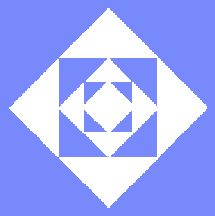


The Advantages of Post-link Code Coverage

Orna Raz, Moshe Klausner, Nitzan Peleg, Gad Haber, Eitan Farchi, Shachar Fienblit, Yakov Filiarsky, Shay Gammer and Sergey Novikov

What is Code Coverage?

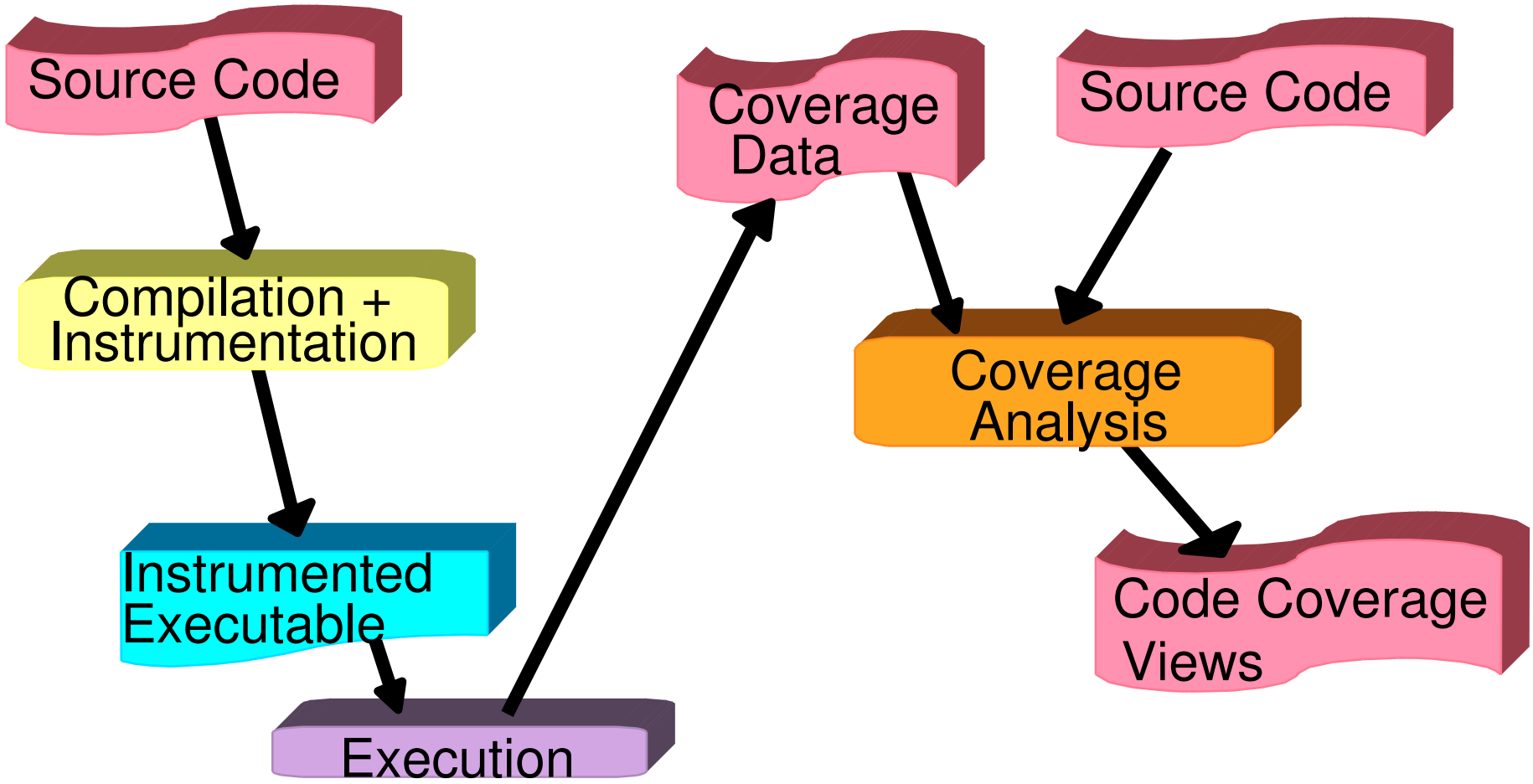
- A quality measurement used in software testing
- A quantitative way to describe the degree to which the **source code** of a program has been tested
  - Feedback for improving the tests

How is Code Coverage Done?

