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;
   }
}
```

The Minimum Set of Places to Instrument

```
if (…)
   S1
else
   S2
S3
if (…)
   S4
else
   S5
```

```
if (…)
   S1
   if (…)
      S2
else
   S3
```

Tracing by Source Level Instrumentation

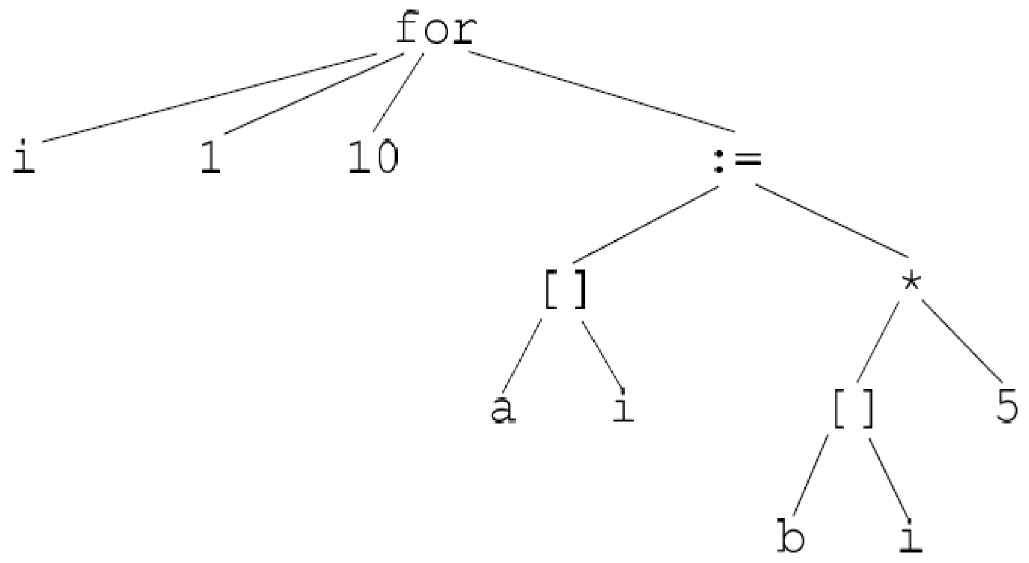
