perf-taint: Taint Analysis for Automatic Many-Parameter Performance Modeling

Marcin Copik, Torsten Hoefler (advisor)

Extra-P [1]: Automatic Black-Box Performance Modelling

Parameters Identification → **Experiment Design** → **Experiment Execution** → **Models Generation**

Select problem size **s** and ranks **p** as model parameters.

Decide to use
- **5** values per parameter
- **5** samples per experiment

25 combinations of **p** and **s**

25 parameter values
5 repetitions per sample
125 instrumented executions

Create performance model for **each function**. Example result:

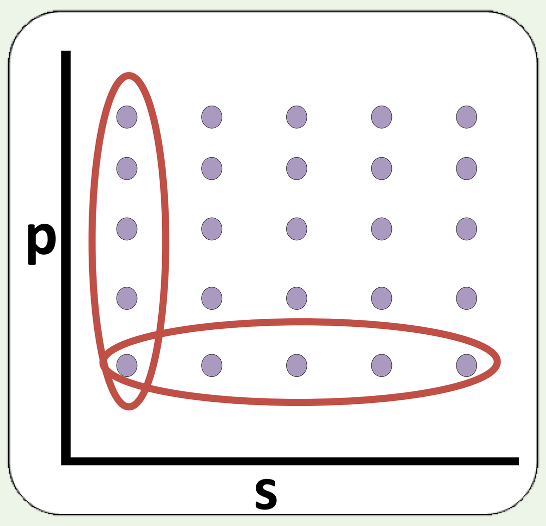
$$2.3\ s^3 + 1.71 \log_2 p - 0.1329$$

Challenge #1 Parameters

```
int nx, ny, nz, nt;
int node_geometry[4];
int nflavors, propinterval;
int warms, trajecs, steps;
int niter, nrestart, prec_pbp;
```

A **subset** of all *su3_rmd* parameters. Which have the largest effect on performance?

Challenge #2 Experiment Samples



How parameters interact with each other?

$p \times s$

25 experiments

$p + s$

9 experiments

Challenge #3 Functions

```
int p = MPI_ranks();
for(int i = 0; i < p - 1; ++i)
  MPI_Send(…);
```

$-10^{-5}s^2 + 1.3\ p + 0.7$

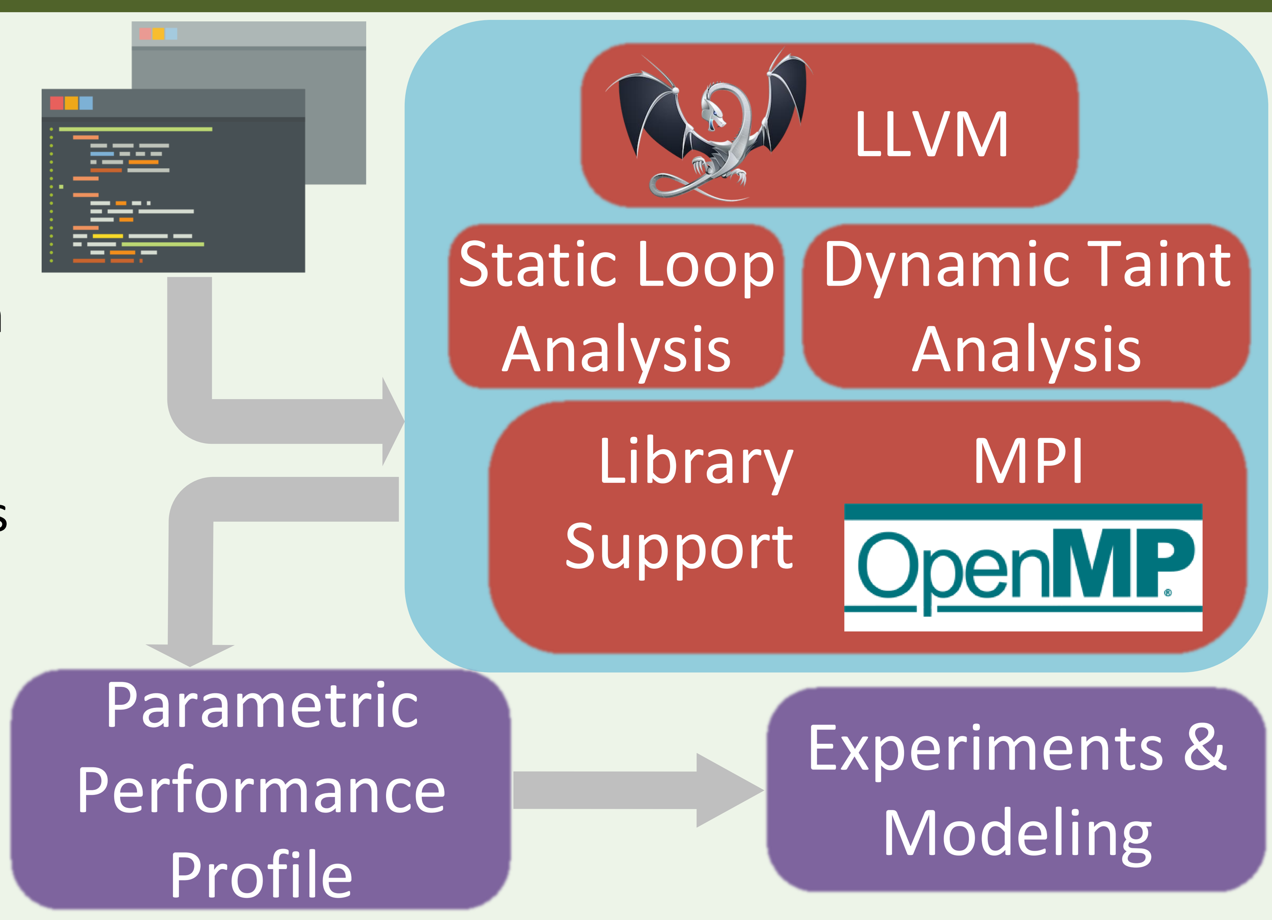
Performance irrelevant functions increase costs of **instrumented execution**. The presence of measurement noise leads to **overfitted models** with incorrect dependencies.