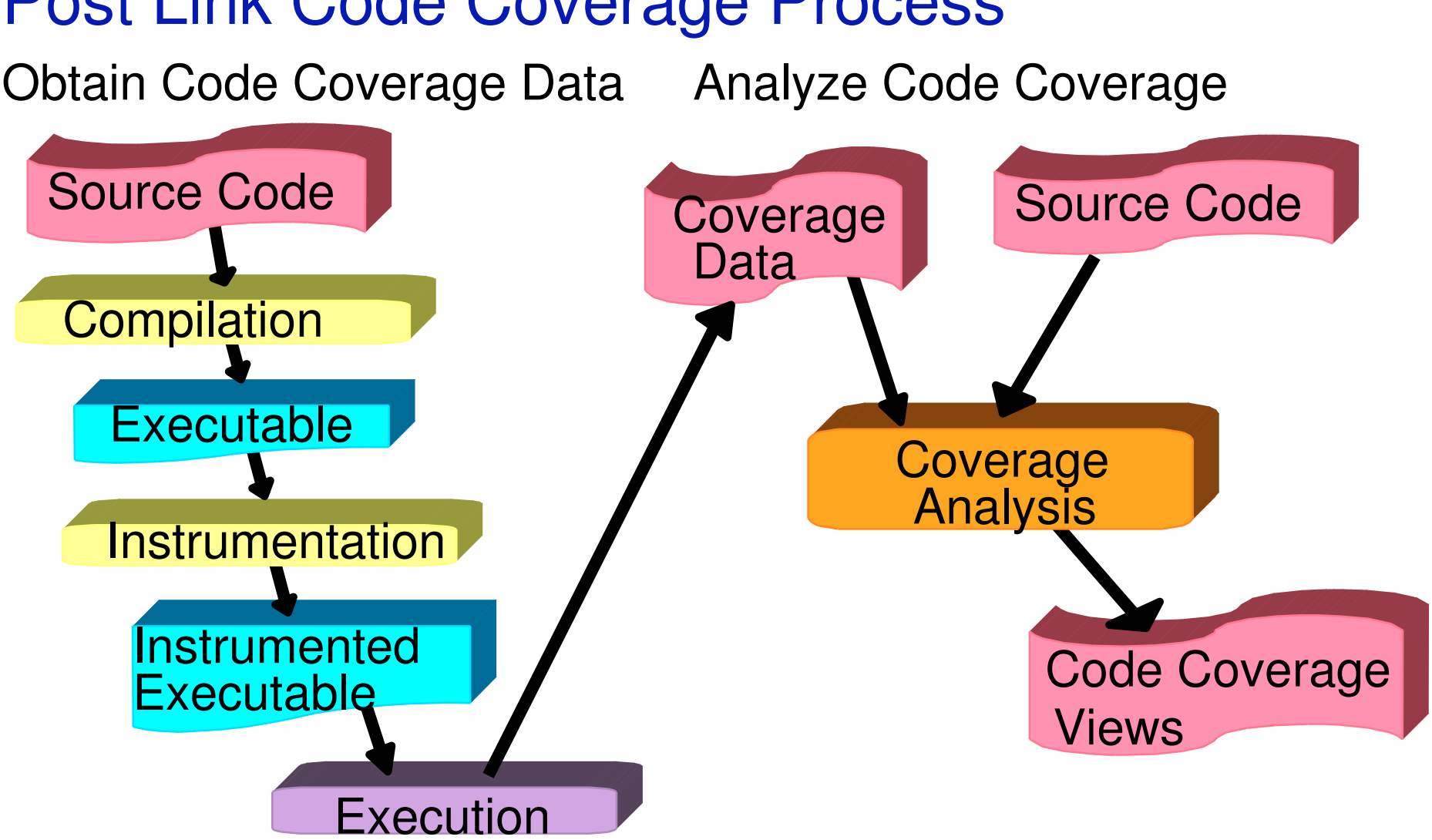
- Via instrumentation
  - Adding additional code to collect coverage data
  - Introduce overhead on execution time
- Source level instrumentation is the common practice
  - Instrumentation done during compilation
- For example gnu tools
  - gcc/gcov – compilation + instrumentation
  - lcov – for analysis

