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

Suman Saha[1], Julia Lawall[123], and Gilles Muller[13]
LIP6-Regal[1]/DIKU[2]/INRIA[3]

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche PARIS - ROCQUENCOURT

Linux is Future of Embedded System

- Linux is used in Mobile phones, PDAs and many more electronic devices.
- Android (modified version of Linux kernel) has become major competitor of Symbian OS
- During third quarter of 2010, 25.5% of smartphones sold, used Android
- Reliability of code used in embedded systems is critical.
  - Handling transient run-time errors

Error Handling Code

- Error Handling code handles exceptions.
  - Returns the system to a coherent state.

```
static int __init reipl_init(void) {
    …
    reipl_kset = kset_create_and_add(...)
    if (rc) {
        kset_unregister(reipl_kset);
        return rc;
    }
    rc = reipl_ccw_init();
    if (rc)
        return rc;
    …
}
```

File name: arch/s390/kernel/ipl.c

- Key to ensuring reliability
- Mistakes cause deadlock and memory leaks

Error Handling strategies in Linux

A typical and initial way to write error code

basic-strategy

```
…

if(!y) {
  free(x);
  return -ENOMEM;
}
...
if(!z) {
  free(x);
  return -ENOMEM;
}
...
```

Problems

- Duplicates code

Error Handling strategies in Linux

A typical and initial way to write error code

basic-strategy

```
if(!y) {
   free(x);
   return -ENOMEM;
}
if(!m) {
   free(x);
   return -ENOMEM;
}
...
if(!z) {
   free(x);
   return -ENOMEM;
}
...
```

Problems

- Duplicates code
- Obscures what error handling code to use for new operations

Error Handling strategies in Linux

A typical and initial way to write error code

basic-strategy

```
if(!y) {
  free(n);
  free(x);
  return -ENOMEM;
}
if(!m){
  free(n);
  free(x);
  return -ENOMEM;
}
...
if(!z) {
  free(n);
  free(x);
  return -ENOMEM;
}
```

Problems

- Duplicates code
- Obscures what error handling code to use for new operations
- Requires updating a lot of existing error handling code when adding a new operation.

Error Handling strategies in Linux

Goto-based strategy

```
...
if(!y)
   goto out;
...
if(!z)
   goto out;
…
out:
   free(x);
   return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function.
- No code duplication

Error Handling strategies in Linux

Goto-based strategy

```
...
if(!y)
   goto out;
…
if(!m)
   goto out;
if(!z)
   goto out;
…
out:
   free(x);
   return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function.
- No code duplication