perf-taint: Taint-Based Performance Profiler

perf-taint applies a hybrid analysis to detect functions which performance depend on any parameter.

A **static analysis** detects functions with non-constant loops and recursion. A **dynamic taint analysis** tracks propagation of parameters and detects functions affected by parameters.

perf-taint is built on top of the **LLVM** framework and the **DfSan** analyzer. It is language, memory and hardware agnostic.



Why not static analysis?

Static techniques are often unable to provide precise answers due to **theoretical** and **practical** limitations.

```
taint_variable(s);
for(i = 0;i < s; ++i)
  j += f(i);
// Is j dependent on s?
while(condition() < j)
  ...
```

```
// Polymorphic type
Solver * ptr = getSolver();
int size = conf->getSize();
// Dynamic call, arguments
ptr->solve(size);
```

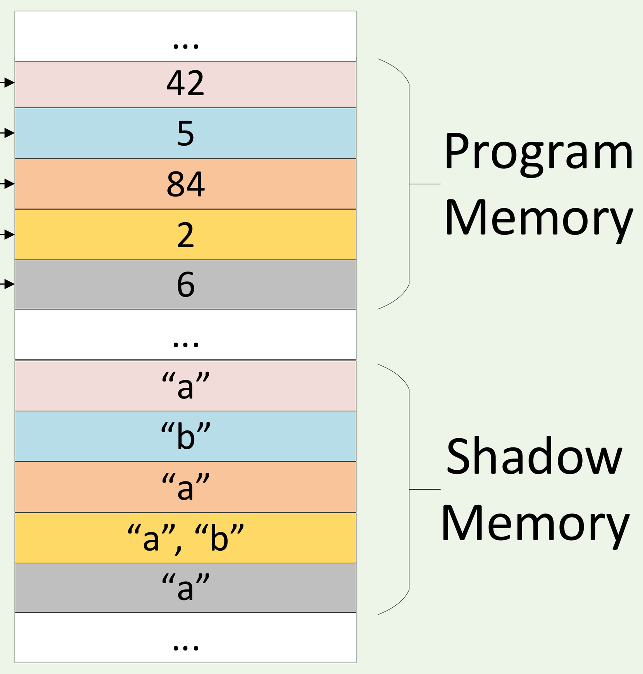
- ❑ **Alias analysis** is challenging in inter-procedural dataflow.
- ❑ **Data-dependent loop** conditions cannot be handled by loop trip counting.
- ❑ **Abstractions** introduced by modern languages decrease the likelihood of successful static analysis.

Taint analysis

```
int a = 42;
int b = omp_get_num_threads();
taint_variable(a);

// Data-flow propagation
int x = 2 * a;
int y = modulo(a, b);

// Control-flow propagation
int z = 10;
if(a != 43)
  z = 6;
```



User adds **taint sources** for parameters to investigate. Implicit taint sources: **MPI_Comm_rank**, **omp_get_num_threads**. Compiler inserts instrumentation to propagate taint labels on instruction result and control-flow.

Parametric Performance Profile

```
void f(int a, int b) {
  taint_variables(a, b);
  g(a, b); h(a, b); i(a, b);
}

void g(int a, int b) {
  for(int i = 0; i < a; ++i)
    j(b);
}

void h(int a, int b) {
  j(a);
}

void j(int c) {
  for(int j = 0; j < c; ++j)
    // compute
}

void i(int a, int b) {
  printf("%d %d\n", a, b);
}
```

