HPC System Software Enhanced by Source Code Analysis

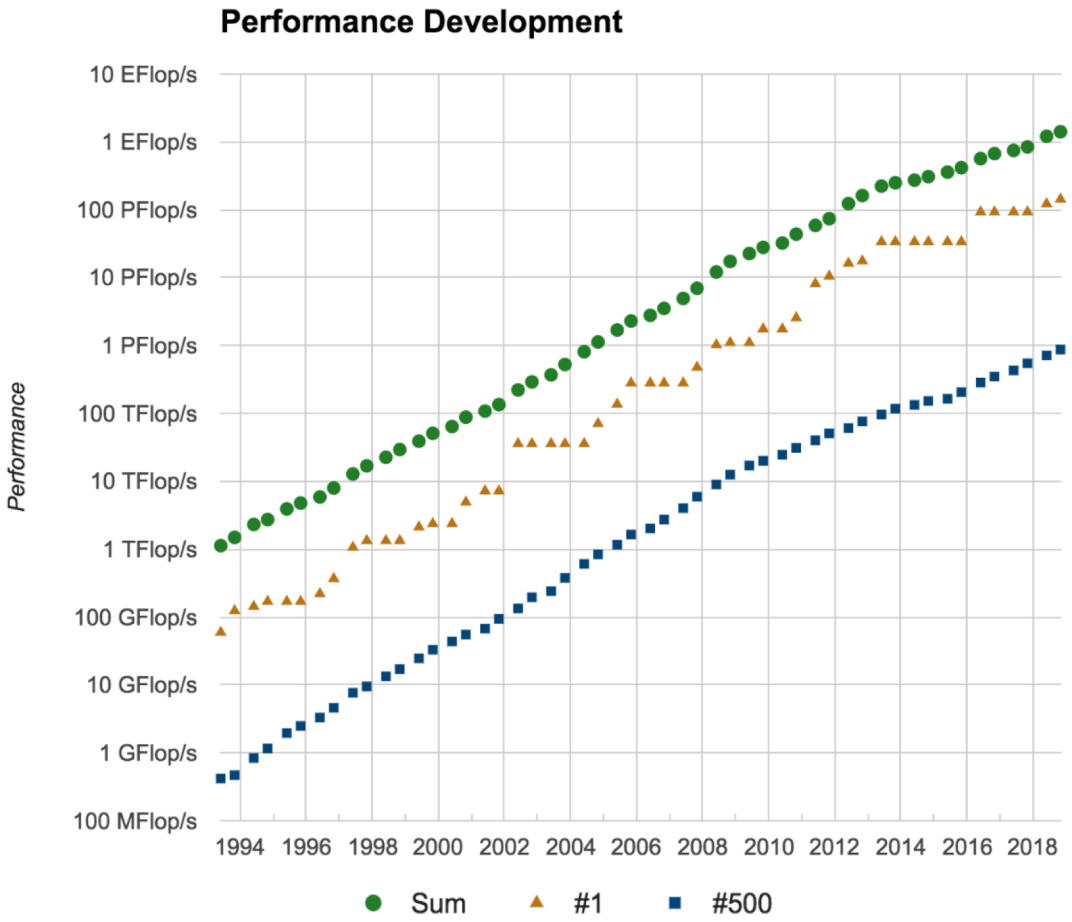
ROSS keynote @Phoenix 2019.06.25

Jidong Zhai

Tsinghua University

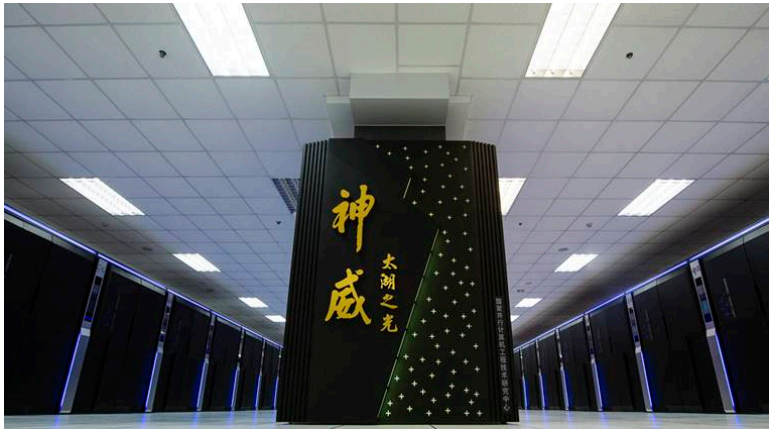
zhaijidong@Tsinghua.edu.cn

Exponential Growth of HPC



Performance Development in TOP500

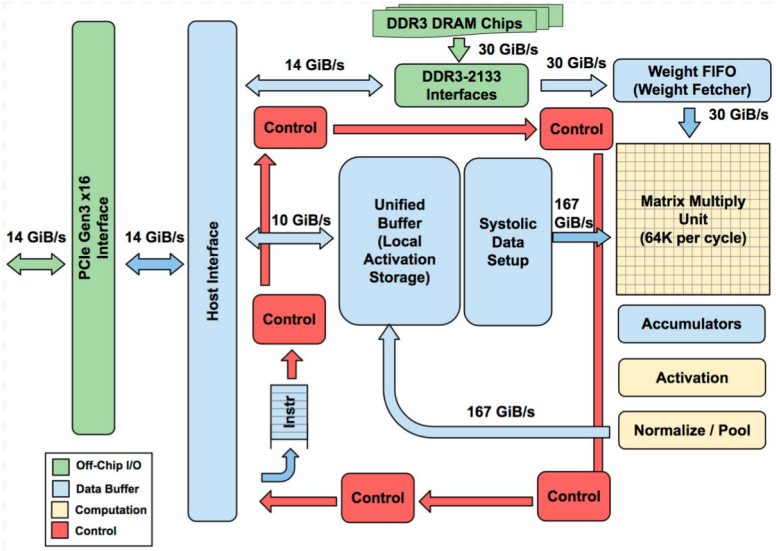
Summit: 2M cores



Sunway: 10M cores

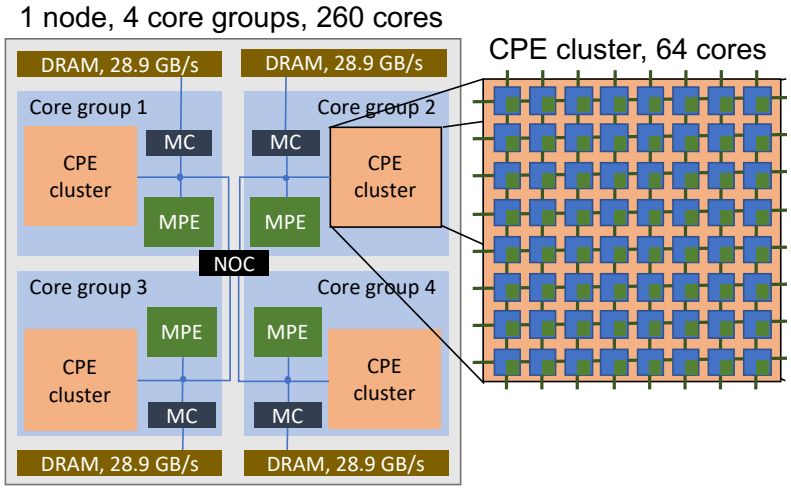
Heterogeneous Acceleration in HPC

- **Hardware Specialization**
  - Google TPU
  - MIT Eyeriss
- **Heterogeneous Acceleration**
  - Sunway TaihuLight
  - EuroEXA Project: ARM+FPGA



Google TPU

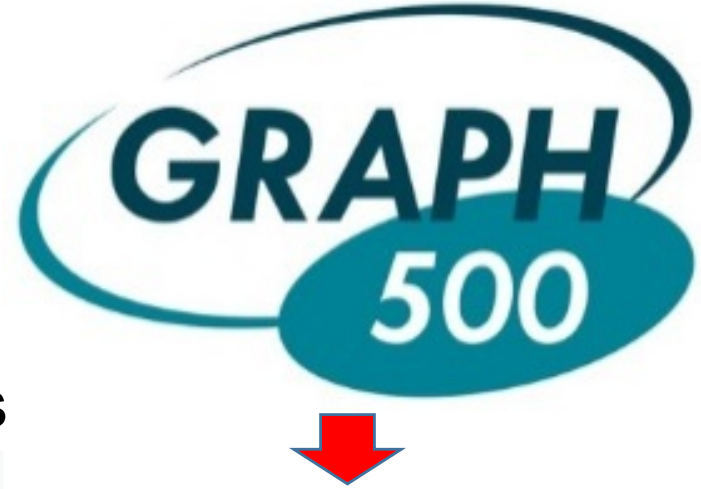
Europe Exascale Project



Sunway Processor

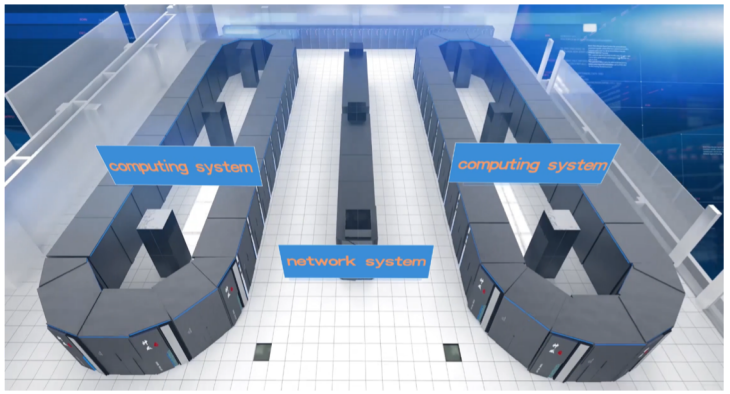
Programming and Performance Challenges

- **Understanding performance**
  - Porting Graph 500 on Sunway
  - **From 100 LOC → 10,000 LOC**
  - **~1 PH.D 1 year effort**
  - Ranked #1 in Heterogeneous HPCs



Top Ten from June 2019 BFS

RANK	MACHINE	VENDOR	INSTALLATION SITE	LOCATION	COUNTRY	YEAR	NUMBER OF NODES	NUMBER OF CORES	SCALE	GTEPS
1	K computer	Fujitsu	RIKEN Advanced Institute for Computational Science (AICS)	Kobe Hyogo	Japan	2011	82944	663552	40	31302.4
2	Sunway TaihuLight	NRCPC	National Supercomputing Center in Wuxi	Wuxi	China	2015	40768	10599680	40	23755.7
3	DOE/NNSA/LLNL Sequoia	IBM	Lawrence Livermore National Laboratory	Livermore CA	USA	2012	98304	1572864	41	23751
4	DOE/SC/Argonne National Laboratory Mira	IBM	Argonne National Laboratory	Chicago IL	USA	2012	49152	786432	40	14982
5	SuperMUC-NG	Lenovo	Leibniz Rechenzentrum	Garching	Germany	2018	4096	196608	39	6279.47



