SCALABLE COMPUTING SOFTWARE LABORATORY

ILLINOIS INSTITUTE OF TECHNOLOGY

Vidya: Performing Code-Block I/O Characterization for Data Access Optimization

HIPC'18 25th IEEE International Conference on High Performance Computing, Data, & Analytics

Hariharan Devarajan, Anthony Kougkas, Prajwal Challa, and Xian-He Sun

hdevarajan@hawk.iit.edu

Complex modern big data applications

- Multi-faceted : programming languages, libraries, algorithms, etc.
  - Montage has 23 million lines of code with 38 executables
  - Cubed-Sphere-Finite-Volume has more than a million lines of code with 23 simulation kernels and 54 analysis kernels.
  - Google has a code base of 2 billion lines with more than 50 languages and frameworks.

Tuning I/O of these applications is crucial in the performance of various systems

Current I/O Profiling tools

- Static analysis tools
  - tracing applications runtime behavior
  - Example: Darshan
- Dynamic analysis tools
  - identifying application's repetitive behavior using statistical or grammer-based prediction models.
  - Example: Omnisc'IO

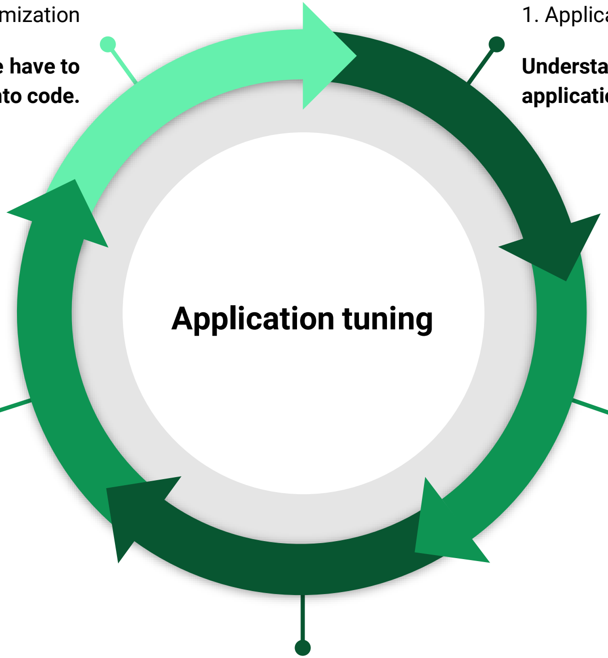
Current I/O tuning process

5. Applying the optimization

Once you identify the optimizations these have to implemented manually into code.

1. Application Analysis

Understand the structure of application and workflow



4. Profiler output analysis

Profiler outputs various graphs showcasing the behavior of application. Understanding it to pin-point optimizations could be a challenge

2. Profiler Choice

Each Profiler extracts its own set of metrics and features

3. Running the Profiler

Link application with profiler and run them (expensive)

Problem

- Static analysis tools are more accurate but have high profiling cost
- Dynamic analysis tools have little profiling cost but its accuracy depends on repetitive patterns

Can we do something better to balance this tradeoff?

Overview

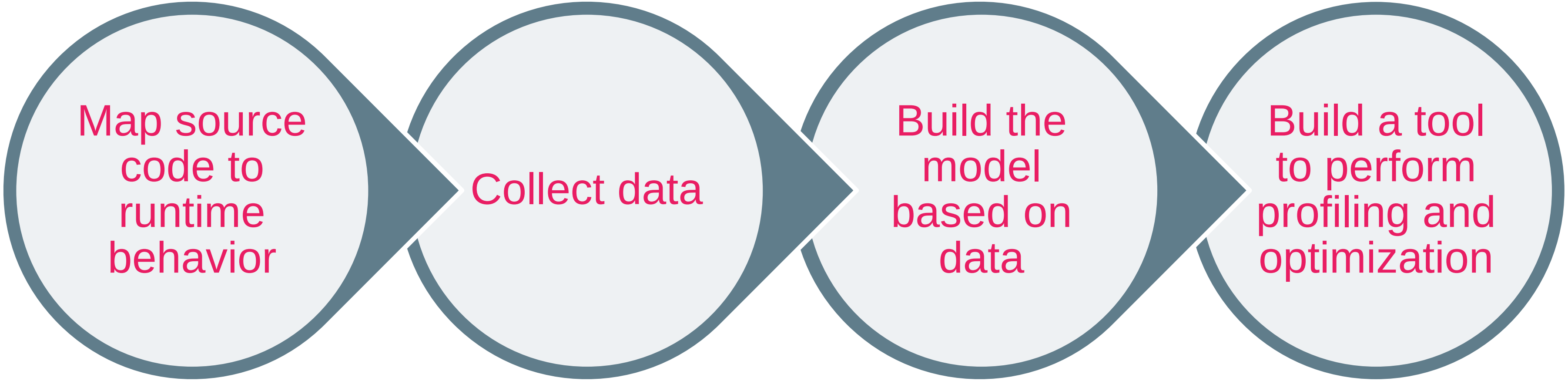
- Approach
- Design
- Results
- Conclusion
- Q & A

