



Dynamic Program Analysis

Xiangyu Zhang

Introduction

- Dynamic program analysis is to solve problems regarding software dependability and productivity by inspecting **software execution**.
- Program executions vs. programs
 - Not all statements are executed; one statement may be executed many times.
 - Analysis on a single path – the executed path
 - All variables are instantiated (solving the aliasing problem)
- Resulting in:
 - Relatively lower learning curve.
 - Precision.
 - Applicability.
 - Scalability.
- Dynamic program analysis can be constructed from a set of primitives
 - Tracing
 - Profiling
 - Checkpointing and replay
 - Dynamic slicing
 - Execution indexing
 - Delta debugging
- Applications
 - Dynamic information flow tracking
 - Automated debugging



Program Tracing

Outline

- What is tracing.
- Why tracing.
- How to trace.
- Reducing trace size.

What is Tracing

- Tracing is a process that faithfully records detailed information of program execution (lossless).
 - Control flow tracing
 - the sequence of executed statements.
 - Dependence tracing
 - the sequence of exercised dependences.
 - Value tracing
 - the sequence of values that are produced by each instruction.
 - Memory access tracing
 - the sequence of memory references during an execution
- The most basic primitive.

Why Tracing

- **Debugging**
 - Enables time travel to understand what has happened.
- **Code optimizations**
 - Identify hot program paths;
 - Data compression;
 - Value speculation;
 - Data locality that help cache design;
- **Security**
 - Malware analysis
- **Testing**
 - Coverage.

Outline

- What is tracing.
- Why tracing.
- How to trace.
- Reducing trace size.
- Trace accessibility

Tracing by Printf

```
Max = 0;
for (p = head; p; p = p->next)
{
    printf("In loop\n");
    if (p->value > max)
    {
        printf("True branch\n");
        max = p->value;
    }
}
```


Tracing by Source Level Instrumentation

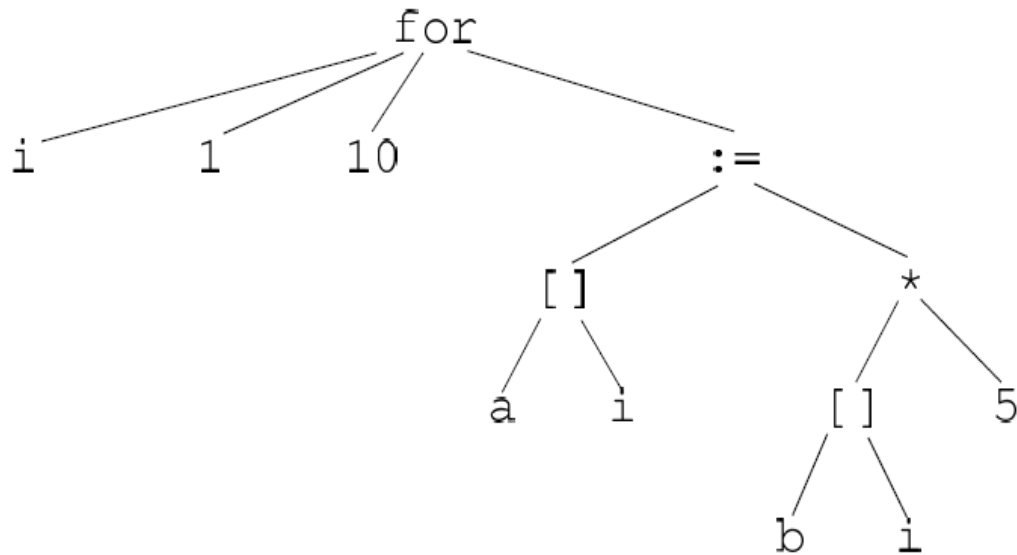
- Read a source file and parse it into ASTs.
- Annotate the parse trees with instrumentation.
- Translate the annotated trees to a new source file.
- Compile the new source.
- Execute the program and a trace produced.

An Example

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:

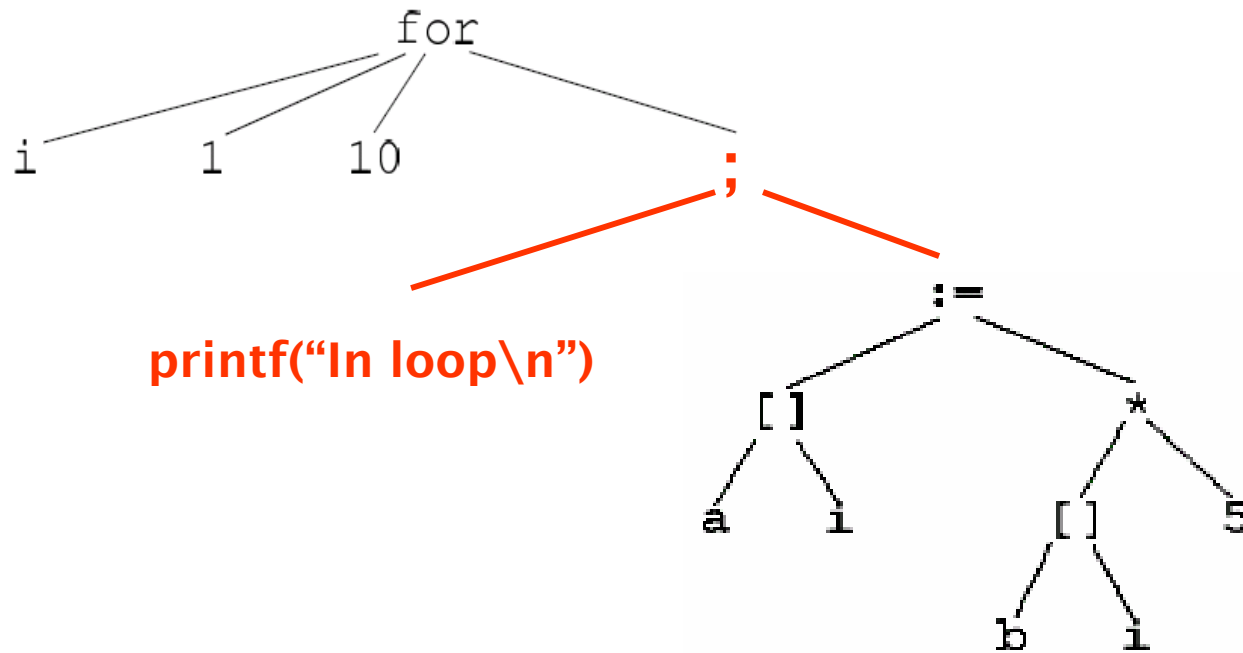


An Example

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:



Limitations of Source Level Instrumentation

- **Hard to handle libraries.**
 - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).
- **Hard to handle multi-lingual programs**
 - Source code level instrumentation is heavily language dependent.
- **Requires source code**
 - Worms and viruses are rarely provided with source code

Tracing by Binary Instrumentation

- What is binary instrumentation
 - Given a binary executable, parses it into **intermediate representation**. More advanced representations such as control flow graphs may also be generated.
 - Tracing instrumentation is added to the intermediate representation.
 - A lightweight compiler compiles the instrumented representation into a new executable.
- Features
 - No source code requirement
 - Easily handle libraries.