Sunway TaihuLight

We need efficient performance tools to understand the performance of both applications and systems

Previous Work on Performance Tools

- **Profiling-based Approaches**
  - e.g., mpiP, gprof etc.
  - ☺ Light-weight
  - ☺ Scalable
  - ☹ Limited information, need expertise knowledge
- **Tracing-based Approaches**
  - e.g., ITC/ITA, Dtrace etc.
  - ☺ Much more fine-grained information
  - ☹ Not scalable
- **Challenges**
  - System scale
  - Problem size
  - E.g., ASCI SMG2000,
    64*64*32 Problem size,
    22,538 Processes → **5 TB communication traces**

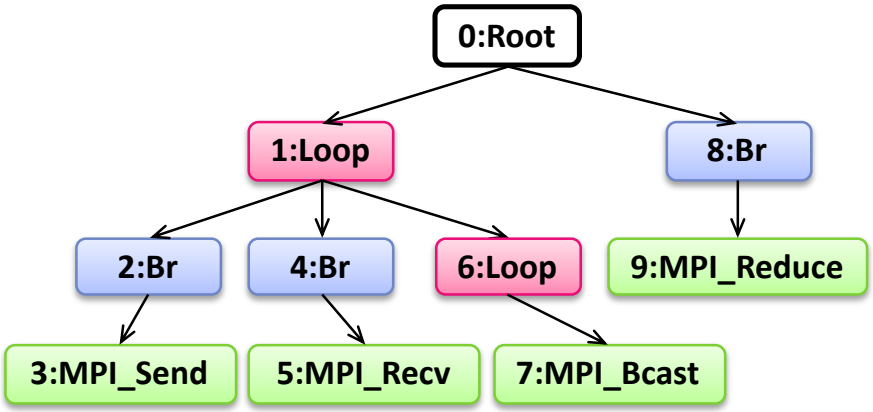
Advantage of Source Code Analysis

- **Previous methods**
  - Runtime analysis
  - **Ignore** source code

Source code analysis ➔

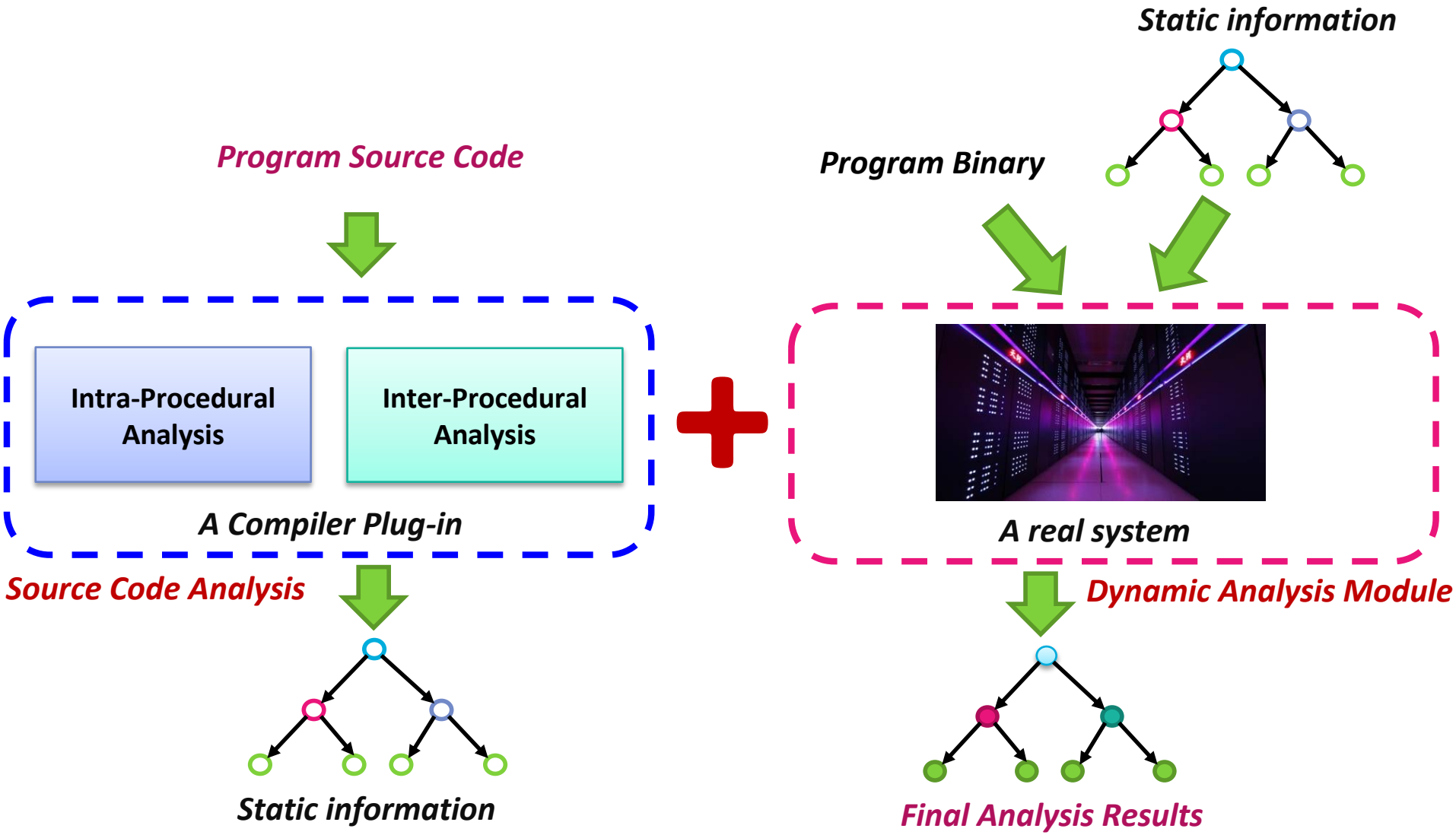
- **Provide reliable insight**
  - Top-down “Big picture”
  - **Guide** runtime analysis
- **Low overhead + Scalability**
  - Low overhead
  - One-time static analysis cost
  - **Independent** of problem size and job scale

Source code Analysis Tools



Information from Source Code

Hybrid Method: Static + Dynamic



Systems based on this idea

- **vSensor**
  - A performance variance detection tool
  - [PPoPP'18]
- **Spindle**
  - An informed memory access monitor
  - [USENIX ATC'18]
- **Cypress**
  - An efficient communication trace compression tool
  - [SC'14, Best Paper Finalist]
- **FACT**
  - A fast communication trace collection tool
  - [SC'09]

Systems based on this idea

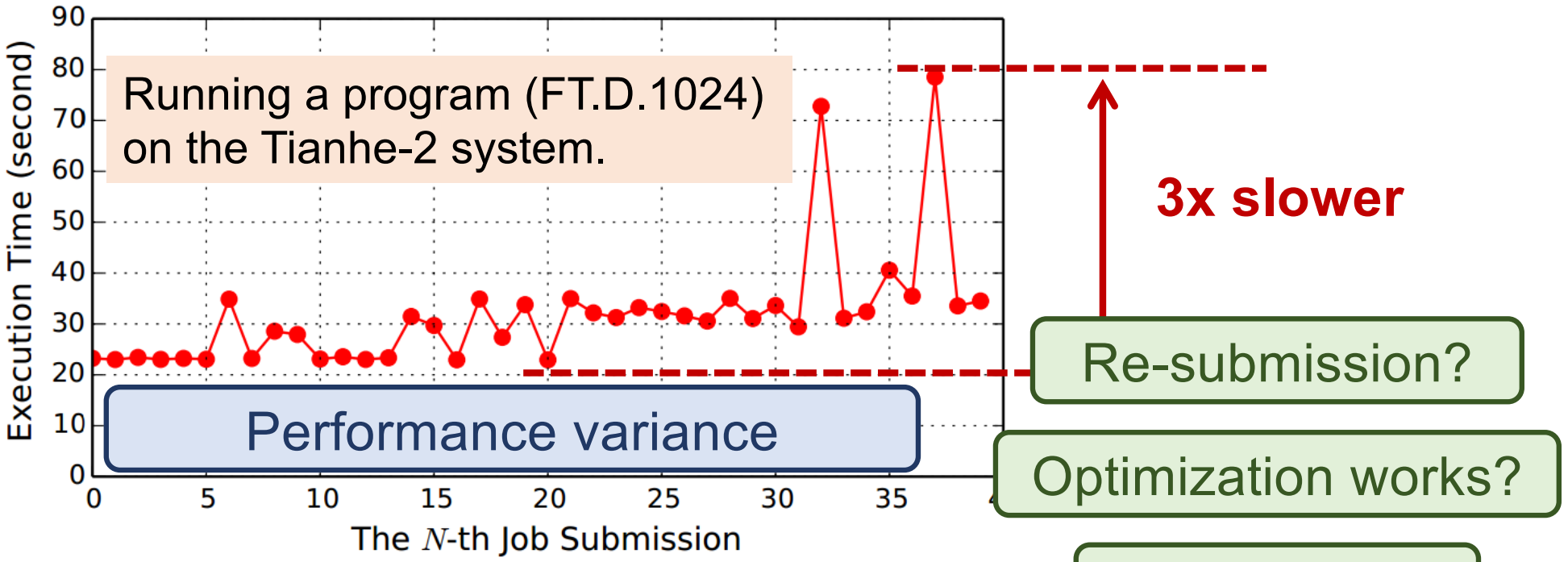
- **vSensor**
  - A performance variance detection tool
  - [PPoPP'18]
- **Spindle**
  - An informed memory access monitor
  - [USENIX ATC'18]
- **Cypress**
  - An efficient communication trace compression tool
  - [SC'14]
- **FACT**
  - A fast communication trace collection tool
  - [SC'09]

vSensor:

Leveraging fixed-Workload Snippets of Programs for Performance Variance Detection

What happens to my program?

Programs get **varied performance** across executions, even on the **same** computing nodes.



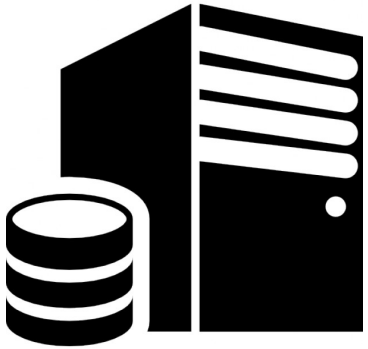
Programs become **slow** and **unstable**.