Approach (Basic Idea)

Goal: Lower Profiling Cost with good accuracy on profiling (add definitions)

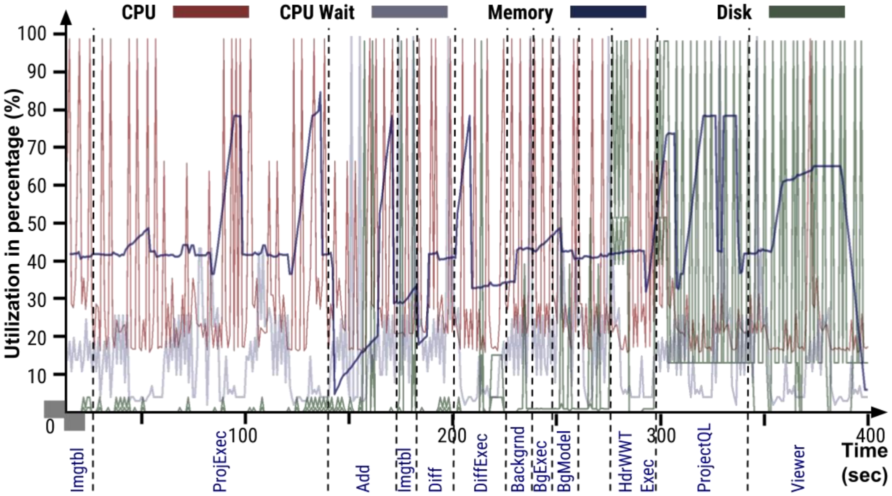
We use the **source code** based approach to achieve this goal.

Approach (Overview)



Co-relate application-behavior with its source code

- Montage
  - 38 million lines, 38 executables, complex end-to-end worklow
- We profile application using existing profiling tools and manually inspect the code with seen behavior
  - Compute-intensive: mImgtbl, mProjExec, and mDiff
  - Data-intensive: mHdrWWTExec, mProjectQL, and mViewer.
  - Balanced: mAdd, mFitExec, and mDiffExec.



Correlate application-behavior with its source code

S.No	Description	Eg. Executable
P_1	loop count containing I/O calls (i.e., number of iterations)	mProjectQL
P_2	number of I/O operations (i.e., count of calls)	mHdrWWTExec
P_3	amount of I/O (i.e., size in bytes)	mHdrWWTExec
P_4	number of synchronous I/O operations	mAdd
P_5	number of I/O operations enclosed by a conditional statement	mAdd
P_6	number of I/O operations that use binary data format	mViewer
P_7	number of flush operations	mViewer
P_8	size of file opened	mHdrWWTExec
P_9	number of sources/destination files used	mProjectQL
P_{10}	space-complexity of code	mProjectQL
P_{11}	function stack size of the code	DiffExec
P_{12}	number of random file accesses	mViewer
P_{13}	number of small file accesses	mProjectQL
P_{14}	size of application (i.e. number of processes)	Application Specific
P_{15}	storage device characteristics (i.e. access concurrency, latency and bandwidth)	System specific

Collecting Data

- Build dataset consists from a variety of applications:
  - graph exploration kernels (BFS, DFS, Page-rank)
  - sorting programs (Tera-sort, external-sort)
  - machine learning kernels (Kmeans, random forest classifications)
  - I/O and CPU benchmarks (IOR, Graph500, HACC)
- We use code-block as a unit (a function/class/branch/loop/line of code)
- **I/O intensity of a code-block** is I/O time by the overall time of the code-block
- final dataset consists of 4200 records.

Build a model (CIOC – Code-block I/O intensity)

- Model all parameters as Variables (more details in the paper)
- Build a linear regression model of the form

$$Y_m(v) = \beta_0 + \sum_{i=1}^{v} \beta_i * X_{im}$$

where

- Y is the dependent variable I/O intensity,
- m is the m^{th} code block,
- v are the variables,
- β are the coefficients of the regression
- X_{im} is the value of the i^{th} variable for m^{th} code-block.