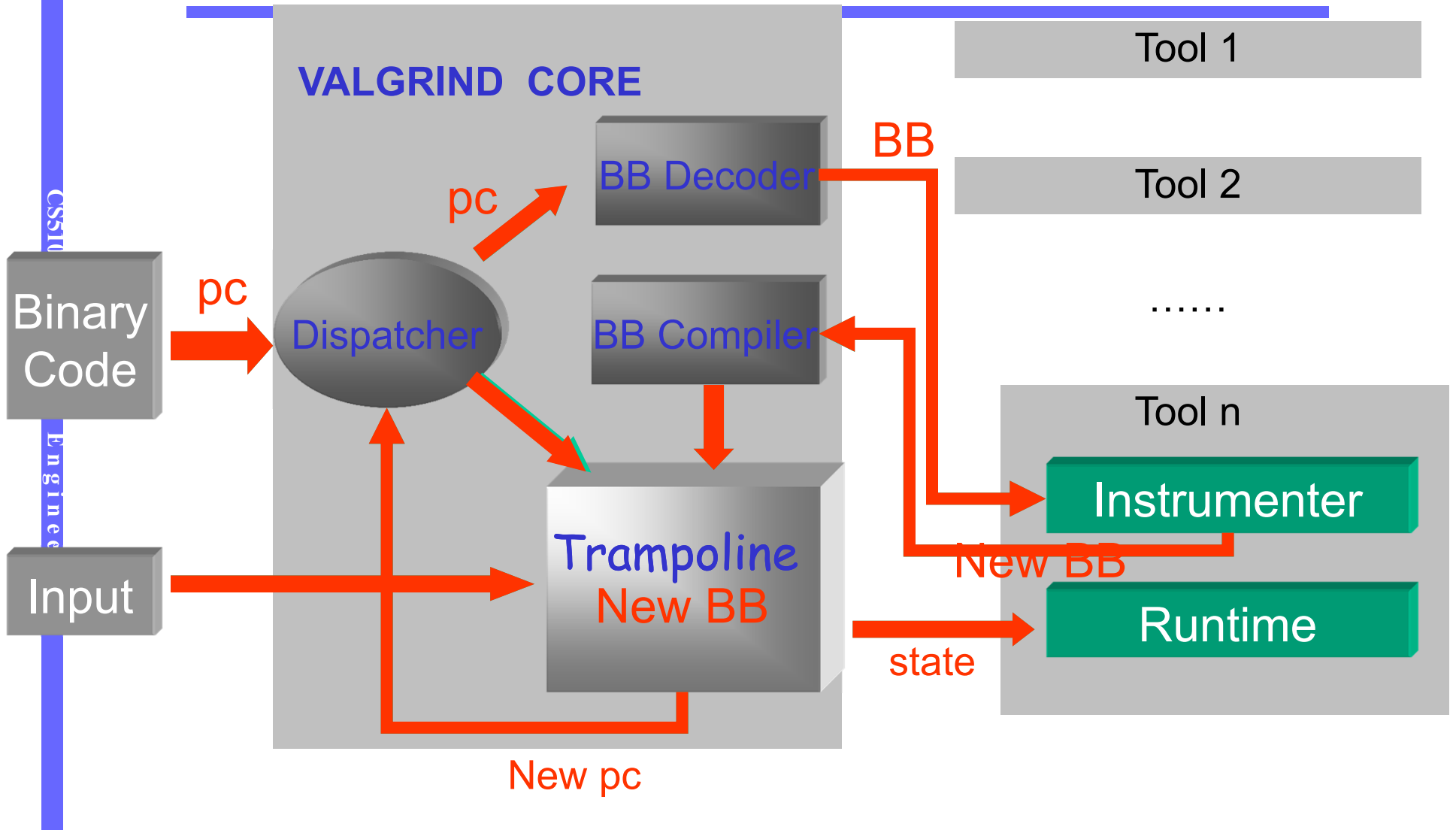
Static vs. Dynamic Instrumentation

- Static: takes an executable and generate an instrumented executable that can be executed with many different inputs
- Dynamic: given the original binary and an input, starts executing the binary with the input, during execution, an instrumented binary is generated on the fly; essentially the instrumented binary is executed.

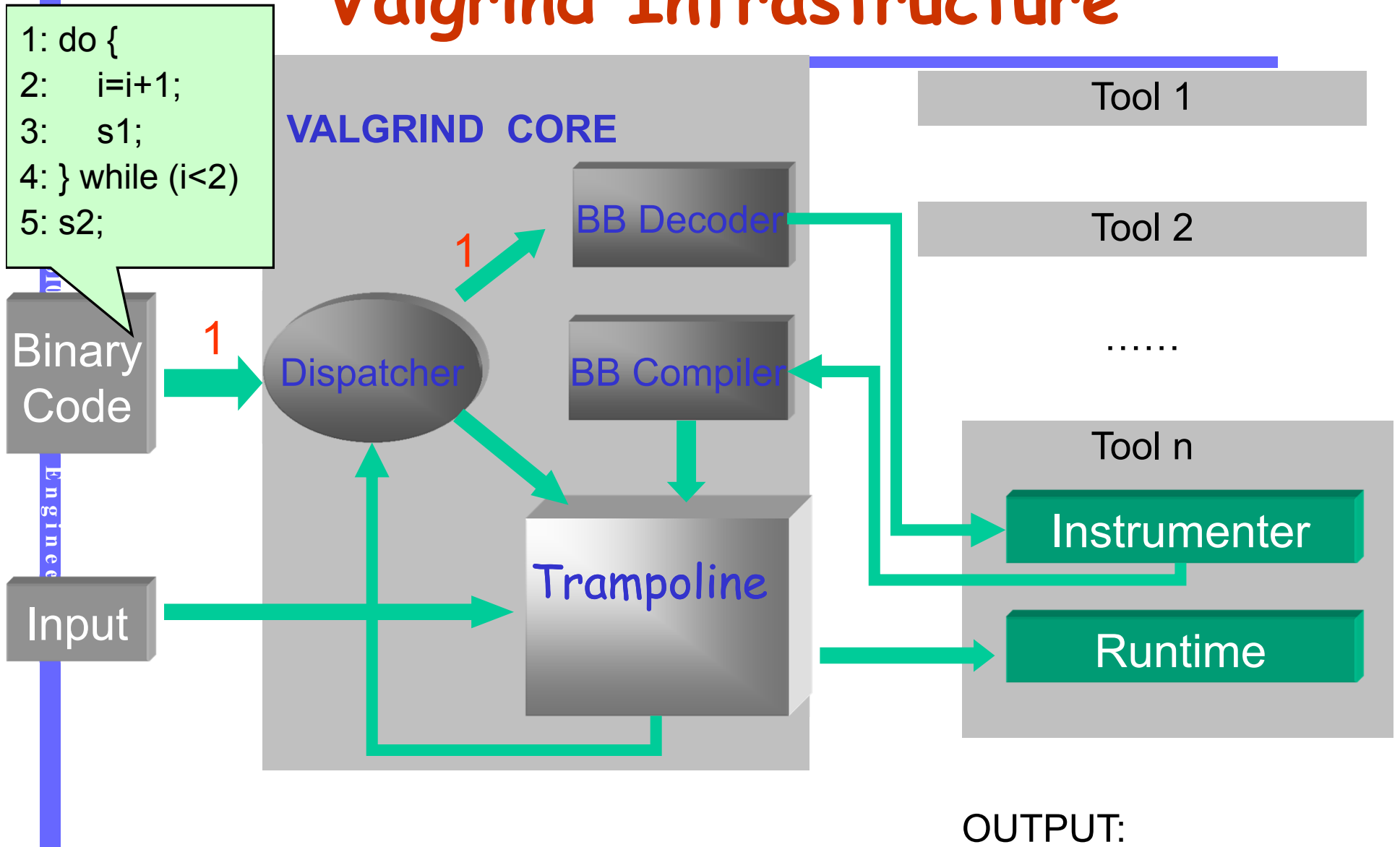
Dynamic Binary Instrumentation - Valgrind

- Developed by Julian Seward at Cambridge University.
 - Google-O'Reilly Open Source Award for "Best Toolmaker" 2006
 - A merit (bronze) Open Source Award 2004
- Open source
 - works on x86, AMD64
- Easy to execute, e.g.:
 - `valgrind --tool=memcheck ls`
- It becomes very popular
 - One of the two most popular dynamic instrumentation tools
 - Pin and Valgrind
 - Very good usability, extendibility, robust
 - 25MLOC
 - Mozilla, MIT, Berkeley-security, Me, and many other places
- Overhead is the problem
 - 5-10X slowdown without any instrumentation
- Reading assignment
 - **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation (PLDI07)**