- Read a source file and parse it into ASTs.
- Annotate the parse trees with instrumentation.
- Translate the annotated trees to a new source file.
- Compile the new source.
- Execute the program and a trace produced.

An Example

Source:

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

AST:

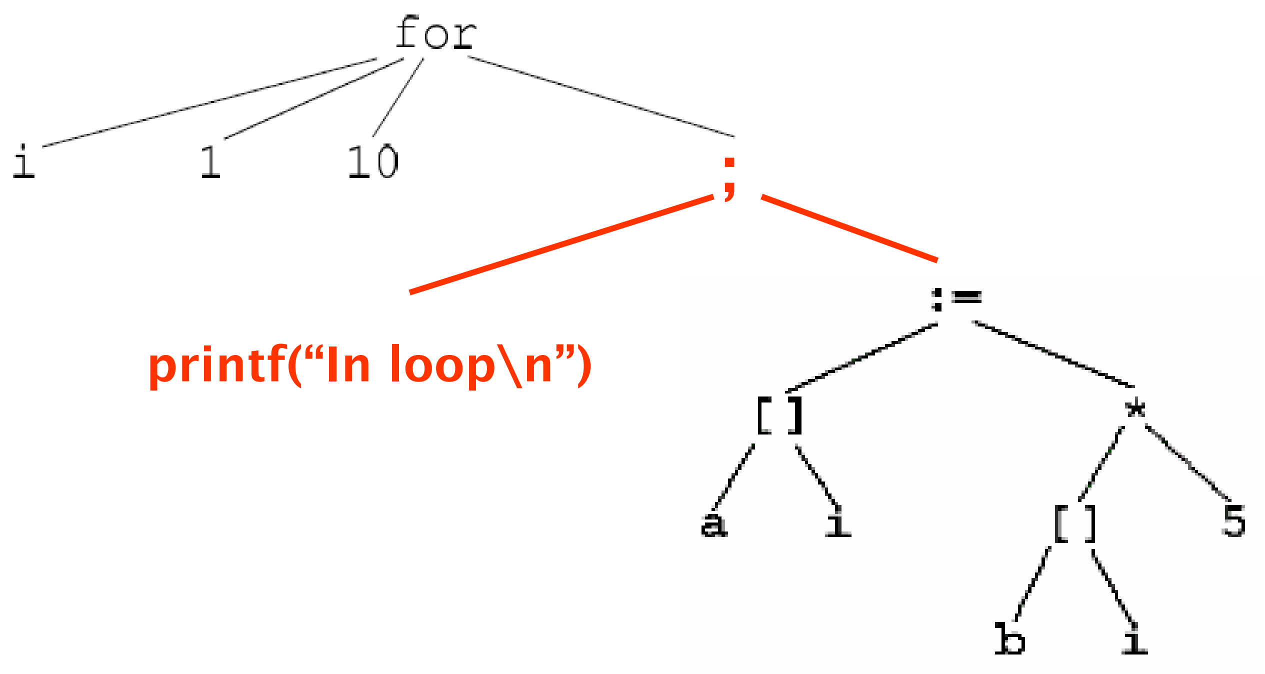