Consume more resources

Hard to understand its behavior

Sources of performance variance

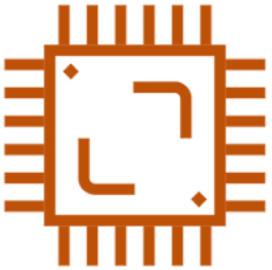
Hardware Error | OS Interruption | Resource Competition



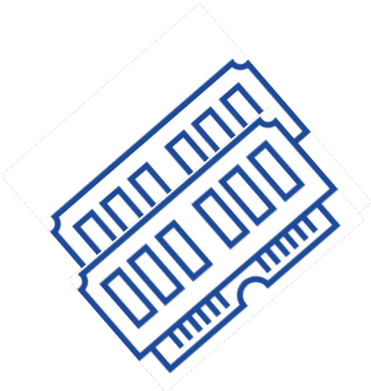
Computer System

Operating System

Users



Processor



Memory

Network

Storage

To detect performance variance

Detecting system performance variance

- Helps users to **understand** programs' performance
- Helps administrators to **investigate** system problems

How to?

Rerun: **easy** but **time consuming**

Model: **accurate** but **not portable**

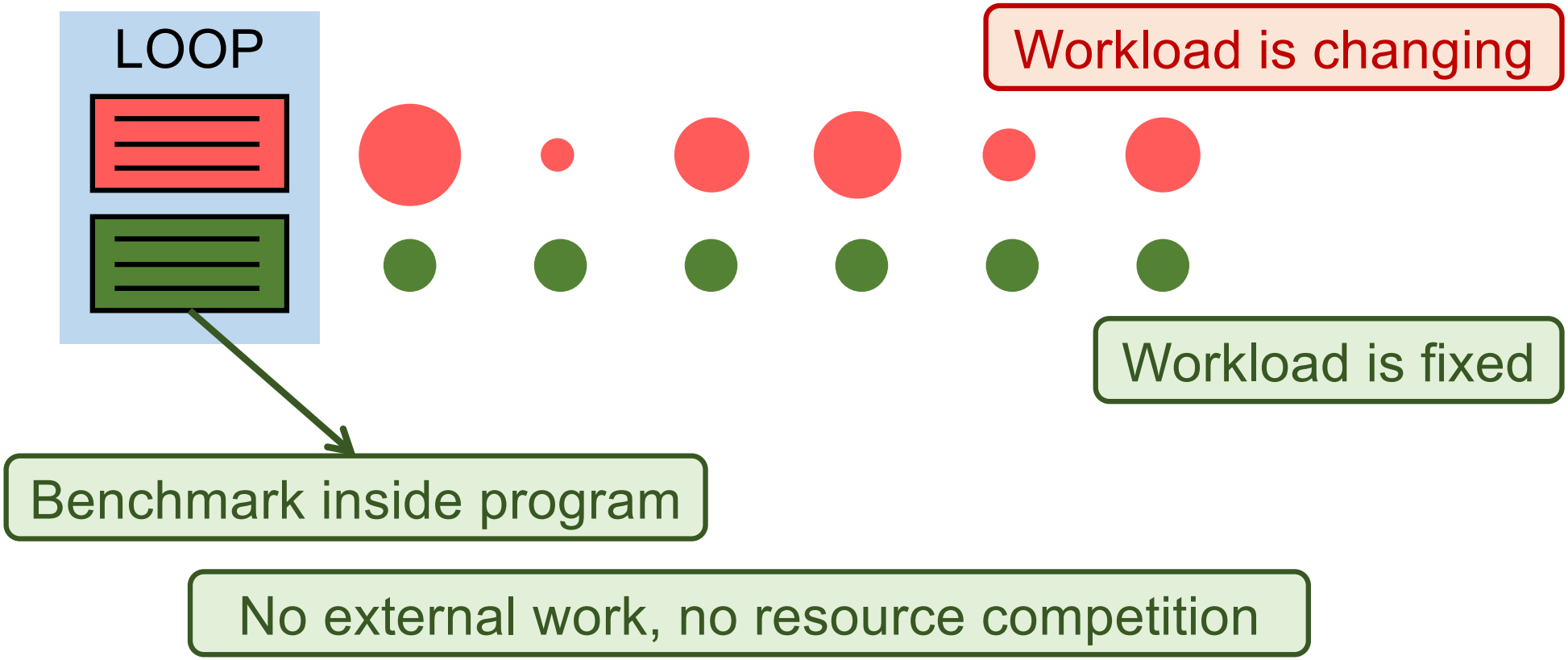
Profile/Trace: **widely used** but **expert knowledge required**

Benchmark: **accurate** but **intrusive**

What if the benchmarks are parts of an application?

Observation

Many programs contain code snippets that are executed **repeatedly** with **fixed workload**.



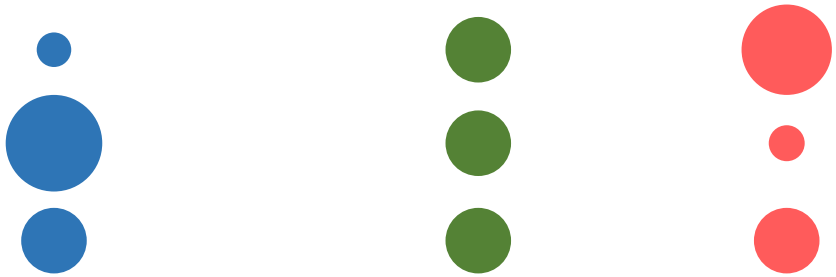
Use fixed-workload snippets as **v**ariance **sensors** (**v-sensors**)

vSensor in a nutshell

Idea:

Find out fixed-workload snippets, and then analyze their time variance to detect system performance variance

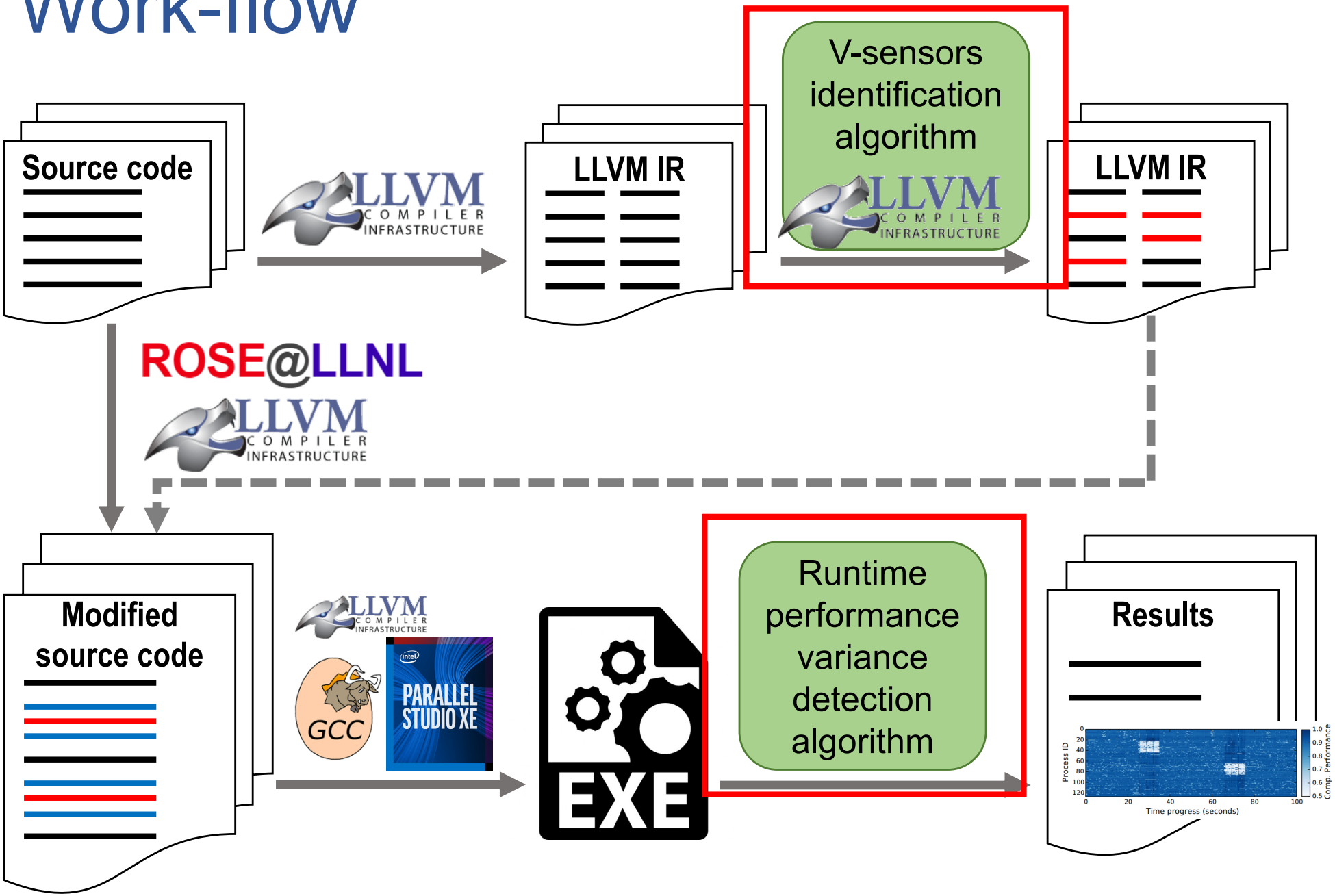
Performance = Workload / Time



Benefits:

- **Easy to detect** and locate performance variance
- **No model** or expert knowledge required
- **Low overhead** and little manual effort

Work-flow


Source code
LLVM COMPILER INFRASTRUCTURE
LLVM IR
V-sensors identification algorithm
LLVM COMPILER INFRASTRUCTURE
LLVM IR
ROSE@LLNL
LLVM COMPILER INFRASTRUCTURE
Modified source code
LLVM COMPILER INFRASTRUCTURE
GCC
PARALLEL STUDIO XE
EXE
Runtime performance variance detection algorithm
Results
Process ID
Time progress (seconds)
Comp. Performance

Fixed-Workload Snippets

What is fixed workload?

Computation:

The same number and same sequence of instructions.

Network communication:

The same message sizes.

