Performance Regression Testing
Target Prioritization via Performance Risk Analysis

Peng Huang, Xiao Ma, Dongcai Shen, Yuanyuan Zhou

University of California, San Diego
University of Illinois at Urbana-Champaign

http://cseweb.ucsd.edu/~peh003/perfscope
Trend #1: Software evolving fast

Lines of code for MySQL over past 10 years grew from ~5 million to ~13 million!
**Trend #1: Software evolving fast**

The average revision rate can be $\geq 100$ commits per day!

<table>
<thead>
<tr>
<th>Software</th>
<th>Avg. Rev. Per Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>~27</td>
</tr>
<tr>
<td>Chrome</td>
<td>~155</td>
</tr>
<tr>
<td>Linux</td>
<td>~170</td>
</tr>
</tbody>
</table>
Trend #1: Software evolving fast

<table>
<thead>
<tr>
<th>Software</th>
<th>Avg. Rev. Per Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>~27</td>
</tr>
<tr>
<td>Chrome</td>
<td>~155</td>
</tr>
<tr>
<td>Linux</td>
<td>~170</td>
</tr>
</tbody>
</table>

Broken functionality or worse performance!
Trend #2: Performance testing, important but slow...

Upgrading MySQL 4.1 to 5.0 in a production e-commerce website:

“Although this is a performance issue, total page rendering time in my web shop would increase from **1 second to 20 seconds** for example if showing a decent amount of products and prices on the same page. Therefore MySQL 5 is no good for production until this bug is fixed.”
Trend #2: Performance testing, important but slow...

Performance critical software

<table>
<thead>
<tr>
<th>Category</th>
<th>Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Server</td>
<td>autobench,Web Polygraph,SPECweb</td>
</tr>
<tr>
<td>Database</td>
<td>pgbench,sysbench,DBT2</td>
</tr>
<tr>
<td>Compiler</td>
<td>CP2K,Polyhedron,SPEC CPU</td>
</tr>
<tr>
<td>OS</td>
<td>Imbench,Phoronix Test Suite</td>
</tr>
</tbody>
</table>

Performance regression testing benchmark
Trend #2: Performance testing, important but slow...

<table>
<thead>
<tr>
<th>Category</th>
<th>Test Suite</th>
<th>Per Run Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Server</td>
<td>autobench,Web Polygraph,SPECweb</td>
<td>3min—1hr</td>
</tr>
<tr>
<td>Database</td>
<td>pgbench,sysbench,DBT2</td>
<td>10min—3hrs</td>
</tr>
<tr>
<td>Compiler</td>
<td>CP2K,Polyhedron,SPEC CPU</td>
<td>1hr—20hrs</td>
</tr>
<tr>
<td>OS</td>
<td>Imbench,Phoronix Test Suite</td>
<td>2hrs—24hrs</td>
</tr>
</tbody>
</table>

Performance regression testing cost
Problem: Catch me (perf. regression) if you can!

Doing [performance testing] between just release kernels means that there will be a two-month lag between telling developers that something pissed up performance. Doing it every day (or at least a couple of times a week) will be much more interesting. [ . . . ] Two months (or half a year) later, and we have absolutely no idea what might have caused a regression. For example, that 2.6.2->2.6.8 change obviously makes pretty much any developer just go : I’ve got no clue.

-- Linus Torvalds
Current practices of perf. regression testing

• Aggregate testing
  • Daily, weekly, per-release

• Prioritize test cases
  • Divide based on comprehensiveness and overhead
  • Multiple levels
Our tool—**PerfScope**—in a nutshell

- Prioritize perf. regression testing *target* with **Performance Risk Analysis**
  - Statically examine a code commit
  - Conduct performance risk analysis
- Lightweight, white-box
- NOT a performance bug detection tool
Our tool—PerfScope—in a nutshell

• Prioritize perf. regression testing target with Performance Risk Analysis
  • Statically examine a code commit
  • Conduct performance risk analysis
• Lightweight, white-box
• NOT a performance bug detection tool
Outline

