G^2: A Graph Processing System for Diagnosing Distributed Systems

Zhenyu Guo[†] Dong Zhou[‡] Haoxiang Lin[†] Mao Yang[†]
Fan Long[‡] Chaoqiang Deng[§] Changshu Liu[†] Lidong Zhou[†]
[†]*Microsoft Research Asia* [‡]*Tsinghua University* [§]*Harbin Institute of Technology*

ABSTRACT

G^2 is a graph processing system for diagnosing distributed systems. It works on execution graphs that model runtime events and their correlations in distributed systems. In G^2, a diagnosis process involves a series of queries, expressed in a high-level declarative language that supports both relational and graph-based operators. Each query is compiled into a distributed execution. G^2's execution engine supports both parallel relational data processing and iterative graph traversal.

Execution graphs in G^2 tend to have long paths and are in structure distinctly different from other large-scale graphs, such as social or web graphs. Tailored for execution graphs and graph traversal operations on those graphs, G^2's graph engine distinguishes itself by embracing *batched asynchronous iterations* that allows for better parallelism without barriers, and by enabling partition-level states and aggregation.

We have applied G^2 to diagnosis of distributed systems such as Berkeley DB, SCOPE/Dryad, and G^2 itself to validate its effectiveness. When co-deployed on a 60-machine cluster, G^2's execution engine can handle execution graphs with millions of vertices and edges; for instance, using a query in G2, we traverse, filter, and summarize a 130 million-vertex graph into a 12 thousand-vertex graph within 268 seconds on 60 machines. The use of an asynchronous model and a partition-level interface delivered a 66% reduction in response time when applied to queries in our diagnosis tasks.

1 INTRODUCTION

Distributed applications in data centers are increasingly important as they power large-scale web and cloud services. Often, the execution of such an application involves a large number of cooperating processes running on different machines, spanning multiple software modules and layers, tolerating and recovering from various machine failures and network disruptions. Increases in both the scale and complexity of such systems have made it difficult to understand and diagnose their runtime (mis-)behavior.

Typical diagnosis tasks start with observing misbehavior or anomaly, navigating through runtime information such as logs to find relevant information, and processing the information to infer root causes. For example, starting with a log entry with an error message, diagnosis could find all relevant log entries to infer the root cause for the error. As another example, given two similar jobs that noticeably perform differently, diagnosis could extract related runtime information to identify major differences. Also, it might be difficult to spot problems from a large number of low-level runtime events. A useful practice is to aggregate performance information at an appropriate layer, identify which aggregated component in that layer is problematic, and then drill down into the next layer of details in an iterative process.

Effective diagnosis depends heavily on the ability to correlate runtime events and to leverage these correlations. Previous work, especially those on path-based analysis [14, 7, 13, 8, 19, 26, 18, 27], has largely addressed the important problem of generating and correlating runtime information from executions of a distributed system. Often the difficulty for diagnosis is not due to lack of information, but due to the inability to navigate through and process a sea of information to find out what is relevant.

In this paper, we propose G^2, a distributed graph processing system for storing runtime information of distributed systems and for processing queries on such information. Runtime information is organized as a graph, where vertices correspond to events and edges correspond to correlations between events. Diagnosis then involves an iterative process of writing queries against the graph and analyzing the results of those queries. G^2 provides a declarative language that supports relational and graph operators that operate on the graph structure. For example, given an error log entry *e*, a G^2 query can be issued to find all events (vertices) that vertex *e* is causally dependent on, where causal dependencies are captured by certain types of edges. This query uses a *slicing* operator that G^2 provides. From a starting vertex *v*, *forward slicing* finds all vertices that causally and transitively depend on *v*, while *backward slicing* finds all vertices that *v* is dependent on.

Graph aggregation and summarization are another effective way of reducing the amount of information to be examined during diagnosis. In an execution graph, each vertex is associated with a context that indicates the aggregation units that the event belongs to. Examples of aggregation units include static ones such as components, classes, and functions, as well as dynamic ones such as machines, processes, and threads. A G^2 query can aggregate information at an appropriate level. For example, to compare executions of two jobs, a query can compute the forward slices from the starting points of two jobs. To make comparison easier, the query can continue to compute a machine-level aggregation from the two slices. This requires a *hierarchical aggregation* graph operator that transforms an input graph into a smaller one: it condenses each continuous segment of events with the same aggregation unit (e.g., machine) to create a single super-node and applies an aggregation function on those events to compute the an associated aggregated value.