IO read/write:

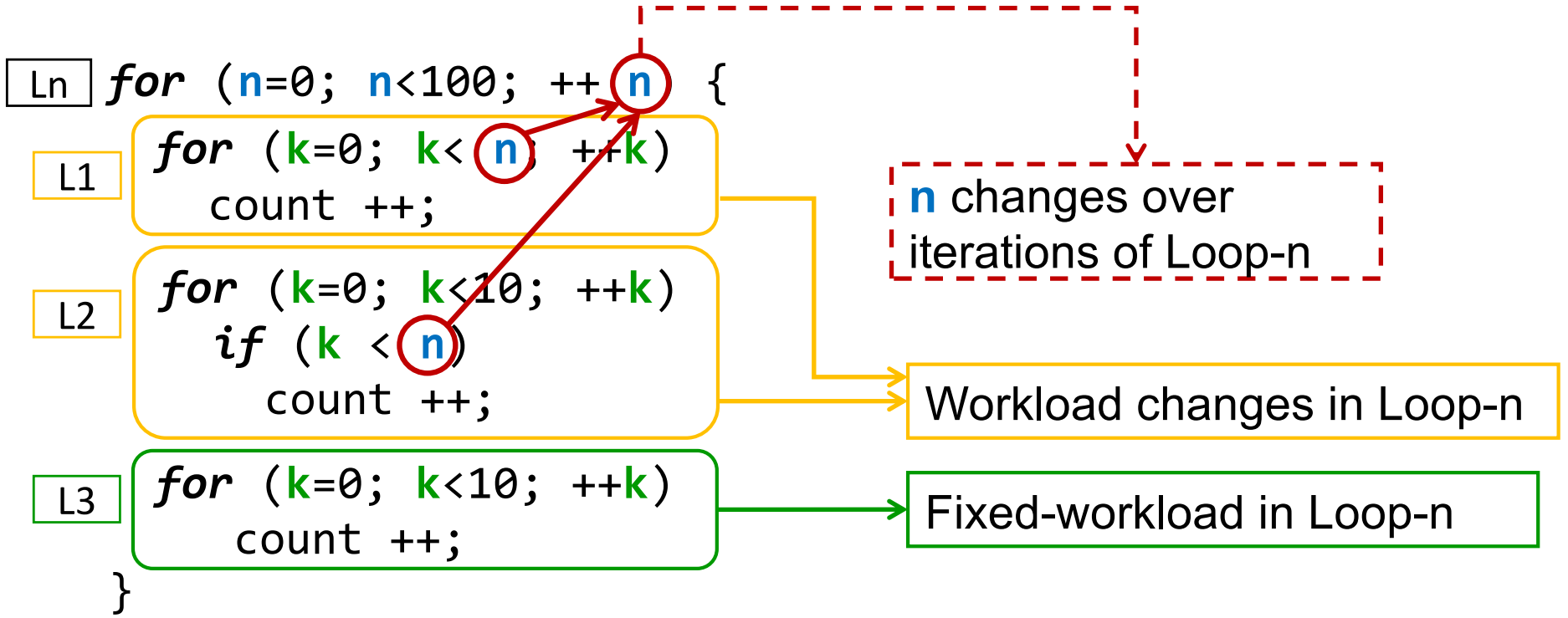
The same IO data sizes.

v-sensor candidates:

Only **loops** and **function calls** are considered as candidates, to filter out very fine-grained snippets(e.g., single instruction).

Intra-procedural Analysis

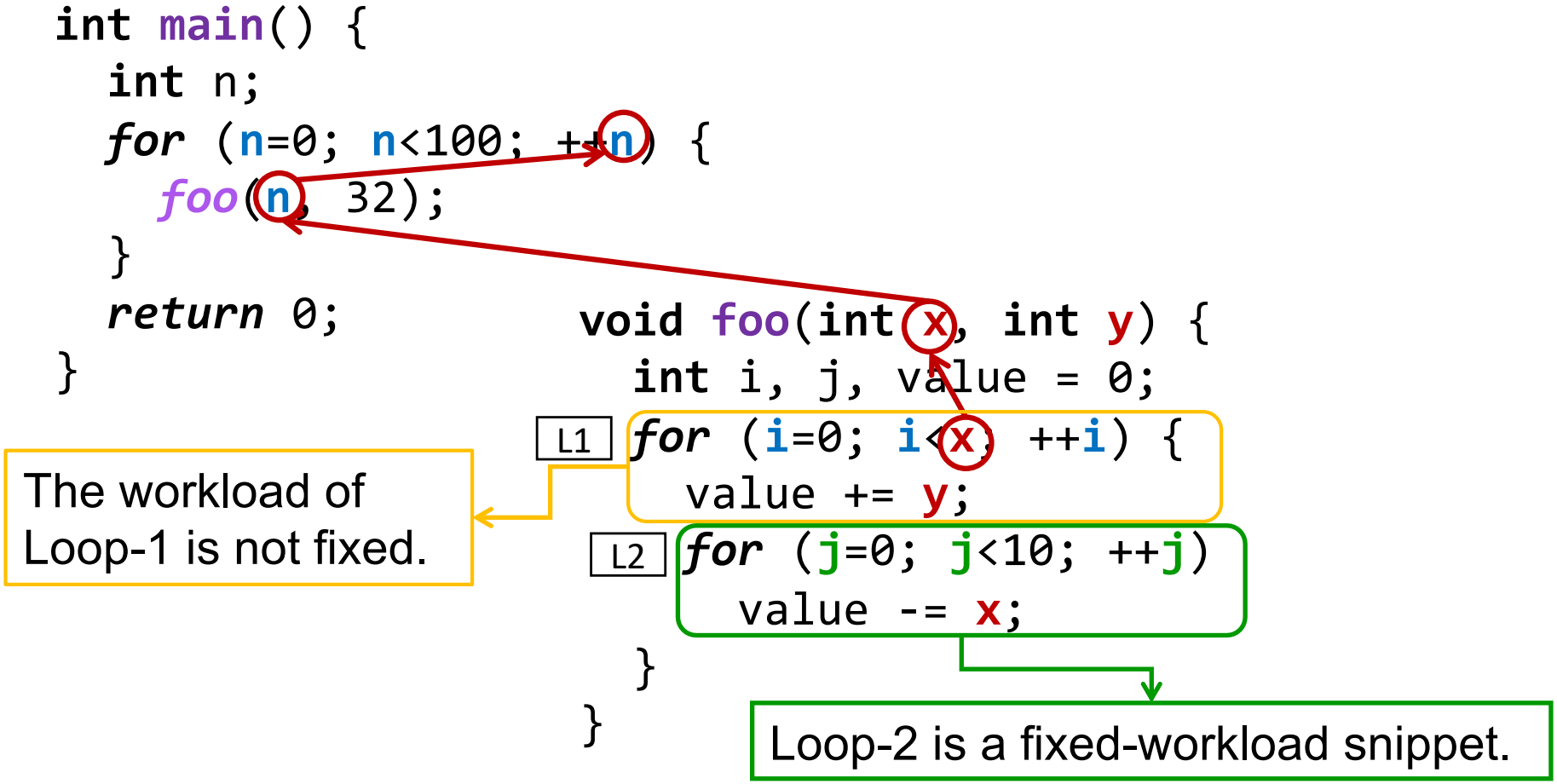
Workload is determined by the **control expressions** of **loops** and **branches**.



*Use-Define Chain technique is used to analyze the dependency.

Inter-procedural Analysis

Workload is also affected by **function arguments**.



Instrumentation

Insert **Tick/Tock** function calls to record execution time and trigger **periodically** analysis.

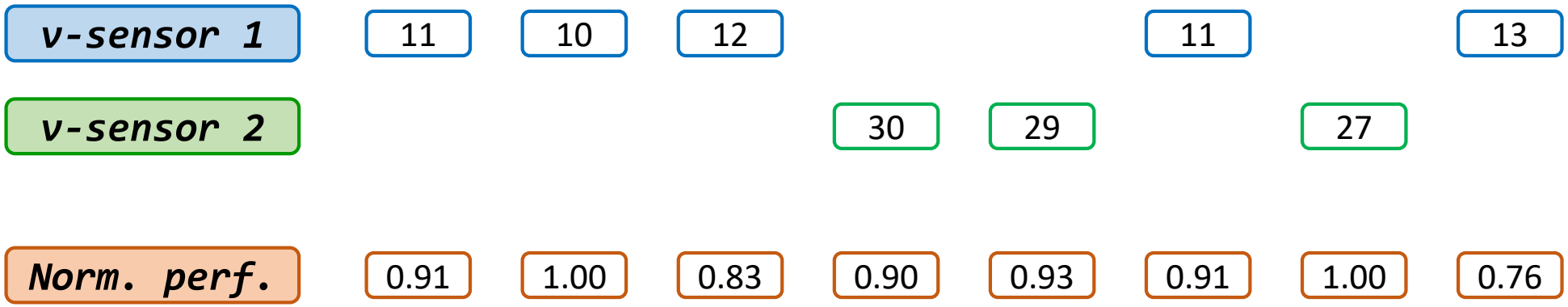
```
for (...) {
    snippet-1
    snippet-2 (fixed-workload)
    snippet-3
}
```

```
for (...) {
    snippet-1
    Tick()
    snippet-2 (v-sensor)
    Tock()
    snippet-3
}
```

Performance normalization

Normalize **execution time** to **relative performance**

Shortest time → relative performance 1
Longer time → smaller relative performance value (0~1)



Merge data from v-sensors of the **same type** (comp/network/IO)

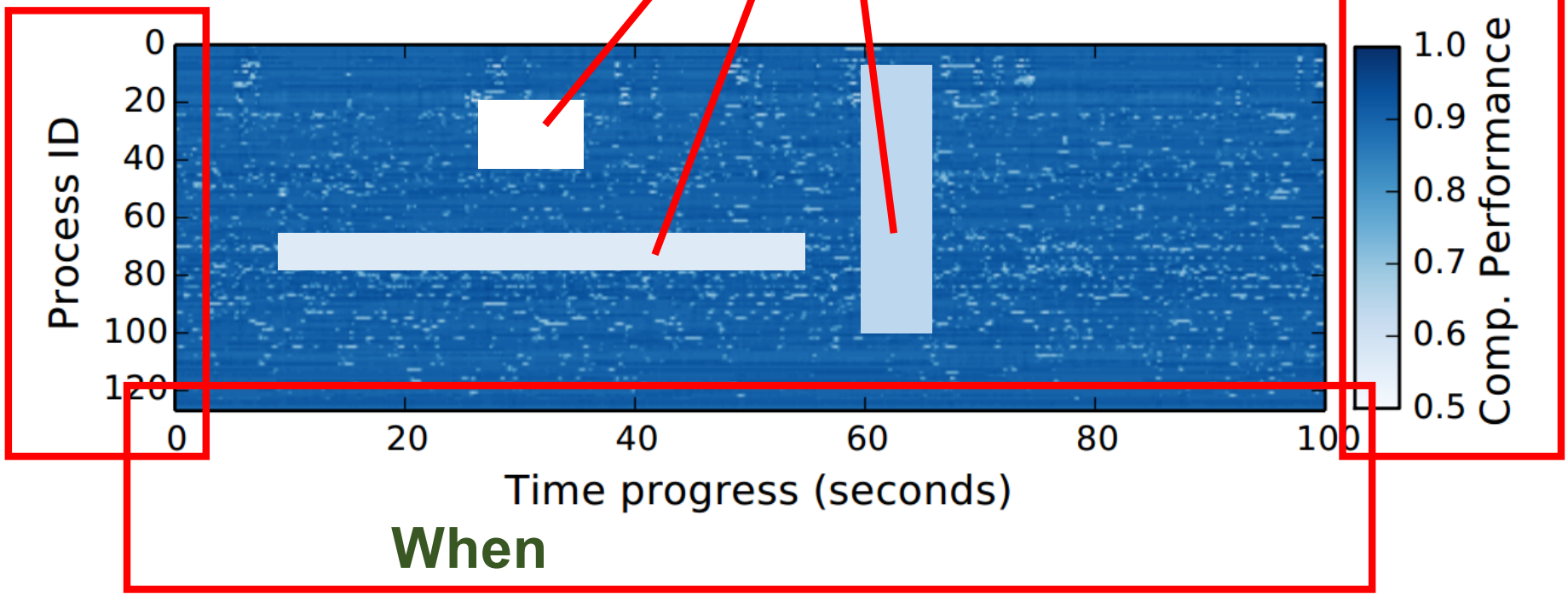
Improve the **coverage** of detection

Result visualization

Light color blocks → bad performance

Darker is better

Where



When

*An example result for computation performance, detected for a 128 processes program.

