# Introduce New Branch Tracer 'perf branch'

Akihiro Nagai

Linux Technology Center,

Yokohama Research Lab.,

Hitachi Ltd.

## Introduce myself

- I’m working at Linux Technology Center of Yokohama Research Lab in Hitachi Ltd.,

- I’m interested in
  - Automated software testing
  - Performance analysis
  - Debugging tools ...etc

## Agenda

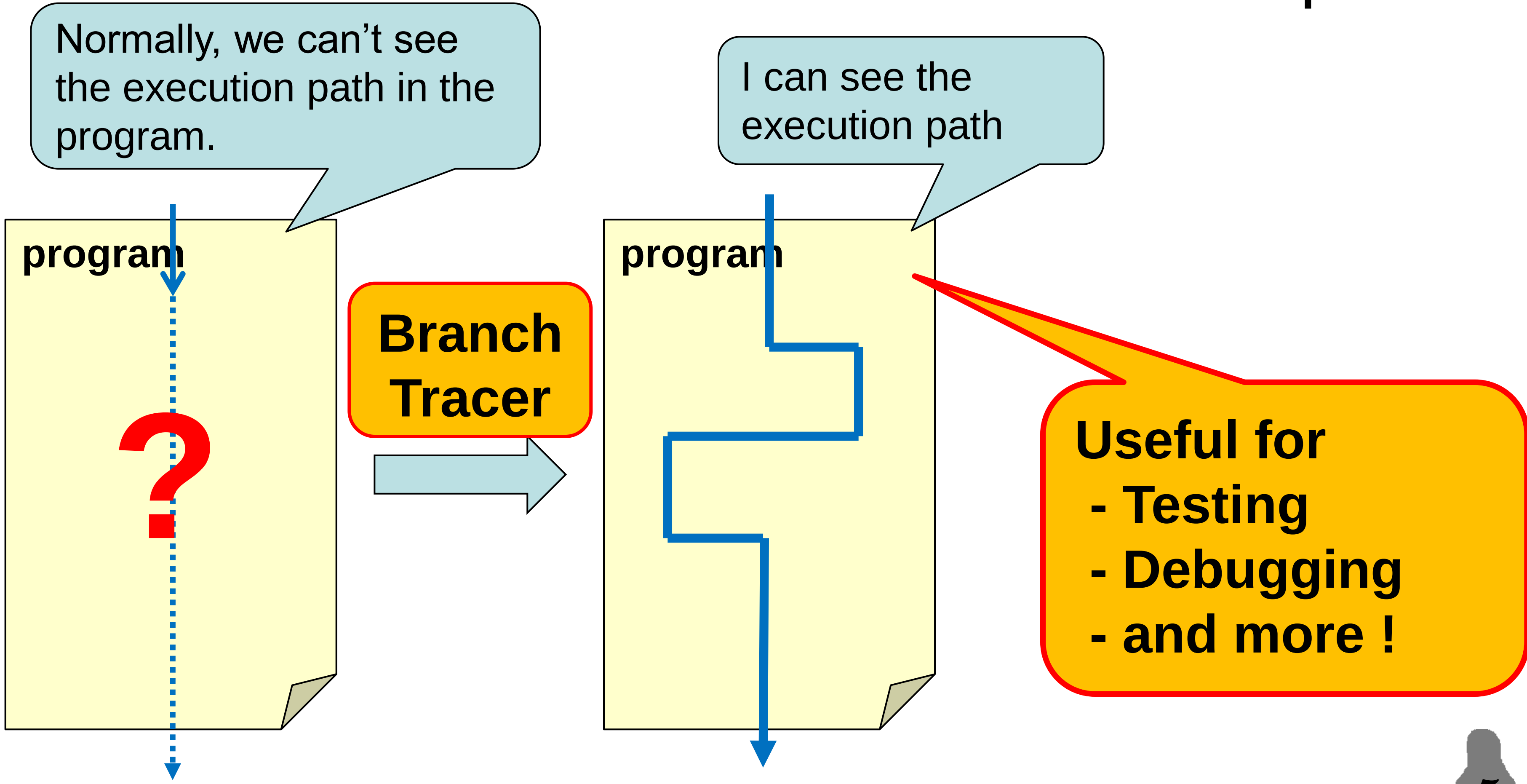
- Background
- perf branch
- Implementation
- Btrax
- Merge plan
- Future plan & Conclusion

## Agenda

- Background
- perf branch
- Implementation
- Btrax
- Merge plan
- Future plan & Conclusion

## Branch Tracer

- ‘perf branch’ is a branch tracer
- Branch tracer can record the execution path


Normally, we can’t see the execution path in the program.
program
?
Branch Tracer
I can see the execution path
program
Useful for
- Testing
- Debugging
- and more !

## Motivation

- Modern processors have HW-based branch tracer
- It’s very useful and interesting function
  - It can be applied in development tools.
    - For example, testing and debugging tools.
- However, there is no way to use it easily in Linux

⇒**perf branch provides the interface to use HW-based branch tracer**

## Agenda

- Background
- perf branch
- Implementation
- Btrax
- Merge plan
- Future plan & Conclusion

## perf

- ‘perf branch’ is implemented as a part of perf

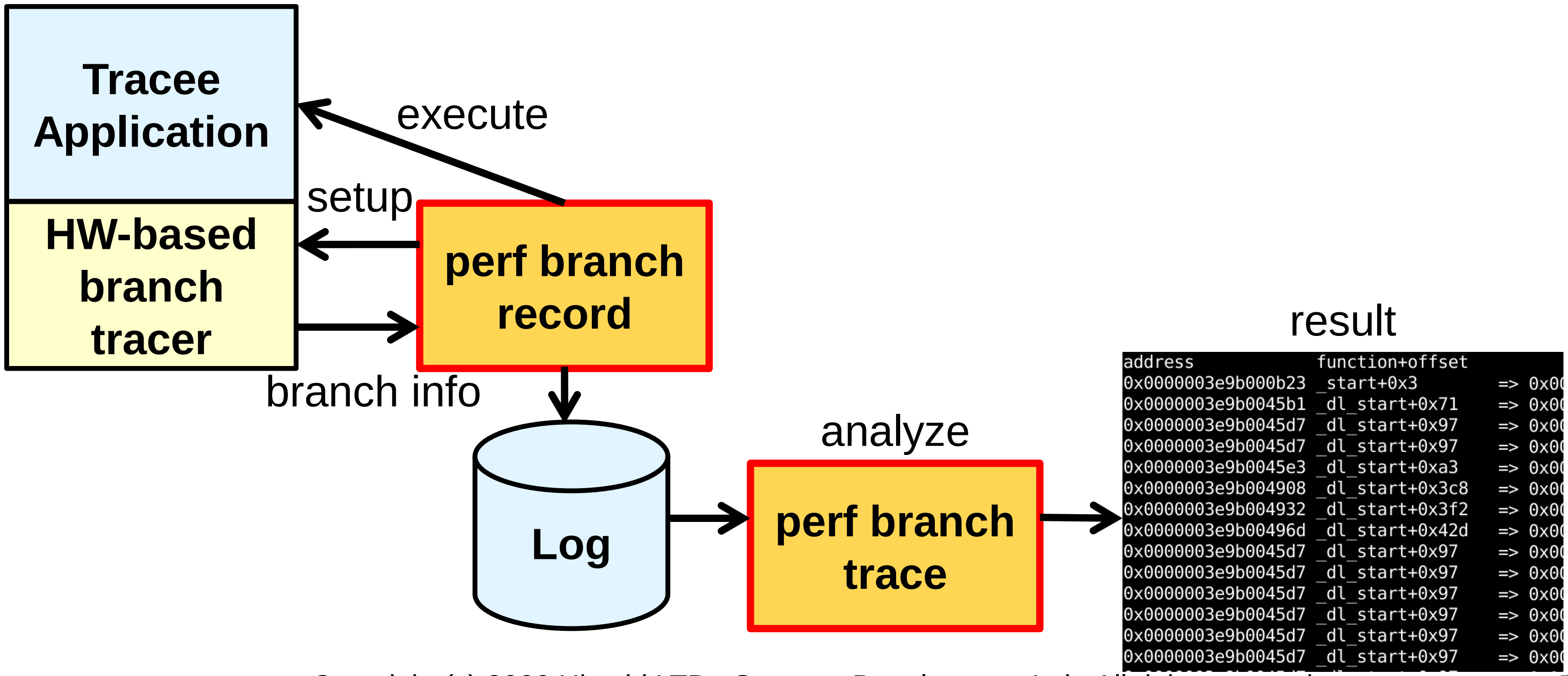
- ‘perf’ is a subsystem of Linux
  - Processors’ performance monitoring facilities
    - HW-based branch tracer is one of them
  - Trace points