An Example

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.

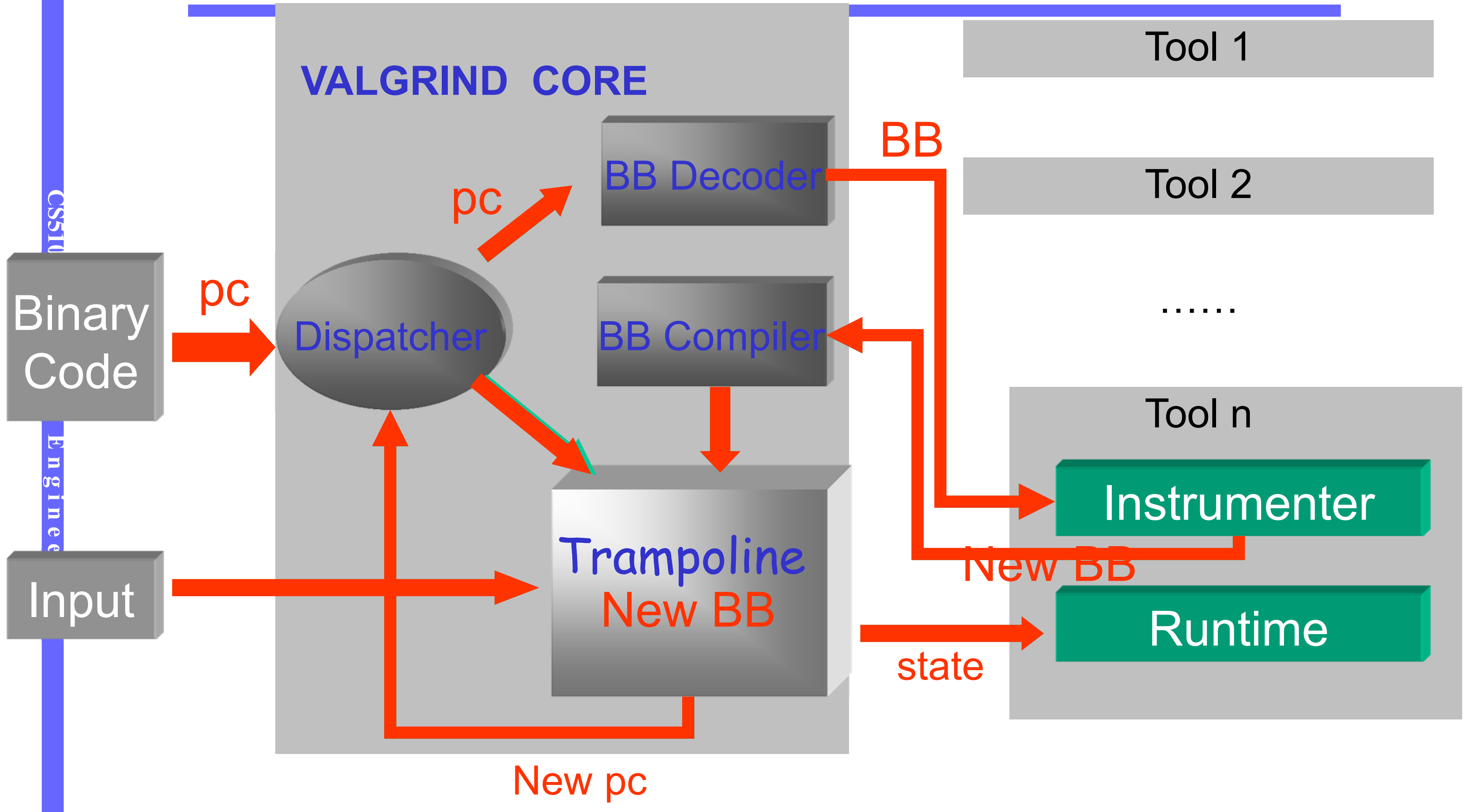
Static vs. Dynamic Instrumentation

- 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.

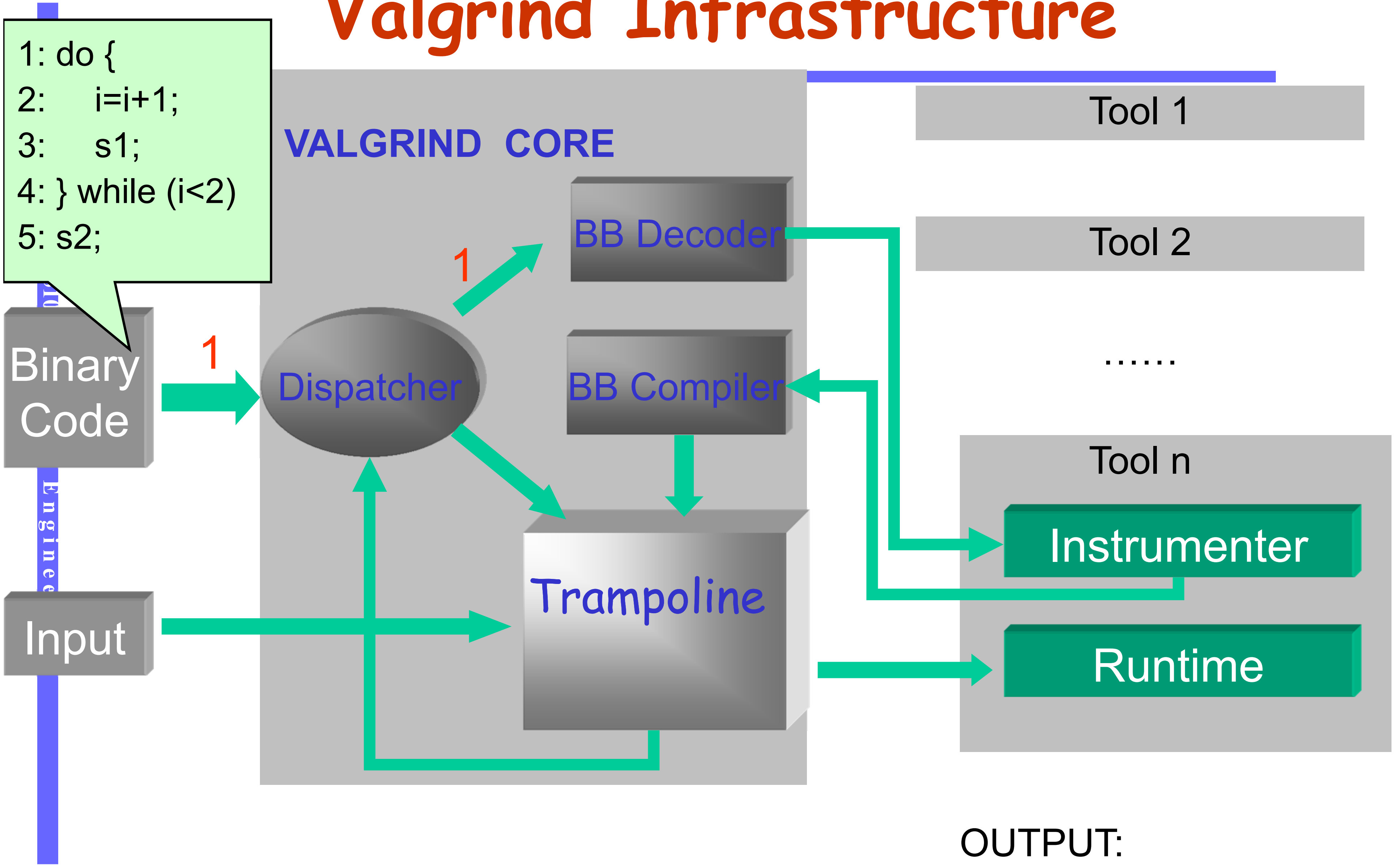
Dynamic Binary Instrumentation - Valgrind