Common Code Coverage Process

Obtain Code Coverage Data Analyze Code Coverage



Post Link Code Coverage Process



Motivation for Post-Link Instrumentation

- Challenging architectures like kernel & embedded systems
  - The current solutions (e.g. GCOV) may not address all the issues
    - No OS facilities
    - Non-terminating code
- Integrate code coverage in all tests along the entire development cycle
  - Same tests for coverage and functional
  - Enables accumulation of coverage data between phases

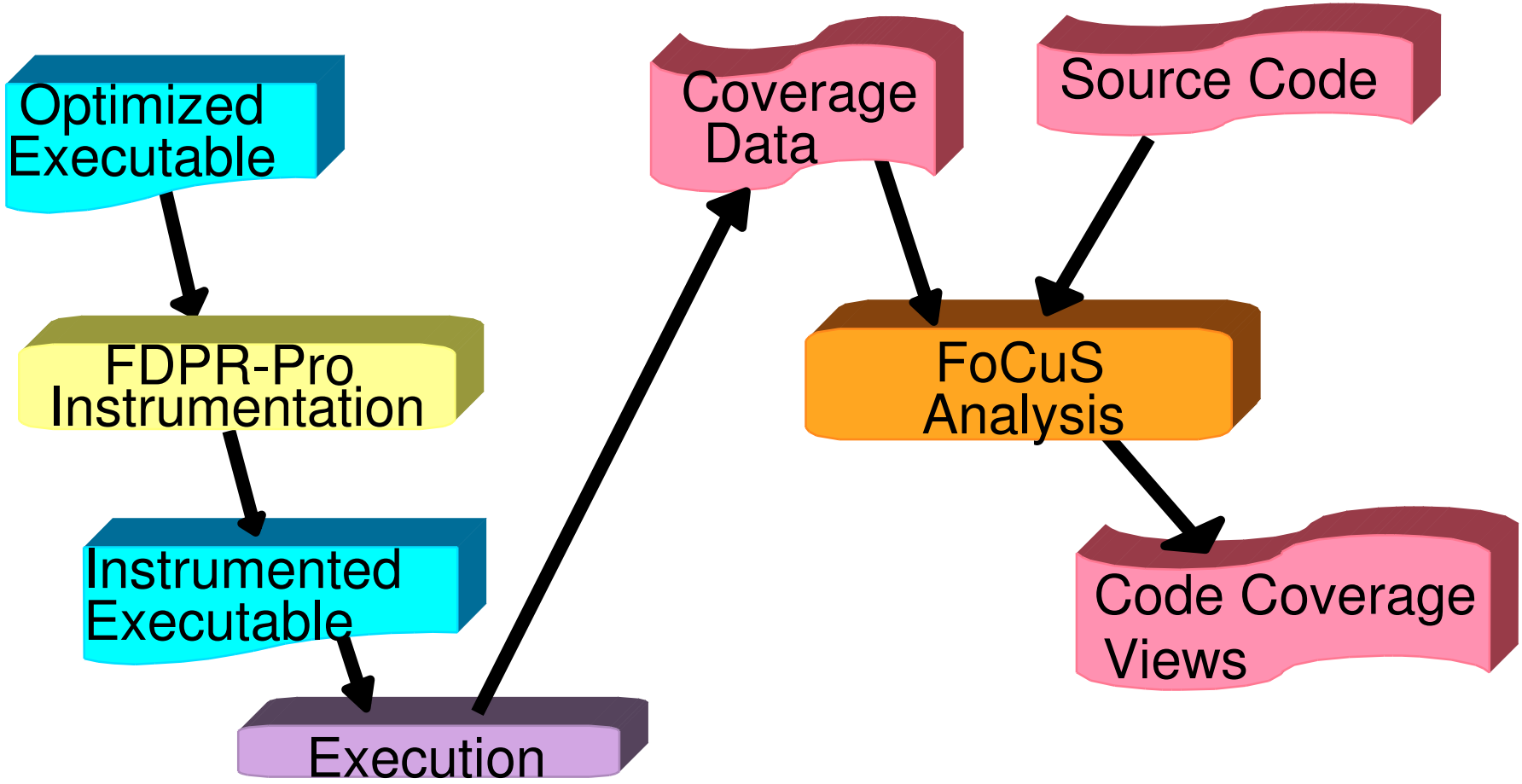
  As a result, coverage:
  - is done on actual (optimized) code (e.g. SVT)
  - instrumentation must have low overhead
    - The tests cycle period should be reasonable
    - The behavior should not be affected (e.g. time outs)