Linear Regression model (CIOC)

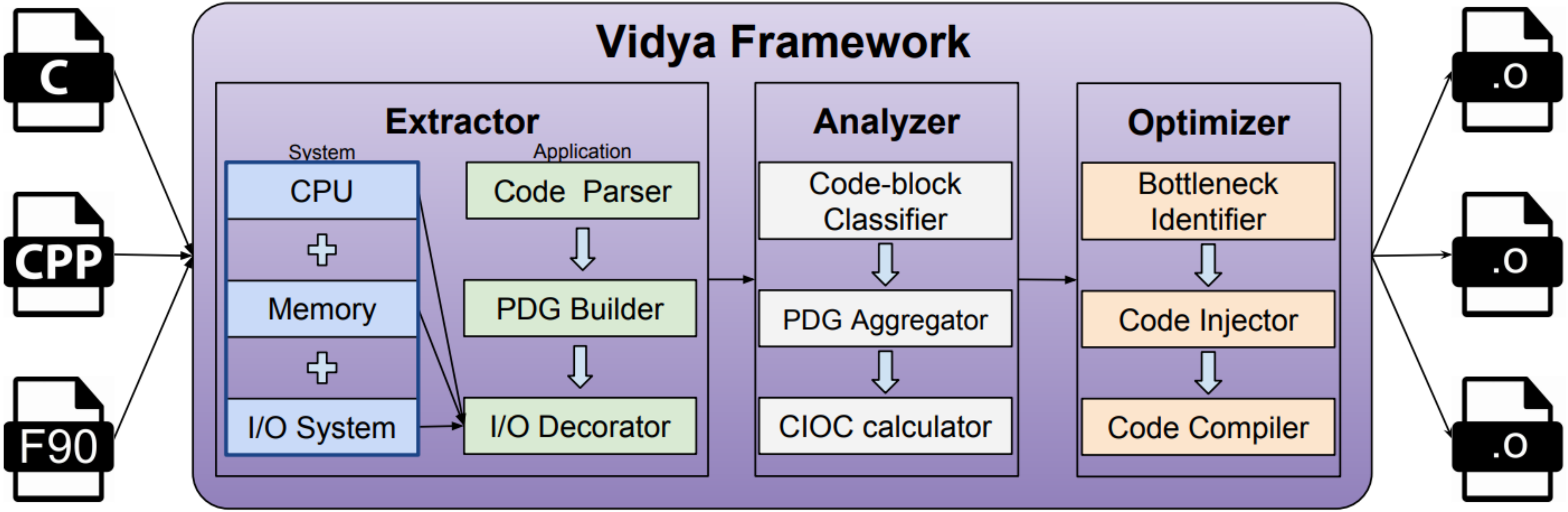
- The linear regression model excludes variables with |t| <2
- Good model fit and predictability
  - High R^2
  - High f-statistic score
- Top two significant variables
  - Amount of I/O
  - Number of files opened

Name	Coefficient	Std. Error	t-ratio
const	−1.99	0.16	−11.92
X_1	0.17	0.33	2.53
X_2	278.80	44.18	6.30
X_3	3706.47	196.81	18.83
X_4	−42612.30	14540.90	−2.93
X_5	Excluded		
X_6	Excluded		
X_7	−10487.80	2511.20	−4.17
X_8	Excluded		
X_9	809.04	93.55	8.64
X_{10}	183996.00	5843.16	31.49
X_{11}	Excluded		
X_{12}	227.98	18.43	12.36
X_{13}	6456.39	2257.85	2.86
X_{14}	0.78	0.10	7.24
X_{15}	Excluded		
X_{16}	Excluded		

Metric	Value
Mean dependent	−6.78
S.D. dep. var	1.69
Sum^2 resid	2675.76
S.E. of reg.	0.79
R^2	0.92
Adjusted R^2	0.91
$F(16, 4183)$	785.13
P-value(F)	0.00