- 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

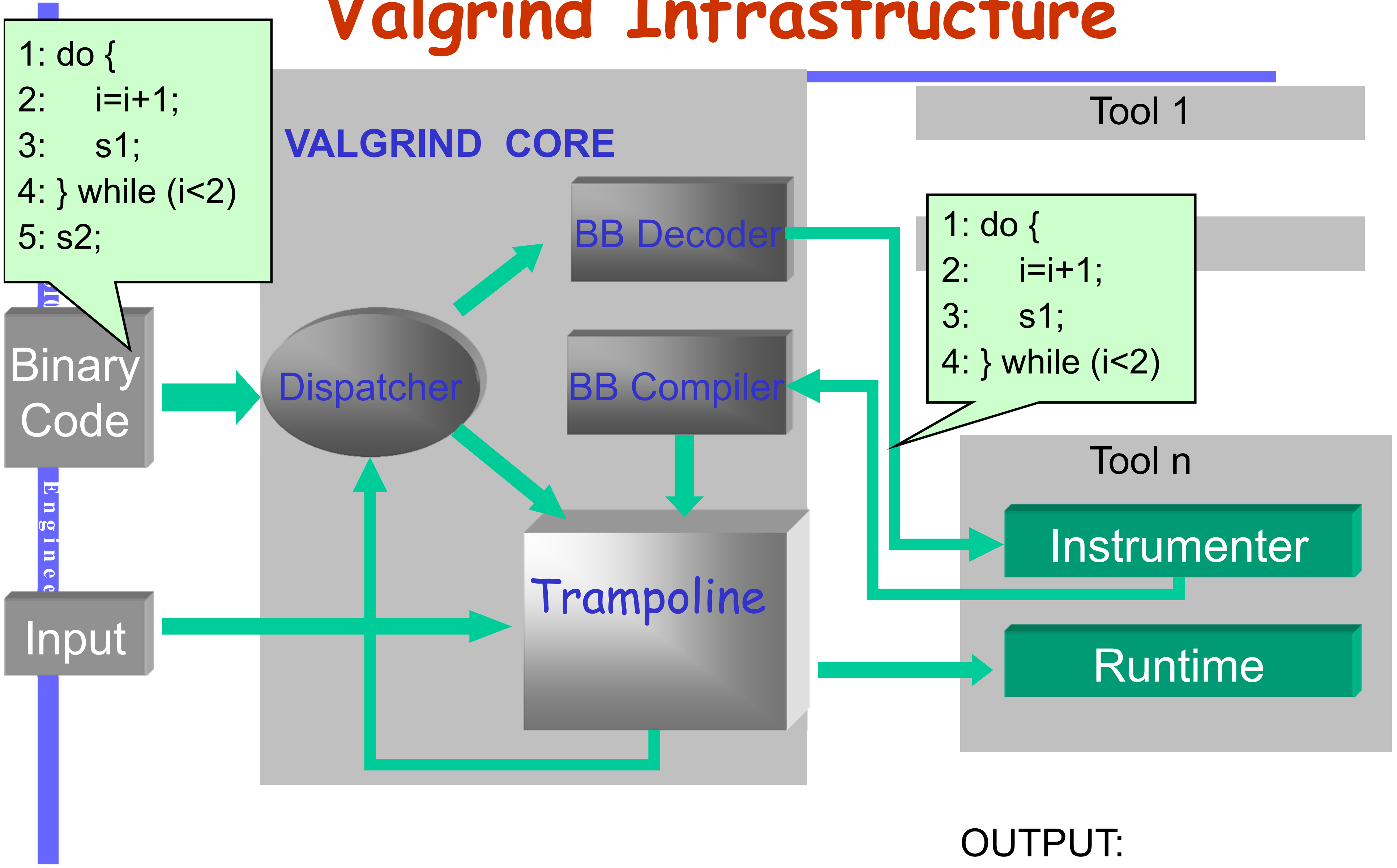


Valgrind Infrastructure



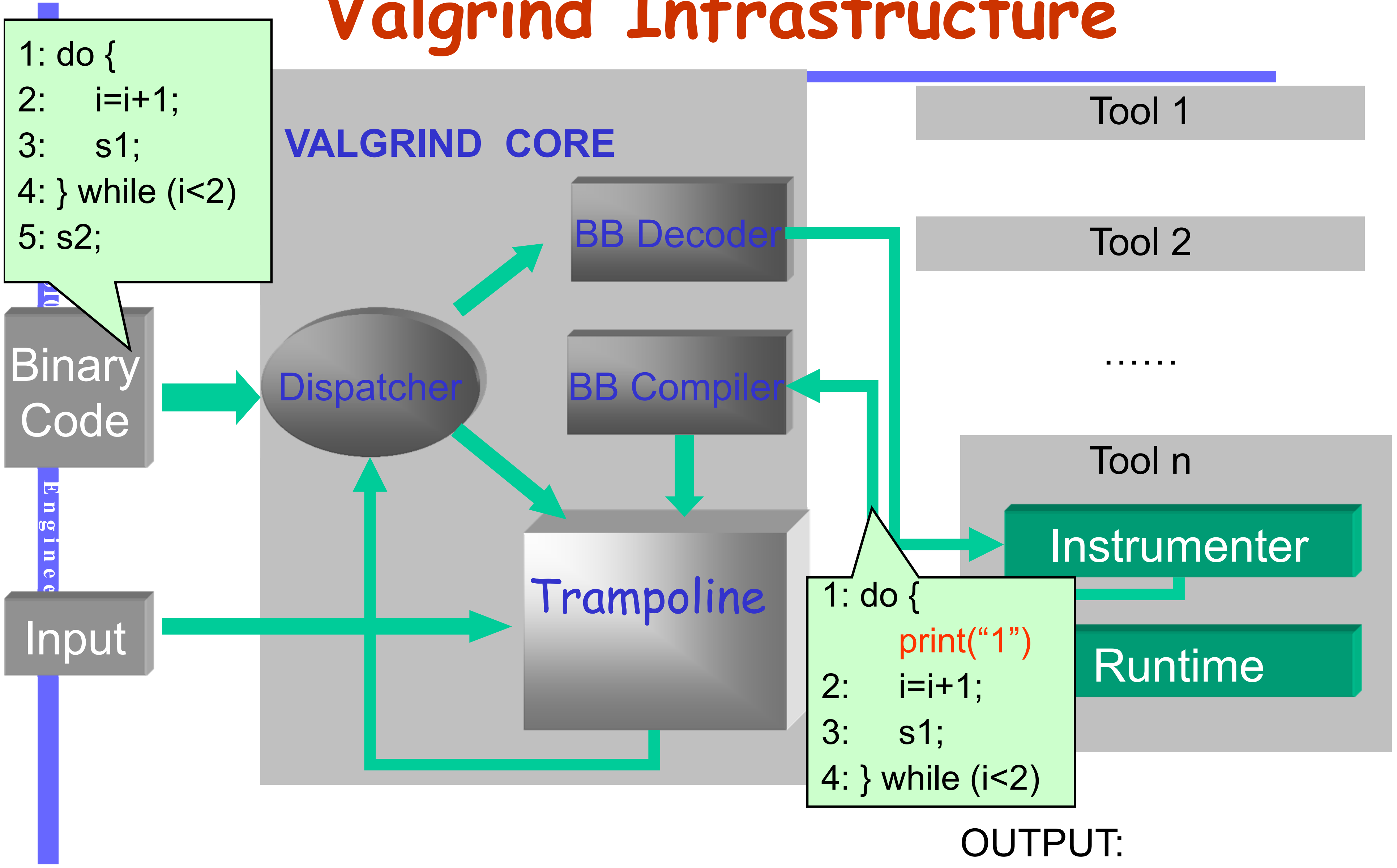
OUTPUT:

Valgrind Infrastructure

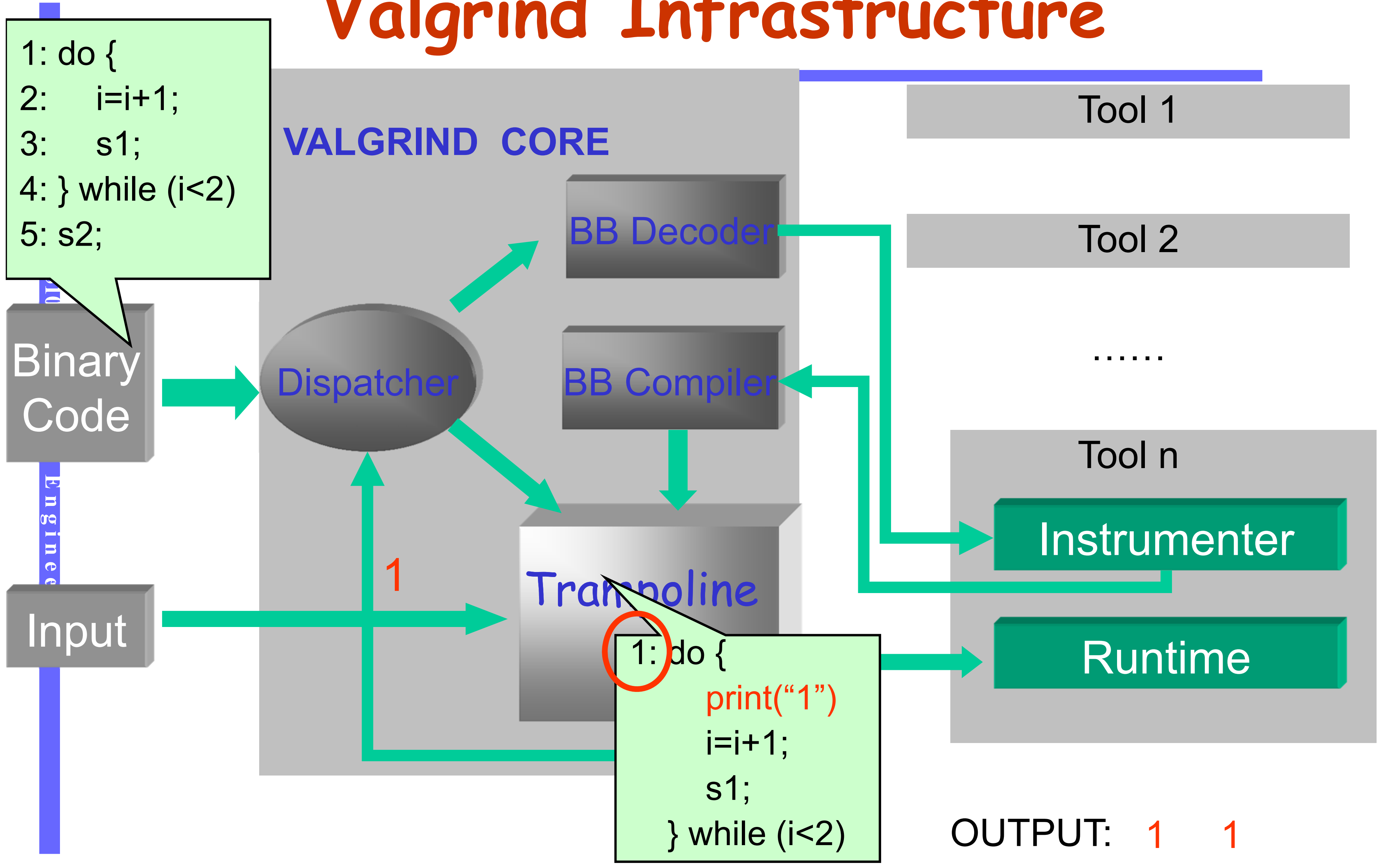


OUTPUT:

Valgrind Infrastructure



Valgrind Infrastructure

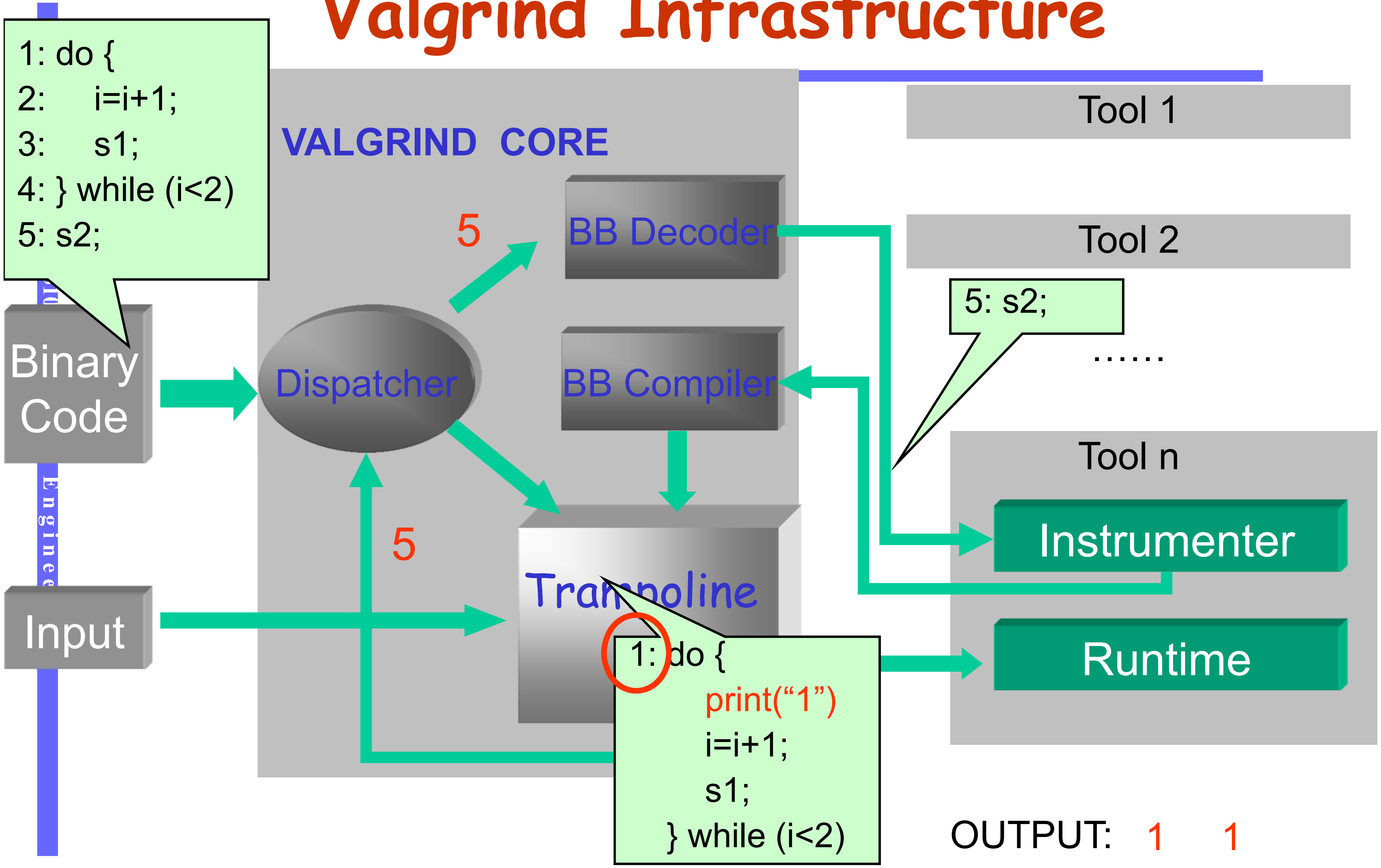