Distributed query execution in G^2 is supported by a distributed storage and execution system that addresses the challenges of storing and processing large execution graphs with millions or even billions of vertices efficiently. In G^2, events and correlations are captured on local machines as they occur during system execution, leading to a natural partitioning of an execution graph.

G^2's execution engine is tailored for execution graphs that exhibit significantly different characteristics from other large graphs, such as social and web graphs. Execution graphs tend to have long paths corresponding to events along a logically related progression of execution, where social and web graphs have relatively small diameters. Graph operations on execution graphs are often in the form of graph traversal, which is again different from iterative graph operations that must proceed in globally synchronized rounds, such as in page-rank computation for example. Consequently, G^2 embraces *batched asynchronous iterations*, where processing on each partition is batched, but does not have to proceed synchronously in lock steps. Both slicing and hierarchical aggregation fall into this model that allows for improved parallelism and efficiency than the bulk synchronous computation model in previous work, such as in Pregel [24]. Barriers are used only at the end of graph traversal or to create global consistent checkpoints for failure recovery. Furthermore, partitions tend to contain long *local* paths before those paths connect to vertices on other partitions due to cross-machine communication. Graph traversal within each partition is therefore significant to the overall graph traversal performance. Instead of a vertex-oriented interface, G^2 exposes a partition-oriented interface that allows partition-level aggregation states to be maintained in an appropriate data structure. This is particularly valuable for hierarchical aggregation, where the choice of partition-level data structure significantly influences performance.

We have built a prototype and applied it to a set of distributed systems, including Berkeley DB [2], SCOPE/Dryad [11, 22], and G^2 itself. Berkeley DB is a replicated distributed key-value database that can be easily linked with applications. SCOPE/Dryad is a production data intensive computation system, which includes a distributed file system, a distributed execution engine (Dryad), and a declarative query language (SCOPE). G^2 is shown to be effective in diagnosis: for instance, using a query in G2, we traverse, filter, and summarize a 130 million-vertex graph into a 12 thousand-vertex graph within 268 seconds on 60 machines. The optimizations we introduce into G^2's execution engine are effective: the use of asynchronous model and partition-level interface delivered up to a factor of 3 performance improvement when applied to graph operators in our diagnosis tasks. We have also studied scalability of G^2 and the checkpointing overhead introduced to enable failure recovery.

The contribution of G^2 is two-fold. First, as a tool, G^2 enables efficient distributed-system diagnosis by allowing users to write declarative queries with both relational and graph operators, and by providing a distributed engine that executes those queries efficiently. Second, as a distributed system, G^2's execution engine targets a different type of graphs with different structural characteristics and with different type of graph operations. It allows a batched asynchronous graph computation model and a partition-level interface, which have contributed significantly to its efficiency.

The rest of the paper is organized as follows. Section 2 introduces the system execution graph data model, and the diagnosis primitives applied to the graph. Section 3 presents the operators, and the language that G^2 supports, as well as several examples expressed in those constructs. The design and optimization of the distributed graph engine is the focus of Section 4, followed by implementation details in Section 5. We evaluate G^2 and share experience in Section 6. Section 7 discusses the related work. Finally, we conclude in Section 8.

2 MODEL

Distributed-system diagnosis in G^2 centers on the data model and the operations defined on the model, which are the topic of this section.

2.1 Text, Paths, and Graphs

Traditionally, system diagnosis treats runtime information (e.g., logs) as *unstructured* text and involves a tedious and ineffective process of going through logs using primitive text-processing tools such as grep. Using grep on a special tag (such as a request id) captures all entries that are explicitly related to that request, but is likely to miss information that has implicit dependencies.

Previous work [14, 7, 13, 8, 20, 18, 27] on correlating runtime information has effectively addressed this shortcoming by capturing common causal relationship in distributed systems. A *path*-like abstraction is often used to track how a request flows through a distributed system. This relatively simple structure is effective for request-centric analysis and modeling, and reflects a good balance between what an abstraction enables, the simplicity of an abstraction, and the complexity involved in supporting operations on an abstraction.

