Sto

A Better Way To Store And Query Profiler Data

Patrick Somaru
Production Engineer
What is profiler data?
Tying things back to code
Why use Profilers?

Profiler Data
Challenges such as?
Introducing **Sto**: by storing profiler data in a DAG, we can reduce it’s footprint by ~10000x. Think, 2gb => 5mb for suboptimal cases.
What We Found

Sto Primitives
What We Found

Primitive 1: Stack Node Data

/path/to/demo.c
01: void simulateWork() { ... }
02:
03: void doBusinessLogic() { simulateWork(); }
04:
05: void doLogging() { simulateWork(); }
06:
07: void applicationLogic() {
08:   if(... 50%) {
09:     doBusinessLogic();
10:     doLogging();
11:   } else {
12:     doBusinessLogic();
13:   }
14: }
15:
16: int main() { applicationLogic(); }

<table>
<thead>
<tr>
<th>id</th>
<th>file</th>
<th>symbol</th>
<th>line number</th>
</tr>
</thead>
<tbody>
<tr>
<td>324</td>
<td>/path/to/demo.c</td>
<td>simulateWork</td>
<td>01</td>
</tr>
<tr>
<td>47653</td>
<td>/path/to/demo.c</td>
<td>doBusinessLogic</td>
<td>03</td>
</tr>
<tr>
<td>2345</td>
<td>/path/to/demo.c</td>
<td>doLogging</td>
<td>05</td>
</tr>
<tr>
<td>56742</td>
<td>/path/to/demo.c</td>
<td>applicationLogic</td>
<td>07</td>
</tr>
<tr>
<td>1234</td>
<td>/path/to/demo.c</td>
<td>main</td>
<td>16</td>
</tr>
</tbody>
</table>
What We Found

Primitive 2: Executable

<table>
<thead>
<tr>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>version</td>
</tr>
<tr>
<td>samples</td>
</tr>
</tbody>
</table>

/path/to/demo

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>version</th>
<th>samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(name,version)</td>
<td>cli argument</td>
<td>cli argument</td>
<td>calculated by cli</td>
</tr>
</tbody>
</table>
**Primitive 3: Stack Node**

- **Executable ID**
- **Position in Stack (via parent id)**
- **Data ID**
- **Stack Node**
- **Sample Count**

### Table: `stack_node`

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int(pk)</td>
</tr>
<tr>
<td>parent_id</td>
<td>int</td>
</tr>
<tr>
<td>exe_id</td>
<td>int</td>
</tr>
<tr>
<td>data_id</td>
<td>int</td>
</tr>
<tr>
<td>samples</td>
<td>int</td>
</tr>
</tbody>
</table>
Conversion Process

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Filename</th>
<th>Line Number</th>
<th>Executable ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>_start</td>
<td>../sysdeps/x86_64/start.S</td>
<td>117</td>
<td>36413</td>
</tr>
<tr>
<td>_libc_start_main@GLIBC_2.25</td>
<td>../csu/libc-start.c</td>
<td>368</td>
<td>36413</td>
</tr>
<tr>
<td>__libc_start_call_main</td>
<td>../sysdeps/nptl/libc_start_call_main.h</td>
<td>58</td>
<td>36413</td>
</tr>
<tr>
<td>main</td>
<td>/path/to/demo.c</td>
<td>22</td>
<td>36413</td>
</tr>
<tr>
<td>applicationLogic</td>
<td>/path/to/demo.c</td>
<td>18</td>
<td>36413</td>
</tr>
<tr>
<td>doLogging</td>
<td>/path/to/demo.c</td>
<td>9</td>
<td>36413</td>
</tr>
<tr>
<td>simulateWork</td>
<td>/path/to/demo.c</td>
<td>5</td>
<td>36413</td>
</tr>
</tbody>
</table>
How we made this data queryable.

```sql
select subtree( rootid: stack_node.id)
from stack_node
inner join executable on stack_node.executable_id = executable.id
where executable.basename = 'demo'
  and executable.build_id = 'one'
  and stack_node.parent_id is null;
```

Low Cardinality Indices
- filename, symbol, line_no
- symbol, filename, line_no
- fulltext(filename, symbol, line_no)

High Cardinality Indices
- parent_id, data_id
- data_id
- exe_id
What We Shared

Recursive Queries
Without Recursive Difficulty
Let’s Find a Regression

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void simulateWork(int x) { for(int i = 10000*x; i>0; i--); }

void doBusinessLogic(int x) { simulateWork(x); }

void doLogging(int x) { simulateWork(x); }

void applicationLogic(int logPct) {
    if(rand() % 100 < logPct) {
        doBusinessLogic(1);
        doLogging(1);
    } else {
        doBusinessLogic(1);
    }
}

int main() {
    printf("%ld\n", (long)getpid());
    while(1){ applicationLogic(50); }
}
```

desktop /home/pat/repos/sto # target/release/cli --pid 15040 --binary demo --version two
What We Shared

Found it, Generically

```sql
select * from findRegressions(start_time: CURRENT_DATE-1);
```
## What We Shared

### What did that regression look like?

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Version</th>
<th>Samples</th>
<th>Raw Data Size</th>
<th>Sto Data Size</th>
<th>Storage Size Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2619496695096638500</td>
<td>demo</td>
<td>one</td>
<td>1228800</td>
<td>34.64 MB</td>
<td>95.21 KB</td>
<td><strong>373x</strong></td>
</tr>
</tbody>
</table>

```
simulateWork:demo.c:5
doBusinessLogic:demo.c:7
applicationLogic:demo.c:14
main:demo.c:22
__libc_start_call_main:libc_start_call_main.h:58
__libc_start_main@GLIBC_2.2.5:libc_start.c:368
_start:start.S:117
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Version</th>
<th>Samples</th>
<th>Raw Data Size</th>
<th>Sto Data Size</th>
<th>Storage Size Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>5600443421022226000</td>
<td>demo</td>
<td>two</td>
<td>1894400</td>
<td>53.37 MB</td>
<td>129.30 KB</td>
<td><strong>423x</strong></td>
</tr>
</tbody>
</table>

```
simulateWork:demo.c:5
doBusinessLogic:demo.c:9
applicationLogic:demo.c:18
main:demo.c:22
__libc_start_call_main:libc_start_call_main.h:58
__libc_start_main@GLIBC_2.2.5:libc_start.c:368
_start:start.S:117
```

demo
Call graphs are graphs. Sto makes them more easily storable and queryable.
Thank You!

All code in this talk is available to play with via one-click deploy at https://github.com/likewhatevs/sto
Please take a look!!!