```
1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;
```

```
1: do {
       print("1")
       i=i+1;
       s1;
   } while (i<2)
```

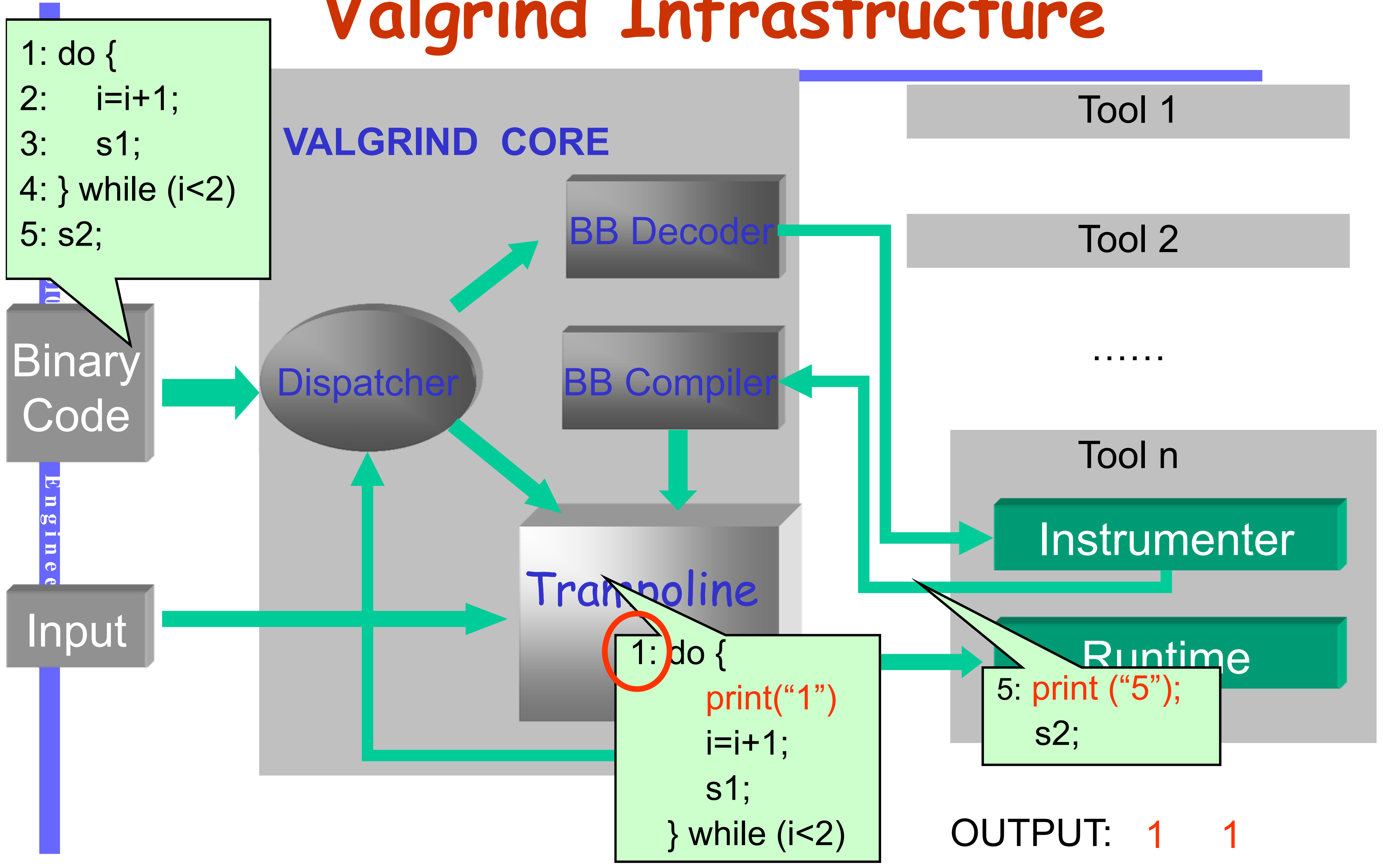
OUTPUT: 1 1

Valgrind Infrastructure

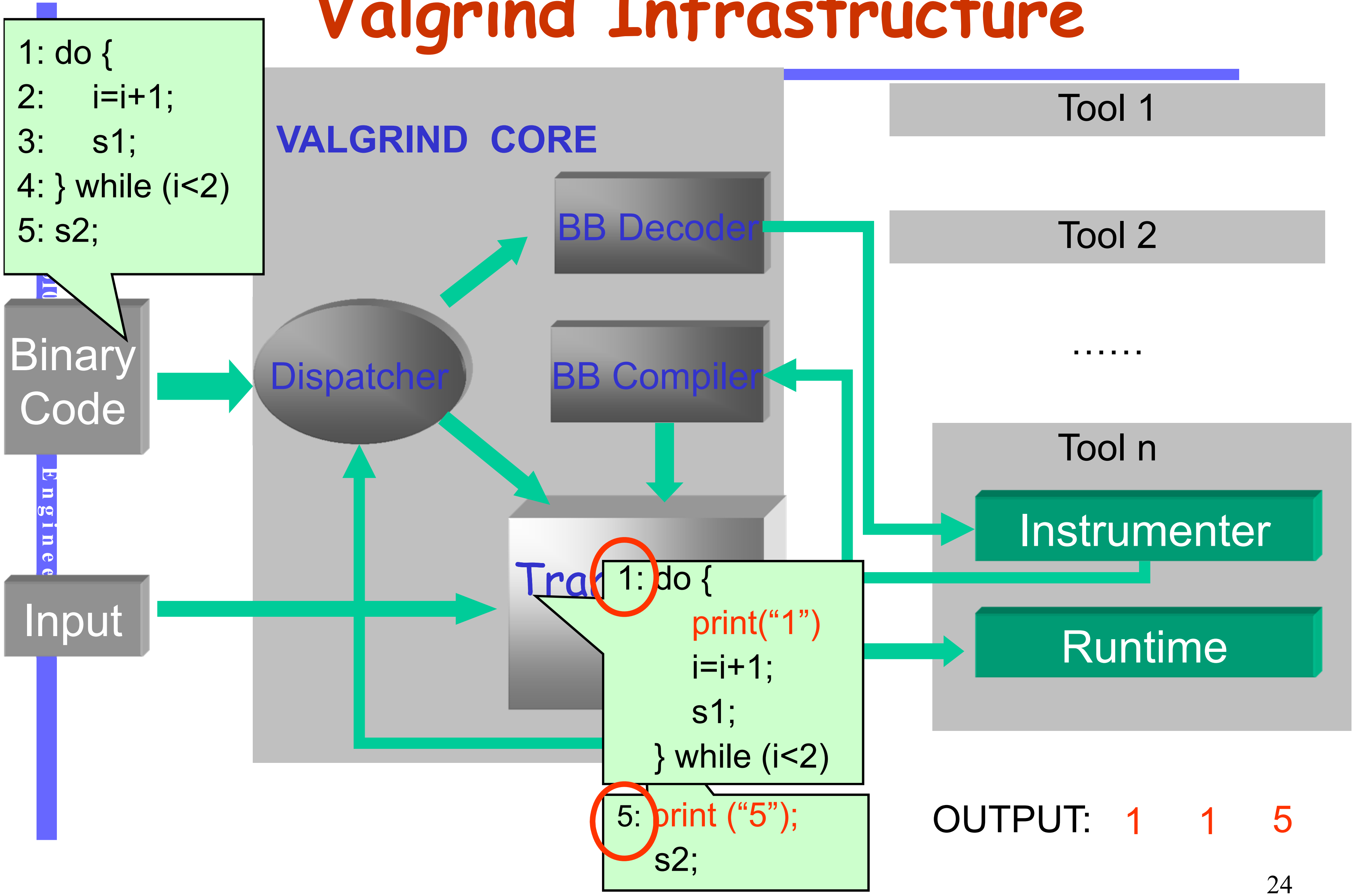


OUTPUT: 1 1

Valgrind Infrastructure



Valgrind Infrastructure



```
1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;
```

```
1: do {
       print("1")
       i=i+1;
       s1;
   } while (i<2)
5: print ("5");
   s2;
```

OUTPUT: 1 1 5

Instrumentation with Valgrind

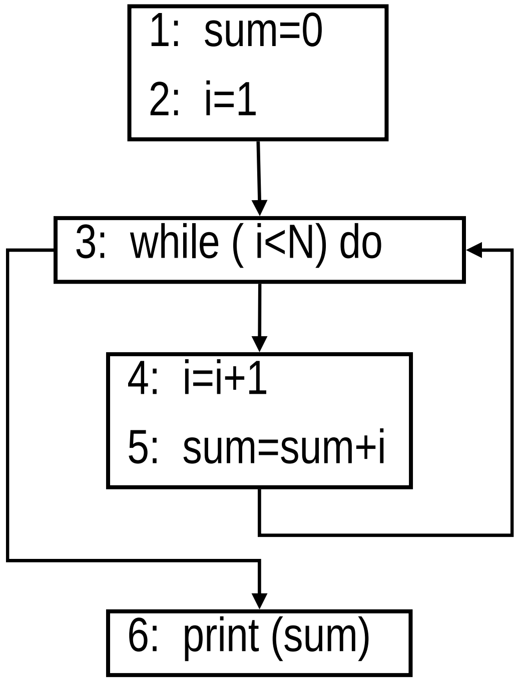
```
UCodeBlock* SK_(instrument)(UCodeBlock* cb_in, …)
{
        …
        UCodeBlock cb = VG_(setup_UCodeBlock)(…);
        …
        for (i = 0; i < VG_(get_num_instrs)(cb_in); i++) {
                u = VG_(get_instr)(cb_in, i);
                switch (u->opcode) {
                      case LD:
 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