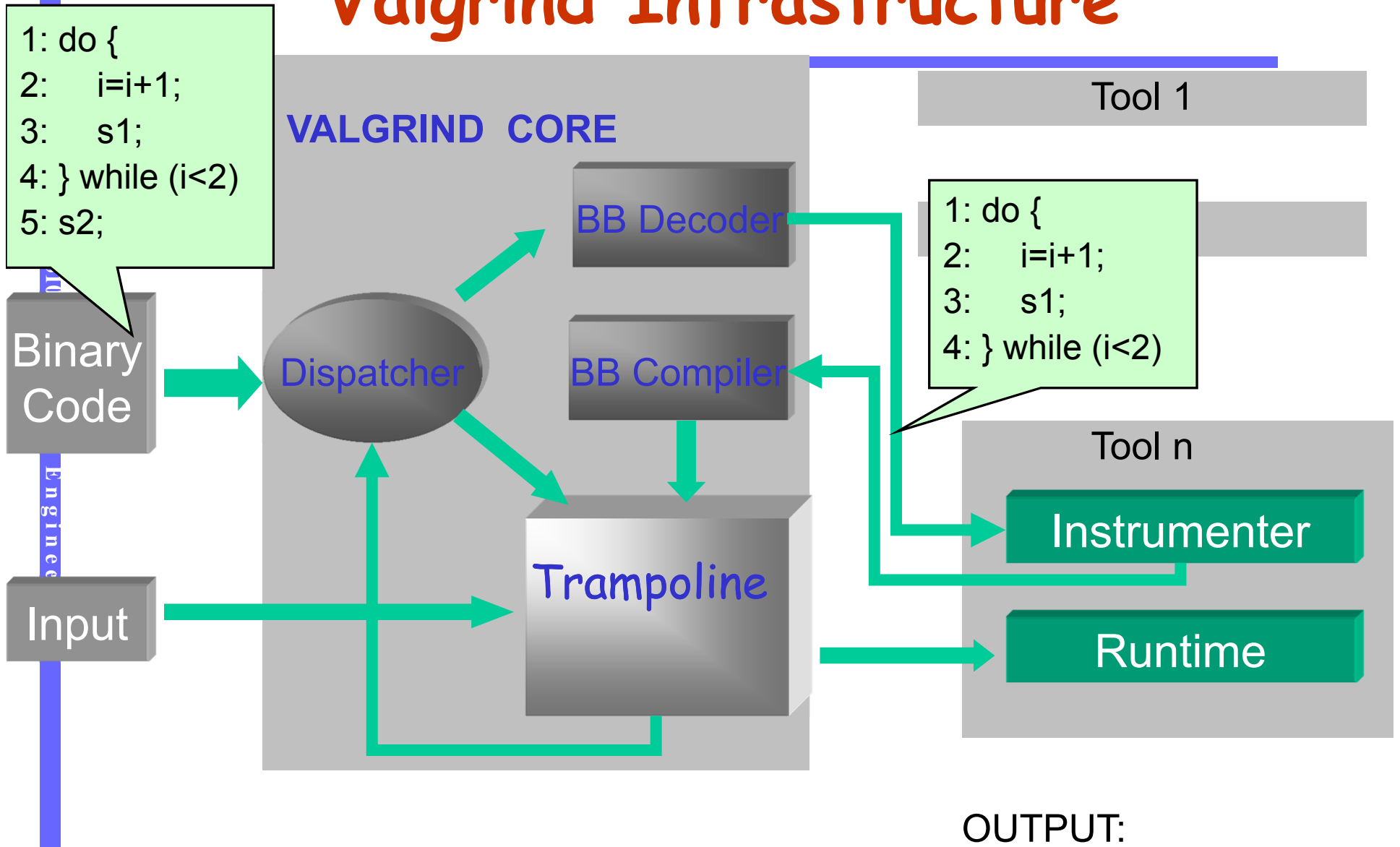
Valgrind Infrastructure



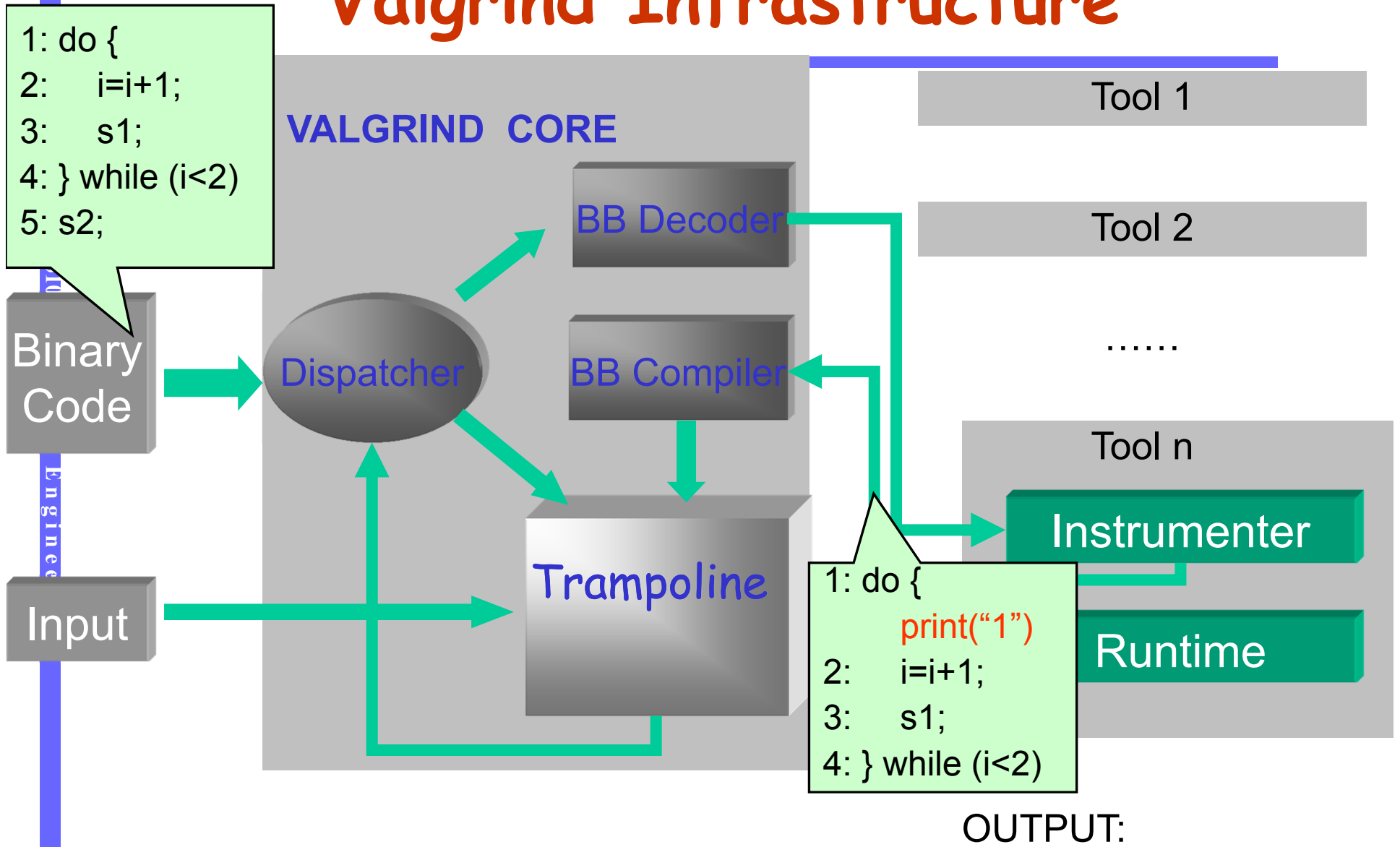
Valgrind Infrastructure



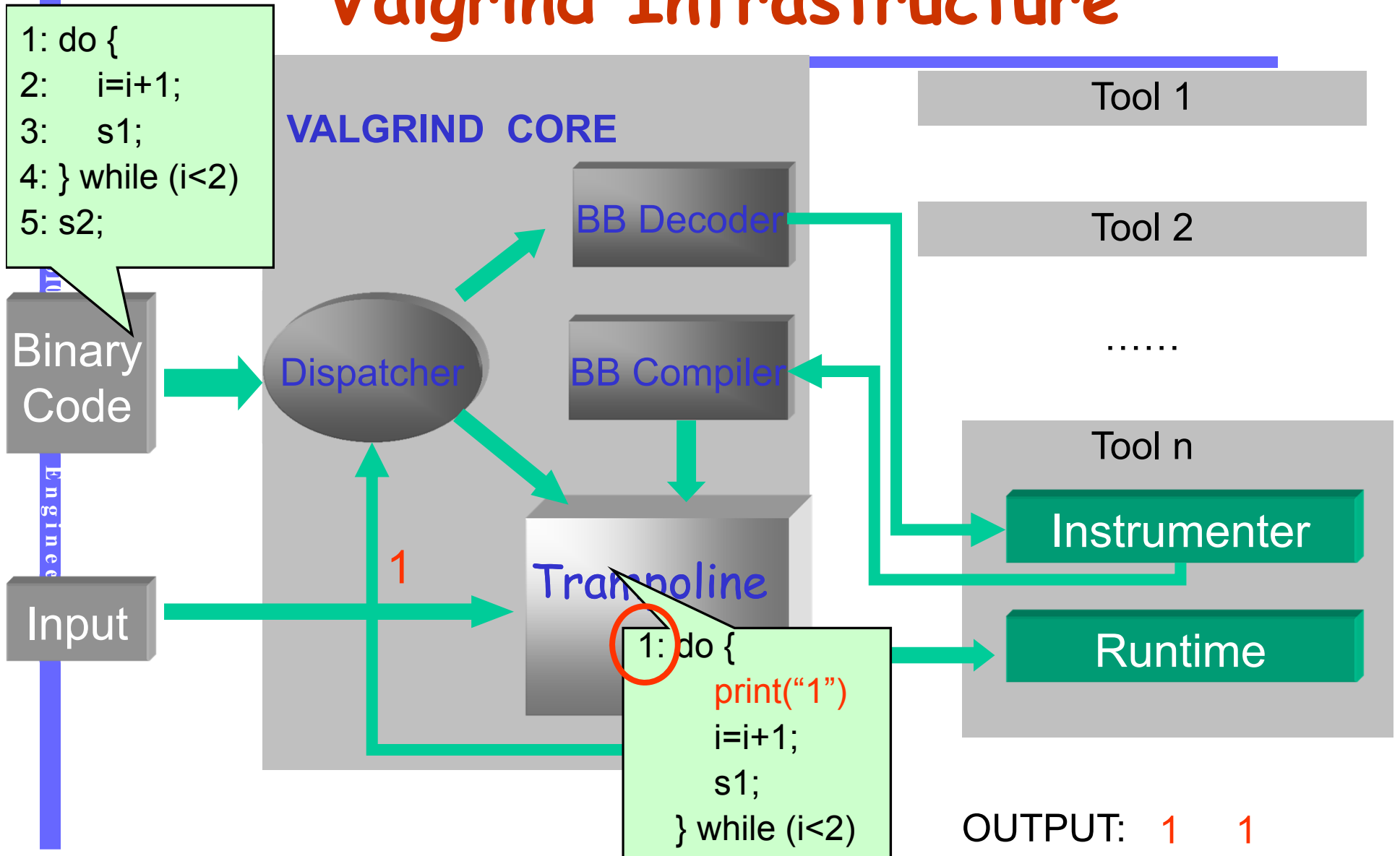
Valgrind Infrastructure



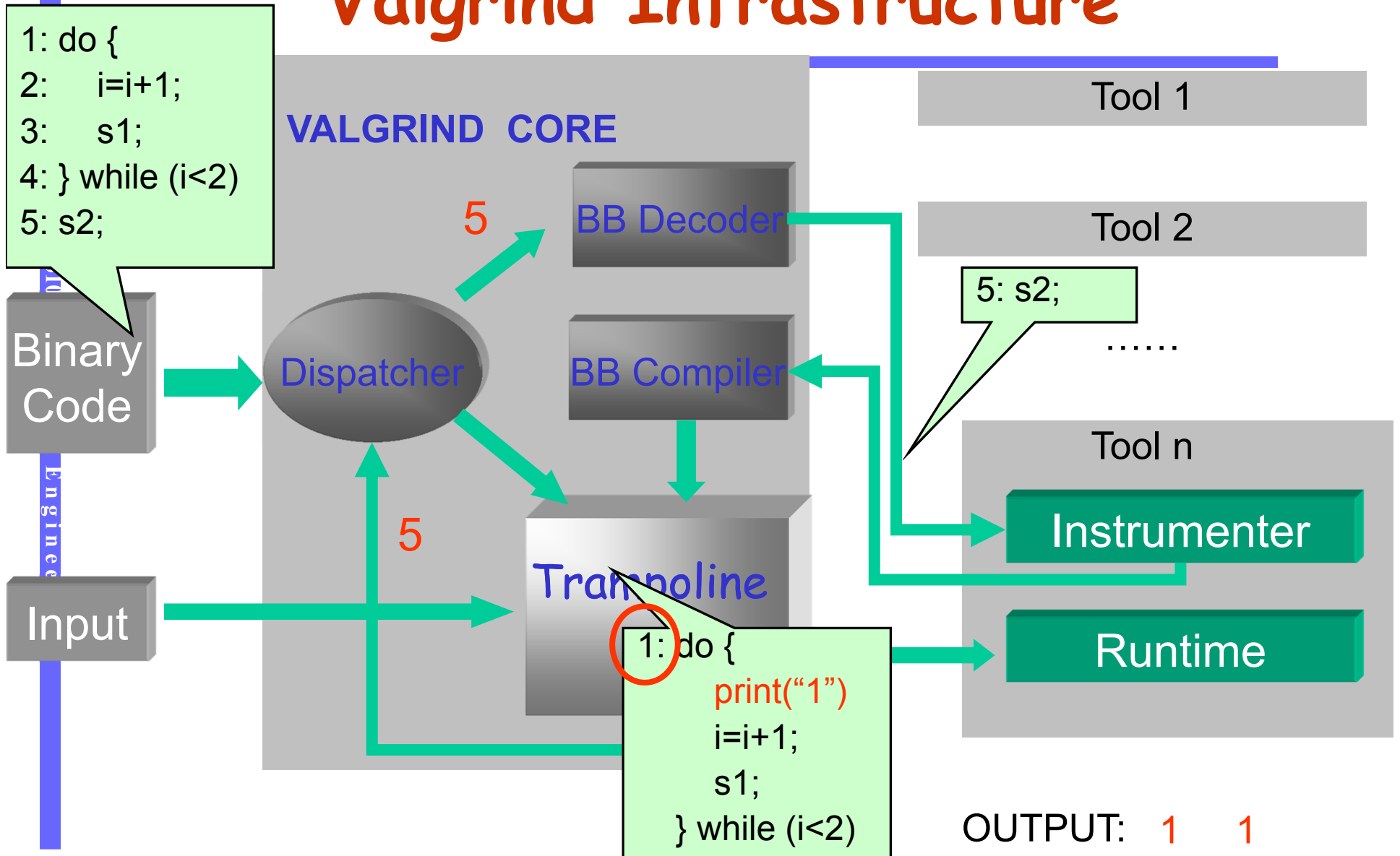
Valgrind Infrastructure



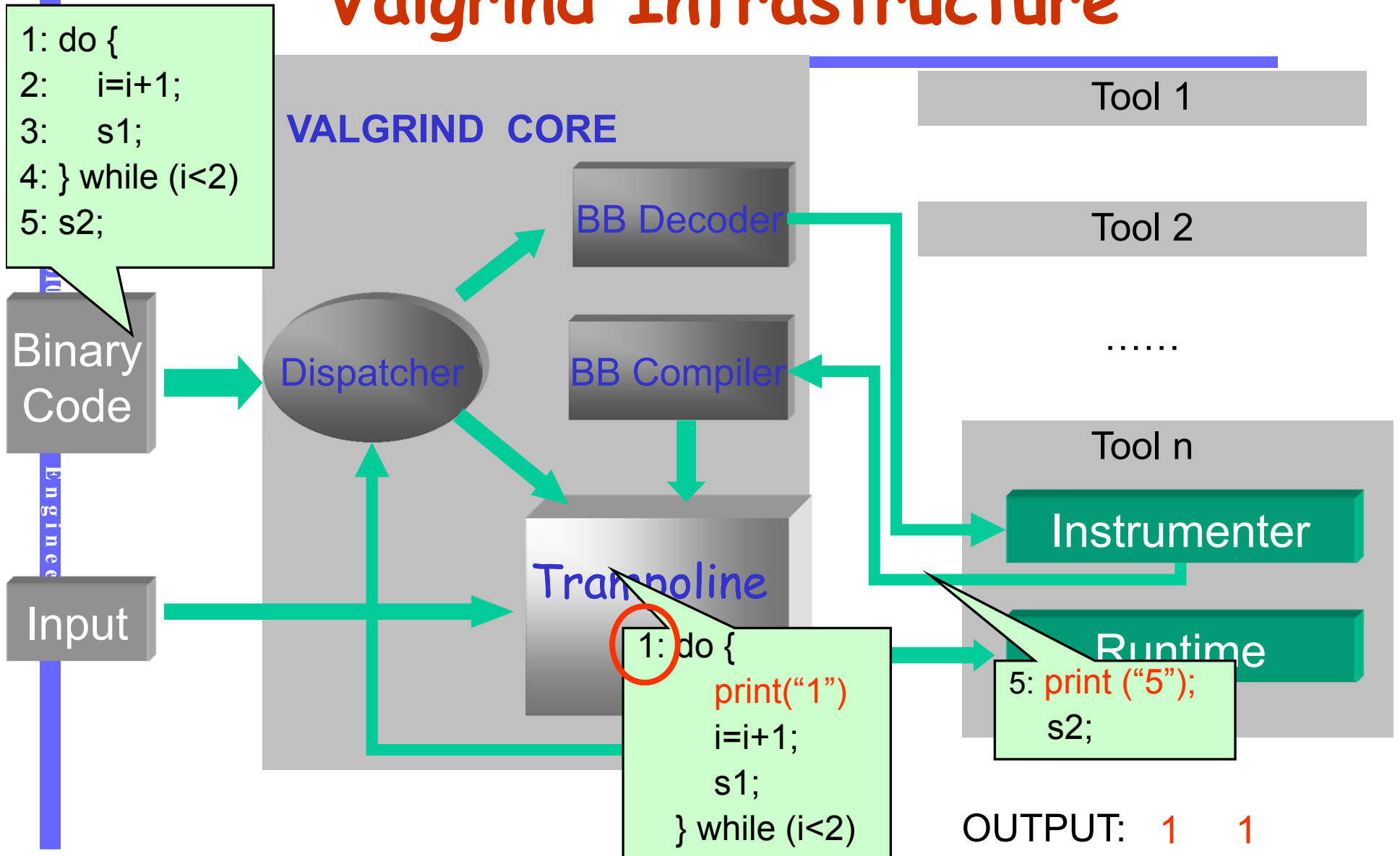
Valgrind Infrastructure



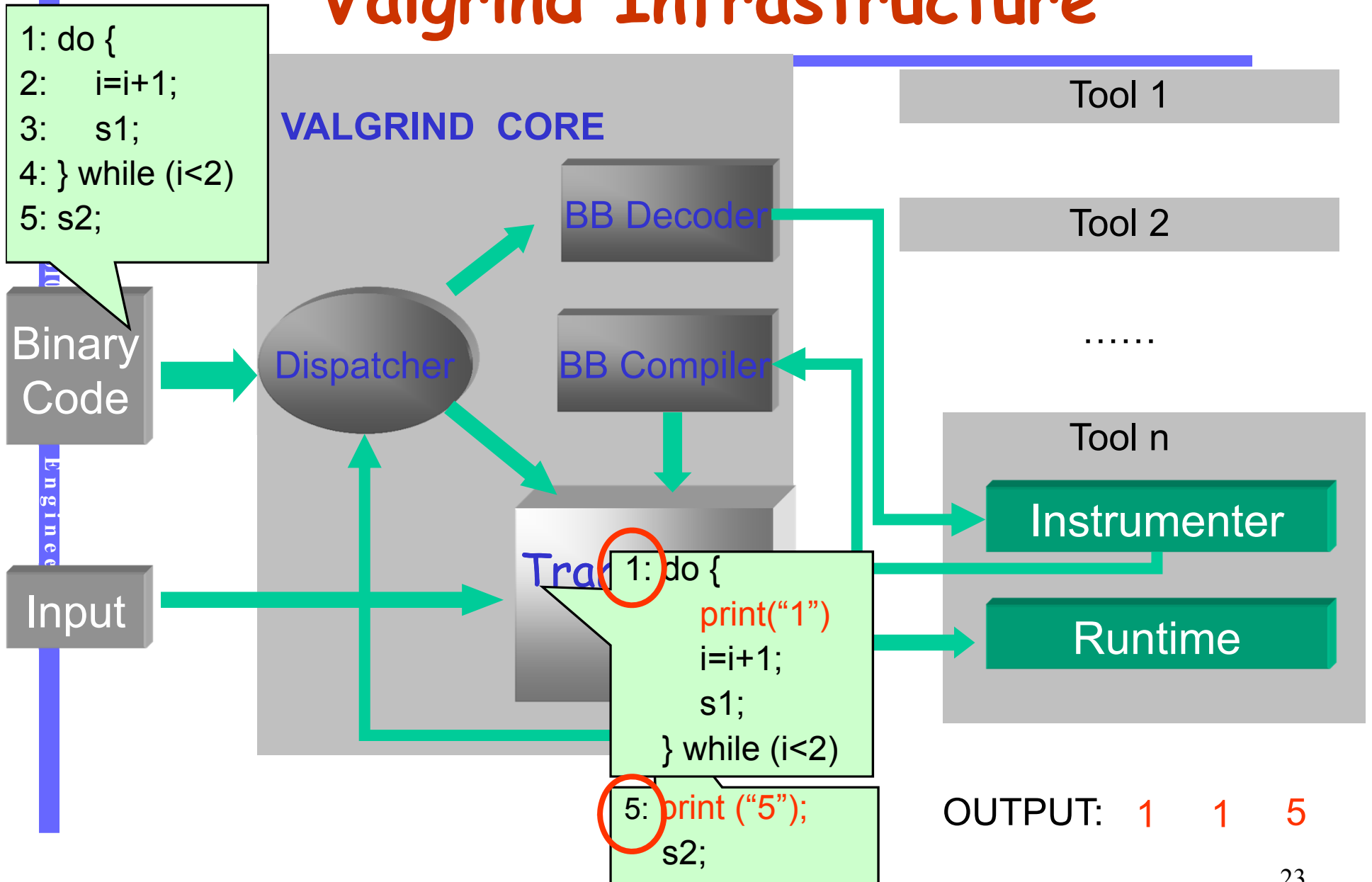
Valgrind Infrastructure



Valgrind Infrastructure



Valgrind Infrastructure



Instrumentation with Valgrind

```
UCodeBlock* SK_(instrument)(UCodeBlock* cb_in, ...)
{
    ...
    UCodeBlock cb = VG_(setup_UCodeBlock)(...);
    ...
    for (i = 0; i < VG_(get_num_instrs)(cb_in); i++) {
        u = VG_(get_instr)(cb_in, i);
        switch (u->opcode) {
            case LD:
                ...
            case ST:
                ...
            case MOV:
                ...
            case ADD:
                ...
            case CALL:
                ...
        }
    }
    return cb;
}
```


Outline

- What is tracing.
- Why tracing.
- How to trace.
- Reducing trace size.



Program Profiling

Xiangyu Zhang

What is Profiling

- Tracing is lossless, recording every detail of a program execution
 - Thus, it is expensive.
 - Potentially infinite.
- Profiling is **lossy**, meaning that it aggregates execution information to finite entries.
 - Control flow profiling
 - Instruction/Edge/Function: Frequency;
 - Value profiling
 - Value: Frequency

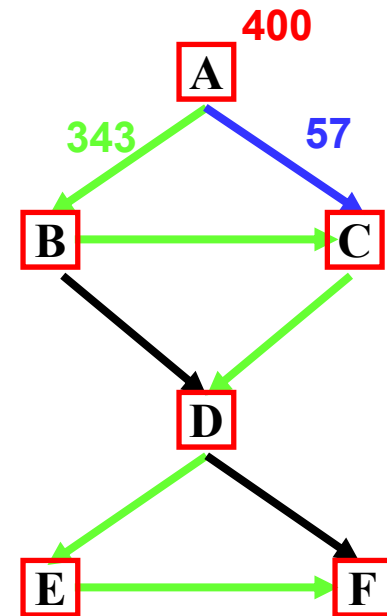
GNU gprof Profiler