## ‘perf branch’

- Provide the easy way to use HW-based branch tracer
- Record and show the execution path of executed programs

# Overview of perf branch

- perf branch consists of two parts
  - perf branch record: recording branches
  - perf branch trace: analyzing recorded log

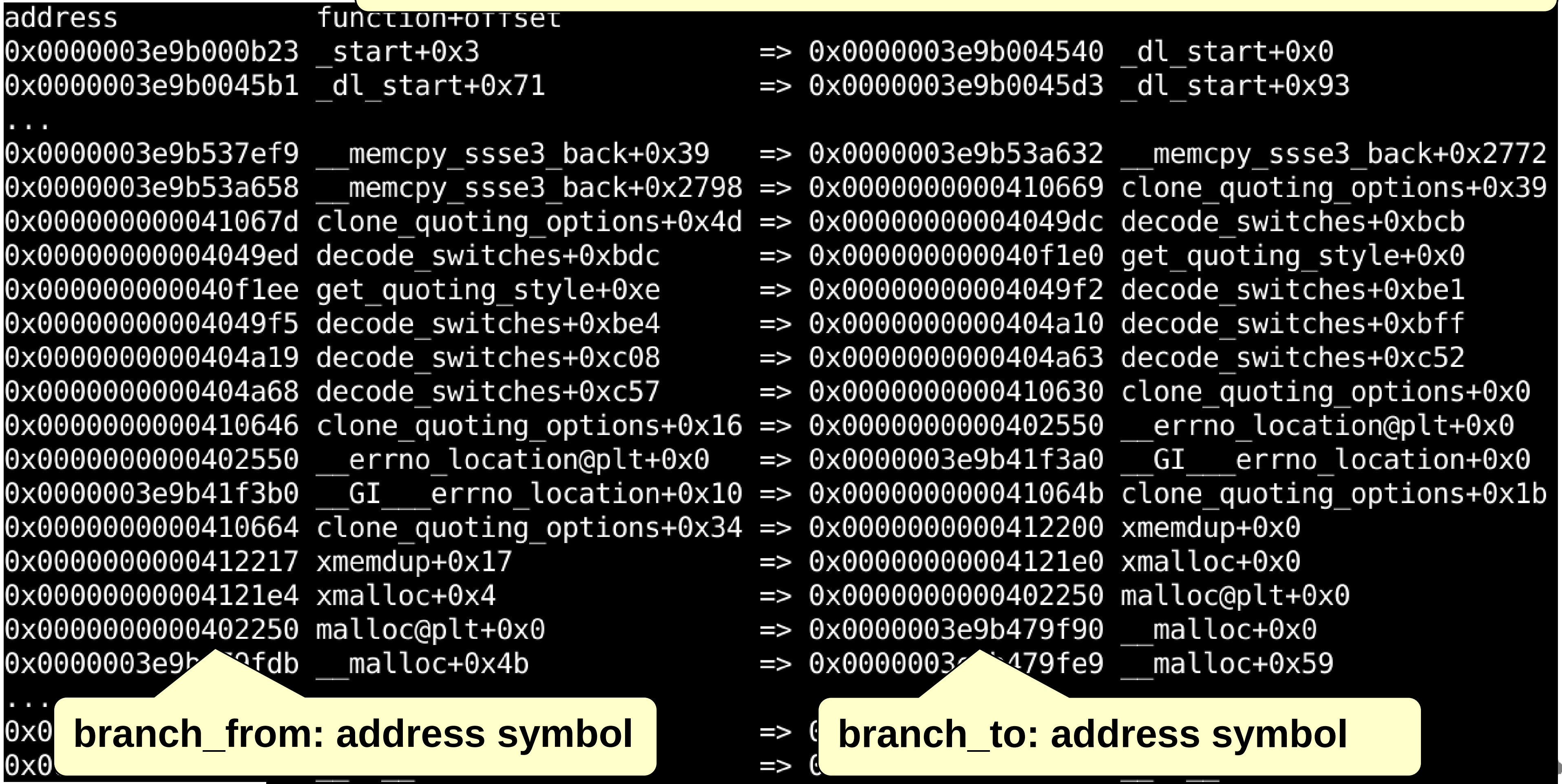


```
address            function+offset
0x0000003e9b000b23 _start+0x3        => 0x00
0x0000003e9b0045b1 _dl_start+0x71    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045e3 _dl_start+0xa3    => 0x00
0x0000003e9b004908 _dl_start+0x3c8   => 0x00
0x0000003e9b004932 _dl_start+0x3f2   => 0x00
0x0000003e9b00496d _dl_start+0x42d   => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
0x0000003e9b0045d7 _dl_start+0x97    => 0x00
```

## Output sample of perf branch

```
# perf branch record ls
# perf branch -as trace
```

specify to display **address** and **symbol** (function+offset)



```
address                function+offset
0x0000003e9b000b23 _start+0x3                  => 0x0000003e9b004540 _dl_start+0x0
0x0000003e9b0045b1 _dl_start+0x71              => 0x0000003e9b0045d3 _dl_start+0x93
...
0x0000003e9b537ef9 __memcpy_ssse3_back+0x39    => 0x0000003e9b53a632 __memcpy_ssse3_back+0x2772
0x0000003e9b53a658 __memcpy_ssse3_back+0x2798 => 0x0000000000410669 clone_quoting_options+0x39
0x000000000041067d clone_quoting_options+0x4d => 0x00000000004049dc decode_switches+0xbcb
0x00000000004049ed decode_switches+0xbdc       => 0x000000000040f1e0 get_quoting_style+0x0
0x000000000040f1ee get_quoting_style+0xe       => 0x00000000004049f2 decode_switches+0xbe1
0x00000000004049f5 decode_switches+0xbe4       => 0x0000000000404a10 decode_switches+0xbff
0x0000000000404a19 decode_switches+0xc08       => 0x0000000000404a63 decode_switches+0xc52
0x0000000000404a68 decode_switches+0xc57       => 0x0000000000410630 clone_quoting_options+0x0
0x0000000000410646 clone_quoting_options+0x16 => 0x0000000000402550 __errno_location@plt+0x0
0x0000000000402550 __errno_location@plt+0x0    => 0x0000003e9b41f3a0 __GI___errno_location+0x0
0x0000003e9b41f3b0 __GI___errno_location+0x10 => 0x000000000041064b clone_quoting_options+0x1b
0x0000000000410664 clone_quoting_options+0x34 => 0x0000000000412200 xmemdup+0x0
0x0000000000412217 xmemdup+0x17                => 0x00000000004121e0 xmalloc+0x0
0x00000000004121e4 xmalloc+0x4                 => 0x0000000000402250 malloc@plt+0x0
0x0000000000402250 malloc@plt+0x0              => 0x0000003e9b479f90 __malloc+0x0
0x0000003e9b[illegible]fdb __malloc+0x4b        => 0x0000003[illegible]479fe9 __malloc+0x59
...
0x0                                            =>
0x0                                            =>
```

**branch_from: address symbol**

**branch_to: address symbol**

## Usage

**There are two steps to use perf branch**

**Recording Branches:**

```
# perf branch record <command>
```

record branch-log while specified command executing

**Analyzing Branches:**

```
# perf branch [options] trace
```

analyze and show the recorded branch log
options to show these items

- address
- command name
- pid
- filepath to executed binary
- function+offset

Default output is human-friendly.
It can output TSV: script-friendly format for external programs.