Post-link Code Coverage Characteristics

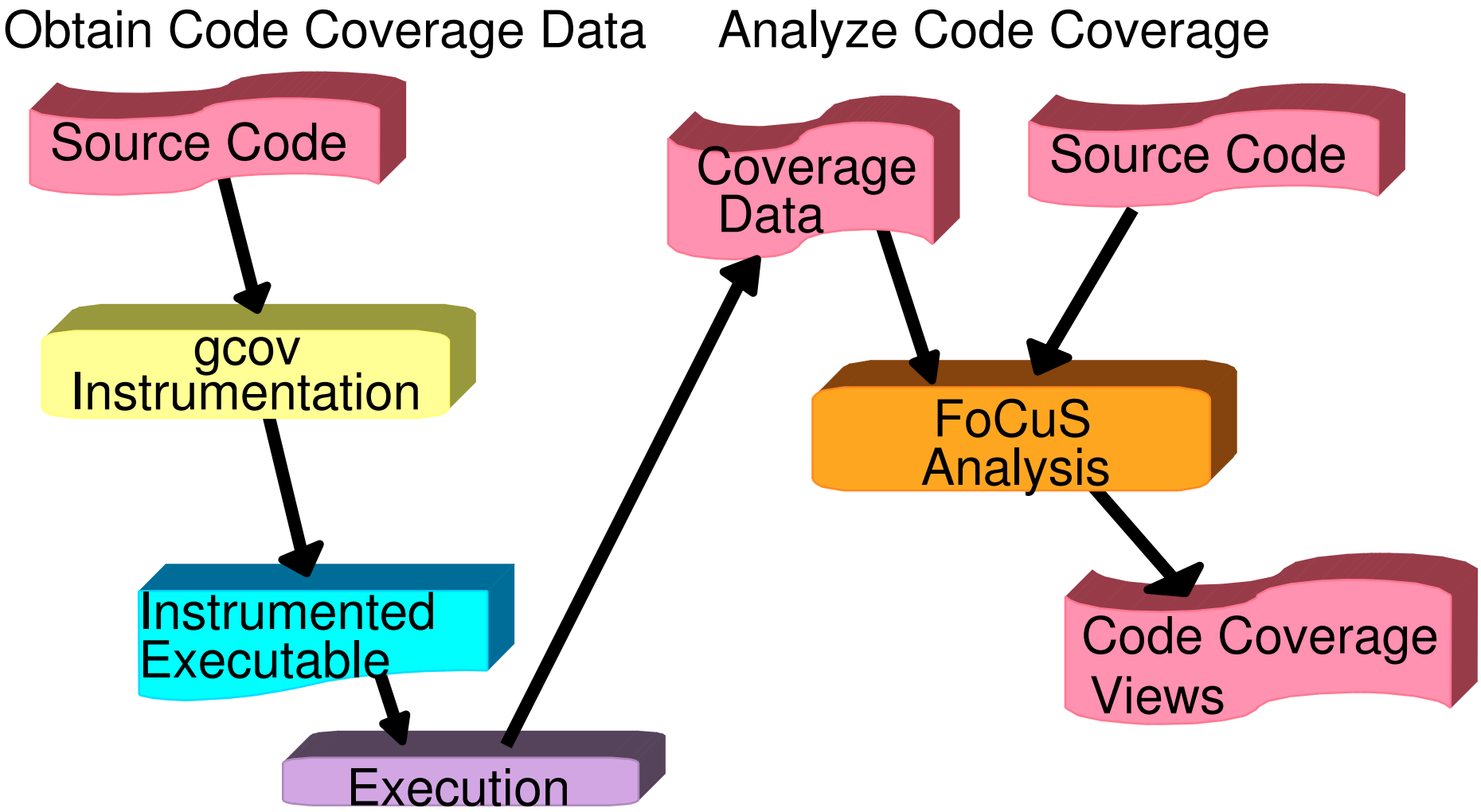
- Works on the running code and does not interfere with compiler optimizations
- The coverage data is influenced by the compiler transformation and optimization
  - Different from source level coverage
  - More information available
- Enables further reduction in instrumentation overhead