Outline

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

Fine-Grained Tracing is Expensive

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:            i=i+1
5:            sum=sum+i
      endwhile
6:   print(sum)
```

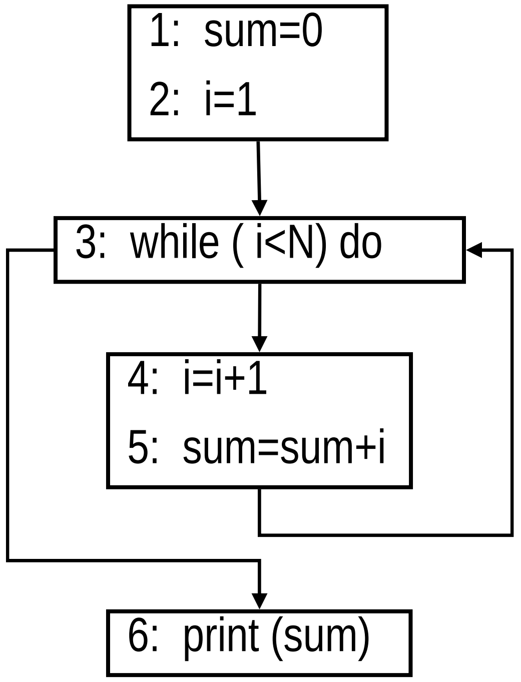


Trace(N=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6

Space Complexity: 4 bytes * Execution length

Basic Block Level Tracing

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:            i=i+1
5:            sum=sum+i
      endwhile
6:  print(sum)
```



Trace(N=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6

BB Trace: 1   3 4   3 4   3 4   3 4   3 4   3 6

More Ideas

- Would a function level tracing idea work?
  - A trace entry is a function call with its parameters.
- Predicate tracing