- Gprof is a profiler for C programs.
- It profiles execution times for each individual functions and produces a call graph with call edges annotated with frequencies.
- Working
 - Gcc with the -p -pg options. -p tells the program to save profiling information, and -pg saves debug information in the compiled executable.
 - Gcc instruments the entry and exit of each function to record the calling frequency of each function.
 - Sampling is used to measure execution time.
 - inaccuracy
 - A gmon.out file will be created at the end.
 - Run *gprof ./a.out* to view the profiler's information.

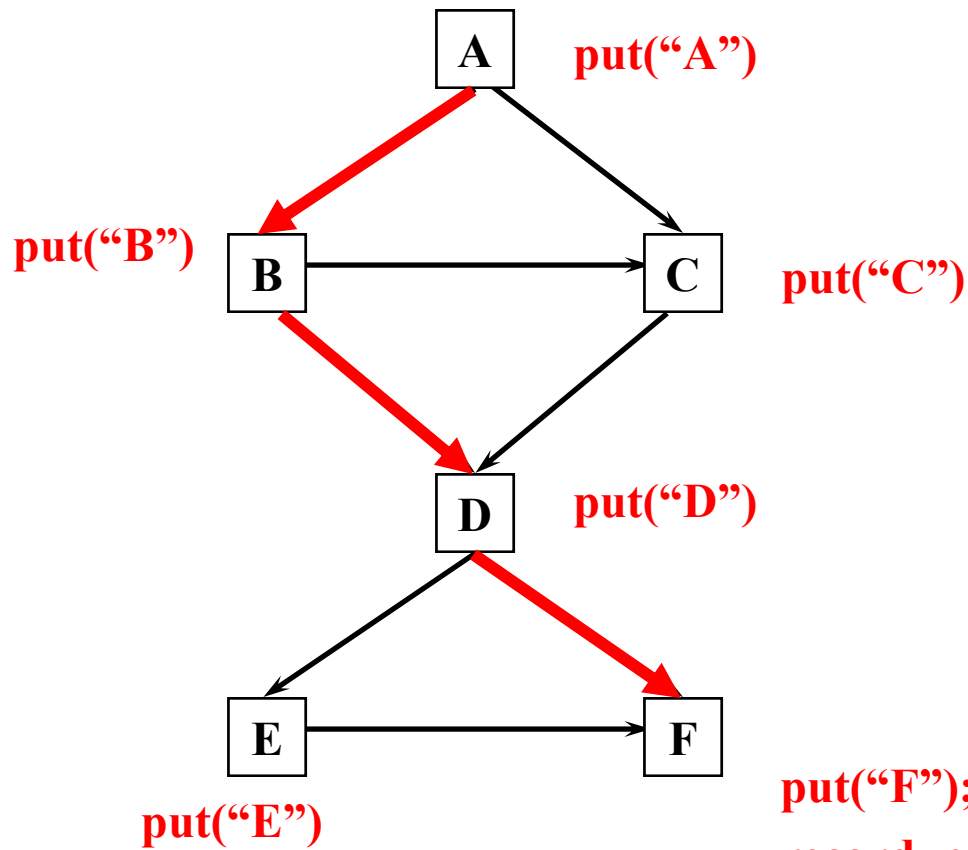
More Advanced: Path Profiling

- How often does a control-flow path execute?
- Levels of profiling:
 - blocks
 - edges
 - paths