Our Post Link Code Coverage Process

Obtain Code Coverage Data Analyze Code Coverage



FoCuS can work with gcov as well (and more)

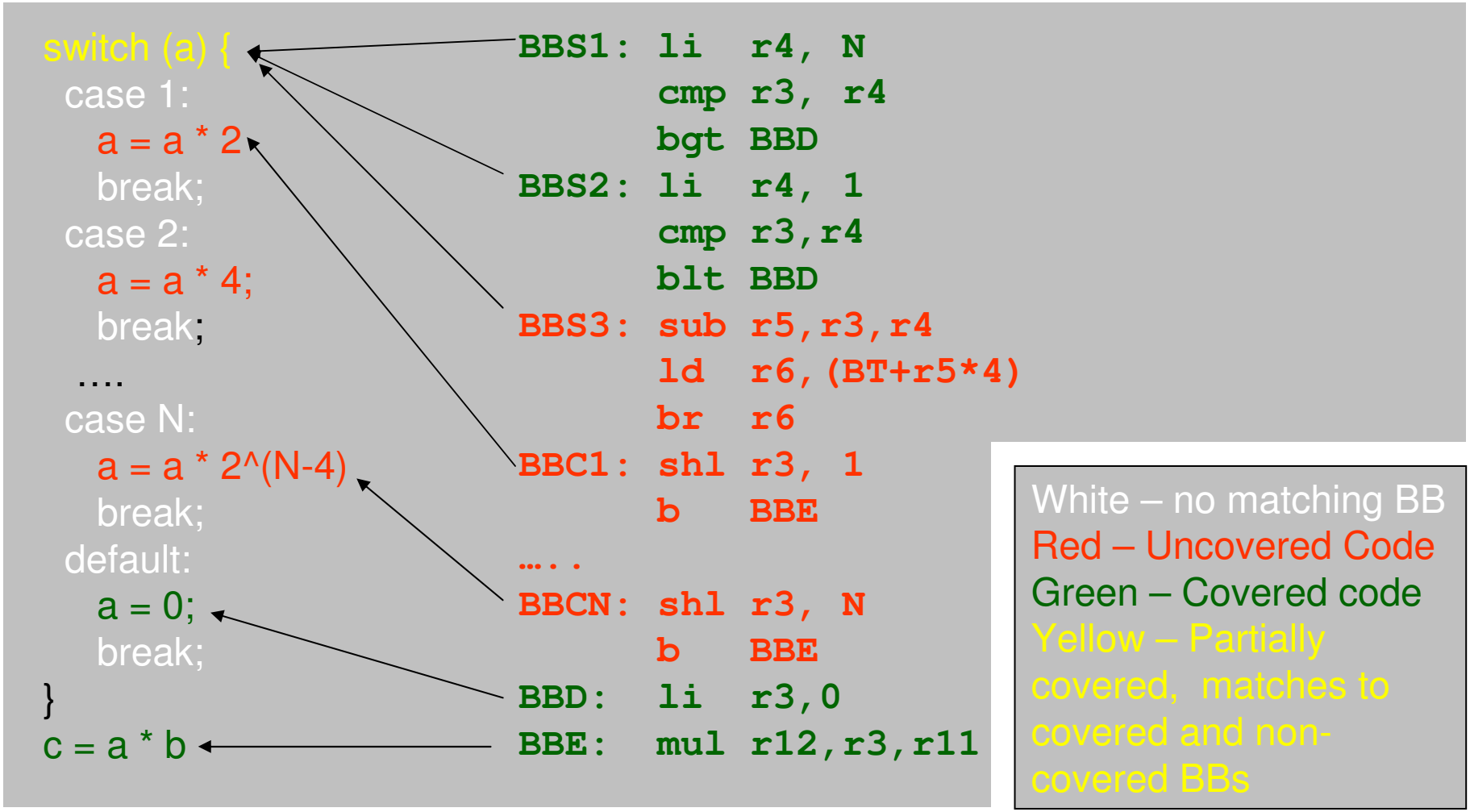


Raw Coverage Information From FDPR-Pro

```
.Perl_do_chop { function } ( size = 632 )
safe bb     size = 60    func = .Perl_do_chop ( prolog ) count = 2019
  0x1000b6c8: 0x7c0802a6 mflr r0              ; doop.c:215
  0x1000b6cc: 0xfb61ffd8 std r27,-40(r1)      ; doop.c:215
  .......
safe bb     size = 16    func = .Perl_do_chop      count = 0
  0x1000b74c: 0xe8828440 ld r4,-31680(r2)     ; doop.c:251
  0x1000b750: 0x7f63db78 or r3,r27,r27        ; doop.c:251
  0x1000b754: 0x38a00000 li r5,0              ; doop.c:251
  0x1000b758: 0x48076901 bl 0x1007fddc        ; doop.c:251
```

Basic Block – stream of instructions with one entry and one exit with no control flow in the middle

FoCuS' Source View with Post-link Mapping



Working with Post-Link Code Coverage

- At the high level same as in regular code coverage
- The new methodology provides a compiler transformations dictionary to help with:
  - White lines – not in the code
    - Due to compiler optimization
    - No need for additional tests to cover these lines
  - Yellow lines – partially covered
    - Due to the translation of language constructs (e.g. switch statement)
    - Due to compiler optimization

Understanding Coverage – Macro Example

Partial coverage

```
398  if (mg = SvTIED_mg((SV*)av, 'P')) {
399    dSP;
...    ...
410    return;
411  }
```

(a) Source code using the SvTIED_mg

```
#define SvTIED_mg(sv,how) (SvRMAGICAL(sv) ? mg_find((sv),(how)) : Null(MAGIC*))
```

(b) The SvTIED_mg macro which uses the ? operator and calls other macros

Understanding Coverage - If Example

Partial coverage

```
176   if (*name == '+' && len>1 && name[len-1] != '\|') { /* scary */
177     mode[1] = *name++;
178     –len;
179   writing = 1:
180 }
```

Understanding Coverage - If Example (cont)

