Path-based Macroanalysis for Large Distributed Systems

Mike Chen[1], Anthony Accardi[2], Emre Kiciman[3]
Dave Patterson[1], Armando Fox[3], Eric Brewer[1]

mikechen@cs.berkeley.edu

UC Berkeley[1], Tellme Networks[2], Stanford University[3]

Motivation

- **Systems have many component boundaries**
  - Design principles: divide and conquer, layering, replication, indirection, virtualization, etc.
- **Problem: execution context is dispersed throughout**
  - Worse yet, these components are dynamic, distributed, heterogeneous, and evolving

Motivation

- **Need to think about how to understand *collections* of interacting components**
  - Existing low-level tools only help with debugging individual components
  - A widening gap between the systems we are building and the tools we have
    - printf() and 20 xterms won't cut it!

Macroanalysis

- **Exploits non-local context to improve reliability and performance**
  - Performance examples: Scout, ILP, Magpie
- **Macro vs. Micro?**
  - Emphasis is on component interaction rather than low-level details
  - Complements microanalysis tools such as app-level logs and gdb
- **Automated statistical analysis is possible**
  - Large number of concurrent, independent requests

Macroanalysis helps with…

- **Normal operation**
  - Profiling
  - Evolution
    - Deducing system structure
    - Versioning
- **Failure handling**
  - Failure detection
  - Failure diagnosis
  - Impact analysis

Path-based Macroanalysis (PMA)

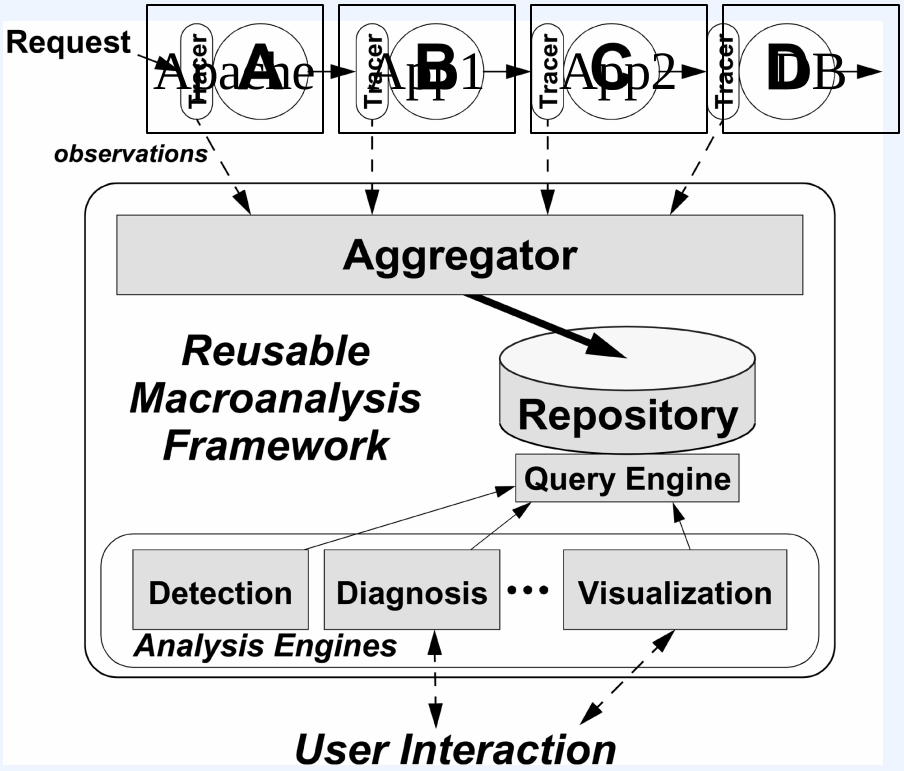
- **Model the target system as a collection of paths**
  - e.g. request/response paths, one-way message paths
- **Our philosophy**
  - Use only dynamic, observed behavior
  - Application-independent techniques when possible
- **Novel use of paths**
  - Traverse heterogeneous, distributed components
  - Discover paths based on runtime data => no a priori input
  - Assume no component knowledge other than entry/exit points

