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 grammar-based prediction models.
  - Example: Omnisc’IO
Current I/O tuning process

1. Application Analysis
Understand the structure of application and workflow

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)

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

5. Applying the optimization
Once you identify the optimizations these have to be implemented manually into code.
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.
Build a tool to perform profiling and optimization

Approach (Overview)

Map source code to runtime behavior

Collect data

Build the model based on data

Build a tool to perform profiling and optimization
Co-relate application-behavior with its source code

- Montage
  - 38 million lines, 38 executables, complex end-to-end workflow
- 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

<table>
<thead>
<tr>
<th>S.No</th>
<th>Description</th>
<th>Eg. Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(_1)</td>
<td>loop count containing I/O calls (i.e., number of iterations)</td>
<td>mProjectQL</td>
</tr>
<tr>
<td>P(_2)</td>
<td>number of I/O operations (i.e., count of calls)</td>
<td>mHdrWWTExec</td>
</tr>
<tr>
<td>P(_3)</td>
<td>amount of I/O (i.e., size in bytes)</td>
<td>mHdrWWTExec</td>
</tr>
<tr>
<td>P(_4)</td>
<td>number of synchronous I/O operations</td>
<td>mAdd</td>
</tr>
<tr>
<td>P(_5)</td>
<td>number of I/O operations enclosed by a conditional statement</td>
<td>mAdd</td>
</tr>
<tr>
<td>P(_6)</td>
<td>number of I/O operations that use binary data format</td>
<td>mViewer</td>
</tr>
<tr>
<td>P(_7)</td>
<td>number of flush operations</td>
<td>mViewer</td>
</tr>
<tr>
<td>P(_8)</td>
<td>size of file opened</td>
<td>mHdrWWTExec</td>
</tr>
<tr>
<td>P(_9)</td>
<td>number of sources/destination files used</td>
<td>mProjectQL</td>
</tr>
<tr>
<td>P(_{10})</td>
<td>space-complexity of code</td>
<td>mProjectQL</td>
</tr>
<tr>
<td>P(_{11})</td>
<td>function stack size of the code</td>
<td>DiffExec</td>
</tr>
<tr>
<td>P(_{12})</td>
<td>number of random file accesses</td>
<td>mViewer</td>
</tr>
<tr>
<td>P(_{13})</td>
<td>number of small file accesses</td>
<td>mProjectQL</td>
</tr>
<tr>
<td>P(_{14})</td>
<td>size of application (i.e. number of processes)</td>
<td>Application Specific</td>
</tr>
<tr>
<td>P(_{15})</td>
<td>storage device characteristics (i.e. access concurrency, latency and bandwidth)</td>
<td>System specific</td>
</tr>
</tbody>
</table>

### APPROACH

### DESIGN

### RESULTS

### DISCUSSION

### CONCLUSION
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,
- \( \beta \) are the coefficients of the regression
- \( X_{im} \) is the value of the \( i^{th} \) variable for \( m^{th} \) code-block.
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

### Results

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

### Metric Values

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean dependent</td>
<td>-6.78</td>
</tr>
<tr>
<td>S.D. dep. var</td>
<td>1.69</td>
</tr>
<tr>
<td>$Sum^2$ resid</td>
<td>2675.76</td>
</tr>
<tr>
<td>S.E. of reg.</td>
<td>0.79</td>
</tr>
<tr>
<td>$R^2$</td>
<td>0.92</td>
</tr>
<tr>
<td>Adjusted $R^2$</td>
<td>0.91</td>
</tr>
<tr>
<td>$F(16, 4183)$</td>
<td>785.13</td>
</tr>
<tr>
<td>P-value($F$)</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Vidya design
High level design
Example (Extractor and Analyzer)

```c
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) Pseudo-code (Does not compile :)

```cpp
1 void main(int argc, char *argv[]) {
2     int loop_count = std::stoi(argv[1]);
3     for (int i = 0; i < loop_count; i++) {
4         std::sort(temp_results.begin(), temp_results.rbegin()-i);
5         fread(read_buf, read_sz, read_cnt, input_fh);
6     }
7     if (myrank == 0)
8     fwrite(result_buf,result_sz, result_cnt,results_fh);
9 }
10 }
11 int loop_count = std::stoi(argv[1]);
12 for (int i = 0; i < loop_count; i++) {
13     vidya::async_prefetch(read_buf, read_sz, read_cnt, input_fh);
14     std::sort(temp_results.begin(), temp_results.rbegin()-i);
15     vidya::buffer_read(read_buf, read_sz, read_cnt, input_fh);
16 }
17     if (myrank == 0)
18     fwrite(result_buf,result_sz, result_cnt,results_fh);
19 }
```
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'I0
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'I0 has the least profiling time/overhead
  - Vidya has the best overall performance (profiling+optimization)
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