Experiments

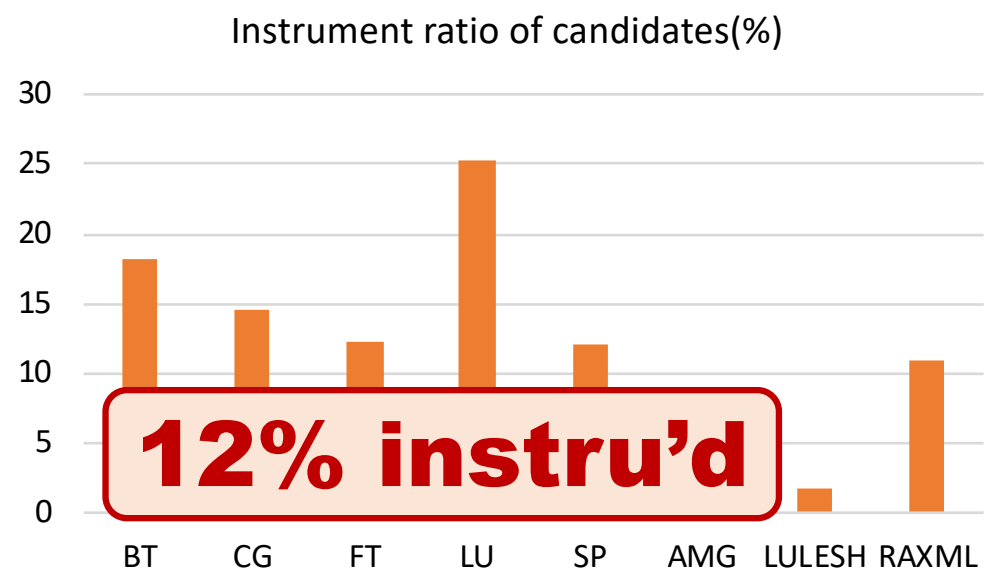
Platforms:

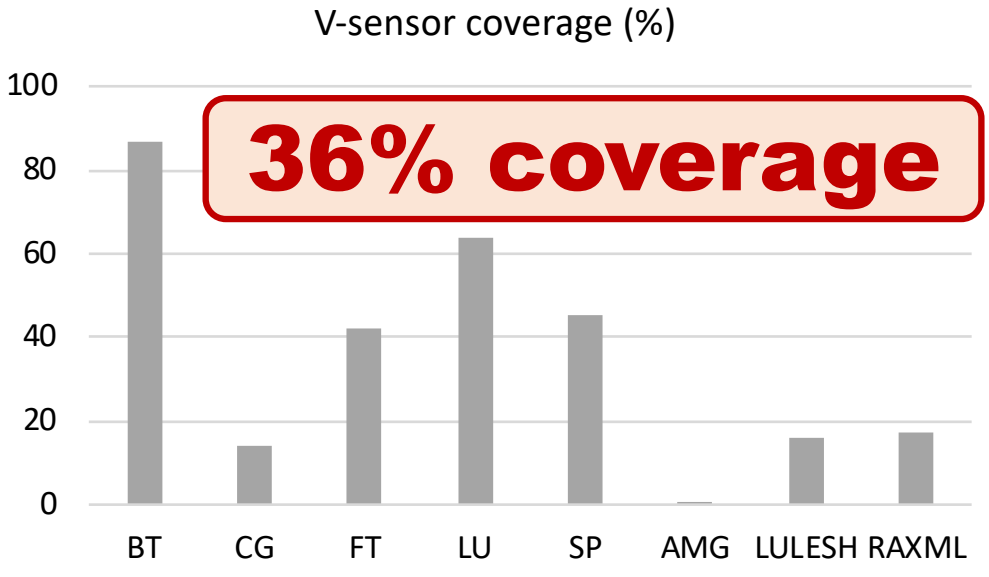
Tianhe-2 system in NSCC-Guangzhou
Dual Xeon E5-2692(v2) (24 cores in total) and 64GB memory.
Gorgon cluster in Tsinghua (for noise injection)
Dual Xeon E5-2670(v3) (24 cores) and 128GB memory

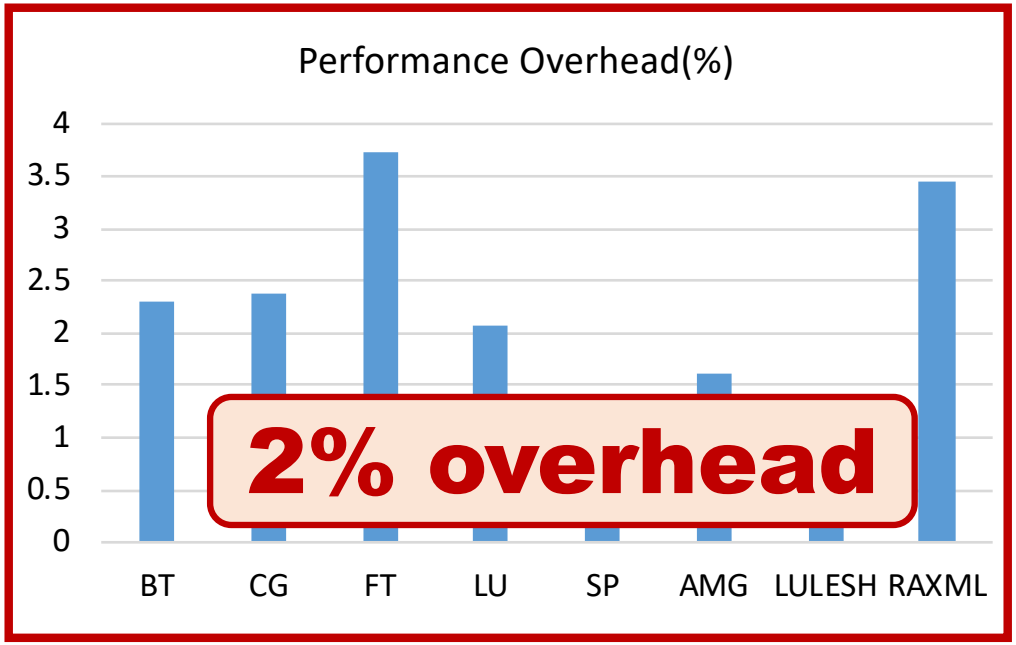
Programs:

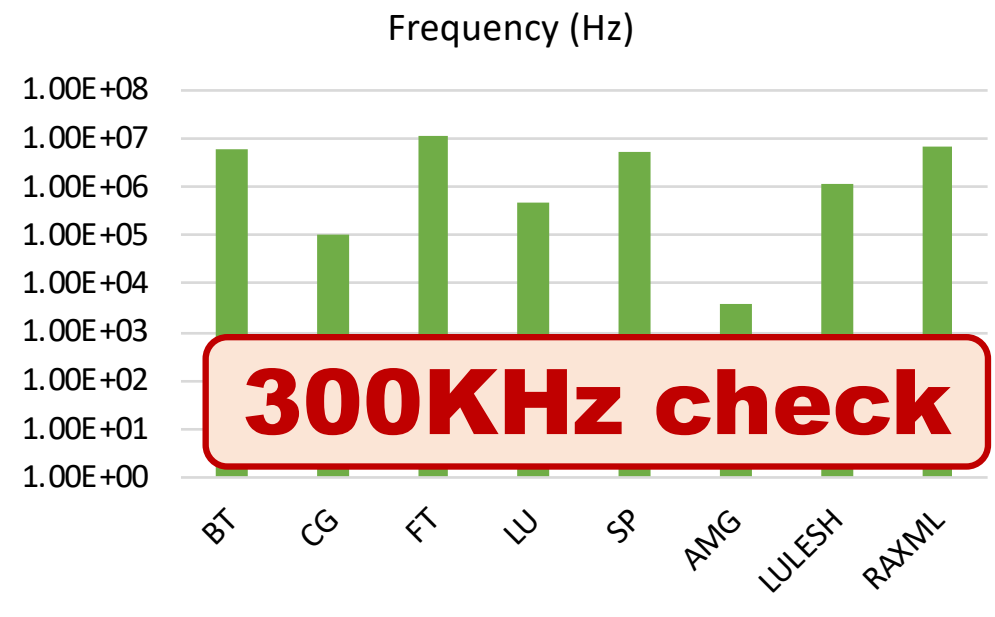
8 HPC programs: BT/CG/FT/LU/SP/AMG/LULESH/RAXML
Up to 16,384 MPI processes