Tainted Execution

$() \to \{\{a, b, a \times b\} + \{a\}\}$

$(f) \to \{a, b, a \times b\}$

$(f) \to \{a\}$

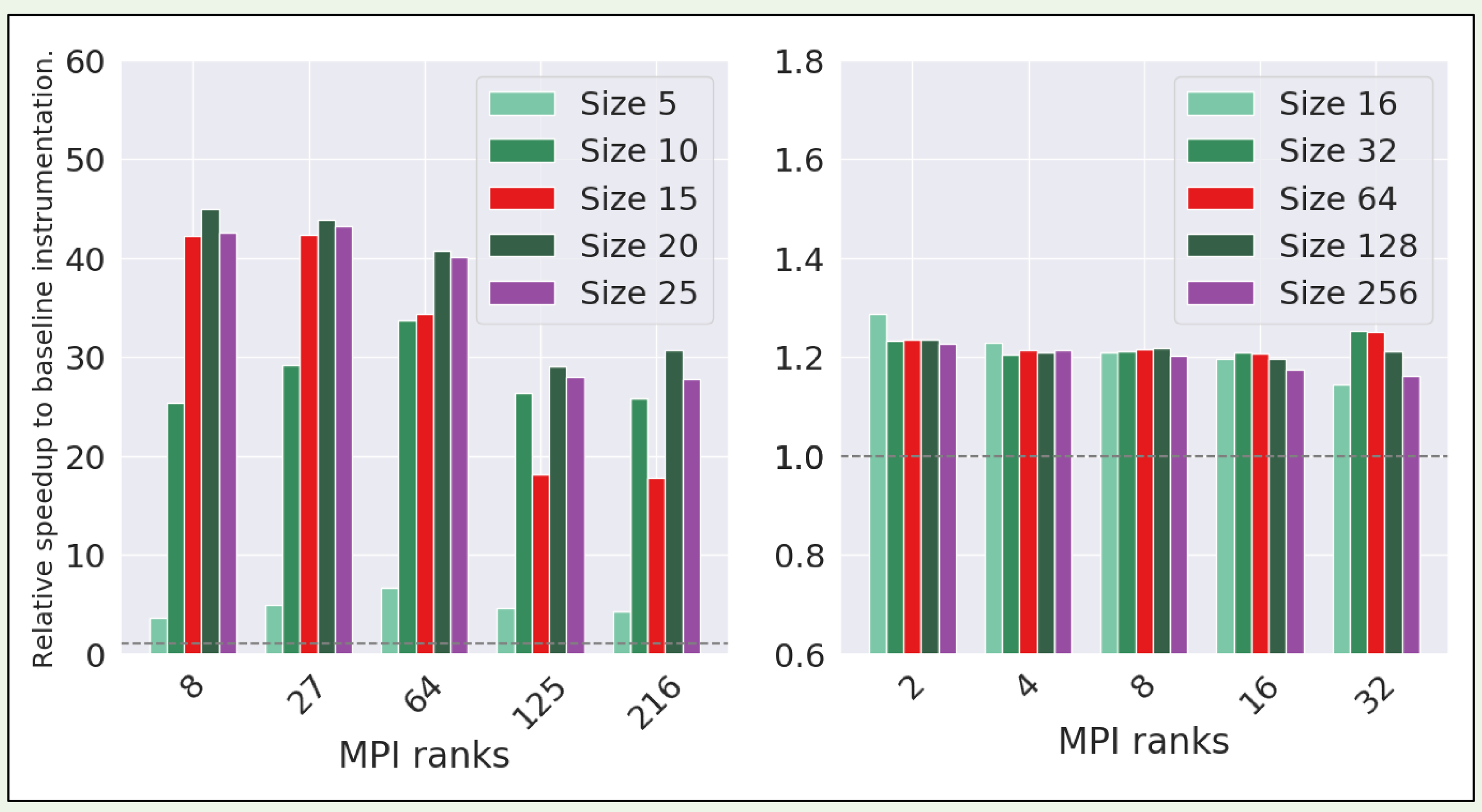
$(f, g) \to \{b\}$

$(f, h) \to \{a\}$

Compiler instruments **performance-critical control flow** with taint sinks. Program execution builds a model storing parametric **complexity of computation volume** for each **callpath**, letting modeler to skip **irrelevant functions** and **prune models** with incorrect dependencies.

Faster experiments...

Speedup of selective instrumentation in Score-P without inlining: *LULESH* (C++, left) and *su3_rmd* (C, right).



... and better models

LULESH, *CalcHourglassControlForElems* computation kernel with complexity $O(s^3)$

✗ $9.7 \times 10^{-7}\ s^{2.5} \log_2 s + 0.0024 \log_2 p - 0.016$

✓ $7.6 \times 10^{-7}\ s^{2.5} \log_2 s - 0.0025$

MILC su3_rmd, *do_gather* communication (s = 2048, p = 1024) -> runtime 0.039 s

✗ $8.2 \times 10^{-12}\ p^3 s^{0.75} \log_2 p + 6.2 \times 10^{-6}$
Prediction 26.7 s

✓ $2.2 \times 10^{-12}\ p^3 \log_2 p + 2.4 \times 10^{-6}$
Prediction 0.023 s

References

[1] A. Calotoiu et al., "Using automated performance modeling to find scalability bugs in complex codes" in SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis Article No. 45. DOI: 10.1145/2503210.2503277

ETH zürich — SC19 Denver, CO