G²: A Graph Processing System for Diagnosing Distributed Systems

Zhenyu GUO, Dong ZHOU, Haoxiang LIN, Mao YANG, Fan LONG, Chaoqiang DENG, Changshu LIU, Lidong ZHOU

System Research Group, MSR Asia
Cost of Debugging

- The huge printing presses for a major Chicago newspaper began malfunctioning ...

$10,000 = $1 + $9,999

Most bugs can be fixed quickly, however identifying the root causes is hard.
Motivation

• Diagnosing distributed systems is frustrating
  – Execution is too complex to comprehend
  – Tons of logs, but correlations are missing
  – Lost in the information sea

• We need a tool that
  – Finds correlated information.
  – Facilitates better summarization and reasoning
  – Is fast and easy to use
Contribution

• Graph based diagnosis for distributed systems
  – Execution graph to capture correlations
  – Graph based diagnosis operators
    • Slicing for finding & filtering
    • Hierarchical Aggregation for summarization

• Declarative diagnosis queries
  – Integrated with Microsoft LINQ

• Distributed engine
  – Integrated relational computation and graph traversal
  – Optimizations based on the characteristics of the execution graph and diagnosis operators
Outline

• Model
• Engine
• Programming
• Evaluation
Capture Correlations

![Diagram showing a master server and two replica servers connected to a disk and receiving requests from clients.](image-url)
Execution is Graph

Client 1

Client 2

Master

printf
req
issue
req
use
reqs
forward
reqs
forward
**Slicing**: Find the correlated subgraph and filter others by traversing the execution graph

```csharp
// Error log analysis
Events
  .Where(e => (e.Val.Type == EventType.LOG_ERROR) && e.Val.Payload.Contains("Write request failed"))
  .Slicing(Slice.Backward)
  .Select(e => Console.WriteLine(e.Val.Payload));
```
**Slicing**: Find the correlated subgraph and filter others by *traversing* the execution graph

```
Events
.Where(e => (e.Val.Type == EventType.LOG_INFORMATION)
  && e.Val.Payload.Contains("Start ClientRequest()"))
.Slicing(Slice.Forward)
.Select(e => Console.WriteLine(e.Val.Payload));
```
**HierarchicalAggregation**: Summarize details by traversing the execution graph

```csharp
// HierarchicalAggregation
Events
  .Where(e => e.Val.Location.Name == "SubmitWriteReq")
  .Slicing(Slice.Forward)
                           evts => evts.First().Val.Process.Machine.Name);
```

**Zoom In**

- **Network**
  - Message::DoExecution (12)
- **Replication**
  - ReplicateWrite (149)
  - WriteRequestFailed (24)
  - SerializedIOWrite (17)
- **I/O**
  - Time

**Time**

- **Machine 0**
  - Primary (440)
- **Machine 1**
  - Secondary 1 (144)
- **Machine 2**
  - Secondary 2 (202)
Understand Execution Graph

- **Execution graph is rather huge**
  - A 2-hour SCOPE/Dryad graph has over 1.2 billion vertices, 0.54 billion edges, and lots of user payload (logs)

- **Connected subgraph is also huge**
  - However, intra-machine interactions are much more than inter-machine ones (91% vs 9% in SCOPE/Dryad graph)

- **Graph structure data is relatively small**
  - User payload is over 64% in storage

- **Iterative access to graph structure data**
  - Concurrent traversals
  - Aggregation follows slicing
Optimize Graph Access

• Diagnosing tool as a distributed system
• Optimal partition on graph data
  – At machine boundary initially. Dynamic partitioning.
  – Local data is stored in database
• Caching
  – Graph structure data in memory
  – Retrieve payload only when necessary
• Prefetching
  – Get vertex properties during slicing, instead of during aggregation
Understand Slicing & Hierarchical Aggregation

• Latency is an issue
  – More than 200 hops sometimes, due to deep paths

• Rigorous synchronization is not efficient
  – Different from Page Rank/Belief Propagation

• Aggregation repeatedly colors local vertices with the same aggregation identity
  – Lots of local messages
Optimize Fast Execution Graph Traversal

- **Batched Asynchronous Graph Traversal**
  - Explore local vertices until reaching cross-partition edges without synchronization

- **Partition-level interface**
  - One traversal worker on each partition
  - Direct access to the whole local graph data
  - Local vertices could be condensed into super nodes in advance

![Slicing Speedup](chart1)

![Aggregation At Component Level](chart2)
Play with $G^2$

- Capture the graph
  - Manual annotation, Binary rewriter and dynamic instrumentation
- Write simple C# queries
  - Reuse existing relational operators in LINQ
  - Slicing(Chopping) / HierarchicalAggregation
  - Local Extensions: Diff, CriticalPath, ...
- Provide diagnosis wizards in Visual Studio
## Evaluation