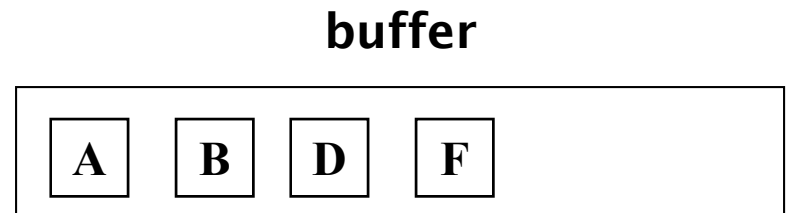
Edge profile equivalent to block profile?



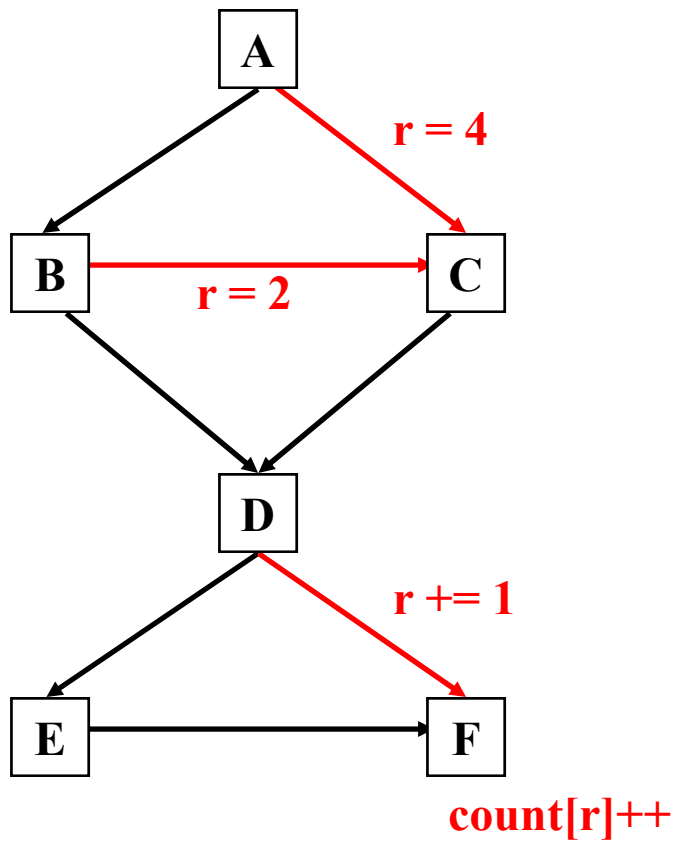
Naive Path Profiling



`put("F");`
`record_path();`

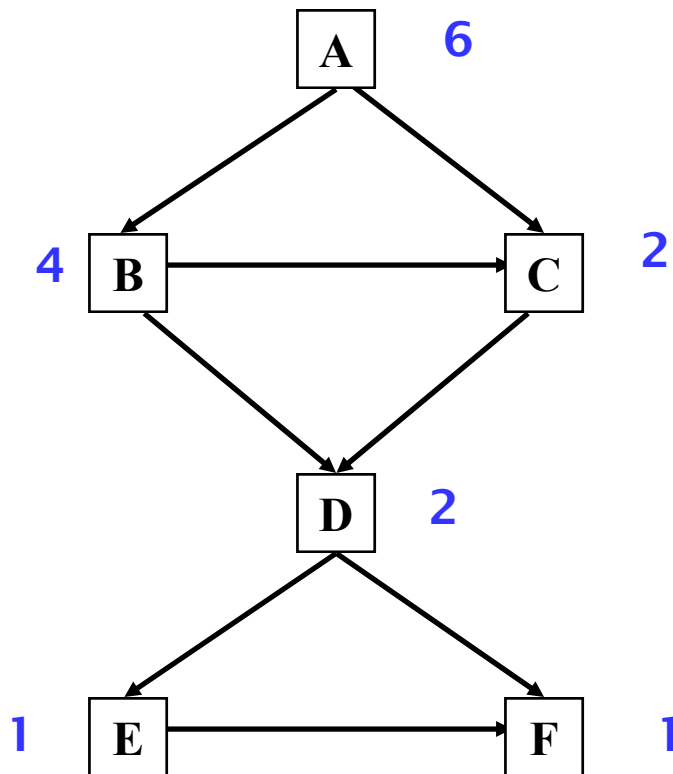


Efficient Path Profiling



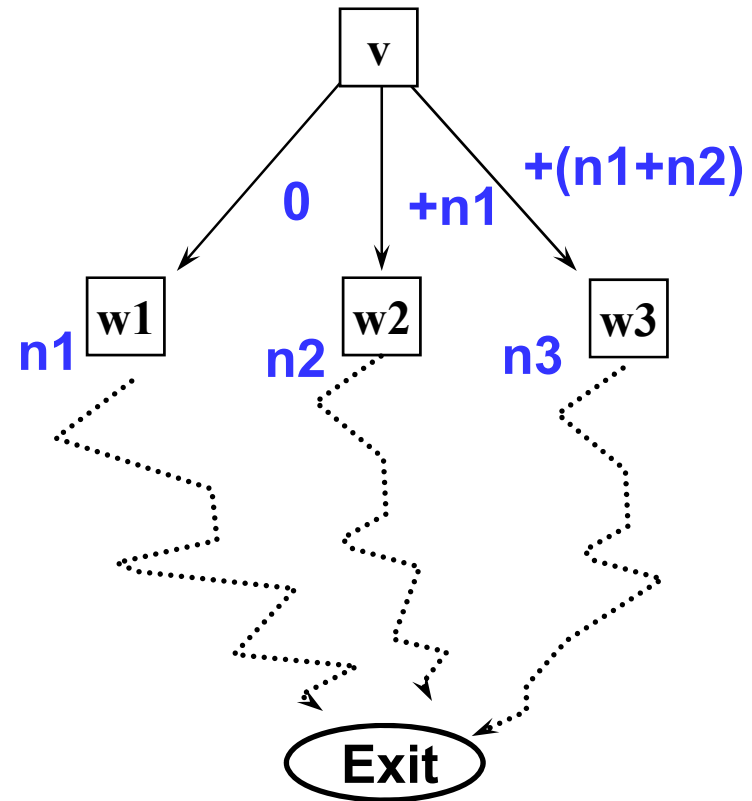
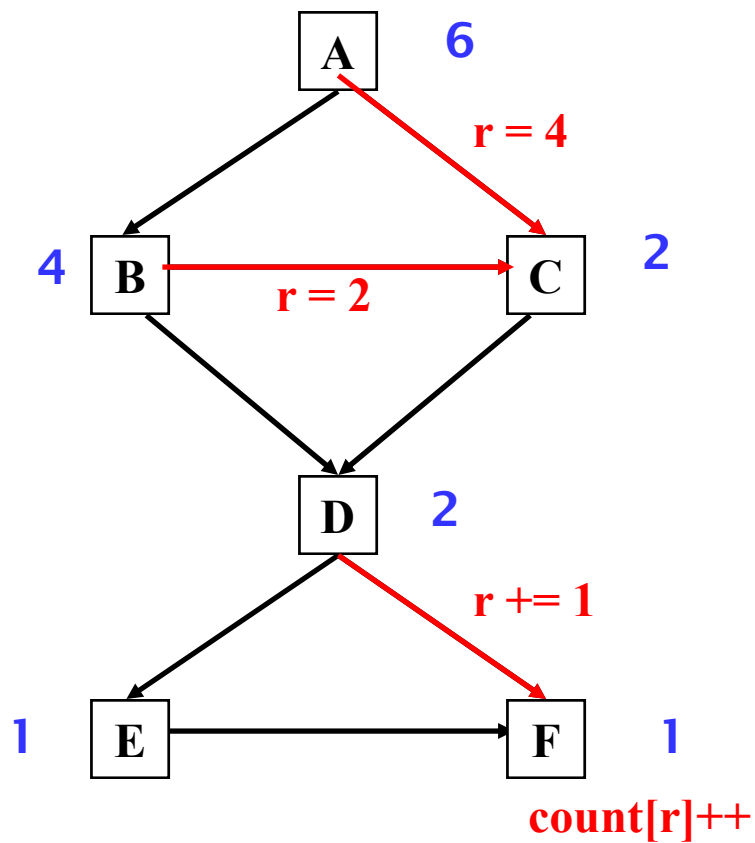
<u>Path</u>	<u>Encoding</u>
ABDEF	0
ABDF	1
ABCDEF	2
ABCDF	3
ACDEF	4
ACDF	5