Observing Runtime Paths

- **Dynamically trace requests through a system at the component level**
  - Record call path + runtime properties
    - e.g. components, latency, success/failure, and resources used to service each request
  - Runtime analysis tells you how the system is *actually* being used, not how it *may* be used
- **Use statistical analysis detect and diagnose problems**

Architecture

- **Tracer**
  - Tags each request with a unique ID and carries it
  - Reports *observations*
- **Aggregator + Repository**
  - Reconstructs paths from observations
- **Query + Analysis Engines**
  - Supports statistical queries on paths
  - Mix of both online and offline analysis



2 Implementations

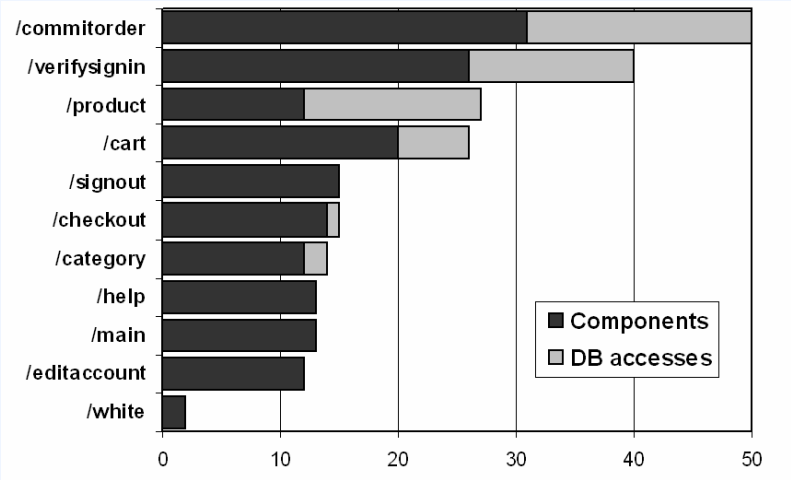
- **Pinpoint**
  - 3-tier J2EE system (httpd, J2EE app server, DB)
    - v1: Sun reference implementation
    - v2: JBoss, another open-source server
  - Several e-commerce apps and benchmarks
- **Tellme Networks (Observation Logs)**
  - An enterprise voice application service provider
    - Multi-tier with telephony frontend and web backend
  - Hosts thousands of VocieXML applications
  - Billions of production requests since end of 2001

Applying PMA

- **Normal operation**
  - Latency profiling
  - Evolution
  - Versioning
- **Failure handling**
  - Failure detection
  - Failure diagnosis
  - Impact analysis

Request-centric Profiling

- **Key idea: paths provide request-centric profiling (vs. aggregate)**
  - Path *length* shows complexity of each request
  - Associates user-perceived latency to internal components