<table>
<thead>
<tr>
<th>Systems</th>
<th>LOC(K)</th>
<th>Func#</th>
<th>Edge#</th>
<th>Event#</th>
<th>Raw(MB)</th>
<th>DB(MB)</th>
<th>Time(min)</th>
<th>Node#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley DB</td>
<td>172</td>
<td>46164</td>
<td>92502</td>
<td>186597</td>
<td>14</td>
<td>29</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>G²</td>
<td>27</td>
<td>267,728</td>
<td>634,704</td>
<td>1,212,778</td>
<td>85</td>
<td>231</td>
<td>17</td>
<td>60</td>
</tr>
<tr>
<td>SCOPE/Dryad</td>
<td>1,577</td>
<td>3,128,105</td>
<td>8,964,168</td>
<td>20,106,457</td>
<td>1,226</td>
<td>3,269</td>
<td>120</td>
<td>60</td>
</tr>
</tbody>
</table>

**Table 1: Per node graph statistics**

<table>
<thead>
<tr>
<th>Systems</th>
<th>Annotated Edge#</th>
<th>Annotated CS#</th>
<th>Instrumented Func#</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley DB</td>
<td>2</td>
<td>2</td>
<td>1,542</td>
<td>23</td>
</tr>
<tr>
<td>G²</td>
<td>9</td>
<td>11</td>
<td>197</td>
<td>10</td>
</tr>
<tr>
<td>SCOPE/Dryad</td>
<td>17</td>
<td>13</td>
<td>730</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 2: Instrumentation statistics**

- 60 machines
- 2 GHZ dual core
- 8 GB memory
- Two 1 TB disk
- 1 Gb Ethernet
5770 random queries on the SCOPE/Dryad Graph

Events.Where (e => ...)
  .Slicing(Slice.Forward)
  .HierarchicalAggregate(e => e.Val.Process.ID);
Related Work

• Execution Model
  – Path based analysis
  – Pure log analysis
  – Static analysis

• Distributed Execution Engine and Storage
  – Graph systems
  – Map-reduce alike systems

• Diagnosis Platform
  – Cloud9: Testing as a service
  – Dapper: path analysis atop of BigTable
Conclusion

• Graph based diagnosis for distributed systems
  – Slicing for filtering the logs
  – HierarchicalAggregation for summarization

• Graph engine with specific requirements
  – Integrated relational computation and graph traversal support
  – Batched asynchronous graph traversal and partition-level interface for better performance
Thanks!
Generations for log structure and related tools:

**Text**

- **Unstructured text**
  - Format: `[AUTO: time, component, log level, pid, tid, location], `printf’ message
  - Aggregation: by the meta information or keywords in the unstructured message

- **Pros**
  - Free style format
  - Easy to process: grep

- **Cons**
  - May miss many implicit dependencies among log entries without shared tag (e.g., request id)
Generations for log structure and related tools:

Paths

• Path-based aggregation
  – Format:
    • [ANNOTATION: path id] + unstructured text
    • Optional: [ANNOTATION] dependencies among log entries belonging to the same path are captured
  – Aggregation: by the user request id (path id)

• Pros
  – Effective for request-centric analysis and modeling
  – The logs are partitioned by request id, and each partition can usually be handled by single machine
  – A nice balance between usability and the overhead

• Cons
  – Cut off interactions between requests, which is common in distributed systems, such as batching
  – Path is statically defined by the pre-defined `requests’ only
Client 1

Client 2

Master
Scaling Performance

![Graphs showing scaling performance](image)

- **Worker Count** graph: Decreasing average latency with increasing worker count.
- **Concurrent Query Count** graph: Comparison of small and large query counts showing different latencies.
Graph Traversal Interface

```csharp
IQueryable<T> GraphTraversal<TWorker>(
    this Graph<TV, TE> g,
    IQueryable<Vertex<TV, TE>> startVertices
) where TWorker : GPartitionWorker<TV, TE, _, T>;

class GPartitionWorker<TV, TE, TMsg, T>
{
    public Vertex<TV, TE> GetLocalVertex(ID VertexID);
    public void SendMessage(ID VertexID, TMsg msg);
    public void WriteOutput(T val);
    public virtual void Initialize(VertexIterator<TV, TE>);
    public virtual void OnMessage(Vertex<TV, TE>, TMsg msg);
    public virtual void Finalize();
}
```
class GPartitionSlicingWorker<TV, TE> : GPartitionWorker<TV, TE, bool, Vertex<TV, TE>>
{
    private HashSet<ID> VisitedVertices;

    public override void Initialize(VertexIterator<TV, TE> inits)
    {
        foreach (var v in inits)
        {
            SendMessage(v.ID, true);
        }
    }

    public override void OnMessage(Vertex<TV, TE> v, bool msg)
    {
        if (VisitedVertices.Contains(v.ID)) return;
        VisitedVertices.Add(v.ID);
        WriteOutput(v);
        foreach (var e in v.OutEdges)
        {
            if (e.IsCausal())
                SendMessage(e.DstVertexID, true);
        }
    }
}
Experience using $G^2$
Deployment Issues

• Capture the correlations
  – Instrument the network and thread pool libraries to capture the asynchronous transitions among threads and machines

• Store and process the logs
  – Option 1: dedicated graph engine ($G^2$)
    • Pros: complete support of $G^2$ diagnosis queries
    • Cons: interference to host systems
  – Option 2: in-app graph engine with latest logs
    • Pros: lightweight, easy to deploy
    • Cons: limited memory cache capacity (latest logs only)