Fluxo: Simple Service Compiler

Emre Kıcıman, Ben Livshits, Madanlal Musuvaththi
{emrek, livshits, madanm}@microsoft.com
Architecting Internet Services

• Difficult challenges and requirements
  – 24x7 availability
  – Over 1000 request/sec
    • CNN on election day: 276M page views
    • Akamai on election day: 12M req/sec
  – Manage many terabytes or petabytes of data
  – Latency requirements <100ms
Flickr: Photo Sharing

Cal Henderson, “Scalable Web Architectures: Common Patterns and Approaches,” Web 2.0 Expo NYC
Common Architectural Patterns

(In no particular order)

- **Tiering**: simplifies through separation
- **Partitioning**: aids scale-out
- **Replication**: redundancy and fail-over
- **Data duplication & de-normalization**: improve locality and perf for common-case queries
- **Queue or batch long-running tasks**
Everyone does it differently!

• Many caching schemes
  – Client-side, front-end, backend, step-aside, CDN
• Many partitioning techniques
  – Partition based on range, hash, lookup
• Data de-normalization and duplication
  – Secondary indices, materialized view, or multiple copies
• Tiering
  – 3-tier (presentation/app-logic/database)
  – 3-tier (app-layer / cache / db)
  – 2-tier (app-layer / db)
Different caching schemes!

Cal Henderson, “Scalable Web Architectures: Common Patterns and Approaches,” Web 2.0 Expo NYC
Cal Henderson, “Scalable Web Architectures: Common Patterns and Approaches,” Web 2.0 Expo NYC
Differences for good reason

• Choices depend on many things
  • Component performance and resource requirements
  • Workload distribution
  • Persistent data distribution
  • Read/write rates
  • Intermediate data sizes
  • Consistency requirements
Differences for good reason

- Choices depend on many things:
  - Component performance and resource requirements
  - Workload distribution
  - Persistent data distribution
  - Read/write rates
  - Intermediate data sizes
  - Consistency requirements

These are all measurable in real systems!
Differences for good reason

• Choices depend on many things
  • Component performance and resource requirements
  • Workload distribution
  • Persistent data distribution
  • Read/write rates
  • Intermediate data sizes
  • Consistency requirements

Except this one!

These are all measurable in real systems!
FLUXO

• Goal: Separate service’s logical programming from necessary architectural choices
  • E.g., Caching, partitioning, replication, ...

Techniques:
1. Restricted programming model
   • Coarse-grained dataflow with annotations

2. Runtime request tracing
   • Resource usage, performance and workload distributions

3. Analyze runtime behavior -> determine best choice
   • Simulations, numerical or queuing models, heuristics...
Architecture

Dataflow Program + Annotations

Environment Info

Runtime Profile

Analysis Module

Program Transform

Deployable Program

Thin Execution Layer

FLUXO Compiler
Dataflow Program

Restrictions
- All components are idempotent
- No internal state
- State update restrictions

UserID

CloudDB::Messages

CloudDB::Friends

List<Msg>

Merge message lists

html
What do We Annotate?

CloudDB::Messages
CloudDB::Friends

UserID

Volatile<0>
Volatile<5hr>

List<Msg>
List<>

Merge message lists

html

Annotate Semantics
- Consistency requirements
- (No strong consistency)
- Side-effects
What do We Measure?

CloudDB::Messages

CloudDB::Friends

UserID

List<Msg>

Merge message lists

html

List<UserID>

On every edge

• Data content/hash
• Data size
• Component performance and resource profiles
• Queue info
How do we transform? Caching

Messages Cache → CloudDB::Friends → Messages Cache

Pick First
How do we transform? Caching

Messages Cache

Messages Cache

Pick First
So, where do we put a cache?

1. **Analyze Dataflow:**
   Identify subgraphs with single input, single output

2. **Check Annotations:**
   Subgraphs should not contain nodes with side-effects; or volatile<0>

3. **Analyze measurements**
   Data size -> what fits in cache size?
   Content hash -> expected hit rate
   Subgraph perf -> expected benefit
Related Work

- **MapReduce/Dryad** – separates app from scalability/reliability architecture but only for batch
- **WaveScope** – uses dataflow and profiling for partitioning computation in sensor network
- **J2EE** – provides implementation of common patterns but developer still requires detailed knowledge
- **SEDA** – event driven system separates app from resource controllers
Conclusion

• **Q: Can we automate architectural decisions?**

• **Open Challenges:**
  – Ensuring correctness of transformations
  – Improving analysis techniques

• **Current Status:** In implementation
  – Experimenting with programming model restrictions and transformations

• If successful would enable easier development and improve agility
Extra Slides
Utility Computing Infrastructure

• On-demand compute and storage
  – Machines no longer bottleneck to scalability
• Spectrum of APIs and choices
  – Amazon EC2, Microsoft Azure, Google AppEngine
• Developer figures out how to use resources effectively
  – Though, AppEngine and Azure restrict programming model to reduce potential problems
Flickr: Photo Sharing

Cal Henderson, “Scalable Web Architectures: Common Patterns and Approaches,” Web 2.0 Expo NYC
Fault Model

• Best-effort execution layer provides machines
  – On failure, new machine is allocated
• Deployed program must have redundancy to work through failures
• Responsibility of Fluxo compiler
Storage Model

• Store data in an “external” store
  – S3, Azure, Sql Data Services
  – may be persistent, session, soft, etc.

• Data written as delta-update
  – Try to make reconciliation after partition easier

• Writes have deterministic ID for idempotency
• Built toy application: Weather service
  – Read-only service operating on volatile data

• Run application on workload traces from Popfly
  – Capture performance and intermediate workload distributions

• Built cache placement optimizer
  – Replays traces in simulator to test a cache placement
  – Simulated annealing to explore the space of choices
Caching choices vary by workload
Example #2: Pre/post compute

Diagram showing a process with nodes A, B, C, and D. The diagram includes a box labeled 'B C D', indicating preprocessed data, and an output labeled 'all inputs'. The flow of information is from A to B, B to C, C to all inputs, and A to X (preprocessed data).