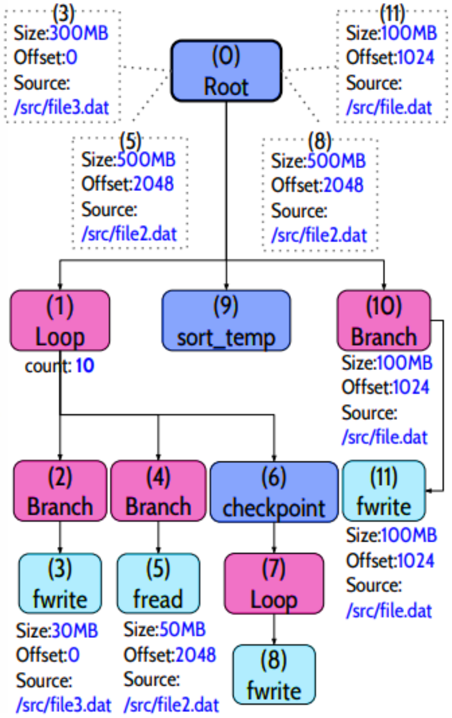
Vidya design

High level design



Example (Extractor and Analyzer)

```
void main(int argc, char *argv[]) {
  int loop_count = std::stoi(argv[1]);
  for (int i = 0; i < loop_count; i++) {
    if (myrank % 2 == 0)
      fwrite(write_buf, write_sz, write_cnt, input_fh);
    else
      fread(read_buf, read_sz, read_cnt, output_fh);
    checkpoint(i);
  }
  sort_temp();
  if (myrank == 0)
    fwrite(result_buf,result_sz,result_cnt,results_fh);
}
int checkpoint(int i) {
  for (int j = 0; j < i; j++)
    fwrite(temp_buf,temp_sz,temp_cnt,intermediate_fh);
}
int sort_temp() {
  std::sort(temp_results.begin(), temp_results.end());
}
```



Example (Optimizer) Psuedo-code(Does not compile :)

```
void main(int argc, char *argv[]) {
  int loop_count = std::stoi(argv[1]);
  for (int i = 0; i < loop_count; i++) {
    std::sort(temp_results.begin(),
              temp_results.rbegin()-i);
    fread(read_buf, read_sz,
          read_cnt, input_fh);
  }
  if (myrank == 0)
    fwrite(result_buf,result_sz,
           result_cnt,results_fh);
}
```

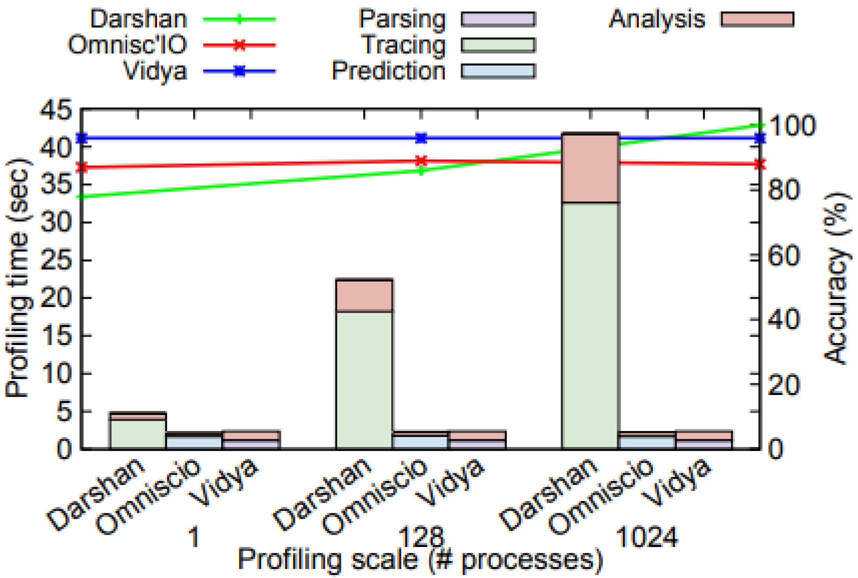
```
void main(int argc, char *argv[]) {
  int loop_count = std::stoi(argv[1]);
  for (int i = 0; i < loop_count; i++) {
    vidya::async_prefetch(read_buf, read_sz,
                          read_cnt, input_fh);
    std::sort(temp_results.begin(),
              temp_results.rbegin()-i);
    vidya::buffer_read(read_buf, read_sz,
                       read_cnt, input_fh);
  }
  if (myrank == 0)
    fwrite(result_buf,result_sz,
           result_cnt,results_fh);
}
```

Evaluation

- Chameleon Cluster
  - 32 client nodes and 8 storage server nodes
  - Each node has 128 GB RAM, 10Gbit Ethernet, and a local 200GB HDD
- Applications used
  - Synthetic Benchmarks
  - CM1
  - WRF
  - Graph500's bfs and GMC
- Baselines
  - Darshan
  - Omnisc'IO