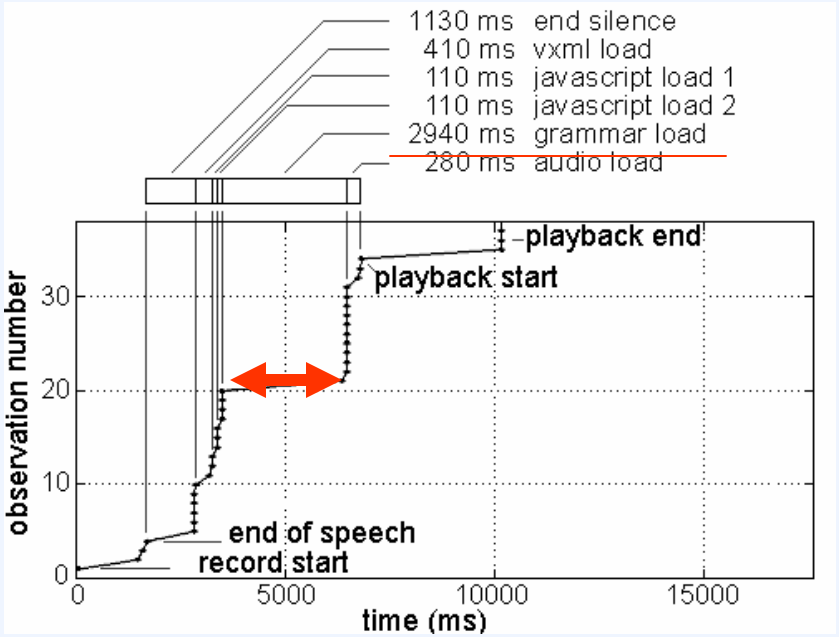


Path length example: # of components and DB accesses in PetStore

- Identifies inefficient DB accesses (30-40% improvement achieved after query optimization).

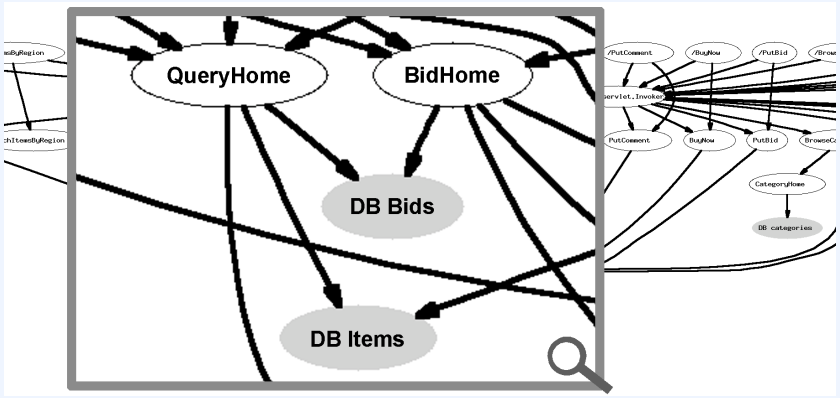
Latency Profiling

- **Why is request XYZ slow?**
  - Drill down to sub-paths to identify bottlenecks



Evolution: System Structure

- **Key idea: paths capture system structure**
- **Verifies manually tracked structure (e.g. UML) against *observed* structure**



- **Automatically derived application structure for RUBiS showing a subset of its 33 components**
- **Database tables are shown in gray**
- **The directed edges represent observed paths**

Evolution: Shared State

- **Key idea: paths associate requests with internal state**
  - Discover state sharing across requests by tracing SQL queries

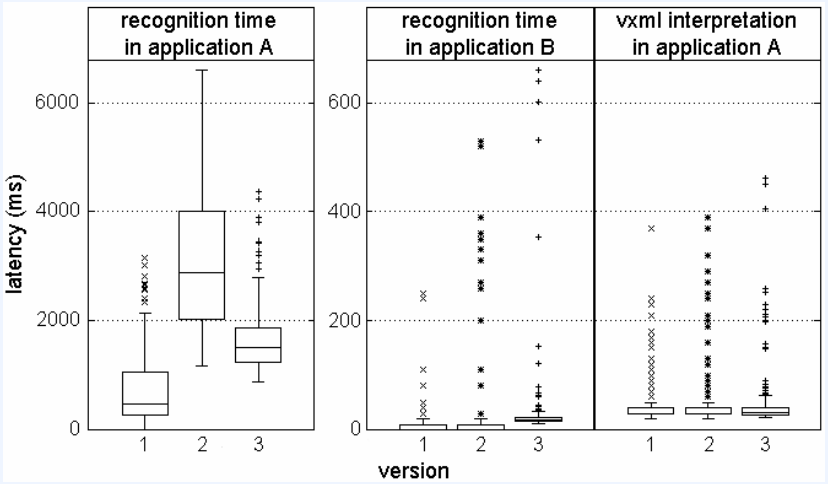
Request types

Database tables

	Product	Item	Signon	Account	Bannerdata	Profile	Sequence	Inventory
/verifysignin	R	R	R	R		R		
/cart	R	R			R			R/W
/product	R	R						
/commitorder	R	R					R/W	W
/category	R							
/language	R	R						
/search	R	R			R			
/productdetails	R	R						R/W
/main					R			
/validatenewaccount			R	R		R		
/checkout								W