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:            i=i+1
5:            sum=sum+i
      endwhile
6:   print(sum)
```

Instruction trace	Predicate trace
1 2 3 6	F
1 2 3 4 5 3 6	T F

  - Lose random accessibility
- Path based tracing

Compression

- Using zlib
  - Zlib is a software library used for data compression. It wraps the compression algorithm used in gzip.
  - Divide traces into trunks, and then compress them with zlib.
  - Disadvantage: trace can only be accessed after complete decompression; slow
- Desired features
  - Accessing traces in their compressed form.
  - Traversing forwards and backwards.
  - fast

Compression using value predictors

- Last n values predictor
  - Facilitated by a buffer that stores the last n unique values encountered
  - If the next value is one of the n values, the index of the value (in [0, n-1]) is emitted to the encoded trace, prefixed with a bit 0 to indicate the prediction is correct.
  - Otherwise (mis-prediction), the original value (32 bits) is emitted to the encoded trace, prefixed with a bit 1 to indicate mis-prediction. The buffer is updated with least used strategy.

Example:

999 333 999 333 999 999 999 333 use last-2 predictor

1 <u>999</u> 1 <u>333</u> 00 01 00 00 00 01 (underlined are 32 bits)

999 333 555 555 999 333 999 999 999 333

Compression using value predictors

- Decompression
  - Take one bit from the encoded trace, if it is 1, emit the next 32 bits. If it is 0, emit the value in the buffer indexed by the next log n bits.
  - Maintain the table in the same way as compression

Compression using value predictors

- Finite Context Method (FCM)
  - Facilitated by a look up table that predicts a value based on the context of left n values. 2-FCM, 3-FCM
  - If the next value can be found in the table through its left context, a bit 0 is emitted to the encoded trace.
  - Otherwise (mis-prediction), the original value (32 bits) is emitted to the encoded trace, prefixed with a bit 1 to indicate mis-prediction. The lookup table is updated accordingly.

Example:

1 2 3 4 5 3 4 5 3 4 5 … 3 4 5 6

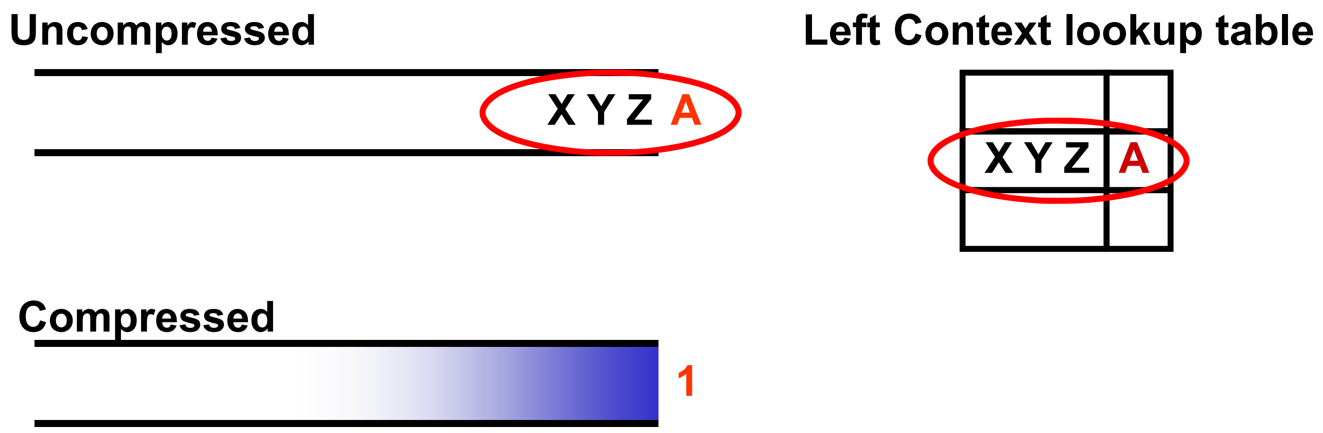
1 $\underline{1}$ 1 $\underline{2}$ 1 $\underline{3}$ 1 $\underline{4}$ 1 $\underline{5}$ 1 $\underline{3}$ 1 $\underline{4}$ 0 0 0 0 … 0 1 $\underline{6}$ (underlined are 32 bits)