Yet, the effectiveness of a path-based model is constrained by its simplifying assumptions: by embracing paths based on requests, the model cuts off interactions between requests that occur in distributed systems. For example, Figure 1 shows a piece of code for a replicated file system. The system receives client requests and appends them in a local cache (line 2-4). When there are enough accumulated requests (line 6), the system batches requests, writes them to local disks (line 11), and forwards them to secondaries for data replication (line 12). The *OnPersistRequests* call in Figure 1 is in fact a batch operation of multiple write requests from clients to the distributed file system. In such a case, it is difficult to assign a path id to the events inside the call (e.g., event h can only share with the path id from e or f, but not both). Two paths might also be correlated when they access the same shared variables. In fact, a more general graph is already used to some extent in previous work such as Pip [26].

G^2 instead explores a different point in the design space. Rather than constraining users to a path-based model a priori, G^2 preserves and presents the full structure captured during the execution of a distributed system as a graph. During diagnosis, users can choose to construct paths from such a graph if paths are appropriate for the diagnosis task at hand, or they can choose to process information in a different way that is more appropriate for that particular task. G^2 does not make that decision for users during the modeling phase. This design choice effectively shifts the burden to the underlying distributed engine, as it must enable efficient operations on a more complicated graph structure.

2.2 Execution Graph

G^2's execution graph model embraces two key concepts: *causality* and *aggregation*. This is based on our observation of common system diagnosis practices: users tend to (i) follow cause-effect relations to find relevant information and (ii) to summarize runtime information at an appropriate aggregation level in a hierarchy in order to find trouble spots for further in-depth analysis.

In an execution graph of G^2, each runtime event from a target system is represented as a *vertex*. In Figure 1, events are shown in small rectangles; examples are *printf* log event b (line 3), asynchronous request define events c and d, and request use event e. A context is associated with an event, indicating the aggregation units that the event belongs to. Multiple levels of aggregation units can be defined. Examples include static constructs, such as modules, classes, and functions, as well as runtime constructs, such as machines, processes, and threads.

Runtime events are correlated, where directed edges in an execution graph are used to represent such correlations. Different types of edges can be defined for different types of correlations. For example, an *use* edge connects a source event that defines/forwards an object with a destination event that consumes that object. Network messages or cross-thread requests are examples of such objects. $\langle c,e \rangle$ and $\langle d,e \rangle$ in Figure 1 are use edges. A *sync* edge indicates synchronization of two events from two different threads in order to ensure exclusive access to a shared object or ensure ordered inter-thread execution. A *fall-through* edge connects two consecutive events in the same thread (e.g., $\langle b,c \rangle$).

G^2 provides primitives to define and customize graph traversal for diagnosis. Two are built-in: *Slicing* finds all causally related events in a graph and *HierarchicalAggregate* summarizes information at an appropriate aggregation level.

2.3 Filter with *Slicing*

Instead of simply "grepping" runtime information with a special tag, *Slicing* filters information using graph structure: it starts from a *root event* and transitively collects *causally dependent* events. Forward and backward traversal yield a *forward slice* and a *backward slice*, respectively.

Computing precise and complete causal dependencies for slicing is usually too costly if not infeasible, where a reasonable approximation is often sufficient in practice. A naive way is to consider all *use* and *fall-through* edges as *causal edges*. Our practical experience has shown that fall-through edges often do not imply causal relations. For example, in a typical implementation of message processing subsystem, a thread will continuously accept new incoming messages and call corresponding message handlers. Fall-through edges between two message handler invocations do not represent any meaningful causal dependencies. Such false causal dependencies could render slicing ineffective. All events in the corresponding message handler should however be considered causally dependent on the message-send event. G^2 introduces *causal scope* to specify, for each *use* edge, the set of events that are causally dependent on the source event of that edge. A causal scope consists of a continuous region from the destination event of each *use* edge: all events within that region are causally dependent on the source event; all fall-though edges within that region are considered causal edges. In Figure 1, large rectangle boxes define causal scopes. The shaded area outlines the forward slice from event a.

