Filtering discards an `if` branch either 1) because its label code contains only a `return` statement or 2) because its label code ends with state restoring code that is not shared with that of any other label code. Columns 3 to 6 of Table 1 show the number of functions for which all of the `if` branches are filtered due to these reasons. Label code may contain only a return statement due to the requirement in the partitioning process that state-restoring code only be added to the label code when there are no subsequent assignments or debugging statements. If an assignments or debugging code occurs just before the `return`, then the label code will not contain any state-restoring operations. The number of functions discarded for this reason is shown in the column "Strict". The lack of shared state-restoring code may occur because a function has only one error-handling `if`

("Single if"), because there are shared state-restoring operations but they appear in the wrong order ("Unordered"), or because there are multiple `if` branches with state-restoring operations but none are shared ("No sharing"). The latter two cases may indicate bugs, and are thus reported to the user for further inspection.

All of the error-handling code in a function may be transformed, or some of the `if` branches may be filtered and thus only a portion of the error-handling code is transformed. Table 1 shows that, depending on the directory, between 16 and 36% of the functions are fully transformed and 1 to 4% more are partially transformed. Overall, 29% of the functions that contain some error-handling code structured according to the basic strategy are transformed.

Branch classification   Transforming a Simple branch replaces the branch's label code by a single `goto`, which is a best case in terms of the reduction in code size. Transforming a Hard branch achieves similar improvement, as it requires only adding a new label. Both the Simple and Hard cases may also introduce an assignment for the return variable. Transforming a Harder `if` branch in itself increases code size, because the label code is copied to the end of the function as is, and a `goto`, a label and possibly an assignment must be introduced. Nevertheless, the filtering process guarantees that at least part of the copied code for a Harder branch is shared with another branch. Finally, transformation of a Hardest branch copies some code and introduces an extra `goto`, but part of its code is again shared with some other branch.

Table 2 shows the number of `if` branches in each category in the various directories, as well as the percentage in each category as compared to the total number of transformed `if` branches in the given directory. In many of the directories, the percentages in all of the categories are roughly similar. Exceptions are `drivers` and `sound`, which have a higher proportion of Simple branches. `arch` has a relatively low proportion of Hardest branches, while `sound` has a very low proportion of Hard branches. `sound` is a collection of sound-card drivers that were split off from the `drivers` directory early in the Linux 2.5 series, and it may be that they have followed a different development pattern than the rest of the code. Finally, the smaller directories ("others") have a relatively low proportion of Simple branches and a higher proportion of Hard and Hardest branches than the other directories.

		Simple	Hard	Harder	Hardest Simple	Hardest Hard	Total
drivers	number	867	630	895	62	29	2483
	%	35	25	36	3	1	100
fs	number	105	111	201	13	9	439
	%	24	25	46	3	2	100
net	number	51	64	83	7	4	209
	%	24	31	40	3	2	100
arch	number	42	43	75	1	4	165
	%	25.5	26	45	1	2	100
sound	number	339	63	335	36	3	776
	%	44	8	43	5	0.5	100
others	number	19	23	46	3	1	92
	%	21	25	50	3	1	100
total	number	1423	934	1635	122	50	4164
	%	34	22	39.5	3	1	100

Table 2. Number and % of Simple, Hard, Harder and Hardest branches of transformed branches in Linux-2.6.34. Hardest branches have a suffix of their state-restoring code that is Simple or Hard.

Branch transformation   Table 3 shows the number of branches (*cf* column "Total" in Table 2) and the number of labels, `goto`s, and return variable initialization statements ("assignment" column) introduced by the transformation. The transformation of a given

branch introduces at most two labels and two gotos (Hardest case), and at most one assignment. A Simple branch, however, introduces no labels, and a Hard or Harder branch introduces only one. The information in this table thus presents another perspective on the number of branches in the various categories (Table 2).

Table 3 shows that the number of labels introduced is always significantly lower than the number of branches, reflecting a good number of Simple branches. The `sound` directory has a particularly high ratio of branches to labels created, reflecting the high rate of Simple branches in this case. The number of `goto`s introduced is overall slightly higher than the number of branches, because each branch introduces at least one `goto` and Hardest branches may introduce two. The number of branches requiring two `goto`s is however small (*cf*, Hardest-Hard case in Table 2). Finally, the number of return variable initializations is also much lower than the number of branches, and for many directories it is lower than the number of labels, indicating that the need to introduce a return variable to abstract over error indicators is not a major burden.

		Branches	Label	Goto	Assignment
drivers	number	2483	1645	2512	1879
	avg per function		1.52	2.32	1.74
fs	number	439	343	448	307
	avg per function		1.45	1.90	1.3
net	number	209	162	213	167
	avg per function		1.45	1.90	1.49
arch	number	165	127	169	120
	avg per function		1.63	2.16	1.54
sound	number	776	440	779	366
	avg per function		1.77	3.14	1.48
others	number	92	74	93	66
	avg per function		1.45	1.82	1.29
total	number	4164	2791	4214	1905
	avg per function		1.54	2.33	1.61

Table 3. Total number and average number of label, goto, assignment created in the transformed Linux-2.6.34 functions.

Code sharing   The goal of the approach is to cause state-restoring code to be shared, to improve robustness in the face of maintenance and to reduce code size. Thus, we should ideally not just have many Simple and Hard branches, but these branches should also contain a good number of state-restoring operations that can then be shared. On the other hand, Harder branches, and to some extent Hardest branches, simply move existing code. Table 4 shows the number of merged and moved lines of code. In most directories, at least 45% more code is merged than moved.