• Understanding real world performance regression issues
• Performance risk analysis design
• Implementation: PerfScope
• Evaluation
• Conclusion
Performance regression study

• What do real world performance regression issues look like?
• Is there opportunity to statically analyze the performance impact of code change?
• If so, based on the real world issues, what static analysis is needed?
# Study subjects

<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
<th># of issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>DBMS</td>
<td>50</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>DBMS</td>
<td>25</td>
</tr>
<tr>
<td>Chrome</td>
<td>Web Browser</td>
<td>25</td>
</tr>
</tbody>
</table>

Studied software of real world performance regression issues
Categorizing problematic code changes

```c
Foo()
{
    ...
    do_add;
    do_add;
    do_add;
    ...
}
```

<table>
<thead>
<tr>
<th>Performance</th>
<th>Performance regression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost of <code>do_add</code></td>
<td>↑</td>
</tr>
<tr>
<td>Execution frequency of <code>do_add</code></td>
<td>↑</td>
</tr>
</tbody>
</table>
Where the problematic change takes place?

```c
int bstream_rd_db_catalogue(...) {
    
    do {
        if (bcat_add_item(cat, &ti.base.base) != BSTREAM_OK)
            return BSTREAM_ERROR;
    } while (ret == BSTREAM_OK);

    int bcat_add_item(...)
    {
        switch (item->type) {
            case BSTREAM_IT_PRIVILEGE:
                Image_info::Dbobj *it1 = info->add_db_object(...);
                
        }
    }

back::Image_info::Dbobj* Backup_info::add_db_object(...) {
    
    if (type == BSTREAM_IT_TRIGGER) {
        + if (type == BSTREAM_IT_TRIGGER) {
            + obs::Obj* tbl_obj=obs::find_table_for_trigger(...);
    }

    
}
```

The new block calls an expensive function. When indirectly executed inside a loop, it can incur 80 times slowdown.
Where the problematic change takes place?

```c
bool test_if_skip_sort_order(...) {
    if (select_limit >= table_records) {
        /* filesort() and join cache are usually
           faster than reading in index order
           and not using join cache */
        if (tab->type == JT_ALL && ...)
            DEBUG_RETURN(0);
    }
    DEBUG_RETURN(1);
}

int create_sort_index() {
    if ((order != join->group_list || ... &&
         test_if_skip_sort_order(...))
        DEBUG_RETURN(0);
    table->sort.found_records = filesort(thd,
                                          table, join->sortorder, ...);
}
```
What the problematic change modifies?

<table>
<thead>
<tr>
<th>Modified program elements</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Chrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expensive function call</td>
<td>21 (42%)</td>
<td>9 (36%)</td>
<td>16 (64%)</td>
</tr>
<tr>
<td>Performance sensitive condition</td>
<td>8 (16%)</td>
<td>6 (24%)</td>
<td>4 (16%)</td>
</tr>
<tr>
<td>Performance critical variable</td>
<td>6 (12%)</td>
<td>5 (20%)</td>
<td>2 (8%)</td>
</tr>
<tr>
<td>Others</td>
<td>15 (30%)</td>
<td>5 (20%)</td>
<td>3 (12%)</td>
</tr>
</tbody>
</table>
What the problematic change modifies?

```c
bool test_if_skip_sort_order(...) {
    if (select_limit >= table_records) {
        /* filesort() and join cache are usually faster than reading in index order and not using join cache */
        if (tab->type == JT_ALL && ...) 
            DBUG_RETURN(0);
    }
    DBUG_RETURN(1);
}
```

The new control flow can change the function return value, which later affects whether an expensive path (with `filesort` call) will be taken or not.