2.4 Summarize with *HierarchicalAggregate*

Aggregation is another effective way of managing a large amount of data, especially with a hierarchy. There are natural hierarchies in distributed systems: a program is

```
class ClientReq {
void OnRecieveClientRequest(...) {
    Log(LOG_INFO, "...");
    IssuePersistRequest(...); }
int PersistMainThread() {
    while (IsEnoughRequest(reqs))
        OnPersistRequests(reqs);
        …}
int OnPersistRequests(list<ClientReq*> reqs) {
    MemBuf* buf = CreateBuf(reqs);
    WriteToLocalDisk(buf);
    ForwardToSecondary(buf, ...); }
```

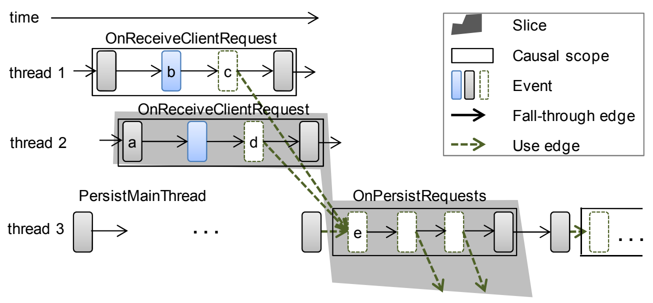

Figure 1: System execution graph, causal scope, and slice.

often made of modules, each module is comprised of classes, and each class contains a set of functions. A distributed-system execution can be aggregated at thread level, then at process level, and further at machine level. A distributed system often consists of multiple logical layers that are application-specific: for example, a system behavior can be analyzed at an RPC layer or at a lower OS layer with a socket interface.

G^2 supports an important notion called *hierarchical aggregation*. The key idea is to construct a condensed graph at an appropriate layer of a hierarchy to summarize system behavior. A continuous segment of events with the same aggregation unit in an execution graph is summarized and condensed into a single higher-level vertex in the resulting graph. G^2 by default attaches signatures of code and runtime location to all events for aggregation. Aggregation in G^2 is customizable: a user can leverage her domain knowledge to specify how to aggregate events and summarize high-level information (e.g., aggregated performance counters) from low-level events.

Figure 2 shows an example of event aggregation when debugging replication in the distributed storage for SCOPE. Numbers inside rectangles are total event-counts within corresponding vertices in the aggregated graphs. An error occurs during a replicated write operation. The upper part of Figure 2 performs event aggregation at machine level: it clearly shows whether the write operation was propagated to all replicas. Once a suspected machine is identified, a user selects that machine and zooms in to see how the write request was processed by each component in this machine, shown in the lower part of Figure 2.

3 Programming in G^2

Programming in G^2 consists of two parts. One is to "program" distributed systems so as to to make them diagnosable by G^2. We defer this to Section 5. The other is for "programming" queries to be executed on G^2, which is the focus of this section.

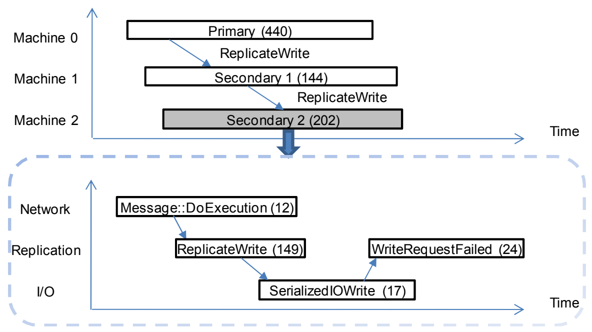

Figure 2: Hierarchical aggregation for a replication implementation. Numbers in rectangles show numbers of events within vertices in the aggregated graphs.

```
1  Graph<TV, TE> Slicing (
2   this Vertex<TV, TE>                       srcVertex,
3   Slice.Type                                type);
4
5  Graph<THighV, THighE>
6  HierarchicalAggregate (
7   this Graph<TLowV, TLowE>                    g,
8   Func<Vertex<TLowV, TLowE>, __out UInt64> labelCb,
9   Func<VertexIterator<TLowV, TLowE>,
10                     __out THighV>   AggreFunc);
```

Figure 3: Graph operators.

3.1 Graph Operators

Figure 3 shows the basic graph operators. Each *Vertex* contains its incoming and outgoing edge lists, and it is a generic type that can be instantiated with $\langle TV, TE \rangle$. Type *TV* describes the data associated with the vertex, such as logs, code locations, and runtime locations, while type *TE* describes the data associated with each edge, such as timestamps for the source and the destination events. A generic *Graph* type can be further defined as a collection of vertices.