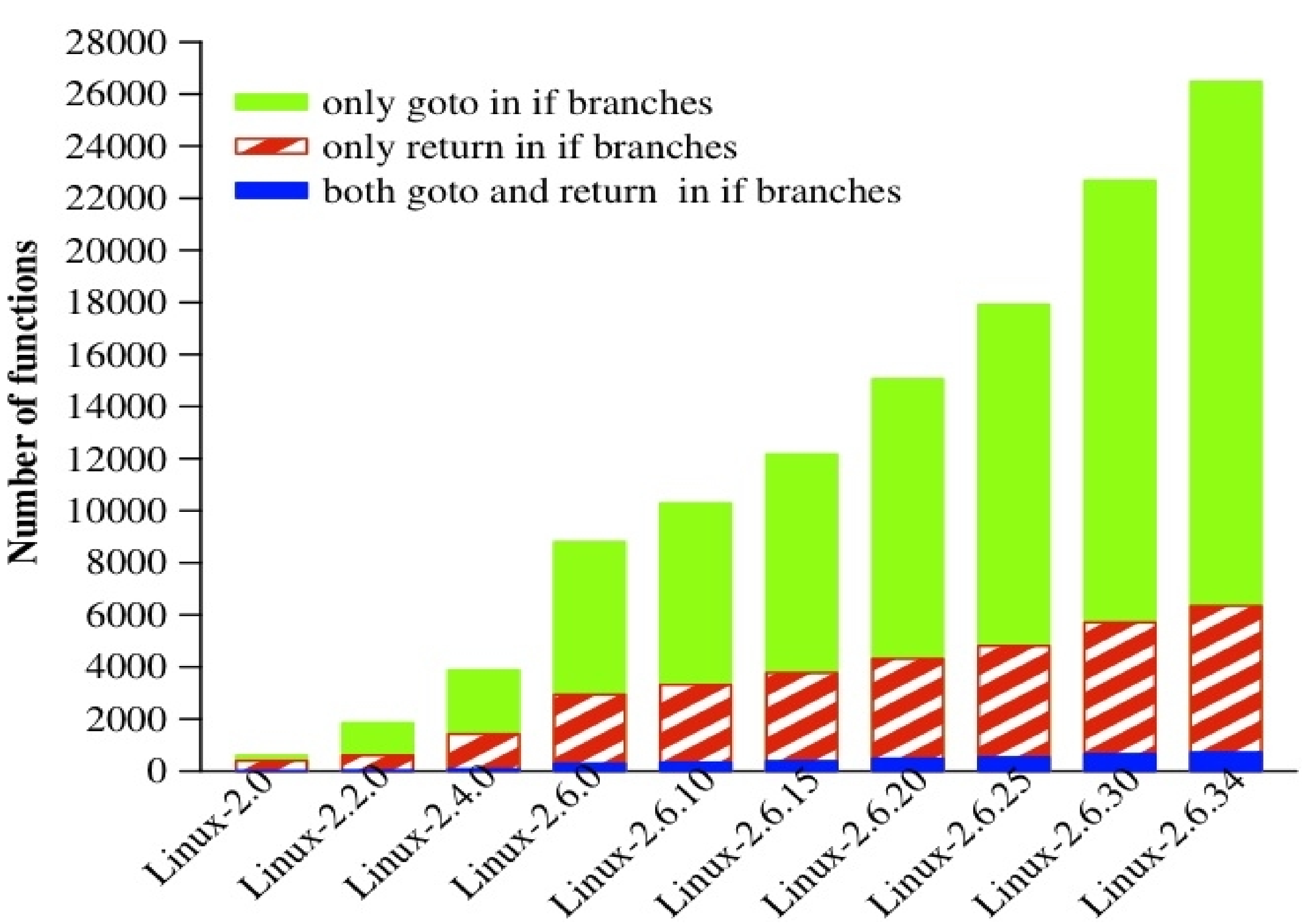
Error Handling strategies in Linux

Goto-based strategy

```
...
if(!y)
   goto out;
…
if(!m)
   goto out;
if(!z)
   goto out;
…
out:
   free(n);
   free(x);
   return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function.
- No code duplication

Goto-based strategy VS Basic strategy



Our Goal: Convert basic strategy code to use the goto strategy.

Three steps:

1. Find error handling code
2. Identify operations for sharing
3. Transformation

1. Find Error Handling Code

- No recognizable error handling abstractions in C code.
- Heuristics:
  - An if branch ending in a return
  - An if branch containing at least one non-debugging function call (something to share)

Examples:

✗ / ✓

```
if(ns->bacct == NULL){
    ...
    if(acct == NULL){
      filp_clode(file, NULL);
      return -ENOMEM;
    }
}
```

✗

```
if(ns->bacct == NULL){
    ns->bacct = acct;
    acct = NULL;
}
```

2. Identify Operations for Sharing

For each branch

- Extract the code that is specific to the error condition
- Extract the code that can be shared with other error handling code.

```
if(x) {
	ret = -ENOMEM;
	k(c);
	h("s");
	f(a);
	return ret;
}
```

```
if(x) {
	ret = -ENOMEM;
	k(c);
	h("s");
	goto out;
}

out:
	f(a);
	return ret;
```

2. Identify Operations for Sharing

Drop some cases where no sharing can be introduced.

No state restoring operations.