Basic results



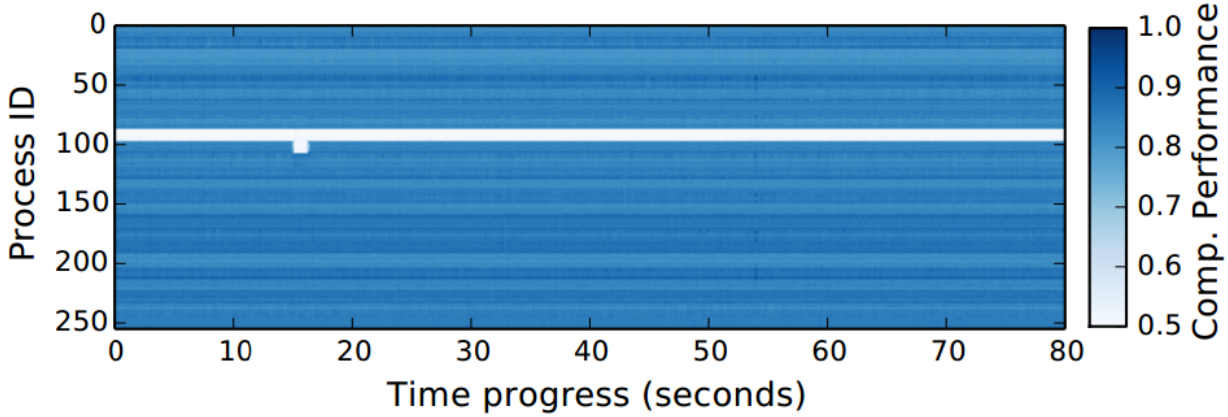






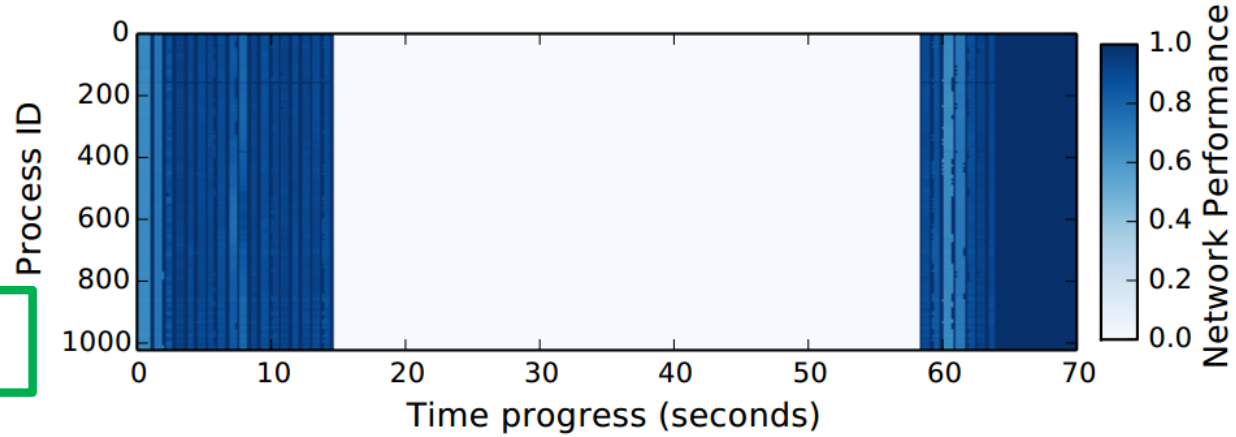
Problems detected on Tianhe-2

A **memory** channel offers only **55%** of typical bandwidth, and causes **21%** performance loss.



Report the bad node.

We know what happens.



An unexpected **network** problem wastes **43** seconds.

Summary

The take away point:

The source code of a program itself can be used to detect system performance variance.

vSensor presented in this work:

vSensor is a toolkit based on LLVM compiler infrastructure. It can efficiently detect and locate where and when the performance variance occurs.

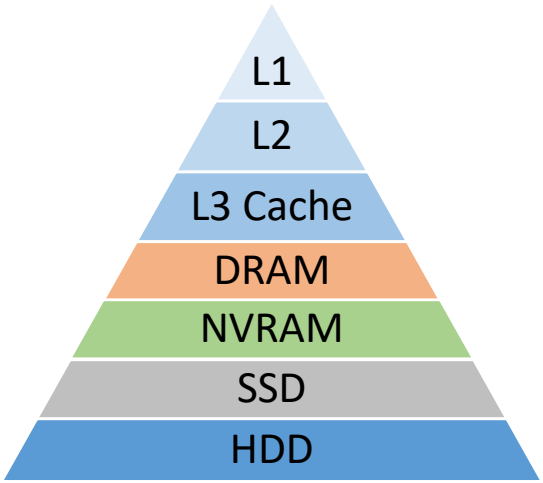
More details in the paper:

Nested v-sensors, the scope of v-sensor, functions in libraries, customize the standard of fixed-workload, handling very short v-sensors, etc..

Spindle:
Informed Memory Access Monitoring

Memory access monitoring is imperative

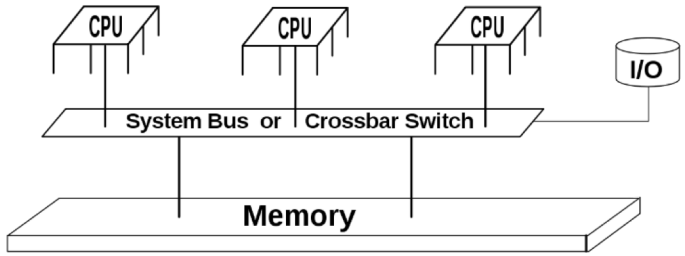
- Memory access is **error prone**
  - Uninitialized read, write after free, data race, deadlock
- Memory access is **a key factor of performance**
  - Big gap between CPU and memory, complex cache hierarchy
- Memory access has **high security risk**
  - Buffer overflow



Memory Hierarchy

- We need memory access monitoring for *bug detection*, *performance optimization*, and *malware analysis*.

Full monitoring for a long-running program introduces large overhead.



Shared memory system

Conventional one-by-one checking is slow

- **Dynamic tools**
  - PIN, Valgrind, DynamoRIO, Dr.Memory
  - Modify instruction flow at runtime to insert checking functions

```
1  for (int i=0; i<N; ++i) {
2    a[i] = i;
3    checkAccess(&a[i]);
4  }
```

Overhead >100x

Trace size ~10GB/s

*Check all memory access **one by one**.*

Record all memory access for tracing.

Still doing redundant work.

Overhead ~ 2x

- Static + dynamic hybrid tools
  - Address Sanitizer, Memory Sanitizer
  - Add checking functions at compile time.

Use access pattern to avoid redundant checks

- What will be the memory access addresses of this code snippet?

```
1  for (int i=0; i<N; ++i) {
2    a[i] = i;
3  }
```

Learn memory access pattern from source code.

```
a, a+4, a+8, a+12, ……, a+4*(N-1)
```

- Only the starting address **a** and loop count **N** are unknown
  - Record at runtime

```
1  recordArrayBase(a);
2  recordLoopFinal(N);
3  for (int i=0; i<N; ++i){
4    a[i] = i;
5  }
```

Knowing a and N, we can compute each a[i].
No other checking is needed.

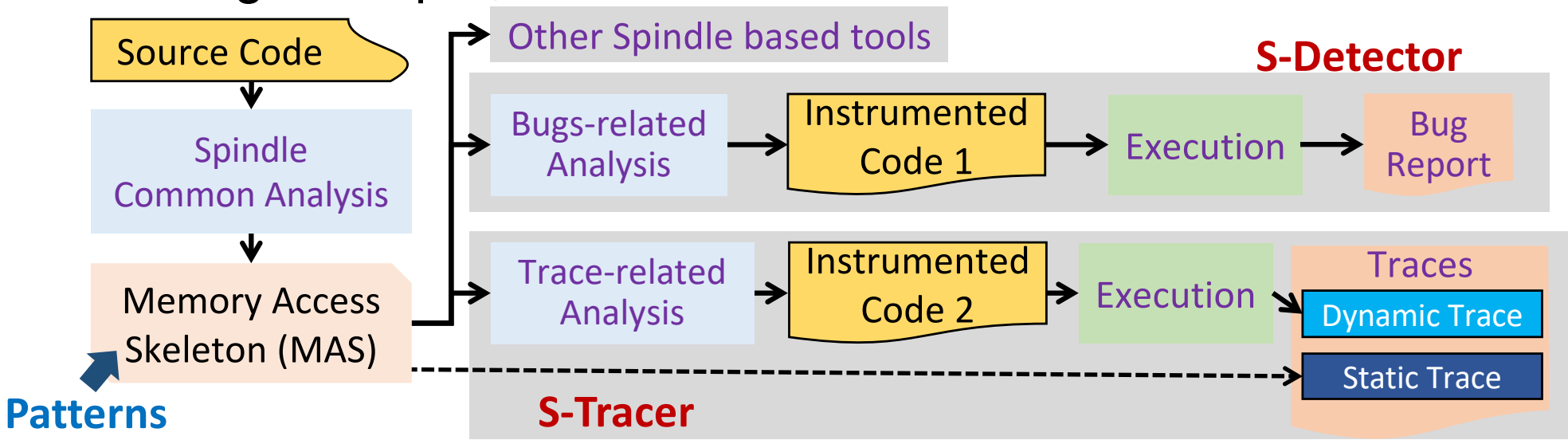
N checks → 2 checks

Compute what is computable, and record the rest.

Spindle: Informed Memory Access Monitoring

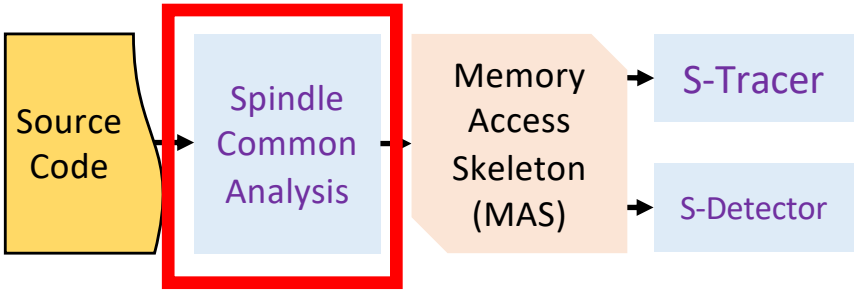
S-Tracer for memory tracing **S-Detector** for memory bug detection

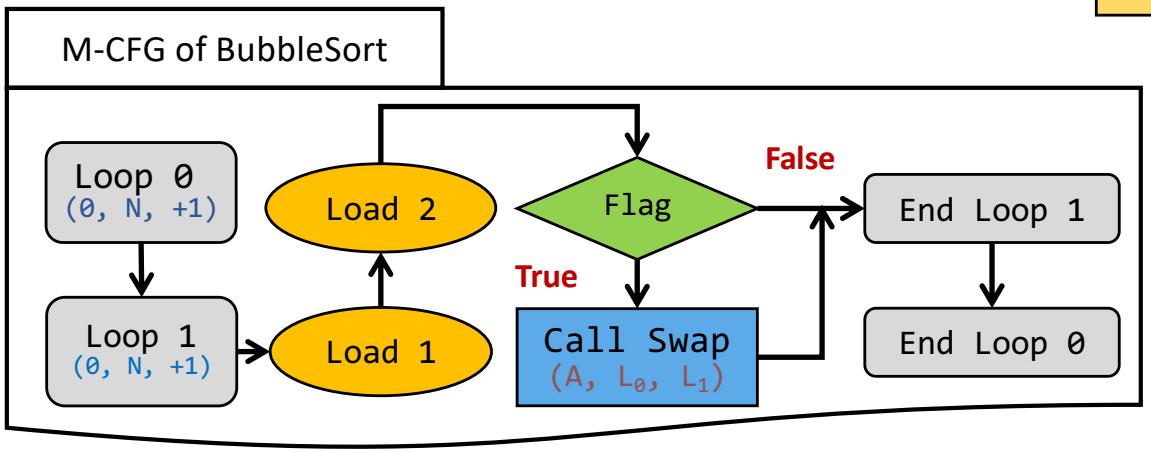
- **Step-1:** Analyze code to find memory access patterns at compile time
  - Loop, branch, array index, structure offset
- **Step-2:** Record non-computable parameters at runtime
  - Program input, random number