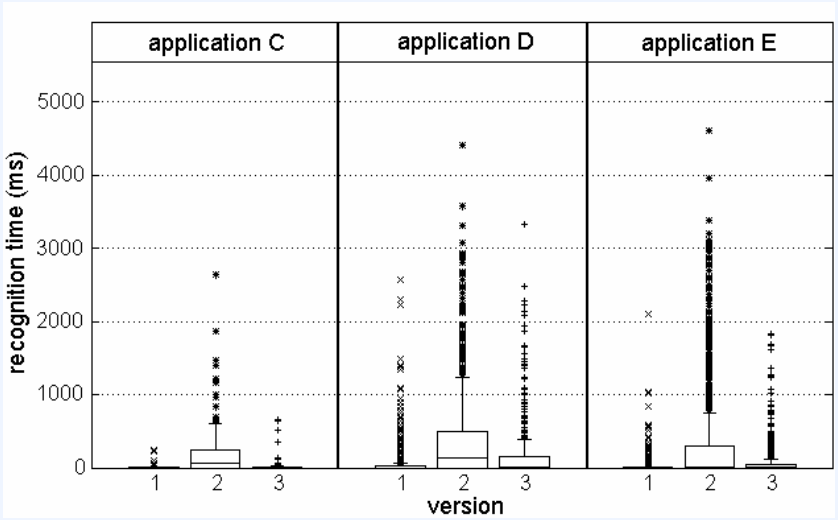
Versioning

- **Challenge: understanding impact of new code**
- **Key idea: paths + statistical analysis = simultaneous detection and identification of performance problems**
  - For both app and system problems
    - App problems are detectable through deviations in specific sub-paths



Versioning: System

- **Look for consistent deviation across all apps**
- **Example:**
  - Significant slower sub-path for system version 2 compared to version 1
  - Version 3 fixed the problem identified



Applying PMA

- **Normal operation**
  - Latency profiling
  - Evolution
  - Versioning
- **Failure handling**
  - Failure detection
  - Failure diagnosis
  - Impact analysis

Failure Detection

- **Key idea: some paths change under failures => detect failures via path changes.**
- **Structural anomaly**
  - Incomplete/interrupted paths
  - Regression
    - Build a model of good paths and look for deviations
- **Latency anomaly**
  - Calls that return without computation or timeouts
    - statistically determine the thresholds to use
- **Can't detect some failures such as arithmetic errors**

Failure Diagnosis

- **Key idea: most bad paths touch root cause(s). Find common features across bad paths.**
- **Single-path diagnosis**
  - Traps *live* failures and complements low-level tools
    - Paths connect the dispersed execution context
    - Often easier to wait for failures to happen than to reproduce them
- **Multi-path diagnosis**
  - Statistically correlate path features with failures
  - Pinpoint PetStore fault-injection experiment
    - 10-30% false negatives and 20-40% false positives
    - Automatic statistical analysis
    - Multi-component faults

Impact Analysis

- **Key idea: possible to compute % of requests that have failed (vs. uptime)**
  - Look for failure signatures
  - Important for billing and SLAs
- **Example:**
  - Audio server failure (0-second playback)
  - Query for paths that have such sub-path (playback < 10ms)

Ongoing Work

- **Key idea: violation of macro invariants are signs of buggy implementation or intrusion**
- **P2P message paths**
  - Challenge: distributed data collection and query
  - Oceanstore routing layer and other DHTs
- **Event-driven systems**
  - Challenge: matching incoming and outgoing events
  - SEDA events and OceanStore
- **Forks and joins**
  - Associate sub-paths using session/transaction ID

Conclusion

- **Macroanalysis provides a holistic view**
  - Useful for problems where local context is insufficient
- **Observe runtime paths and statistically infer macro properties**
  - Paths connect the dispersed execution context
  - Possibly transparent to apps for hosted apps
- **An analysis framework that is reusable across many systems**

Questions?

- **http://www.cs.berkeley.edu/~mikechen/**
  - HotOS paper and SOSP submission