Efficient Path Profiling



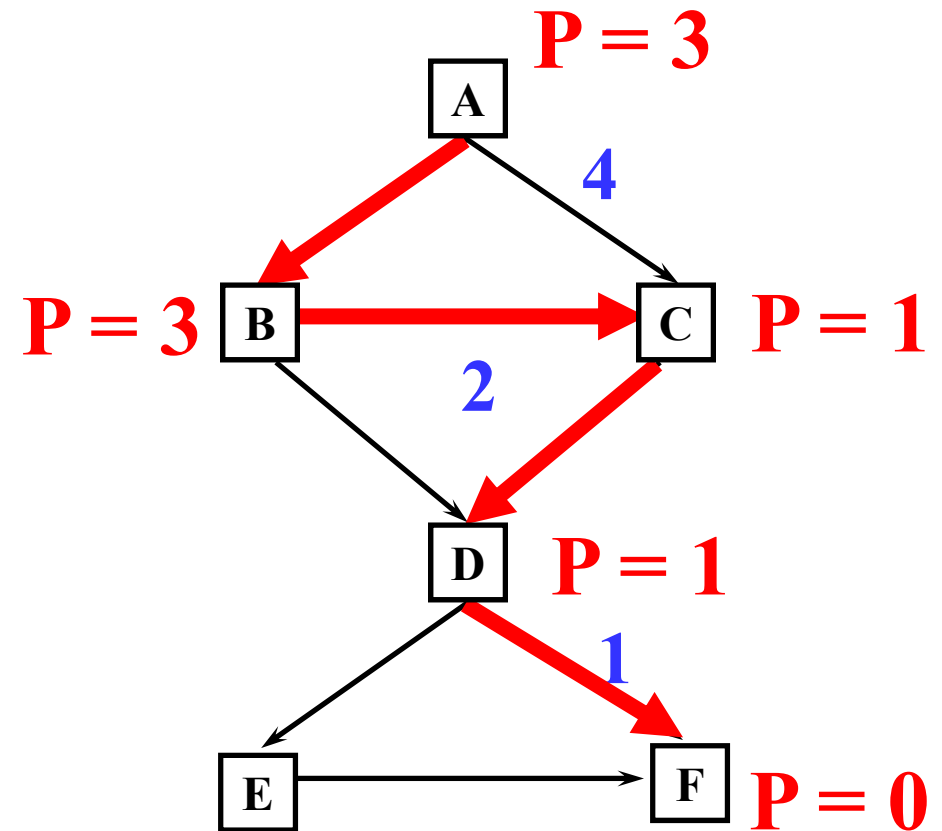
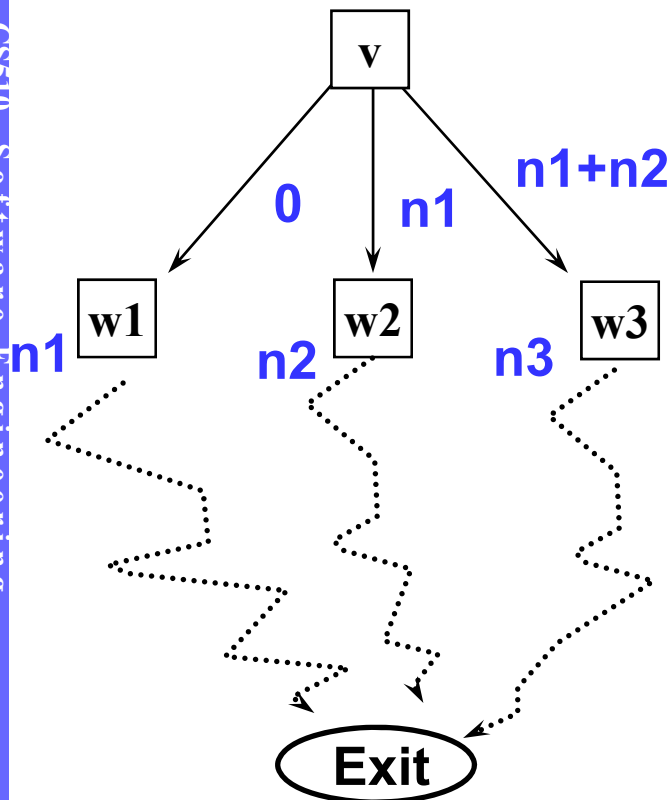
- Each node is annotated with the number of paths from that node to the end.
- $\text{Num}(n) = \sum_{(\text{child } i)} \text{Num}(i)$

Efficient Path Profiling

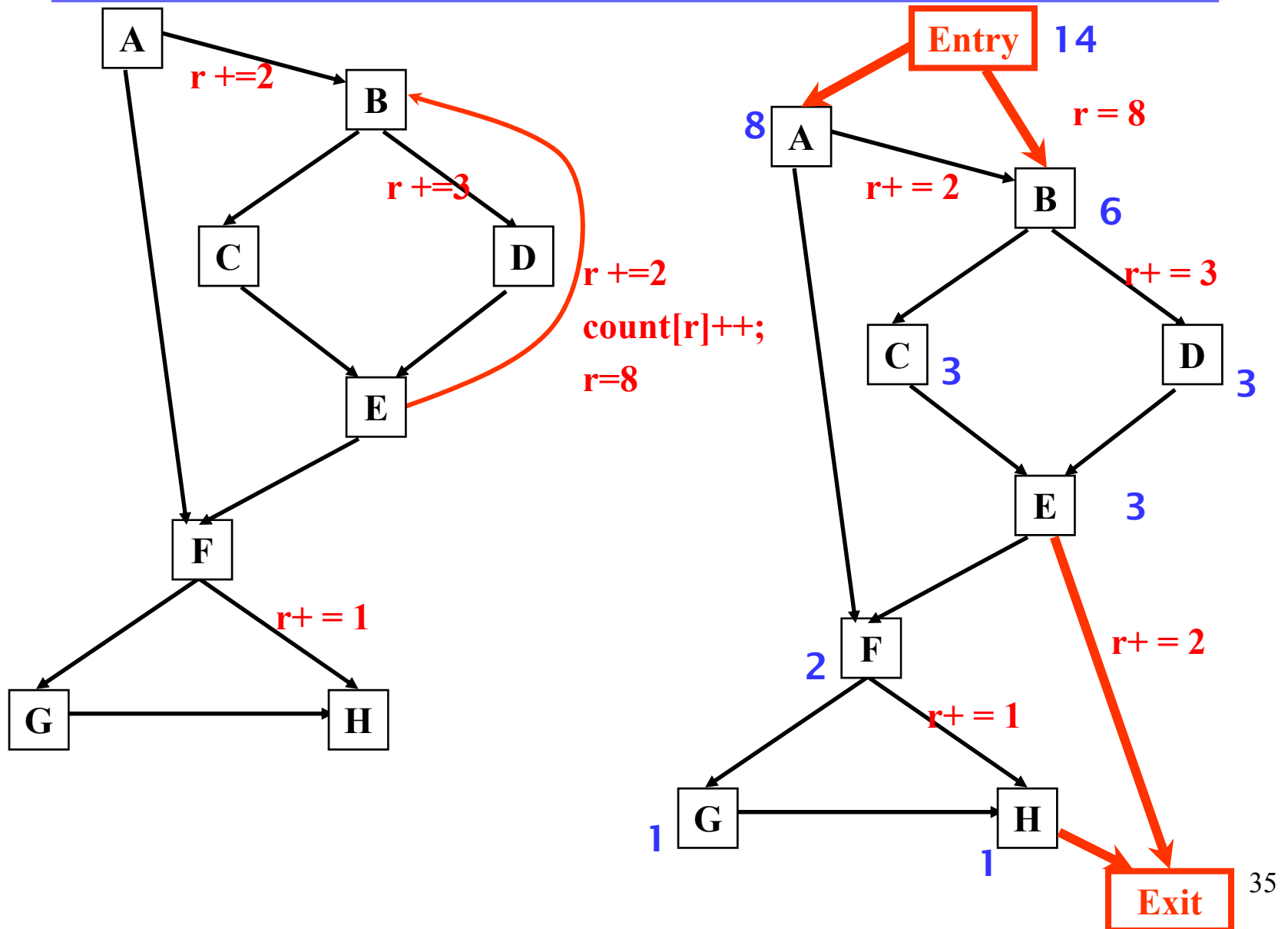


Path Regeneration

Given path sum P , which path produced it?