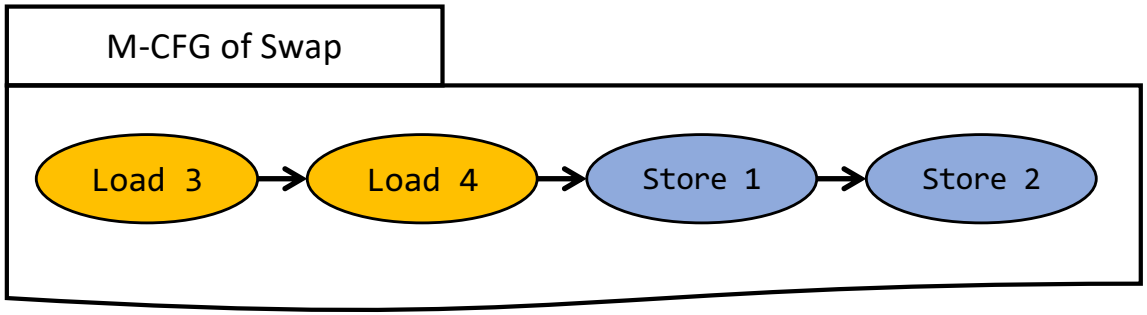


Spindle common source code analysis

- Control flow analysis



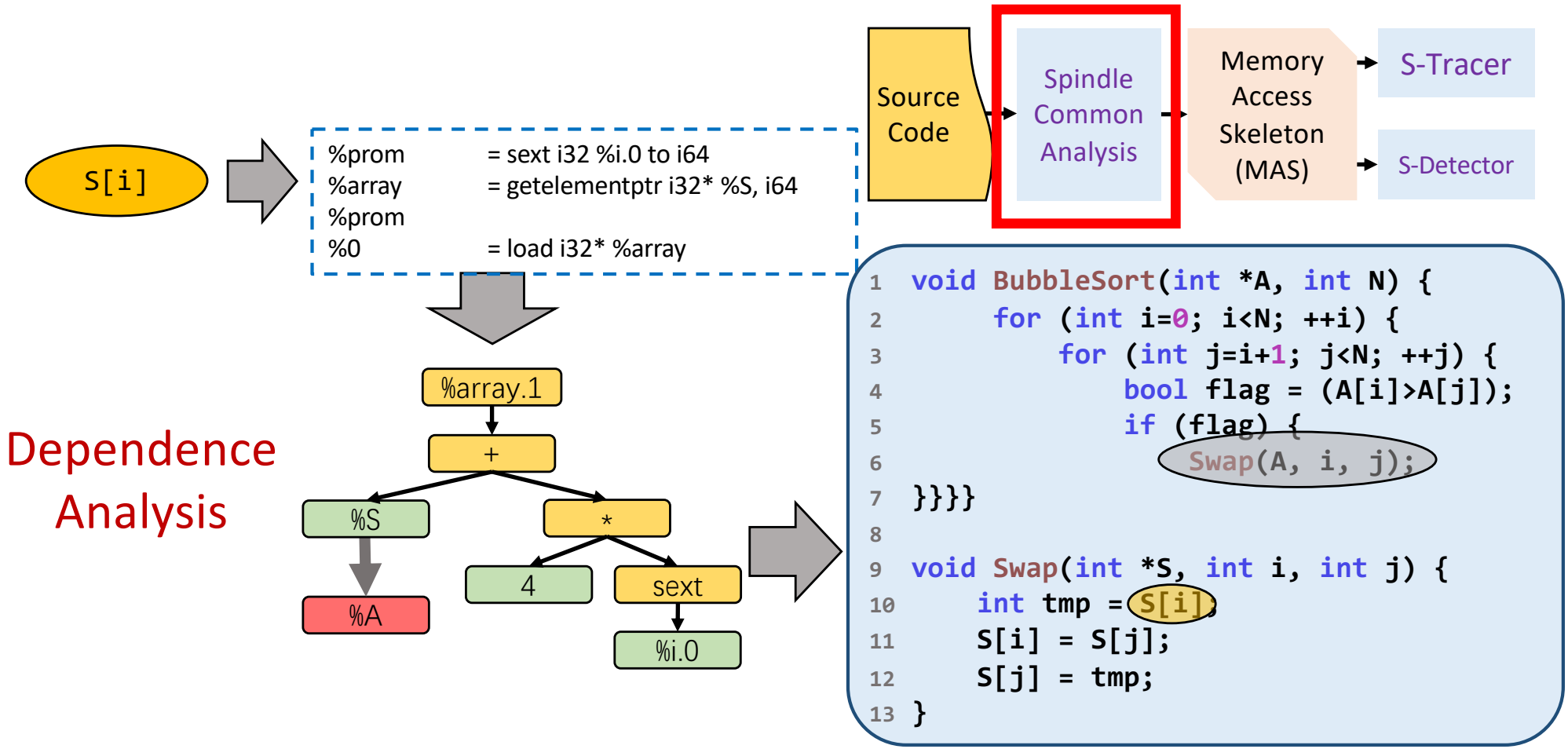




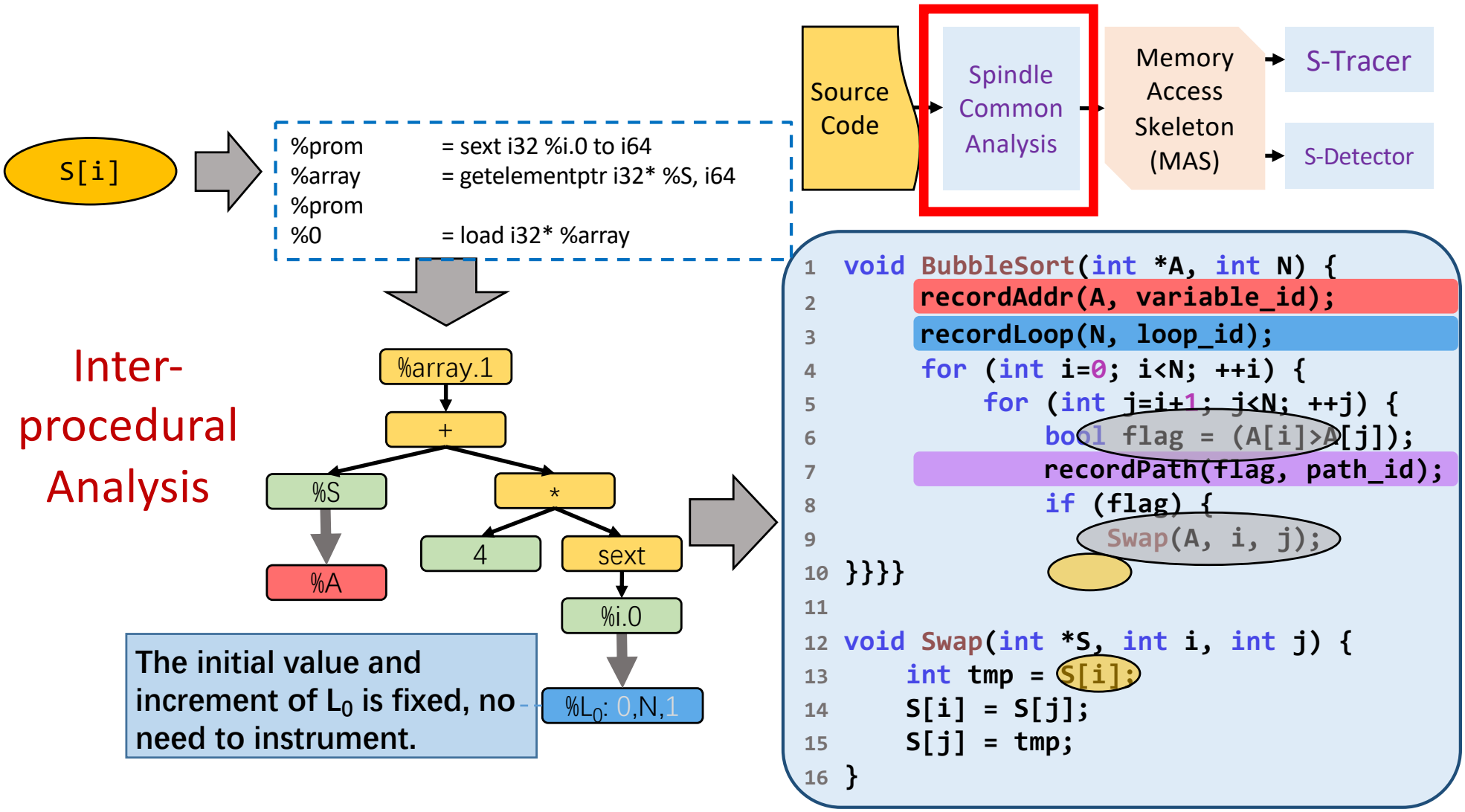

```
void BubbleSort(int *A, int N) {
  for (int i = 0; i  < N; ++i) {
    for (int j = i+1; j < N; ++j) {
      bool flag = A[i] > A[j];
      if (flag) {
        Swap(A, i, j);
}}}}
```

```
void Swap(int *S, int i, int j) {
  int tmp = S[i];
  S[i] = S[j];
  S[j] = tmp;
}
```

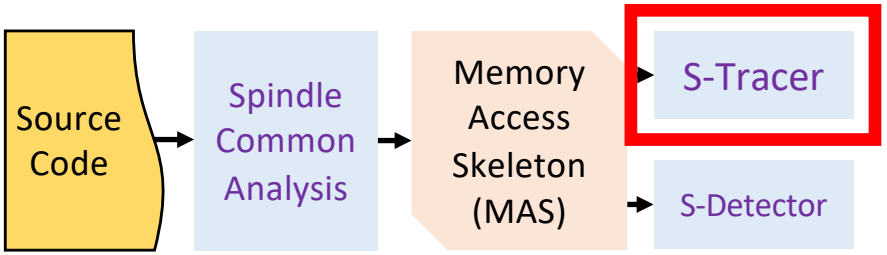
Spindle common source code analysis



Spindle common source code analysis



S-Tracer Example




```
1  void BubbleSort(int *A, int N){
2    for (int i=0; i<N; ++i) {
3      for (int j=i+1; j<N; ++j){
4        bool flag = (A[i]>A[j]);
5        if (flag){
6          Swap(A, i, j);
7  }}}}
8
9  void Swap(int *S, int i, int j){
10   int tmp = S[i];
11   S[i] = S[j];
12   S[j] = tmp;
13 }
```

Static Trace

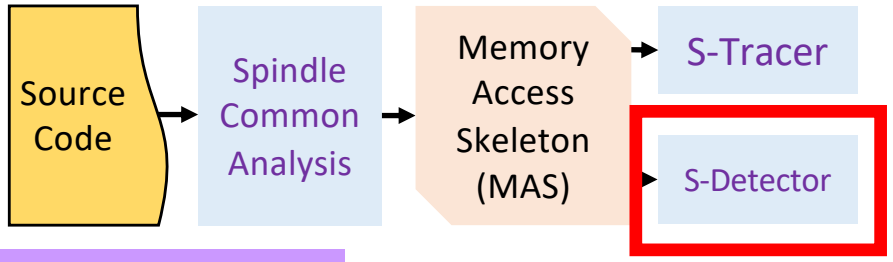
```
Function BubbleSort(dyn_A, dyn_N) {
 Loop0: L0, 0, dyn_N, 1 {
  Loop1: L1, L0, dyn_N, 1 {
   Load1: dyn_A+L0; Load2: dyn_A+L1;
    Branch: dyn_flag {
     Call Swap(dyn_A, L0, L1);
}}}}
Function Swap(S, i, j) {
 Load3 : S+i; Load4 : S+j;
 Store1: S+i; Store2: S+j;
}
```