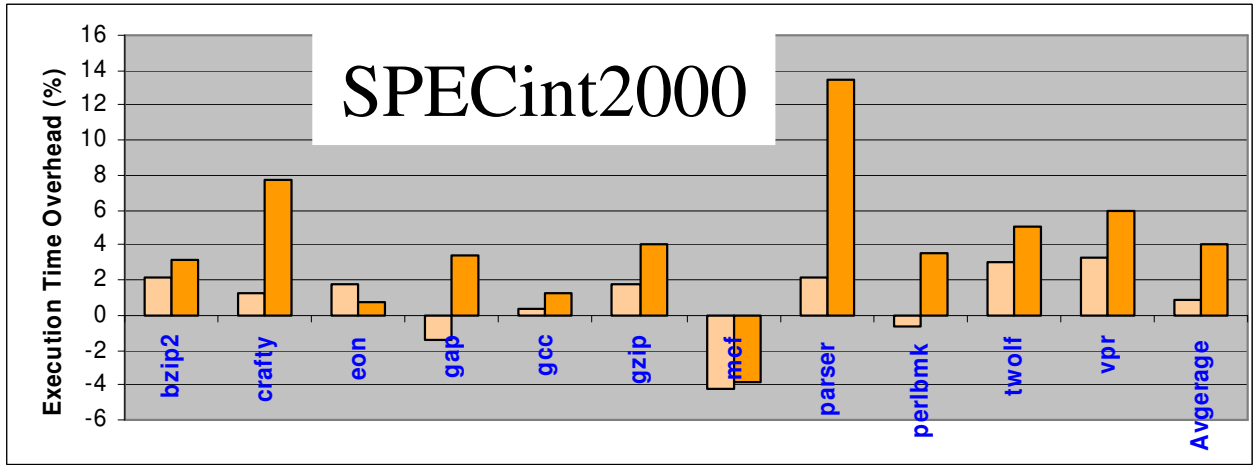
Partial coverage

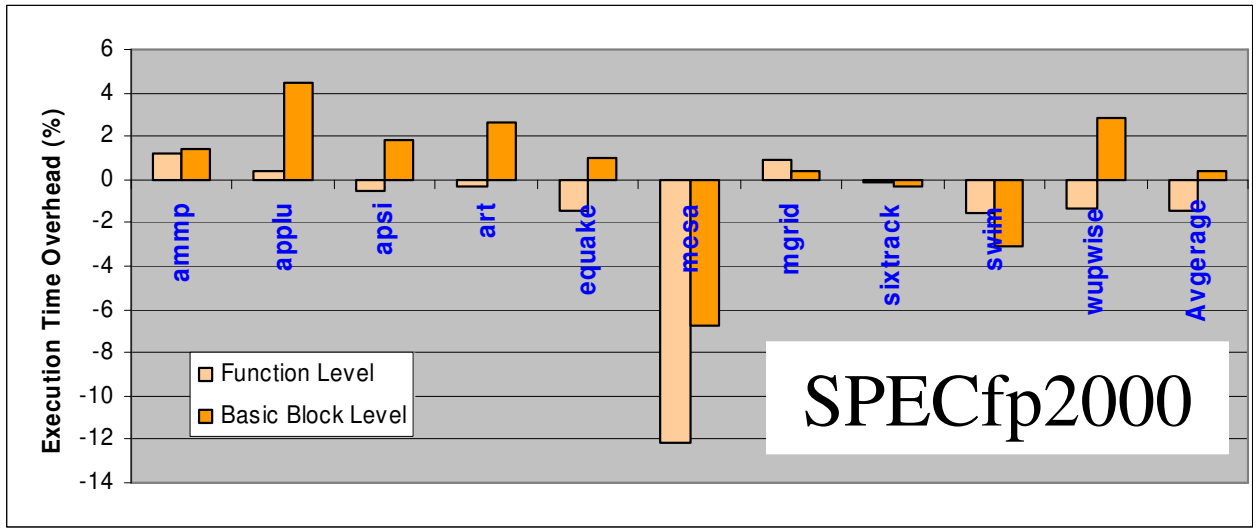
```
176-1   if (*name == '+') {
176-2     if (len>1) {
176-3       if (name[len-1] != '\|') { /* scary */
177           mode[1] = *name++;
178           –len;
179         writing = 1;
180-1     }
180-2   }
180-3 }
```

Low Overhead Instrumentation at Post-Link

- Motivation: integrating coverage along the entire development process
  - More constraints as we approach shipment
  - And beyond – at the customer site
- Post link – a better starting point
  - Works on optimized code
  - Lend itself for reducing instrumentation overhead
    - Only covered/no-covered is needed
    - Using self modifying instrumentation

Low Overhead Instrumentation - Results





Linux on POWER

gcc 4.1 –O3

Using Self-modifying Code

Average

INT: 4% / 0.9%

FP: 0.4% / -1.4%

Conclusion

- Post-link code coverage is useful
  - Can be mapped back to source code
  - Enables compiler optimization
  - Supported by compiler transformation dictionary
- In some cases post-link code coverage is better
  - Enables integrating coverage in the development process
    - Near shipment and beyond
  - More flexible for low overhead instrumentation
- Future work - enhancing the visual aids to improve the understanding of more complex compiler optimizations

Questions ?

FoCuS: www.alphaworks.ibm.com/tech/focus

FDPR-Pro: www.haifa.il.ibm.com/projects/systems/cot/fdpr

Line Number Information and optimization

- GCC man

  “Unlike most other C compilers, GCC allows you to use `-g' with `-O‘”

- xlc man

  “If you specify the **-qlinedebug** option, the inlining option defaults to **-Q!** (no functions are inlined).”

  But inline can be forced and the line-number info will be correct