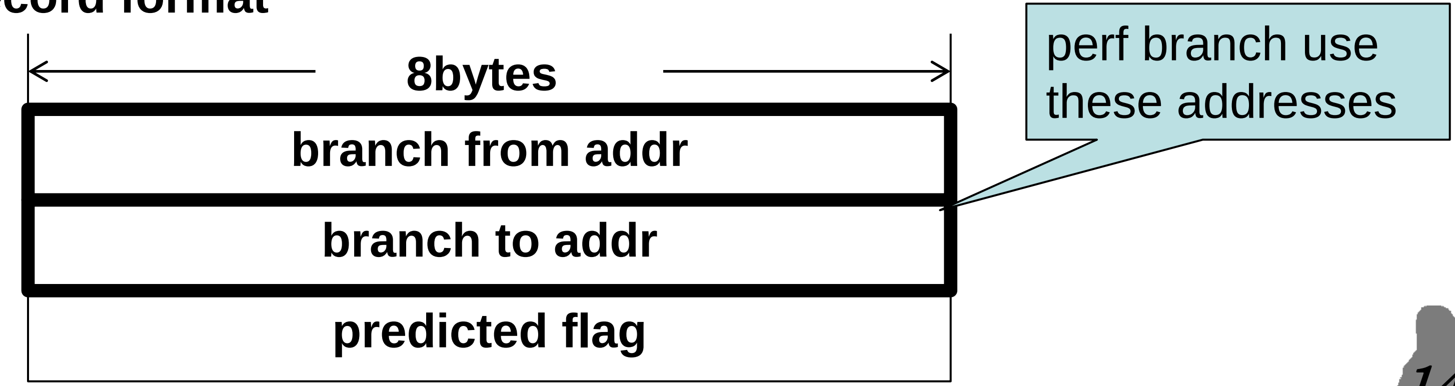
## Agenda

- Background
- perf branch
- Implementation
- Btrax
- Merge plan
- Future plan & Conclusion

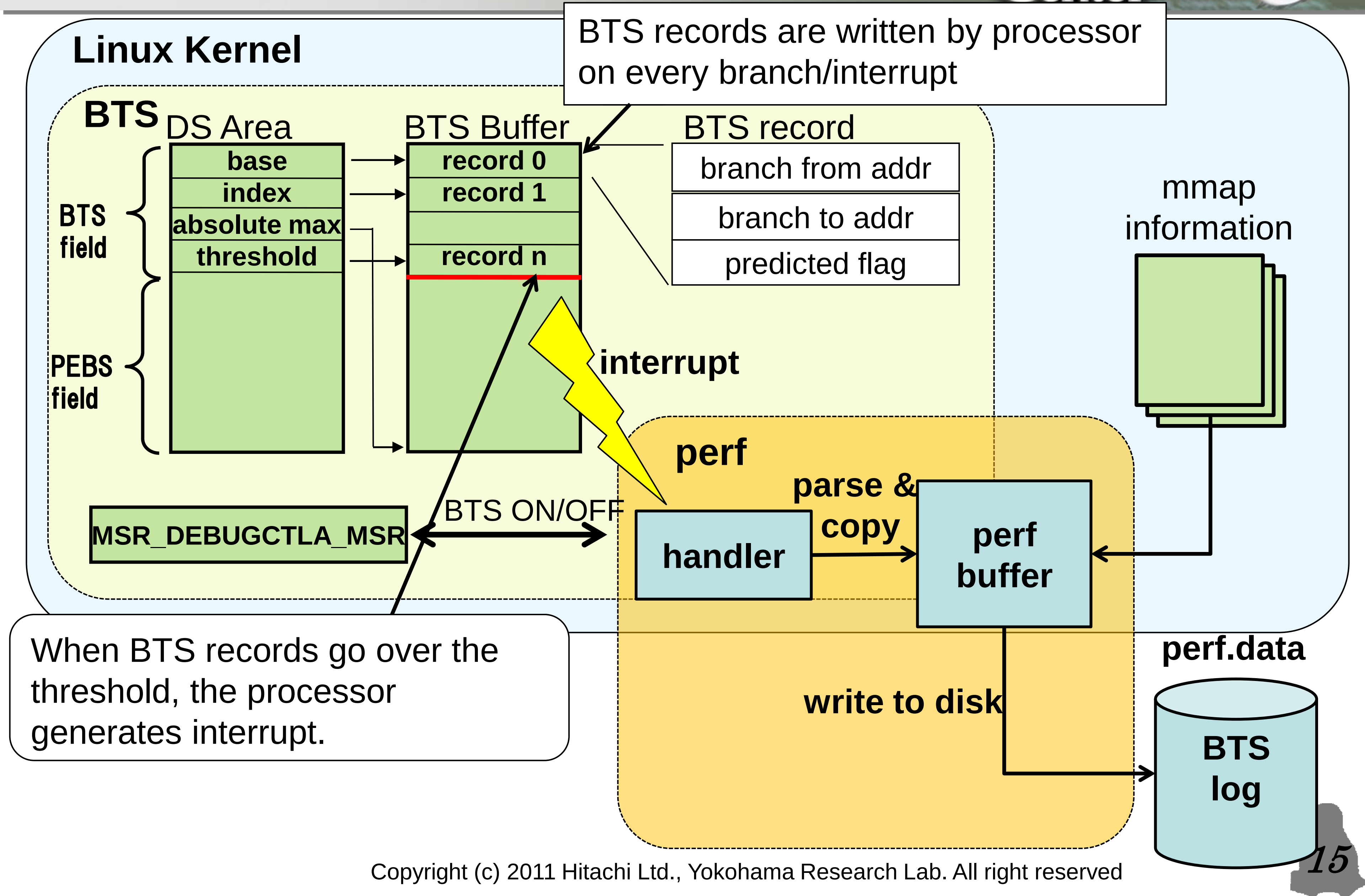
# BTS (Branch Trace Store)

- HW-based branch tracer ‘perf branch’ uses
- Record every branch/interrupt
  - Record as BTS record that includes addresses branch from/to and flag whether branch prediction succeed.
- Intel x86 processors’ facility
  - Available on Pentium4 or later processors