- **Performance sensitive condition**
- **Performance critical variable**
- **Expensive function call**

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Chrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance sensitive condition</td>
<td>21 (42%)</td>
<td>9 (36%)</td>
<td>16 (64%)</td>
</tr>
<tr>
<td>Performance critical variable</td>
<td>8 (16%)</td>
<td>6 (24%)</td>
<td>4 (16%)</td>
</tr>
<tr>
<td>Expensive function call</td>
<td>6 (12%)</td>
<td>5 (20%)</td>
<td>2 (8%)</td>
</tr>
<tr>
<td>Others</td>
<td>15 (30%)</td>
<td>5 (20%)</td>
<td>3 (12%)</td>
</tr>
</tbody>
</table>
What the problematic change modifies?

```c
uint make_join_readinfo(JOIN *join, ulonglong options)
{
    for (i=join->const_tables; i < join->tables; i++) {
        JOIN_TAB *tab=join->join_tab+i;
        + if (table->s->primary_key != MAX_KEY &&
        +     table->file->primary_key_is_clustered())
        +     tab->index= table->s->primary_key;
        + else
        +     tab->index=find_shortest_key(table, ...);
    }
}
int join_read_first(JOIN_TAB *tab)
{
    if (!table->file->inited)
        table->file->ha_index_init(tab->index, tab->sorted);
}
```

The new logic prefers clustered primary index over secondary ones. It degrades performance for certain workloads.
## How a change impacts performance?

<table>
<thead>
<tr>
<th>Type of performance impact</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Chrome</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct</strong></td>
<td>34 (68%)</td>
<td>11 (44%)</td>
<td>12 (48%)</td>
</tr>
<tr>
<td>Indirect</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via function return value</td>
<td>7 (14%)</td>
<td>7 (28%)</td>
<td>3 (12%)</td>
</tr>
<tr>
<td>Via function parameter</td>
<td>5 (10%)</td>
<td>4 (16%)</td>
<td>1 (4%)</td>
</tr>
<tr>
<td>Via class member</td>
<td>1 (2%)</td>
<td>1 (4%)</td>
<td>3 (12%)</td>
</tr>
<tr>
<td>Via global variable</td>
<td>1 (2%)</td>
<td>0 (0%)</td>
<td>1 (4%)</td>
</tr>
<tr>
<td>Others</td>
<td>2 (4%)</td>
<td>2 (8%)</td>
<td>5 (20%)</td>
</tr>
</tbody>
</table>
Outline

• Understanding real world performance regression issues
• Performance risk analysis design
• Implementation: PerfScope
• Evaluation
• Conclusion
Performance Risk Analysis (PRA)

• Goal: statically analyze code change’s risk in incurring performance regression

• Two pieces of information:
  - Cost of changed operation
  - Execution frequency of changed operation
class CostModel {
    protected:
        virtual unsigned getArithmeticInstrCost(...);
        virtual unsigned getMemoryOpCost(...);
        virtual unsigned getCallCost(...);
        ...
    public:
        virtual unsigned getInstructionCost(...);
        virtual unsigned getBasicBlockCost(...);
        virtual unsigned getLoopCost(...);
        virtual unsigned getFunctionCost(...);
};
Static cost model

class CostModel {
    protected:
    virtual unsigned getArithmeticInstrCost(...);
    virtual unsigned getMemoryOpCost(...);
    virtual unsigned getCallCost(...);
    ...

    public:
    virtual unsigned getInstructionCost(...);
    virtual unsigned getBasicBlockCost(...);
    virtual unsigned getLoopCost(...);
    virtual unsigned getFunctionCost(...);
}
Execution frequency estimation

- Static loop iteration count estimation
  - If cannot determine -> frequent
- Recursive function -> frequent
- Inter-procedural
### Risk matrix

<table>
<thead>
<tr>
<th>Cost</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Frequent</td>
</tr>
<tr>
<td>Expensive</td>
<td>Extreme</td>
</tr>
<tr>
<td>Normal</td>
<td>High</td>
</tr>
<tr>
<td>Minor</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