```
if(x) {
      ret = -ENOMEM;
      k(c);
      f(a);
      h("s");
      return ret;
}
```

```
if(x) {
      ret = -ENOMEM;
      k(c);
      f(a);
      h("s");
      goto out;
}

out:
      return ret;
```

2. Identify Operations for Sharing

Drop some cases where no sharing can be introduced.

Only one branch to transform

```
f(){
    ...
    x = allocate();
    …
    y = noallocate();
    if(y) {
      ...
      free(x);
      return ret;
    }
}
```

2. Identify Operations for Sharing

Drop some cases where no sharing can be introduced.

Nothing in common with other error handling code

```
…
if(y) {
  free(x);
  return ret;
}
...
if(z) {
  free(x);
  return ret;
}
...
free(x);
...
if(l) {
  action(z);
  return ret;
}
```

3. Transformation

Classify the branches according to how difficult they are to transform

Simple

Reuse an existing label

```
...
  if (!sl->data) {
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Simple

```
...
  if (!sl->data) {
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

→

```
...
  if (!sl->data)
    goto out;

...
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Hard

- Need to insert a new label.
- No code is moved.

```
...
  if (!sl->data) {
    unlock_kernel();
    return ret;
  }
...
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Hard

```
...
  if (!sl->data) {
    unlock_kernel();
    return ret;
  }
…
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

→

```
...
  if (!sl->data)
    goto out1;

…
out:
    clear_bit(n,sbi-symlink_bitmap);
out1:
    unlock_kernel();
    return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Harder

- Create a new label.
- Move code to that label

```
...
  if (!sl->data) {
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
  if (!ent) {
    kfree(sl->data);
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
```

3. Transformation

Classify the branches according to how difficult they are to transform

Harder

```
...
  if (!sl->data) {
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
  if (!ent) {
    kfree(sl->data);
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
```

→

```
...
  if (!sl->data) {
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
  }
...
  if (!ent)
    goto out;
...
out:
    kfree(sl->data);
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Hardest

Combination of Harder and Simple

```
...
  if (!ent){
     kfree(sl->data);
     clear_bit(n,sbi-symlink_bitmap);
     unlock_kernel();
     return ret;
  }
…
return 0;
out:
     clear_bit(n,sbi-symlink_bitmap);
     unlock_kernel();
     return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Hardest

```
...
  if (!ent){
     kfree(sl->data);
     clear_bit(n,sbi-symlink_bitmap);
     unlock_kernel();
     return ret;
  }
…
return 0;
out:
     clear_bit(n,sbi-symlink_bitmap);
     unlock_kernel();
     return ret;
```

→

```
...
   if (!ent)
     goto out1;
…
return 0;
out1:
     kfree(sl->data);
out:
     clear_bit(n,sbi-symlink_bitmap);
     unlock_kernel();
     return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Hardest

Combination of Harder and Hard

```
...
  if (!ent)
    kfree(sl->data);
    unlock_kernel();
    return ret;
…
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

3. Transformation

Classify the branches according to how difficult they are to transform

Hardest

```
...
  if (!ent)
    kfree(sl->data);
    unlock_kernel();
    return ret;
…
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

→

```
...
  if (!ent)
   goto out1;
…
out:
    clear_bit(n,sbi-symlink_bitmap);
out2:
    unlock_kernel();
    return ret;
out1:
    kfree(sl->data);
    goto out2;
```

Evaluation

- Implemented as 1300 lines of OCaml code (excl. parser)
- Applied to Linux 2.6.34 kernel (8 million LOC).
- 25 minutes on one core of an 8-core 3GHz machine with 16GB memory

Results

- 59% of basic strategy functions have only single *if*. So, those are not transformed.
- 12% of basic strategy functions are not transformed due to lack of sharing.
- 29% of basic strategy functions transformed to the goto strategy.
  - 2% are partially transformed
  - 27% are fully transformed

Conclusion and Future work

We proposed an automatic transformation that converts basic strategy into goto-based strategy of error handling code.

- The algorithm identifies many opportunities for code sharing.

Future Work

Find and fix defects such as missing or misplaced state restoring operations in error handling code

Thank You