Compression using value predictors

- Decompression
  - Take one bit from the encoded trace, if it is 1, emit the next 32 bits. If it is 0, emit the value looked up from the table using the left n values.
  - Maintain the table in the same way as compression

Compression using value predictors

- FCM (finite context method).
  - Example, FCM-3

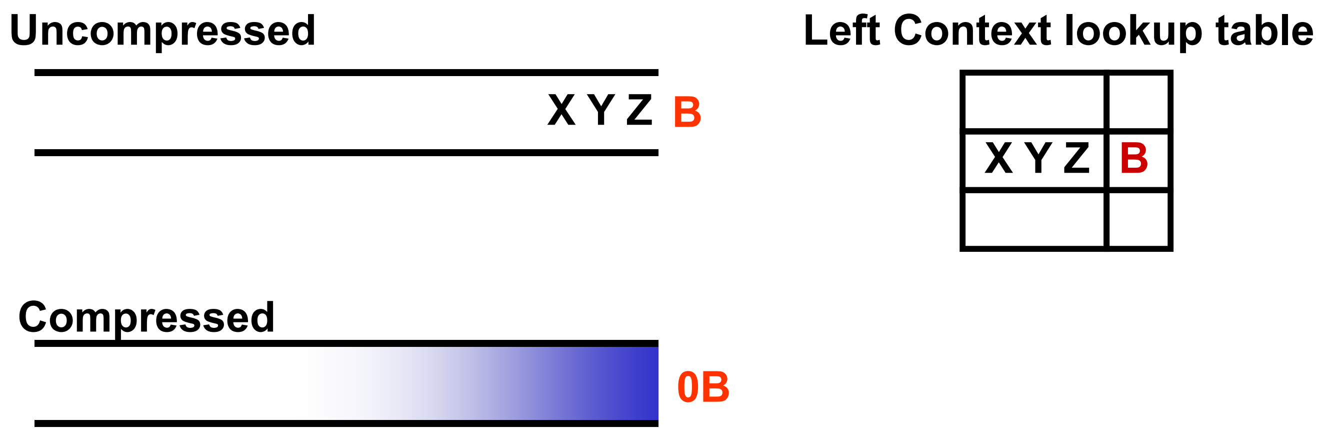


Intel PT

- https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing

Compression using value predictors

- FCM (finite context method).
  - Example, FCM-3



Length(Compressed) = n/32 + n*(1- prediction rate)

It was shown that predictors are better than zlib;

It works so well because the repetitive pattern caused by loops;

Only forward traversable;

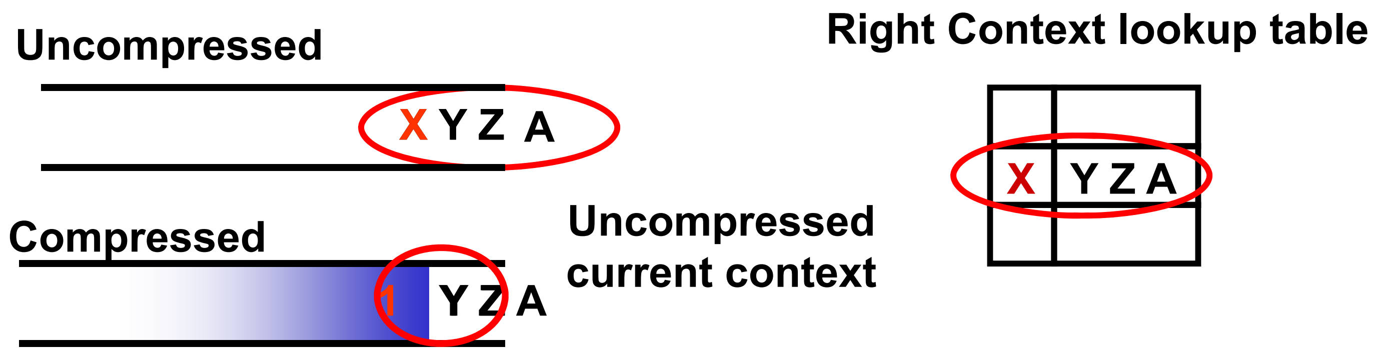
Bidirectional Compression

- Allow trace traversal in the compressed form.
- Bidirectional.
- Fast.
- Good compression.

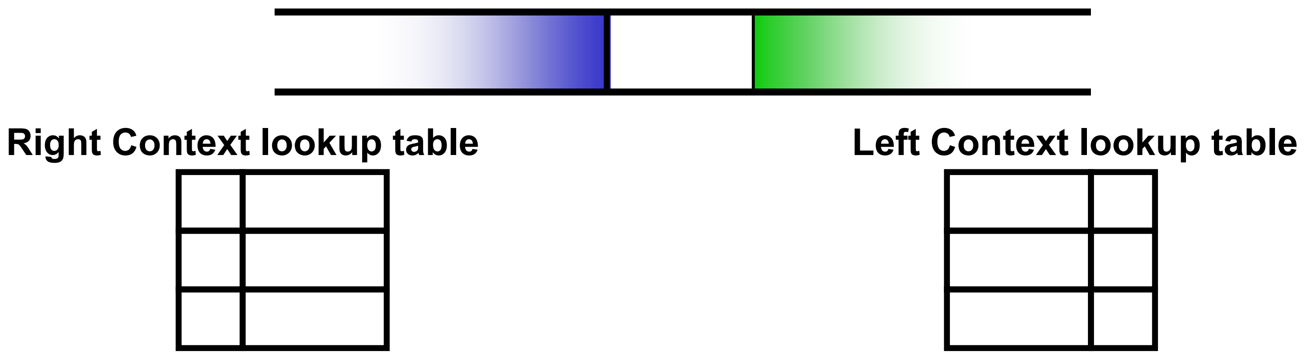
- Methodology:
  - Have a small sliding window on the compressed string.
  - The string in the window is plain text (decompressed)
  - The strings on the left and the right of the window are compressed.

Enable bidirectional traversal

- Forward compressed, backward decompressed FCM
  - Traditional FCM is forward compressed, forward decompressed



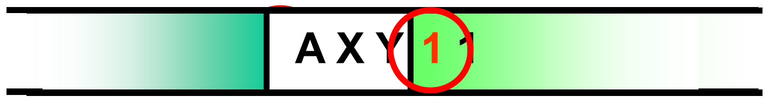
- Bidirectional FCM



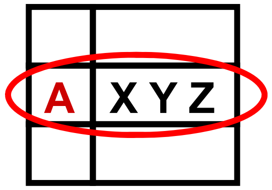
- Left-context look up table
  - Predict the next value based on its left context
- Right-context look up table
  - Predict the next value based on its right context

- Moving the plain text window of size n one step forward
  - Decompress using the left-context lookup (now get a window of size n+1)
  - Compress the first value of window using the right-context lookup table (again we get a window of size n)
- Moving the window one step barward
  - The opposite actions.

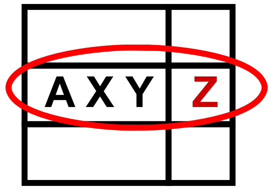
Bidirectional FCM - example



Right Context lookup table



Left Context lookup table



Characteristics of bidirectional predictors

- High compression rate
  - The compression rate is nearly the SAME as unidirectional predictors;
- Fast compression and de-compression
  - Roughly TWO times slower than unidirectional predictors;