Performance risk matrix given cost and frequency information
Outline

• Understanding real world performance regression issues
• Performance risk analysis design
• Implementation: PerfScope
• Evaluation
• Conclusion
PerfScope architecture
PerfScope

- On top of LLVM infrastructure
- Currently support C/C++
- Open sourced in
  http://cseweb.ucsd.edu/~peh003/perfscope
Outline

• Understanding real world performance regression issues
• Performance risk analysis design
• Implementation: PerfScope
• Evaluation
• Conclusion
Evaluation on studied perf. regression commits

<table>
<thead>
<tr>
<th>Software</th>
<th>Problematic Commits</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>39</td>
<td>35</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>64</strong></td>
<td><strong>58 (91%)</strong></td>
</tr>
</tbody>
</table>
Evaluation on new perf. regression commits

• 600 new commits from 6 popular, large-scale software
• Obtained “ground truth” by running standard perf. testing suite

<table>
<thead>
<tr>
<th>Software</th>
<th>LOC</th>
<th>Studied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>1.2M</td>
<td>Yes</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>651K</td>
<td>Yes</td>
</tr>
<tr>
<td>GCC</td>
<td>4.6M</td>
<td>No</td>
</tr>
<tr>
<td>V8</td>
<td>680K</td>
<td>No</td>
</tr>
<tr>
<td>Squid</td>
<td>751K</td>
<td>No</td>
</tr>
<tr>
<td>Apache</td>
<td>220K</td>
<td>No</td>
</tr>
</tbody>
</table>
Evaluation on new perf. regression commits

<table>
<thead>
<tr>
<th>Software</th>
<th>Test Commits</th>
<th>Risky Commits</th>
<th>Recommended (Reduction)</th>
<th>Miss (Coverage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>100</td>
<td>9</td>
<td>22 (78%)</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>100</td>
<td>6</td>
<td>16 (84%)</td>
<td>0</td>
</tr>
<tr>
<td>Squid</td>
<td>100</td>
<td>5</td>
<td>12 (88%)</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>100</td>
<td>6</td>
<td>14 (86%)</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>600</td>
<td>39</td>
<td>100 (83%)</td>
<td>2 (95%)</td>
</tr>
</tbody>
</table>

PerfScope can **reduce** at least **78%** of the performance regression testing candidates and is still able to **alarm 95%** of the risky ones.
# Running time of PerfScope

<table>
<thead>
<tr>
<th>Software</th>
<th>LOC</th>
<th>Analysis Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>1.2M</td>
<td>235</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>651K</td>
<td>194</td>
</tr>
<tr>
<td>GCC</td>
<td>4.6M</td>
<td>289</td>
</tr>
<tr>
<td>V8</td>
<td>680K</td>
<td>344</td>
</tr>
<tr>
<td>Squid</td>
<td>751K</td>
<td>34</td>
</tr>
<tr>
<td>Apache</td>
<td>220K</td>
<td>9</td>
</tr>
</tbody>
</table>
Outline

• Understanding real world performance regression issues
• Performance risk analysis design
• Implementation: PerfScope
• Evaluation
• Conclusion
Limitations and future work

- Cost modeling is simple
- No offsetting for delete/replace changes
- Mainly for CPU cost
  - Can be extended for I/O
- Combine with perf. test case prioritization
  - Already know which code region is risky, associate with coverage information.
Conclusion

• Software evolves fast that can inevitably worsen perf..
• Performance testing is an effective way to catch performance regression but it is costly.
• We propose performance risk analysis to prioritize performance testing target.
• Evaluation shows our tool is light-weight and effective in recommending performance-risky commits
• [http://cseweb.ucsd.edu/~peh003/perfscope](http://cseweb.ucsd.edu/~peh003/perfscope)
Thanks!

The authors are unable to attend the conference and do Q&A due to Visa issues 😞
If you have any questions, please reach Peng at ryanhuang@cs.ucsd.edu