Another perspective on the same information is to consider how many functions have only Simple and Hard branches, implying that the state-restoring operations are already available at the end of the function, how many have one Harder branch and then only Simple and Hard branches, implying that all of the state-restoring code is being shared, and how many have multiple Harder branches or have Hardest branches, implying the need for extra blocks of state-restoring code at the end of the function. As shown in Table 5, across all directories, most functions have only one Harder branch, and then the rest of the branches are Simple or Hard. A large number of functions also have only Simple and Hard branches, providing the best case for reduction in code size. Finally, only a few functions have multiple Harder branches or have Hardest branches. These results show that overall the Linux code is well suited to the use of the `goto`-based strategy.

5. Related Work

A number of studies have classified the kinds of exceptions that can occur, the kind of exception handling that is required, and the kinds

		Merged	Moved	Merged/ Moved	Total Transformed
drivers	number	3581	2074	1.73	5655
	avg per function	3.31	1.91		5.22
fs	number	608	420	1.45	1028
	avg per function	2.58	1.78		4.36
net	number	281	179	1.57	460
	avg per function	2.51	1.60		4.11
arch	number	219	150	1.46	469
	avg per function	2.81	1.92		4.73
sound	number	1017	633	1.61	1650
	avg per function	4.1	2.55		6.65
others	number	113	90	1.26	203
	avg per function	2.21	1.76		3.98
total	number	5819	3546	1.64	9365
	avg per function	3.22	1.96		5.18

Table 4. Total number and average number of lines that are merged, moved, and transformed in the transformed functions.

Harder	Hardest	drivers	fs	net	arch	sound	others	total
0	0	303	59	35	24	26	12	459
1	0	657	146	61	42	183	33	1122
>1	0	64	13	6	8	8	3	102
≥0	≥1	58	18	10	4	31	3	124

Table 5. The number of functions with various numbers of Harder and Hardest branches.

of exception handling abstractions that are provided by current programming languages, as well as proposing new exception handling abstractions [3, 7, 8]. We consider exceptions that simply abort the subcomputation, and our proposed improvement to exception handling stays within the constructs available in C. Our work addresses the issues of readability and uniformity, which these studies have identified as critical.

Bruntink *et al.* study properties of exception handling in a large industrial C code base [2]. They focus on the detection and logging of exceptional conditions, while we focus on the associated cleanup code. Filho *et al.* present a technique to transform the exception handling code of a Java program into an aspect [5], providing modularity and reuse. Mortensen and Ghosh apply aspects to convert code that uses return codes, as done in Linux, to use C++ exception handling abstractions, to ensure that all exceptions are handled [10]. Bruntink performs a similar study on C code, using hypothetical `try` and `catch` constructs [1]. C does not provide any such abstractions. Our work is concerned with improving the structure of the error handling code that is present, potentially helping the user find problems in the usage of the state-restoring operations, rather than ensuring that all error conditions are checked for.

Weimer and Necula present a static data flow analysis on exception handling code for finding bugs [14]. Their flow-sensitive analysis found over 1300 defects in over 5 million lines of Java code. Their results suggest that improper management of error handling code introduces bugs in a system. They propose a programming language feature to help programmers avoid such mistakes. We do not propose new programming language features, but instead show how to restructure error-handling code to make it less error-prone.

Our transformation can be considered to be a form of refactoring, since it changes the structure, but not the semantics of the code [6]. Few tools support refactoring C code. Eclipse provides the CDT development environment for C and C++ code, but the support for refactoring seems to be incomplete [4]. McCloskey and Brewer propose Asfact for refactoring C code, but have only considered refactorings that involve changes of function names [9].

6. Conclusion

This paper has focused on the structuring of error handling code in the Linux kernel. The Linux kernel coding style guidelines advocate organizing such code using labels and `gotos`, but a substantial part of the Linux kernel source code still does not follow this strategy. We have proposed an automatic transformation that converts error-handling code that is dispersed and duplicated throughout the body of a function such that it uses the `goto`-based strategy. We have found that our transformation applies to many functions across the Linux kernel, and that it identifies many opportunities for code sharing. This includes code in the `drivers` and `arch` directories, which may be of particular interest to embedded systems.

Our algorithm only restructures the error-handling code that is provided; it does not fix defects such as missing or misplaced state-restoring operations. Such defects, however, can cause bugs such as deadlocks and memory leaks, when resources are not deallocated correctly. In future work, we will consider these issues. We will also consider the applicability of our approach to other software.

References

[1] M. Bruntink. Reengineering idiomatic exception handling in legacy C code. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 133–142, Athens, Greece, Apr. 2008.

[2] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *28th International Conference on Software Engineering (ICSE)*, pages 242–251, Shanghai, China, May 2006.

[3] P. A. Buhr and W. Y. R. Mok. Advanced exception handling mechanisms. *IEEE Trans. Software Eng.*, 26(9):820–836, 2000.

[4] CDT. CDT/User/FAQ – Eclipsepedia, 2010. http://wiki.eclipse.org/CDT/User/FAQ.

[5] F. C. Filho, C. M. F. Rubira, R. de A. Maranhão Ferreira, and A. Garcia. Aspectizing exception handling: A quantitative study. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 255–274. Springer, 2006.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[7] A. F. Garcia, C. M. F. Rubira, A. B. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[8] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[9] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. In *ESEC/FSE-13*, pages 21–30, Lisbon, Portugal, 2005.

[10] M. Mortensen and S. Ghosh. Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects. In *Industry Track of the International Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver, Canada, Mar. 2007.

[11] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Compiler Construction (CC'09)*, pages 109–125, York, UK, Mar. 2009.

[12] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[13] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, USA, Mar. 2011.

[14] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.

[15] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.