**BTS record format**



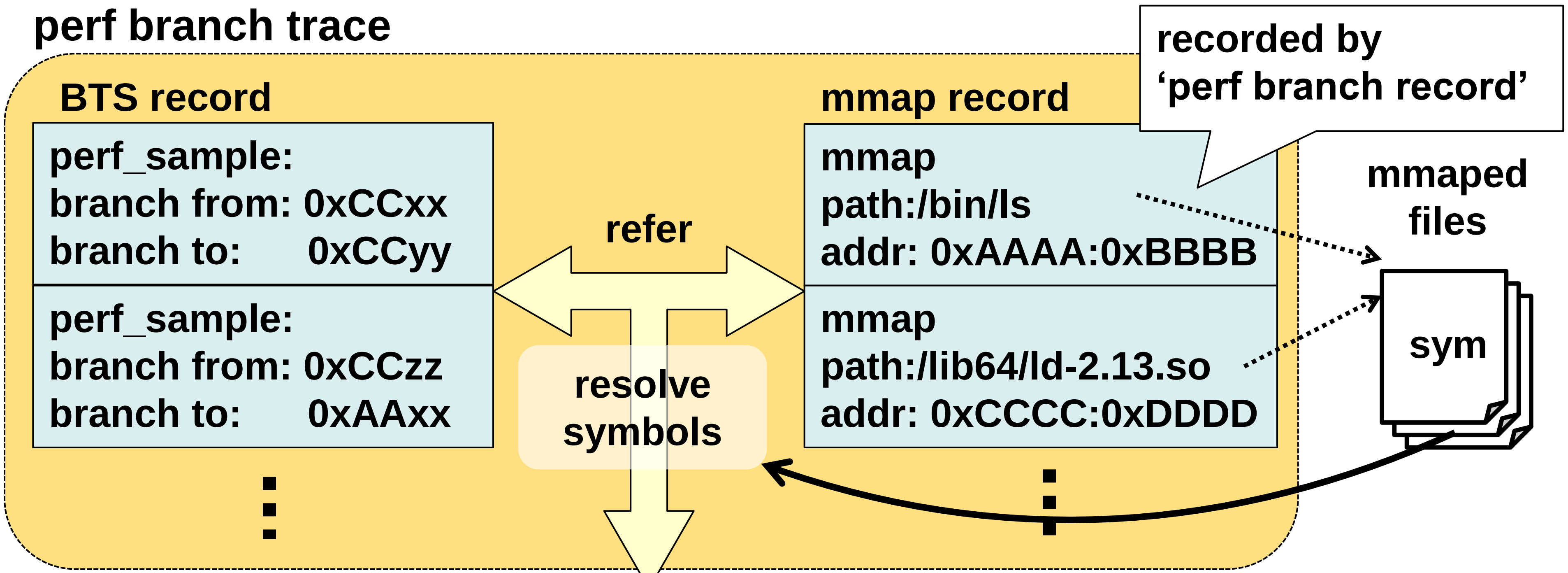
# Recording Phase

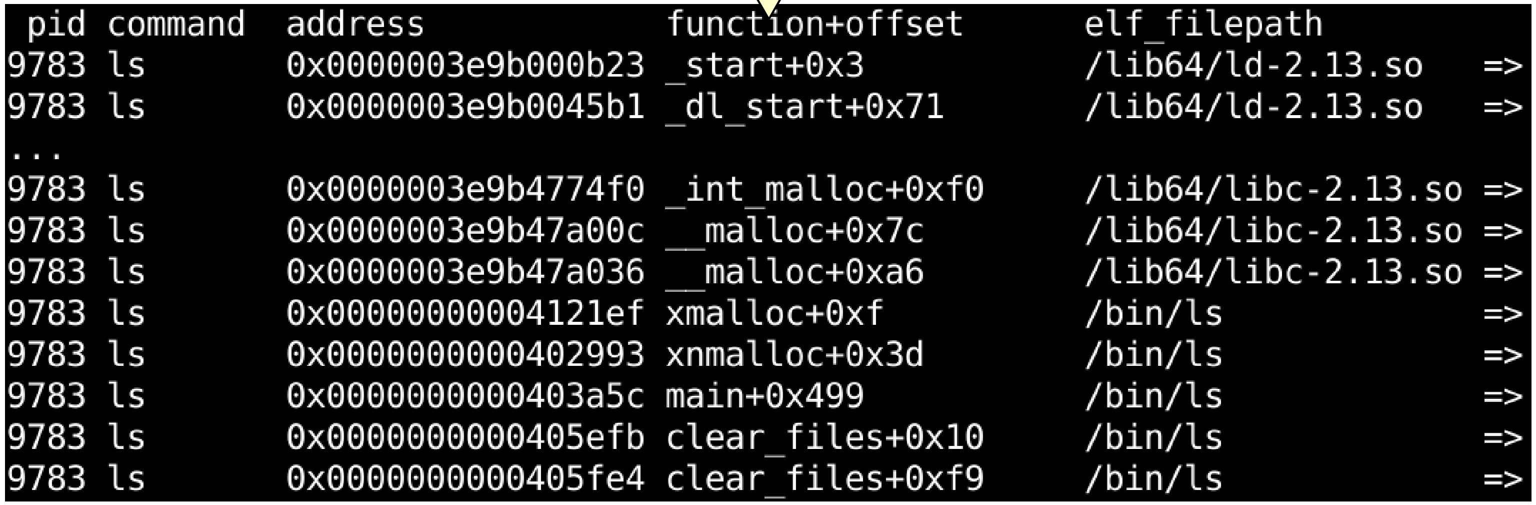


BTS records are written by processor on every branch/interrupt

When BTS records go over the threshold, the processor generates interrupt.

# Analysis Phase




```
 pid command   address            function+offset       elf_filepath
9783 ls        0x0000003e9b000b23 _start+0x3            /lib64/ld-2.13.so   =>
9783 ls        0x0000003e9b0045b1 _dl_start+0x71        /lib64/ld-2.13.so   =>
...
9783 ls        0x0000003e9b4774f0 _int_malloc+0xf0      /lib64/libc-2.13.so =>
9783 ls        0x0000003e9b47a00c __malloc+0x7c         /lib64/libc-2.13.so =>
9783 ls        0x0000003e9b47a036 __malloc+0xa6         /lib64/libc-2.13.so =>
9783 ls        0x00000000004121ef xmalloc+0xf           /bin/ls             =>
9783 ls        0x0000000000402993 xnmalloc+0x3d         /bin/ls             =>
9783 ls        0x0000000000403a5c main+0x499            /bin/ls             =>
9783 ls        0x0000000000405efb clear_files+0x10      /bin/ls             =>
9783 ls        0x0000000000405fe4 clear_files+0xf9      /bin/ls             =>
```

## Agenda

- Background
- perf branch
- Implementation
- Btrax
- Merge plan
- Future plan & Conclusion

# What & Why Btrax?

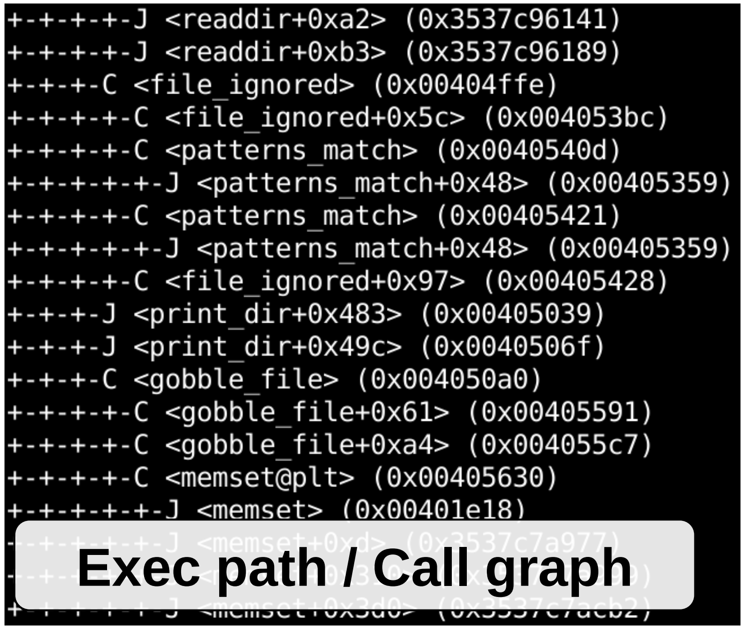
- Branch Tracer for Linux
- Previous project of ‘perf branch’
  - Btrax supports old kernel 2.6.9 – 2.6.30

- Btrax is an example of BTS application
  - Show execution path
  - Analyze code coverage
    - Code coverage means how many executed/unexecuted codes are in the program. It is sometimes used as test progress.
- Next perf branch’s enhancement point is Btrax’s functionality

⇒So, this chapter shows where ‘perf branch’ aims and, what BTS can do with concrete example Btrax.

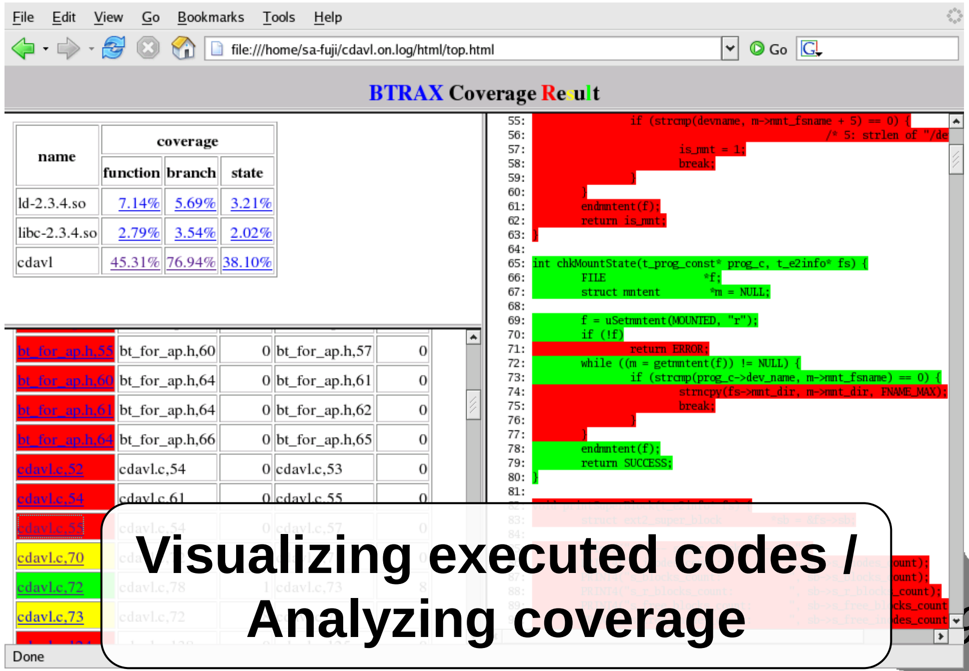
# Functions of Btrax

- Btrax has following functions
  - Showing execution path
  - Drawing call graph
  - Visualizing executed/unexecuted code
  - Analyzing code coverage



```
+-+-+-+-J <readdir+0xa2> (0x3537c96141)
+-+-+-+-J <readdir+0xb3> (0x3537c96189)
+-+-+-C <file_ignored> (0x00404ffe)
+-+-+-+-C <file_ignored+0x5c> (0x004053bc)
+-+-+-+-C <patterns_match> (0x0040540d)
+-+-+-+-+-J <patterns_match+0x48> (0x00405359)
+-+-+-+-C <patterns_match> (0x00405421)
+-+-+-+-+-J <patterns_match+0x48> (0x00405359)
+-+-+-+-C <file_ignored+0x97> (0x00405428)
+-+-+-J <print_dir+0x483> (0x00405039)
+-+-+-J <print_dir+0x49c> (0x0040506f)
+-+-+-C <gobble_file> (0x004050a0)
+-+-+-+-C <gobble_file+0x61> (0x00405591)
+-+-+-+-C <gobble_file+0xa4> (0x004055c7)
+-+-+-+-C <memset@plt> (0x00405630)
+-+-+-+-+-J <memset> (0x00401e18)
+-+-+-+-+-J <memset+0xd> (0x3537c7a977)
[illegible]
+-+-+-+-+-J <memset+0x3d0> (0x3537c7acb2)
```