As shown in the figure, operator *Slicing* takes *srcVertex* as the root event and *type* as the direction (forward or backward) for slicing. Operator *HierarchicalAggregate* condenses a graph to a higher-level graph (line 5). Given a vertex in the original graph,

```
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));
```

(a) Error log analysis.

```
var req = Events
.Where(e => e.Val.Location.Name=="SubmitWriteReq");
req.Slicing(Slice.Forward)
.HierarchicalAggregate(
   e => e.Val.Process.Machine.Signature,
   evts => evts.First().Val.Process.Machine.Name)
.CriticalPath(req,dst,e=>{e.Val.SrcTs, e.Val.DstTs});
```

(b) Machine level critical path analysis.

```
var s1 = Events.Where(t => t.VertexID == 1)
.Slicing(Slice.Forward)
.HierarchicalAggregate(...aggregate by component...);
var s2 = Events.Where(t => t.VertexID == 2)
.Slicing(Slice.Forward)
.HierarchicalAggregate(...aggregate by component...);
s1.Diff(s2, e => {e.Val.SrcTs, e.Val.DstTs});
```

(c) Component level performance regression analysis.

Figure 4: Sample diagnosis queries.

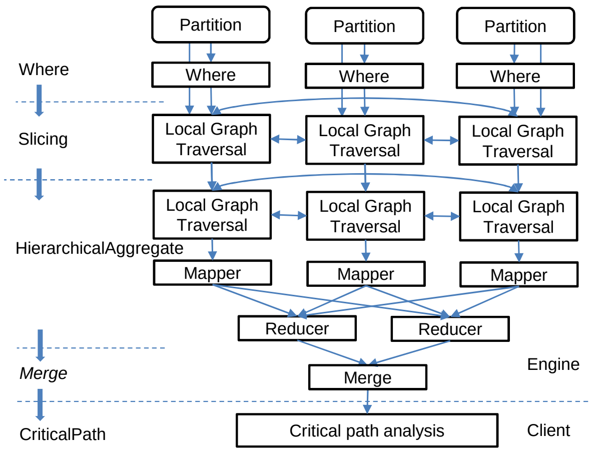

Figure 5: Data flow for the machine-level critical-path analysis query in Figure 4 (b).

the *labelCb* callback returns its aggregation-unit label. The *AggreFunc* callback aggregates a continuous sequence of vertices with the same label (line 9) into a new vertex at the high-level graph (line 10) (Note the structure is determined by G^2, and the associated value(*THighV*) is defined by the callback).

All those operators are built on top of two distributed primitives: *GraphTraversal* and *MapReduce*. *GraphTraversal* starts with a set of vertices in a graph and traverses the graph by following edges forward, backward, or bi-directionally. A user can customize graph traversal by deriving a graph traversal class, which defines computations on vertices, messages passed along edges, as well as final output during graph traversal. *MapReduce* is standard with a map function and a reduce function for aggregation. Details of these distributed primitives and how they are used to build graph operators are left to Section 4.

3.2 Composing Graph Operators

Extensibility and composability are two key features of G^2 design for programming. *Slicing* and *HierarchicalAggregate* both consume a graph and produce another, so they can be composed. We further leverage the extensibility of the LINQ framework [3] in .Net, so that developers can write diagnosis queries using our new operators, LINQ's relational operators, and even customized local analysis modules, such as finding critical path (*CriticalPath*) and comparing two aggregated graphs (*Diff*). Figure 4 shows a set of examples; all from real diagnosis practice.

The first query returns the logs in a backward slice rooted from an error log event. The query first uses *Where* in LINQ to locate the error event and then invokes *Slicing*. The second query aims to find straggler machines during processing of a request. The query first calculates the forward slice from the point of request submission, aggregates the slice into a machine-level graph via *HierarchicalAggregate*, and computes the critical path for request processing. Each vertex in the returned critical path summarizes a continuous execution on a machine with the start and stop times of the execution, from which stragglers can be easily identified. The last query intends to find components responsible for an instance of slower-than-normal request processing. It extracts forward slices rooted from the slow request and normal ones, aggregates at the component level, and outputs differences. If needed, users could drill down into problematic components and investigate further at the function level or lower.

4 Distributed Engine

A distributed engine is responsible for transforming diagnosis queries into distributed jobs to be executed on the set of machines storing the execution graphs.