Handling Loops



Overhead and Others

- EPP causes 40% overhead on average.
- The path explosion problem.
 - If the number of paths is too large to enumerate, which is not very uncommon, hash maps have to be used.
- Can be used to achieve efficient tracing.
- Reading assignment
 - *Efficient Path Profiling*, by T. Ball and J. Larus, Micro 1996
 - The optimization (chord algorithm) is not required.

Challenge

- Encoding backtrace

- A very useful feature of GDB is backtrace, which captures the sequence of call sites leading from the main function to the current execution point. For instance, in case of a segfault, the user can use the "bt" command to investigate the current call sequence.
- Consider the sequence of call sites as a path in the call graph from the main function to the current execution point. Please design an efficient encoding scheme for call paths. The requirement is that it should be able to distinguish the multiple call paths to the same program point. The scheme ought to be minimal, meaning using the minimal number of ids. Apply your technique to the following program.

```
A () { B () { C () { D () { E () { F () {
  B ();   D ();   D ();   E ();   segfault;
  C ();   E ();   }       F ();   }       }
}         }         }         }         }
```

Delta Debugging

Xiangyu Zhang

Problem

- In 1999 Bugzilla, the bug database for the browser Mozilla, listed more than 370 open bugs
- Each bug in the database describes a scenario which caused software to fail
 - these scenarios are not simplified
 - they may contain a lot of irrelevant information
 - a lot of the bug reports could be equivalent
- Overwhelmed with this work Mozilla developers sent out a call for volunteers
 - Process the bug reports by producing simplified bug reports
 - Simplifying means: turning the bug reports into minimal test cases where every part of the input would be significant in reproducing the failure

An Example Bug Report

- Printing the following file causes Mozilla to crash:

```
<td align=left valign=top>
```

```
<SELECT NAME="op sys" MULTIPLE SIZE=7>
```

```
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION
```

```
VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
```

```
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION
```

```
VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
```

```
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac
```

```
System 7.5">Mac System 7.5<OPTION VALUE="Mac
```

```
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System
```

```
8.0<OPTION VALUE="Mac System 8.5">Mac System
```

```
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System
```

```
9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
```

```
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION
```

```
VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
```

```
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION
```

Continued in the next page


```
VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION
VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION
VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT></td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION
VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION
VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION
VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

Delta-Debugging

- It is hard to figure out what the real cause of the failure is just by staring at that file
- It would be very helpful in finding the error if we can simplify the input file and still generate the same failure
- A more desirable bug report looks like this
Printing an HTML file which consists of:
`<SELECT>`
causes Mozilla to crash.
- The question is: Can we automate this?
- Andreas Zeller


Overview

- Let's use a smaller bug report as a running example:

When Mozilla tries to print the following HTML input it crashes:
<SELECT NAME="priority" MULTIPLE SIZE=7>

- How do we go about simplifying this input?
 - Manually remove parts of the input and see if it still causes the program to crash
- For the above example assume that we remove characters from the input file

Bold parts remain in the input, the rest is removed



1	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
2	<SELECT NAME="priority" MULTIPLE SIZE=7 >	P
3	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
4	<SELECT NAME="priority" MULTIPLE SIZE=7 >	P
5	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
6	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
7	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
8	<SELECT NAME="priority" MULTIPLE SIZE=7 >	P
9	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
10	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
11	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
12	<SELECT NAME="priority" MULTIPLE SIZE=7 >	P
13	<SELECT NAME="priority" MULTIPLE SIZE=7>	P

F means input caused failure
P means input did not cause failure (input passed)

14	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
15	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
16	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
17	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
18	<SELECT NAME="priority" MULTIPLE SIZE=7>	F
19	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
20	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
21	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
22	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
23	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
24	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
25	<SELECT NAME="priority" MULTIPLE SIZE=7>	P
26	<SELECT NAME="priority" MULTIPLE SIZE=7>	F

Example

- After 26 tries we found that:

Printing an HTML file which consists of:

<SELECT>

causes Mozilla to crash.

- Delta debugging technique automates this approach of repeated trials for reducing the input.

A Simplified Description of the Algorithm

Initially, $n=2$

(1) Divide a string S equally into $\Delta_1, \Delta_2, \dots, \Delta_n$ and the respective complements are $\nabla_1, \nabla_2, \dots, \nabla_n$.

(2) Test each $\Delta_1, \Delta_2, \dots, \Delta_n$ and $\nabla_1, \nabla_2, \dots, \nabla_n$.

if (all pass) {

$n=2n$;

 if ($n > |s|$) return the most recent failure inducing substring.

 else goto (1)

} else if (Δ_+ fails) {

$n=2$; $s = \Delta_+$

 if ($|s| == 1$) return s

 else goto (1)

} else { /* ∇_+ fails */

$s = \nabla_+$; $n=n-1$; goto (1);

}

Examples

- a b c d e f * h
 - Program fails on any substrings containing '*'
- a b c d e f g h
 - Any strings containing a g h fail
- *abcdef*",
 - the program fails if both *s appear in the input

Minimality

- A test case $c \subseteq c_F$ is called the *global minimum* of c_F if
for all $c' \subseteq c_F$, $|c'| < |c| \Rightarrow \text{test}(c') \neq F$
- Global minimum is the smallest set of changes which will make the program fail
- Finding the global minimum may require us to perform exponential number of tests

Minimality

- A test case $c \subseteq c_F$ is called a local minimum of c_F if for all $c' \subseteq c$, $\text{test}(c') \neq F$
- A test case $c \subseteq c_F$ is n -minimal if for all $c' \subseteq c$, $|c| - |c'| \leq n \Rightarrow \text{test}(c') \neq F$
- The delta debugging algorithm finds a 1-minimal test case

Ex: AAAABBBBCCCC, program fails when
 $|A|=|B|=|C|>0$

Monotonicity

- The super string of a failure inducing string always induces the failure
- DD is not effective for cases without monotonicity.

Case Studies

- The following C program causes GCC to crash

```
#define SIZE 20
double mult(double z[], int n)
{
    int i , j ;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] *(z[0]+1.0);
    }
    return z[n];
}
```

Continued in the next page

```
void copy(double to[], double from[], int count)
{
    int n = count + 7) / 8;
    switch(count % 8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while ( --n > 0);
    return mult(to, 2);
}
int main(int argc, char *argv[])
{
    double x[SIZE], y[SIZE];
    double *px = x;
    while (px < x + SIZE)
        *px++ = (px - x) * (SIZE + 1.0);
    return copy(y, x, SIZE);
}
```

Case Studies

- The original input file 755 characters
- Delta debugging algorithm minimizes the input file to the following file with 77 characters

```
t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*  
(z[0]+0);}return[n];}
```

- If a single character is removed from this file then it does not induce the failure

Isolating Failure Inducing Differences

- Instead of minimizing the input that causes the failure we can also try to isolate the differences that cause the failure
 - Minimization means to make each part of the simplified test case relevant: removing any part makes the failure go away
 - Isolation means to find one relevant part of the test case: removing this particular part makes the failure go away
- For example changing the input from
<SELECT NAME="priority" MULTIPLE SIZE=7>
to
SELECT NAME="priority" MULTIPLE SIZE=7>
makes the failure go away
 - This means that inserting the character < is a failure inducing difference
- Delta debugging algorithm can be modified to look for minimal failure inducing differences
 - Although it is not as popular, it is quite useful in some applications.

Failure Inducing Differences: Example

- Changing the input program for GCC from the one on the left to the one on the right removes the failure

This input causes failure

```
#define SIZE 20
double mult(double z[], int n)
{
    int i , j ;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] *(z[0]+1.0);
    }
    return z[n];
}
```

This input does not cause failure

```
#define SIZE 20
double mult(double z[], int n)
{
    int i , j ;
    i = 0;
    for (j = 0; j < n; j++) {
        i + j + 1;
        z[i] = z[i] *(z[0]+1.0);
    }
    return z[n];
}
```

Modified statement is shown in box

Discussions

- DD on scheduling decisions:
 - Given a thread schedule for which a concurrent program works and another for which the program fails, delta debugging algorithm can narrow down the differences between two thread schedules and find the locations where a thread switch causes the program to fail.
- Chipping
 - Given two versions of a program such that one works correctly and the other one fails, delta debugging algorithm can be used to look for changes which are responsible for introducing the failure
- Fault Localization – apply DD to memory state