Exec path / Call graph



BTRAX Coverage Result

| name | coverage | | |
|---|---|---|---|
| | function | branch | state |
| ld-2.3.4.so | 7.14% | 5.69% | 3.21% |
| libc-2.3.4.so | 2.79% | 3.54% | 2.02% |
| cdavl | 45.31% | 76.94% | 38.10% |

| | | | | |
|---|---|---|---|---|
| bt_for_ap.h,55 | bt_for_ap.h,60 | 0 | bt_for_ap.h,57 | 0 |
| bt_for_ap.h,60 | bt_for_ap.h,64 | 0 | bt_for_ap.h,61 | 0 |
| bt_for_ap.h,61 | bt_for_ap.h,64 | 0 | bt_for_ap.h,62 | 0 |
| bt_for_ap.h,64 | bt_for_ap.h,66 | 0 | bt_for_ap.h,65 | 0 |
| cdavl.c,52 | cdavl.c,54 | 0 | cdavl.c,53 | 0 |
| cdavl.c,54 | cdavl.c,61 | 0 | cdavl.c,55 | 0 |
| cdavl.c,55 | cdavl.c,54 | 0 | cdavl.c,57 | 0 |
| cdavl.c,70 | [illegible] | [illegible] | [illegible] | 0 |
| cdavl.c,72 | cdavl.c,78 | 1 | cdavl.c,73 | 8 |
| cdavl.c,73 | cdavl.c,72 | [illegible] | [illegible] | [illegible] |

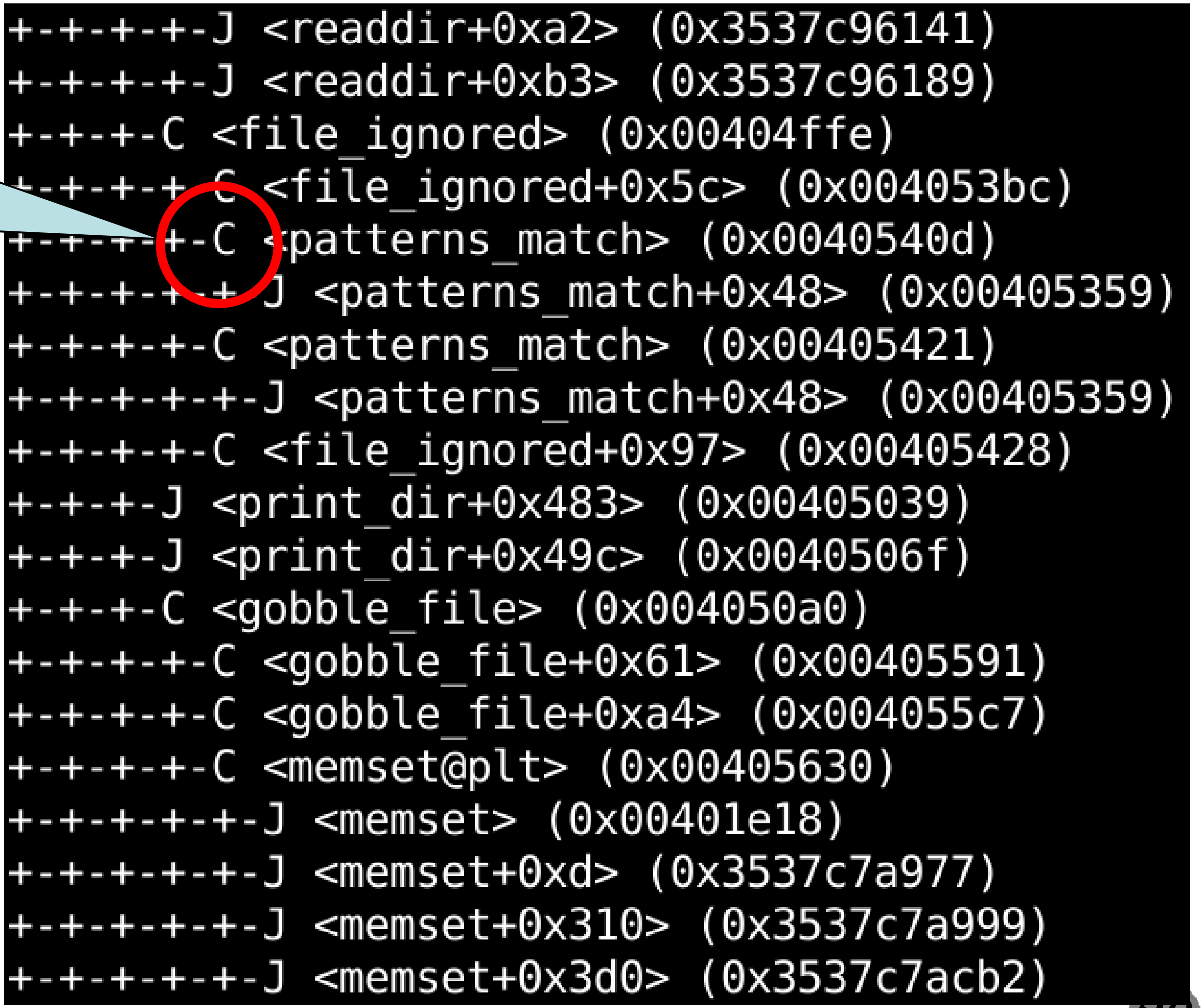
```
                if (strcmp(devname, m->mnt_fsname + 5) == 0) {
                                                /* 5: strlen of "/de
                        is_mnt = 1;
                        break;
                }
        }
        endmntent(f);
        return is_mnt;
}

int chkMountState(t_prog_const* prog_c, t_e2info* fs) {
        FILE                    *f;
        struct mntent           *m = NULL;

        f = uSetmntent(MOUNTED, "r");
        if (!f)
                return ERROR;
        while ((m = getmntent(f)) != NULL) {
                if (strcmp(prog_c->dev_name, m->mnt_fsname) == 0) {
                        strncpy(fs->mnt_dir, m->mnt_dir, FNAME_MAX);
                        break;
                }
        }
        endmntent(f);
        return SUCCESS;
}

void printSuperBlock(t_e2info* fs) {
        struct ext2_super_block         *sb = &fs->sb;
```

