Performance Profiling with EndoScope, an Acquisitional Software Monitoring Framework

Alvin Cheung
Sam Madden
MIT CSAIL
August 25, 2008
Collecting System Runtime Data

- Many uses
  - Real-time system monitoring
  - Detect security breaches
  - Dynamic recompilation

- However, collecting such information is often difficult
Example: Software Profiling

- Sampling / statistical profilers
  - Gprof, oprofile
  - Might not be accurate
  - Can only be used to collect certain types of statistics

- Augment source code / Binary instrumentation
  - ATOM, valgrind, dtrace
  - Tedious work
  - Create substantial overhead

- Want: a unifying infrastructure that can be used to collect and reason about program’s runtime data that is easy to use and introduces low overhead
Our Contributions

- Easy to use interface
  - Use declarative queries

- Uniform data model that represents all sorts of runtime data
  - Model them using the streaming data model

- Small footprint / overhead
  - Be acquisitional: queries drive what data is collected
    - you only pay to collect data you asked for
  - Decouple program running site and monitoring site
  - Use both sampling and instrumentation techniques
  - Query evaluation tricks
Outline

- Data Model
- Query Evaluation Techniques
- Experiments
- Conclusions
Program Runtime Data as Streams

- EndoScope provides a number of basic streams that represent data coming from the runtime environment:
  - function start (function name, time)
  - variable value (name, value, time)
  - cpu usage (% busy, % idle, time)
  - ...

- Users can define additional streams on top of basic streams

- Streams are defined into two categories:
  - Enumerable streams are those that have discrete values in time (e.g., function start stream)
  - Non-enumerable streams are those that have infinite values in time (e.g., CPU usage stream)
  - Non-enumerable streams need to be quantified before they can be used (e.g., in an iterator)
Operations on Streams

- Quantify
  - Sample non-enumerable streams at points in time
- Select
- Project
- Aggregate
- Window-based join
Conditions / Triggers

- Specify actions to be performed when certain event occurs

- Action examples:
  - Start monitor CPU / heap usages
  - Generate report to user
  - Update machine learning models
Query Examples

- when

  select  avg(f1.duration) > 5 sec and
          avg(f2.duration) > 5 sec
  from    function_duration f1, f2
  where   f1.function_name = "foo" and
          f2.function_name = "bar"

then

  sample cpu_load every 1 min

- select *
  from    function_start fs, cpu_load cl
  where   fs.function_name = "foo" and
          cl.busy > 70%
Outline

- Data Model
- Query Evaluation Techniques
- Experiments
- Conclusions
Remote Monitoring Site

Query Plan Optimizer

Stream Processing Engine

Code Instrumentor

Profiled Program

Program Execution Site

Query Plan Optimizer

Stream Processing Engine

Query Plan

Query Plan Optimizer

User

query results

query plan

tuples

collected data

instrumentation plan

instrumentation instructions

query

query plan

Architectural Diagram
Optimizing Query Execution

- Goal: introduce as little runtime overhead as possible while providing reasonable query execution performance

- Three levels of optimization
  - Execution site
  - Query plan
  - Stream implementation
Execution Site Selection

- Query plan can be executed on a program running site or remote monitoring site.

- Aspects to consider:
  - CPU bound vs. network bound
  - Amount of data needed to be sent
  - Number of monitoring sites

- System conditions change over time!
  - Change query plans adaptively (future work)
select *
from   function_start fs, cpu_load cl
where fs.function_name = "foo" and cl.busy > 70%

Join evaluation strategy 1:
- Monitor all “foo” call sites and cpu usage at all times

Join evaluation strategy 2:
- Instrument all “foo” call sites
- Every time when “foo” is called, sample cpu usage, check if > 70%

Join evaluation strategy 3:
- Do not instrument “foo”
- Continuously sample cpu usage
- If sampled usage is > 70%, then instrument “foo” call sites
Query Plan Optimization (2)

- Need cost model
- Simple cost model:
  \[
  \text{Extra instructions from data collecting operations} \times \text{Frequency of such operations}
  \]
- Challenge
  - Frequency estimates changes over program lifetime!
    - Change query plans adaptively (future work)
Optimizing Stream Implementation

- Implementing function start stream
  - Exact
    - Instrument all call sites
    - Use code analysis to reduce # functions to instrument
  - Approximate
    - Sample stack trace and check if function is called

- Need cost model, and understand how much approximation user can tolerate
Outline

- Data Model
- Query Evaluation Techniques
- Experiments
- Conclusions
Experiments Setup

- Implemented a simple profiler for Java programs on top of EndoScope
- Monitored performance of 3 apps
  - SimpleApp included with Apache Derby
  - TPC-C implementation using Derby
  - Petstore app hosted on Tomcat that uses Derby
- Measured runtime overhead
Runtime Overhead Experiment

- Rank all functions by their call frequencies over program run

- Issue query to system
  - Progressively increase the % of functions monitored, with the least frequently called function chosen first
  - Compare time overhead with other profilers
25-50% less overhead when compared to other profilers
Join Operator Ordering Expt

- Query on top of TPC-C implementation
  - SELECT *
    FROM function_start fs, cpu_load cl
    WHERE fs.function_name in (f1,f2..)
    AND cl.busy > 70%

- Quantify the effects of operators ordering by measuring the time overhead of 3 different query plans
Plan Comparison on TPC-C Single Threaded

- Unprofiled
- Plan 1: Monitor Functions and Sample CPU
- Plan 2: CPU Sampling Triggered by Function Monitoring
- Plan 3: Function Monitoring Triggered by CPU Sampling

Less overhead when # functions monitored is small

Less overhead with continuous CPU monitoring
Outline

- Data Model
- Query Evaluation Techniques
- Experiments
- Conclusions
Contributions

- Introduced a low overhead, query driven, acquisitional software monitoring framework
- Proposed data model, a declarative query language, and query evaluation techniques
- Implemented a simple profiler for Java programs and validated on real-world systems