4.1 Overview

In G^2, events and correlations between events are captured and recorded locally, and transformed into appropriate graph representations. G^2 therefore naturally partitions original system execution graphs based on where events occur. Such a partitioning method tends to exhibit good locality as distributed systems are usually designed to minimize cross-machine traffic.

A *job manager* initiates a *job* when a query is submitted. Each machine storing graph partitions runs a *daemon*. The job manager coordinates executions of phases by communicating with these daemons. A job involves multiple phases that can be represented as a data flow graph. Figure 5 shows the data flow graph for the machine-level critical-path analysis query in Figure 4 (b). It consists of 5 phases: Where, Slicing, HierarchicalAggregate, Merge, and CriticalPath. The distributed engine takes care of the first 4 phases and sends the aggregated results to clients for local critical-path analysis. The Merge phase does not appear in the original query and is added automatically during compilation. Both Slicing and HierarchicalAggregate involve graph traversal, where the latter consumes the graph created by the former and outputs an aggregated graph for client analysis. In particular, the mappers during HierarchicalAggregate shuffle the vertices according to which high level vertex they belong to, and the reducers aggregate the vertices inside one high level vertex using the *AggreFunc* callback provided by the queries.

The part of the data flow graph without graph traversal is similar to directed acyclic graphs (DAG) in previous data-parallel computation engines, such as Map/Reduce and Dryad. Graph traversal however requires a different type of coordination to support loops and barriers. G^2's graph traversal support distinguishes itself from previous graph engines (e.g., Pregel [24]) in several noticeable ways. First, for operations such as slicing and hierarchical aggregation, G^2 supports batched asynchronous iterations, where partitions batch operations locally, but do not have to be synchronized using a barrier in each iteration. Second, G^2 exposes a partition-level interface, rather than a vertex-level interface, to allow better batching and aggregation for graph computation. This is particularly important for enabling efficient implementation of hierarchical aggregation. These optimizations can be applied not only to G^2 but also to other distributed graph traversal problems such as shortest path computation.

4.2 Batched Asynchronous Iterations

A typical graph engine implements *synchronous iterations* through loops and barriers. For graph computation such as page-rank computation and belief propagation, all participants must synchronize with each other in each iteration via a barrier, and in each iteration the participants can only traverse one hop. Such synchronization is easily done with the help of a job manager.

In G^2, we observe that graph traversal for slicing and hierarchical aggregation is inherently *asynchronous*. Take forward slicing for example, each partition has a set of vertices to start with in each iteration (except the first one where only one partition has the root vertex). For one local iteration, a partition starts graph exploration from those vertices following causal edges until it reaches cross-partition edges without synchronization with others after every one hop traversal. When this iteration ends, a partition reports to the job manager with pointers to lists of vertices to other partitions for further exploration. The job manager will notify other partitions of the availability of these lists. A partition finishing the current iteration can fetch the lists of new vertices from other partitions and start the next iteration. It does not have to wait to get lists from all other partitions before initiating the next iteration.

G^2 does support global barriers for two cases. In the first case, completion of a graph-traversal stage is through a global barrier: all participants must have completed their last iteration locally. The job manager initiates the next phase of computation only after that global barrier is established. In the second case, the job manager can periodically introduce a barrier to an ongoing graph-traversal stage for failure recovery; the barrier is used to perform a globally consistent snapshot.

```
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> {
 Vertex<TV, TE> GetLocalVertex(ID VertexID);
 void SendMessage(ID VertexID, TMsg msg);
 void WriteOutput(T val);
 virtual void Initialize(VertexIterator<TV, TE>)=0;
 virtual void OnMessage(Vertex<TV, TE>, TMsg) = 0;
 virtual void Finalize() = 0;
};
```

(a) *GraphTraversal* interface.

```
class GPartitionSlicingWorker<TV, TE>
: GPartitionWorker<TV, TE, bool, Vertex<TV, TE>> {
 HashSet<ID> VisitedVertices;
 void Initialize(VertexIterator<TV, TE> inits) {
   foreach (var v in inits)
     SendMessage(v.ID, true);
 }
 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.OutEdgeIterator)
     if (e.IsCausal())
       SendMessage(e.DstVertexID, true);
 }
 void Finalize() {}
}
```

(b) *GPartitionSlicingWorker* for forward slicing.