Dynamic Trace

```
BubbleSort {
 dyn_A:
  0x7fffdfc58320;
 dyn_N:
  10;
 dyn_flag:
  {0,0,1,1,0,...,1,1};
}
```

S-Detector Example

```
1  while (pos-1 && red_cost >
2          (cost_t)new[pos/2-1] .flow) {
3      new[pos-1].tail = new[pos/2-1].tail;
4      new[pos-1].head = new[pos/2-1].head;
5      // More codes.
6      pos = pos/2;
7      new[pos-1].tail = tail;
8      // More codes
9  }
```



In-struct accesses

Struct type $cost\_t$'s range:

$$[struct\_base, struct\_base + struct\_size)$$

Struct access to array new's member:

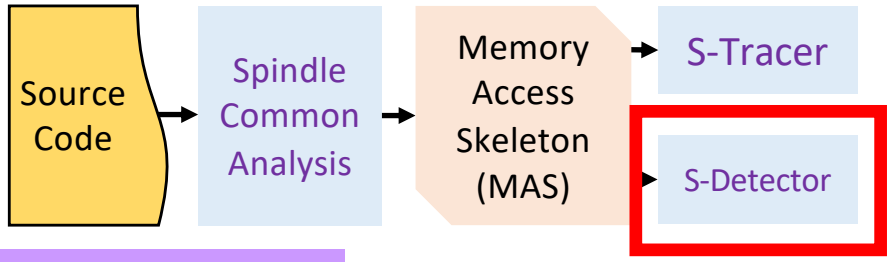
$$addr = struct\_base + offset$$

Such accesses are valid only if:

$$offset < struct\_size$$

and accesses to array new is valid.

S-Detector Example




```
1  while (pos-1 && red_cost >
2          (cost_t)new[pos/2-1] .flow) {
3      new[pos-1].tail = new[pos/2-1].tail;
4      new[pos-1].head = new[pos/2-1].head;
5      // More codes.
6      pos = pos/2;
7      new[pos-1].tail = tail;
8      // More codes
9  }
```

Only have to record: ***pos_init***, ***pos_end***, ***new_size***. (***struct_size*** is known at compilation time)

In-struct accesses

Struct type $cost\_t$'s range:

$$[struct\_base, struct\_base + struct\_size)$$

Struct access to array new's member:

$$addr = struct\_base + offset$$

Such accesses are valid only if:

$$offset < struct\_size$$

and accesses to array *new* is valid.

In-loop accesses

Loop induction variable pos's range:

$$[pos\_init, pos\_final)$$

Array new's size:

$$new\_size$$

Array accesses valid only if:

$$(pos\_init - 1) * struct\_size < new\_size \quad (1)$$

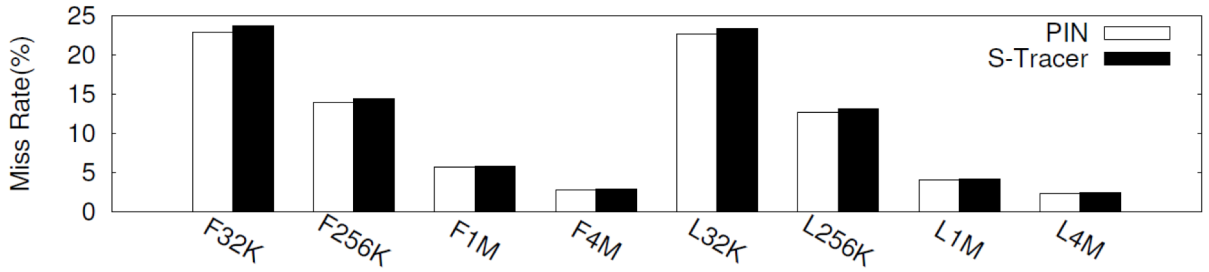
$$\&\& \quad pos\_end/2 - 1 \geq 0 \quad (2)$$

Test Platform and Programs

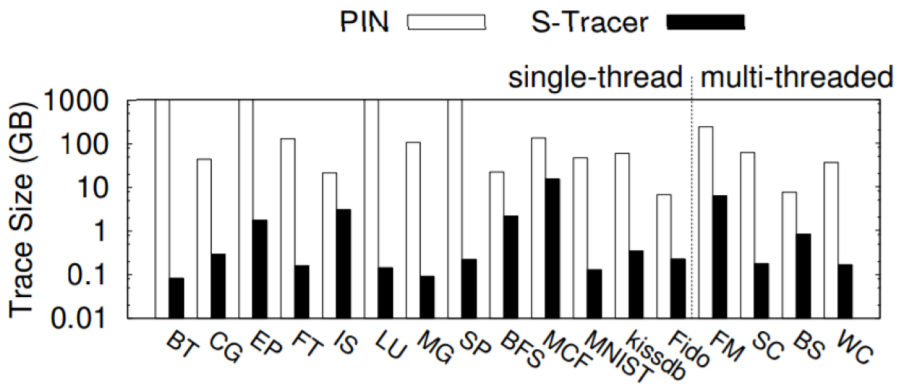
- Platform
  - Intel Xeon E7-8890 (v3), running CentOS 7.1
  - 128GB of DDR3 memory
  - 1TB SATA-2 hard disk
- Benchmark
  - NPB: BT, CG, EP, FT, IS, LU, MG, SP
  - SPECCPU: 400.perlbench, 401.bzip2, 403.gcc, 429.mcf, 433.milc, 445.gobmk, 456.hmmer, 458.sjeng, 464.h264ref, 470.lbm, 482.sphinx3
  - Parsec: streamcluster(SC), freqmine(FQ), blackscholes(BS)
- Applications
  - BFS from Graph 500
  - MNIST, Kissdb, Fido, Mapreduce word count(WC)

Evaluation

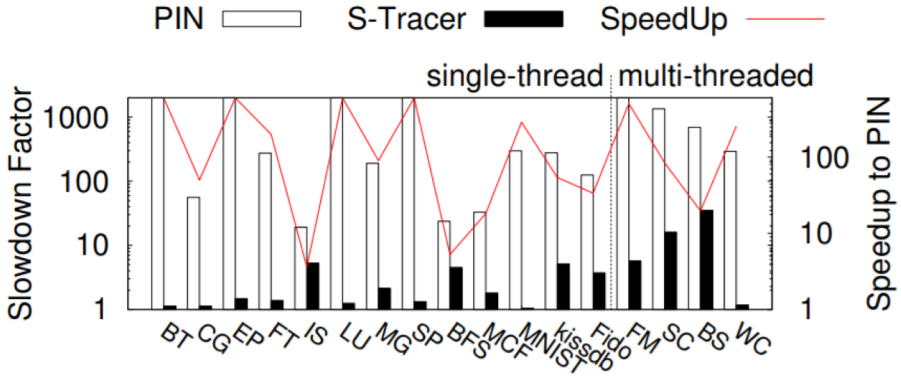
- **Compilation overhead:**
  - 2% to 35% of the original LLVM compilation cost(average at 10%).
- **Correctness:**
  - Full trace size difference between 0.5% and 6% compare with PIN (median at 3.2%)
  - Heap trace difference ratio between 0.0% and 4.7% compare with PIN (median at 1.5%)
- Cache simulation for worst case: BFS



Evaluation: S-Tracer



Trace size comparison

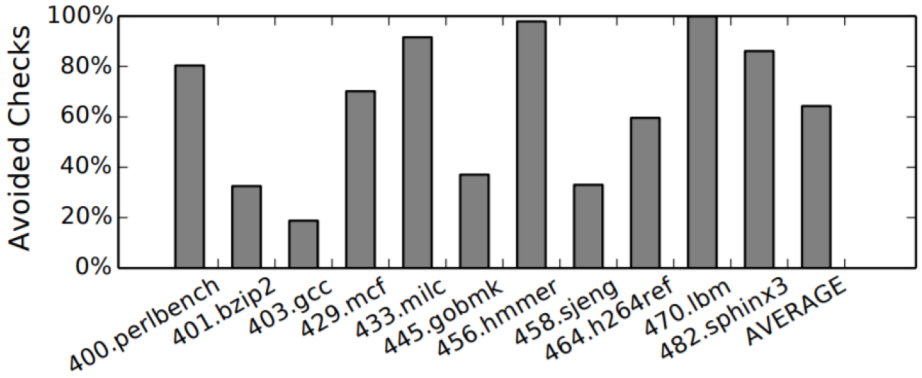


Application slowdown by S-Tracer and PIN with I/O (left) and S-Tracer speedup over PIN (right)

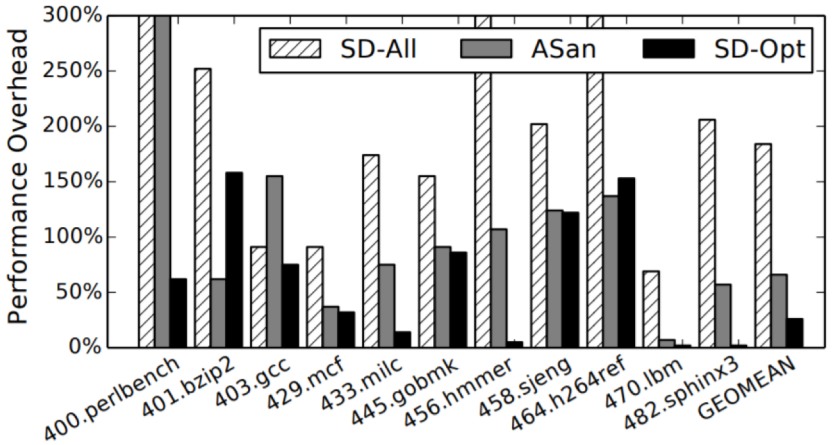
S-Tracer achieves orders of magnitude reduction in trace size from the PIN baseline.

S-Tracer reduces slowdown from PIN by a factor of 61X on average.

Evaluation: S-Detector



Reduction in runtime memory checks.



Overhead comparison (bars over 300% truncated).

Spindle enables S-Detector to cut runtime memory checks by 64%.

S-Detector brings down runtime overhead to geometric mean of 26% compared with ASan.

Take Away Messages

- **Static information is useful**
  - Provides reliable insight
  - Relatively cheap, with scale-independent cost
- **Challenges of Source code analysis**
  - Alias analysis
  - Without source code
  - Input dependent
  - Only binary

Q&A

Thank you!

PACMAN

pacman.cs.tsinghua.edu.cn