Visualizing executed codes / Analyzing coverage

## ScreenShot of Btrax

- Execution path and call graph

**C: Call**
**J: Jump**
**I: Interrupt**



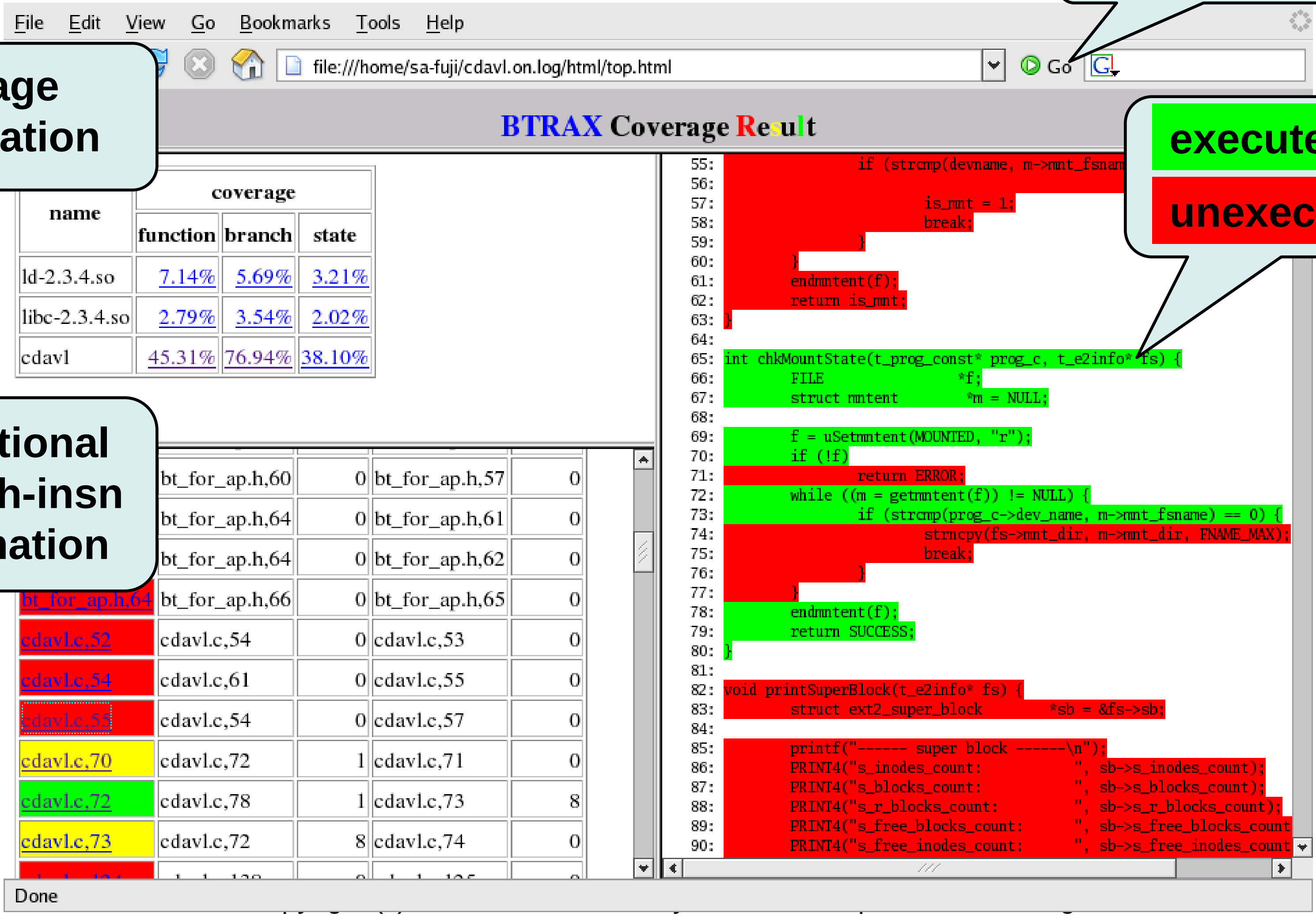
```
+-+-+-+-J <readdir+0xa2> (0x3537c96141)
+-+-+-+-J <readdir+0xb3> (0x3537c96189)
+-+-+-C <file_ignored> (0x00404ffe)
+-+-+-+-C <file_ignored+0x5c> (0x004053bc)
+-+-+-+-C <patterns_match> (0x0040540d)
+-+-+-+-+-J <patterns_match+0x48> (0x00405359)
+-+-+-+-C <patterns_match> (0x00405421)
+-+-+-+-+-J <patterns_match+0x48> (0x00405359)
+-+-+-+-C <file_ignored+0x97> (0x00405428)
+-+-+-J <print_dir+0x483> (0x00405039)
+-+-+-J <print_dir+0x49c> (0x0040506f)
+-+-+-C <gobble_file> (0x004050a0)
+-+-+-+-C <gobble_file+0x61> (0x00405591)
+-+-+-+-C <gobble_file+0xa4> (0x004055c7)
+-+-+-+-C <memset@plt> (0x00405630)
+-+-+-+-+-J <memset> (0x00401e18)
+-+-+-+-+-J <memset+0xd> (0x3537c7a977)
+-+-+-+-+-J <memset+0x310> (0x3537c7a999)
+-+-+-+-+-J <memset+0x3d0> (0x3537c7acb2)
```

**Btrax has disassembler**

- To get nest depth, Btrax disassembles branched insn, and pickup “call” and “ret” insn.
- To distinguish branch type, Btrax disassembles branched insn, call/jump/others.

## ScreenShot of Btrax

- Visualizing executed/unexecuted codes
- Analyzing code coverage


File Edit View Go Bookmarks Tools Help
file:///home/sa-fuji/cdavl.on.log/html/top.html
Go
BTRAX Coverage Result
Done

HTML Report

Coverage Information

| name | coverage | | |
|---|---|---|---|
| | function | branch | state |
| ld-2.3.4.so | 7.14% | 5.69% | 3.21% |
| libc-2.3.4.so | 2.79% | 3.54% | 2.02% |
| cdavl | 45.31% | 76.94% | 38.10% |

Conditional Branch-insn Information

| | | | | |
|---|---|---|---|---|
| | bt_for_ap.h,60 | 0 | bt_for_ap.h,57 | 0 |
| | bt_for_ap.h,64 | 0 | bt_for_ap.h,61 | 0 |
| | bt_for_ap.h,64 | 0 | bt_for_ap.h,62 | 0 |
| bt_for_ap.h,64 | bt_for_ap.h,66 | 0 | bt_for_ap.h,65 | 0 |
| cdavl.c,52 | cdavl.c,54 | 0 | cdavl.c,53 | 0 |
| cdavl.c,54 | cdavl.c,61 | 0 | cdavl.c,55 | 0 |
| cdavl.c,55 | cdavl.c,54 | 0 | cdavl.c,57 | 0 |
| cdavl.c,70 | cdavl.c,72 | 1 | cdavl.c,71 | 0 |
| cdavl.c,72 | cdavl.c,78 | 1 | cdavl.c,73 | 8 |
| cdavl.c,73 | cdavl.c,72 | 8 | cdavl.c,74 | 0 |

executed

unexecuted

```
                        if (strcmp(devname, m->mnt_fsnam

                                is_mnt = 1;
                                break;
                        }
                }
                endmntent(f);
                return is_mnt;
}

int chkMountState(t_prog_const* prog_c, t_e2info* fs) {
        FILE                    *f;
        struct mntent           *m = NULL;

        f = uSetmntent(MOUNTED, "r");
        if (!f)
                return ERROR;
        while ((m = getmntent(f)) != NULL) {
                if (strcmp(prog_c->dev_name, m->mnt_fsname) == 0) {
                        strncpy(fs->mnt_dir, m->mnt_dir, FNAME_MAX);
                        break;
                }
        }
        endmntent(f);
        return SUCCESS;
}

void printSuperBlock(t_e2info* fs) {
        struct ext2_super_block         *sb = &fs->sb;

        printf("------ super block ------\n");
        PRINT4("s_inodes_count:         ", sb->s_inodes_count);
        PRINT4("s_blocks_count:         ", sb->s_blocks_count);
        PRINT4("s_r_blocks_count:       ", sb->s_r_blocks_count);
        PRINT4("s_free_blocks_count:    ", sb->s_free_blocks_count
        PRINT4("s_free_inodes_count:    ", sb->s_free_inodes_count
```

## Agenda

- Background
- perf branch
- Implementation
- Btrax
- Merge Plan
- Future plan & Conclusion

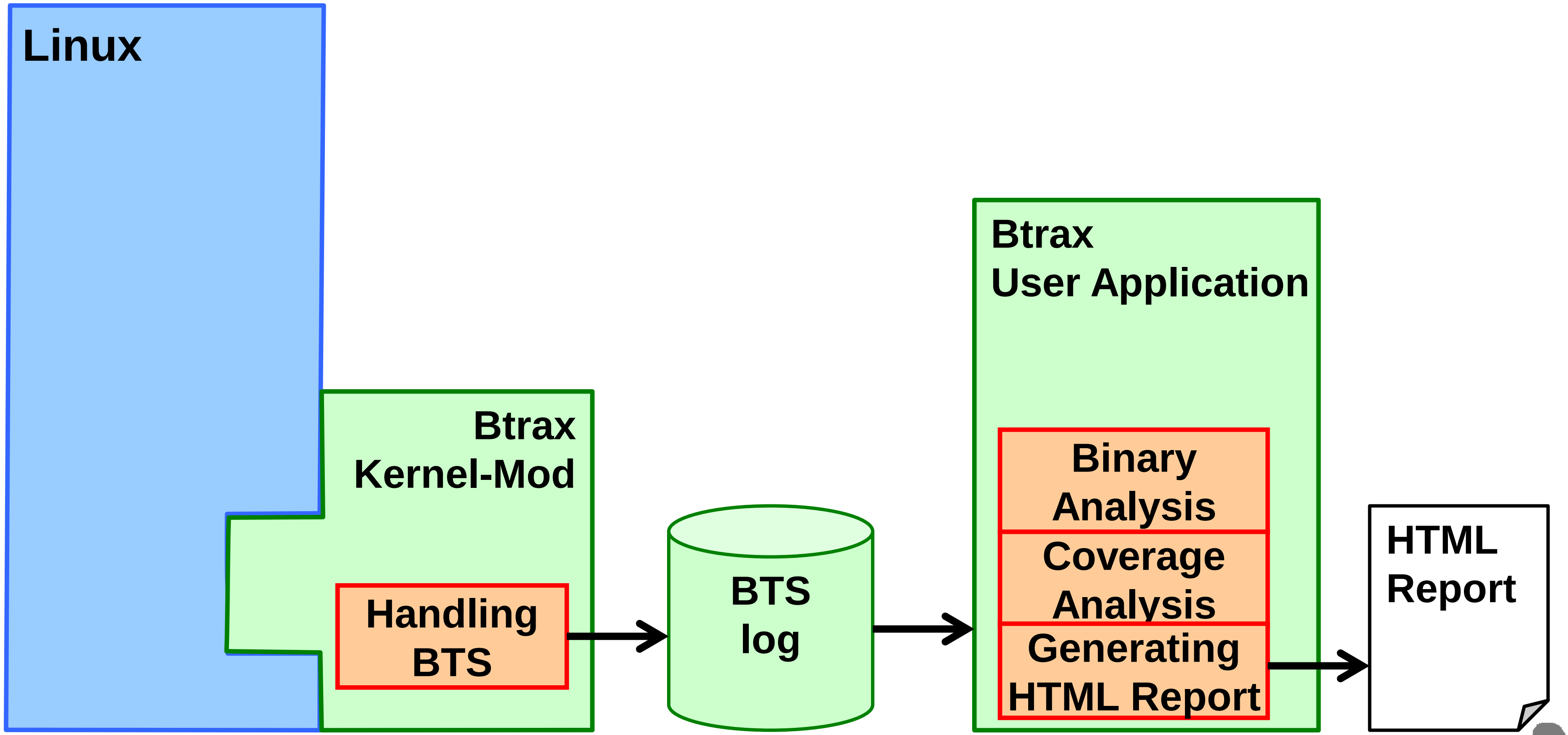
## Merge into perf branch

- Btrax has many useful functions
  - execution path / call graph
  - analyze code coverage / visualize executed codes

⇒I want to merge into perf branch!