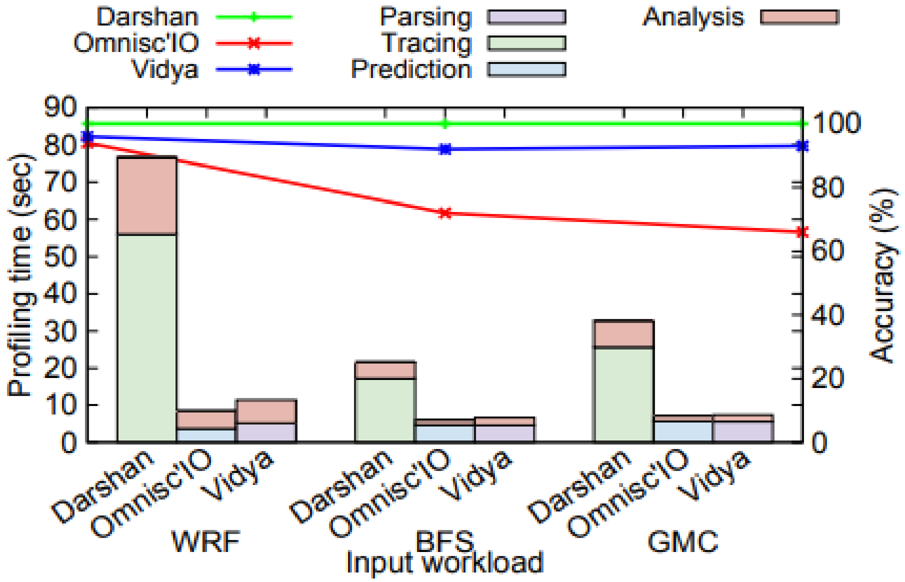
Profiling Performance

- Profiling scale
  - Sensitive for Darshan
  - Application CM1
  - Prediction I/O intensity
- Results
  - Vidya's parsing or Omnisc'IO is not affected
  - Darshan's accuracy is better if the tracing is done close actual running scale but that decreases profiling performance.



Profiling Performance

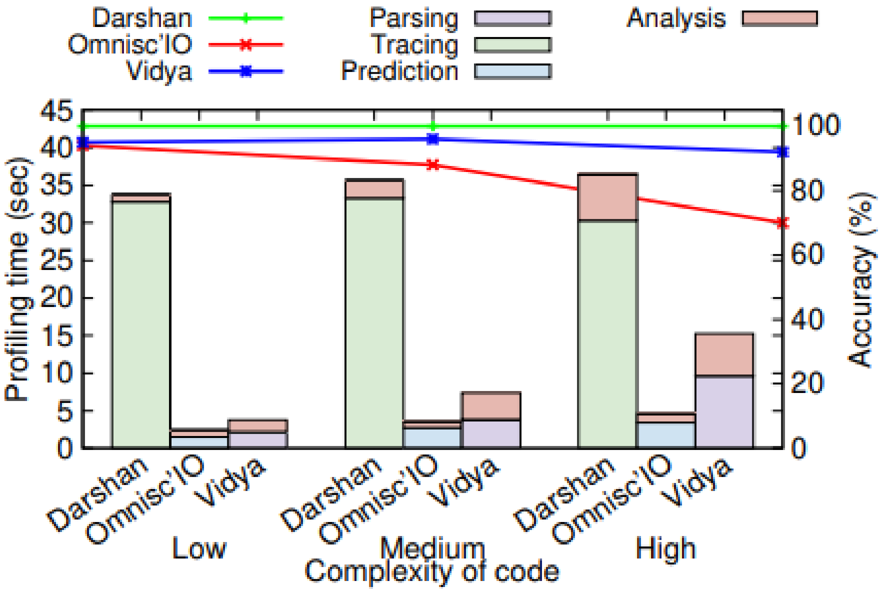
- Workload irregularity
  - Sensitive for Omnisc'IO
  - Applications: WRF, BFS, GMC
  - Prediction I/O intensity
- Results
  - Vidya's parsing or Darshan's tracing is not affected
  - Omnisc'IO has a known limitation irregular patterns