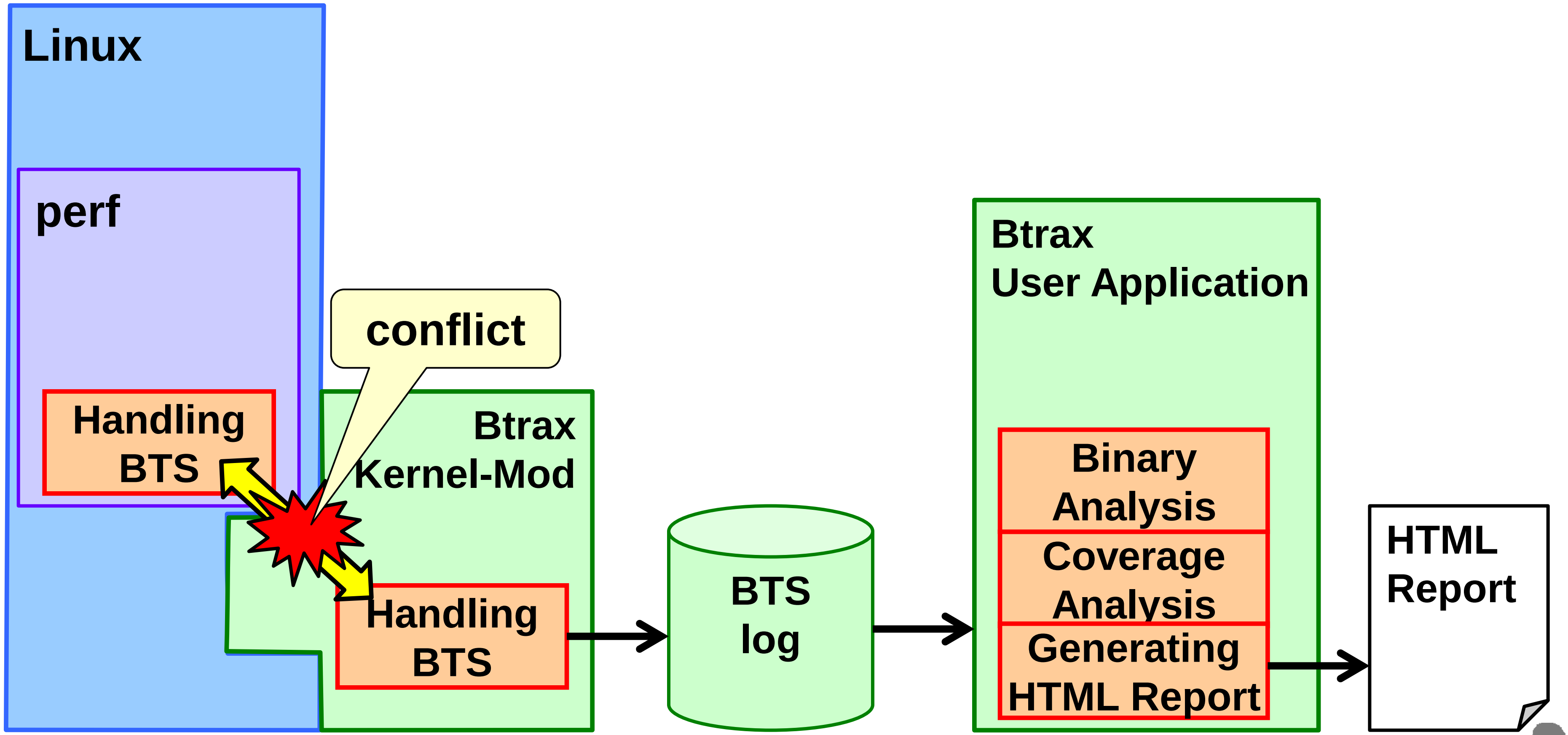
## Old Btrax Structure

Btrax consists of two parts:
kernel module and user application.



## Old Btrax Structure

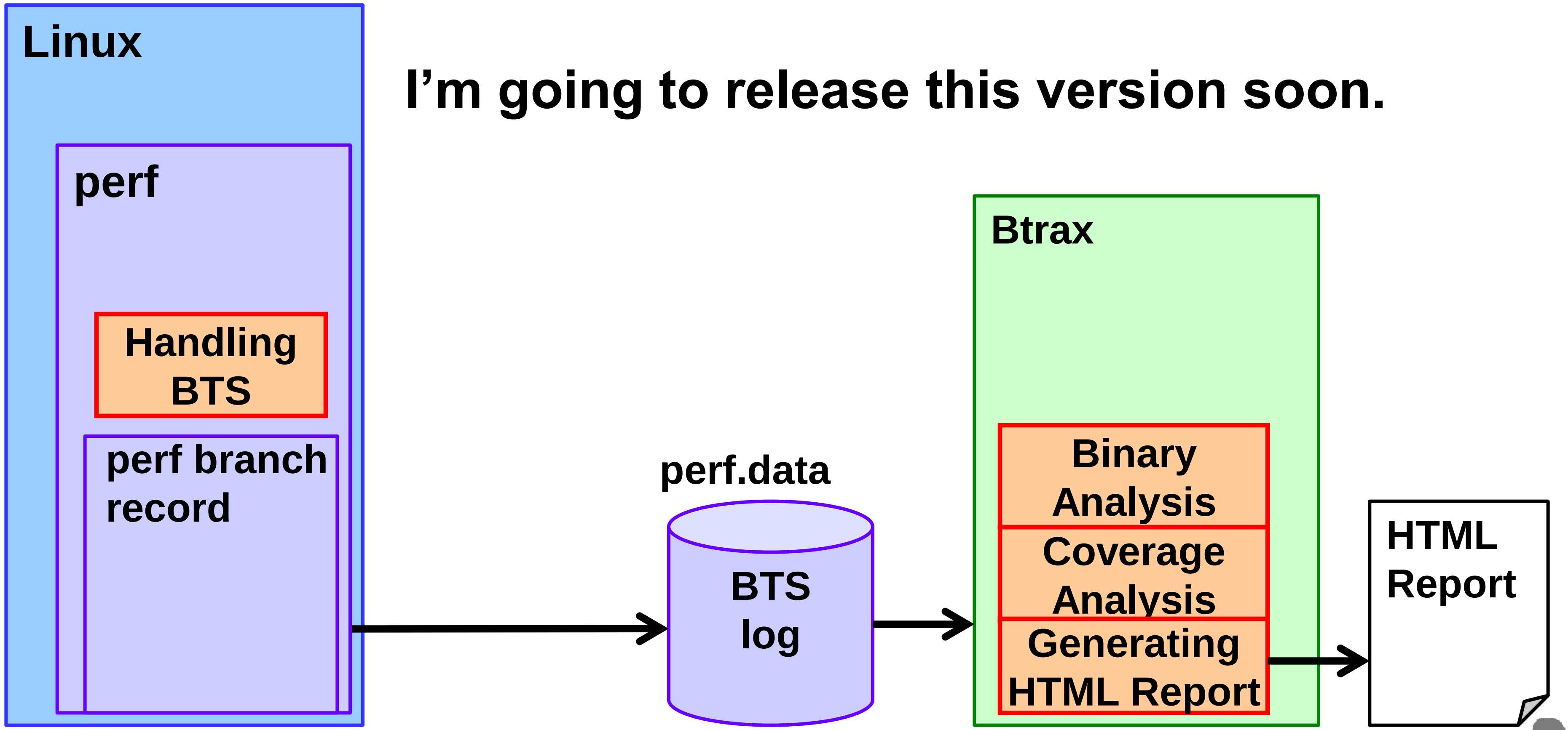
**kernel module to use BTS conflicts with perf's function in recent kernel (2.6.33-)**



## Next Btrax Structure

Next Btrax uses 'perf branch record' instead of kernel-module.
This makes Btrax works on newer kernel.

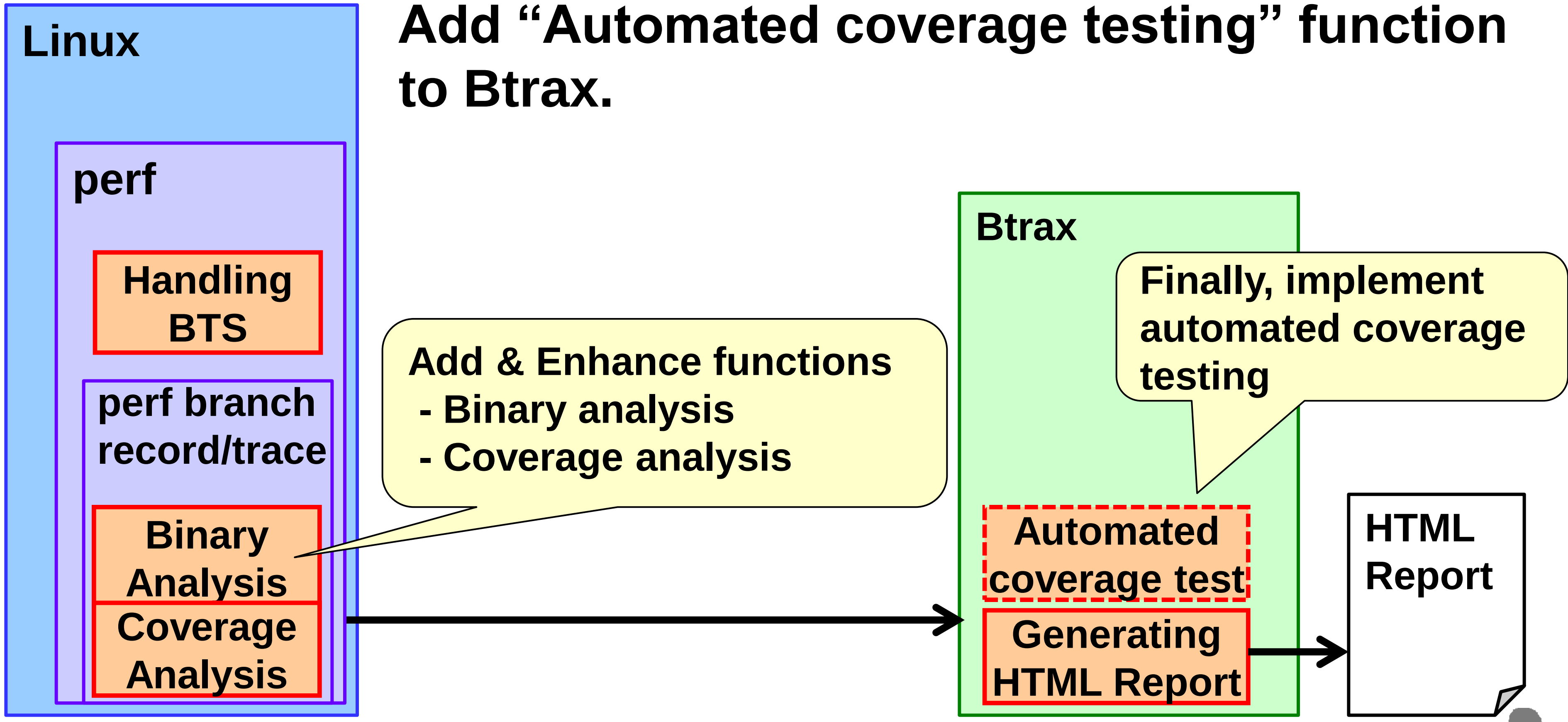
I'm going to release this version soon.



# Future plan of Btrax

In the future, add the functions derivered from Btrax to perf branch.

Add “Automated coverage testing” function to Btrax.



## Agenda

- Background
- perf branch
- Implementation
- Btrax
- Merge plan
- Future plan & Conclusion

## Current upstream status

- I've sent 'perf branch' patches to LKML
  - Latest patchset is version 4
  - I got some requests to implement functions
    - Drawing call graph
    - Visualize executed codes
  - David Ahern & Frederic Weisbecker suggested me that 'perf branch' implements on 'perf script'
    - 'perf script' is subcommand of perf to use perf's output with script languages: perl or python.

I'll continue to work with upstream developers

## Future Plan

- Implement functions like Btrax
  - Make perf branch more informative
    - Source file path, line number, disassembler
- Support kernel and driver test
  - Currently, perf branch can trace only user-space
  - Enable branch-tracing in kernel-space
- Cooperate with other perf functions
  - Processor’s performance monitoring facilities
  - Trace point to get variable information
- Reduce BTS log size
  - Test-range filtering by probe-point ...etc

## Conclusion

- Introduce new branch tracer ‘perf branch’
- perf branch provides the interface to use BTS easily for application developers.
- perf branch has a potential to make useful development tools like Btrax

## Reference

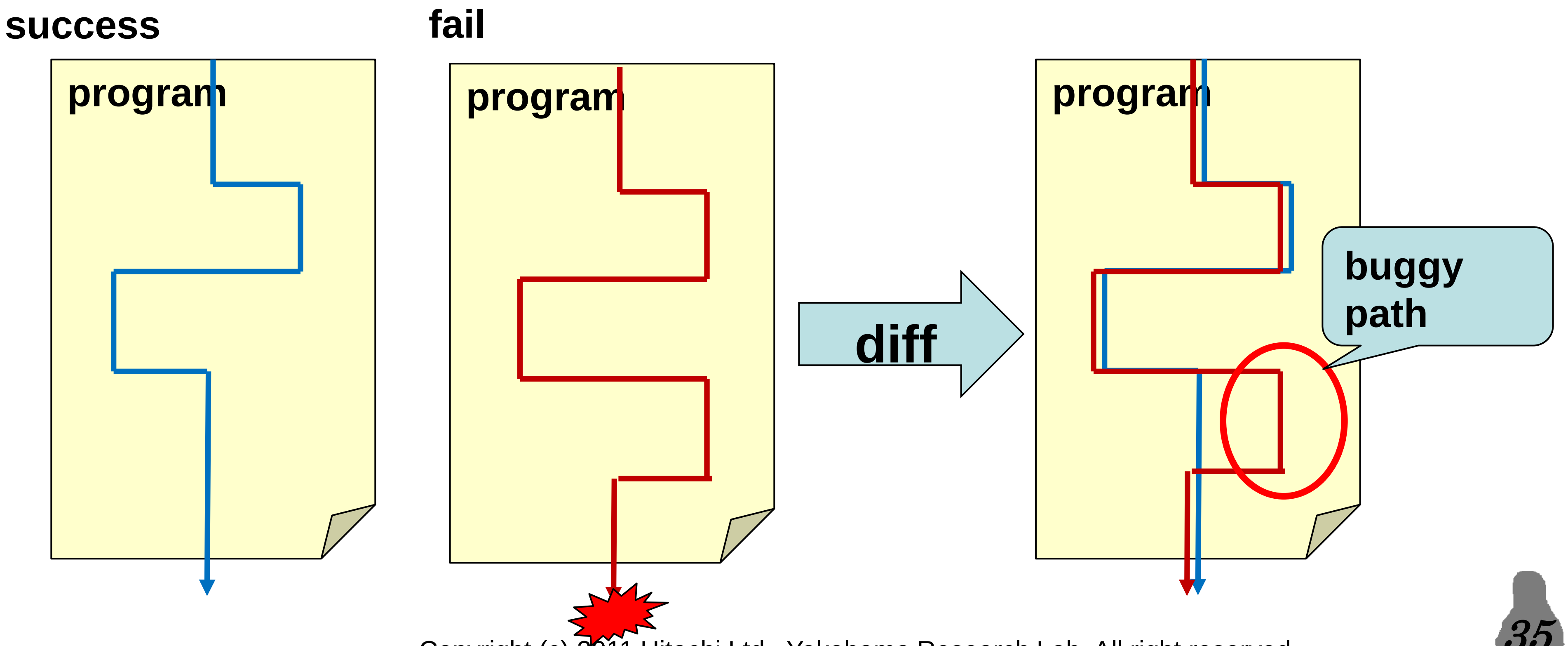
- Btrax WebPage
  - http://sourceforge.net/projects/btrax/
- perf Wiki
  - https://perf.wiki.kernel.org/

# Thank you

# Appendix

# Finding bugs

- Compare the execution path of succeed pattern and failed pattern
- Extract buggy execution path


success
fail
program
program
diff
program
buggy path