Profiling Performance

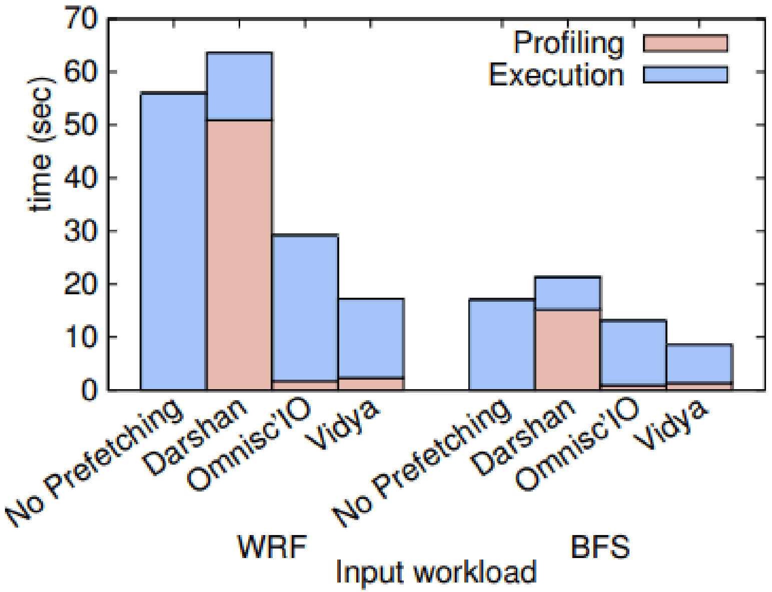
- Complexity of code
  - Sensitive for Vidya
  - Application: Synthetic
  - Complexity: loops, functions,classes and files
  - Prediction I/O intensity

- Results
  - The parsing time for Vidya extractor increases
    - still 3x faster than tracing
    - But 2x slower than Omnisc'IO



I/O Optimization

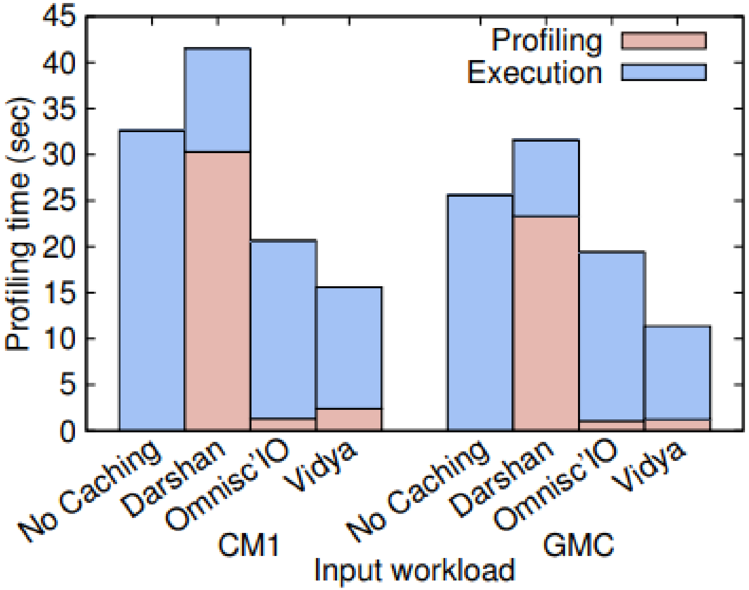
- Prefetching Optimization
  - Applications: WRF and BFS
  - Characteristics: Irregular workloads with simple code.
  - Prediction if prefetching is required (based on opportunity to overlap)
- Results
  - Darshan has the best optimized code
  - Omnisc'IO has the least profiling time/overhead
  - Vidya has the best overall performance (profiling+optimization)



(a) Prefetching On/Off

I/O Optimization

- Caching Optimization
  - Applications: CM1 and GMC
  - Characteristics: repetitive with complex code structures.
  - Prediction if caching is required (based on I/O interference)
- Results
  - Darshan has the best optimized code
  - Omnisc'IO has the least profiling time/overhead
  - Vidya has the best overall performance (profiling+optimization)



(b) Write-cache On/Off

Discussion & Limitations

- Discussion: Measurement Vs Prediction
  - it is a trade-off between accuracy and cost of profiling

- Limitation: Source code approach
  - Dynamic runtime flows
  - Dynamic code generation
  - Dynamic library linking

Conclusions

- Vidya proposes a tradeoff of accuracy to profiling performance.
- Results show that Vidya can make profiling of applications faster by 9x while having a high accuracy of 98%.
- Vidya can be used to optimize applications up to 3.7x.

Q & A

Hariharan Devarajan hdevarajan@hawk.iit.edu