Figure 6: *GraphTraversal* interface and example

4.3 Partition vs. Vertex

G^2 provides a GraphTraversal interface so that users can implement their own custom graph-traversal algorithms. Previous graph processing systems such as Pregel allow users to specify actions on each vertex, which is natural for a large number of graph computation algorithms. However, we have found a partition-level interface offers additional opportunities for better performance.

Graph Traversal Interface. Figure 6 (a) shows the signature of *GraphTraversal*. It starts from a set of initial vertices (line 3), with traversal polices designated by *TWorker* derived from *GPartitionWorker* (line 4). When a graph traversal phase starts, G^2 creates an instance of *GPartitionWorker* on every graph partition of *g* (line 2), and the job manager coordinates the workers to perform multiple iterations of computation: a first round for *Initialize* (line 9), followed by multiple rounds of graph traversal via message exchanges among vertices (line 10) until all workers reach the completion barrier, and a last round for *Finalize* (line 11). In each round, a worker creates remote messages for other partitions. Those remote messages are eventually transported to appropriate partitions and serve as the input for next-round computation on those partitions.

Forward Slicing. Figure 6 (b) shows a sample that implements forward slicing. During *Initialize*, the worker sends a message to the initial vertices of the graph traversal via *SendMessage* (lines 5,6). After initialization, each worker invokes *OnMessage* (line 8) on each message, inside which a worker can read/write partition-local states (lines 9,10), produces partial outputs via *WriteOutput* (line 11), and send messages to other vertices via *SendMessage* (line 14) by following the edges of the current vertex (line 12). *OnMessage* does *not* cause a real network message to be sent: for a local destination, the worker again applies *OnMessage* on the destination vertex in the current round. Only messages destined to a remote partition are gathered and made available to other partitions at the end of this iteration. After a worker completes the current round, it fetches the available remote messages for its partition from other partitions, and starts a new round. This process ends when all workers have completed the current rounds with no new remote messages. In the final round, the workers invoke *Finalize* (line 16), which usually produces outputs of this traversal phase from final local states. It is empty in this case because output is generated during traversal (line 11).

Hierarchical Aggregation. The value of exposing a partition-oriented interface is more evident in the implementation of *HierarchicalAggregate*.

Figure 7 (a) shows a simple vertex-oriented implementation from a vertex's perspective. Each vertex has a label based on its context; for example, the label is its process id for process-level aggregation. We use *AggId* to identify a set of vertices that have already been aggregated together: those vertices will have the same value *v.AggId*. Every vertex uses its own *ID* as the initial *AggId* (line 3), and broadcasts both its label and *AggId* to its neighbors (lines 4,5). A message is ignored when a receiving vertex has a different label (line 8), indicating a boundary of aggregation. Otherwise, if an incoming *AggId* is smaller than the current one, a vertex changes its own *AggId* and propagates the change to its neighbors (lines 9-12). The traversal ends when all vertices are assigned the smallest label of the vertices to be aggregated together.

```
1  void Initialize(VertexIterator inits) {
2    foreach (Vertex v in inits) {
3      v.AggId = v.ID;
4      foreach (Vertex iv in neighbour vertices)
5        SendMessage(iv, {v.ID, v.Label});
6    } ...
7  void OnMessage(Vertex v, MSG msg) {
8    if (msg.Label != v.Label) return;
9    if (msg.AggId < v.AggId) {
10     v.AggId = msg.AggId;
11     foreach (var e in connected edges)
12       SendMessage(e.DstVertexID, msg);
13   } ...
```

(a) Vertex oriented implementation.

```
1  Map<ID, ID>   VertexLeader;  // vertex->leader
2  Map<ID, ID>   LeaderAggIds;  // leader -> aggId
3  Map<ID, ID[]> RemoteVertexGroup;//leader->rvertices
4  void Initialize(VertexIterator inits) {
5    ... local aggregation to initialize the maps ...
6    ... send messages to remote vertices ...
7  }
8  void OnMessage(Vertex v, MSG msg) {
9    if (msg.Label != v.Label) return;
10   ID leaderId = VertexLeader[v.ID];
11   int oldAggId = LeaderAggIds[leaderId];
12   if (msg.AggId < oldAggId) {
13     ... update aggId for the group ...
14     foreach (var vid in RemoteVertexGroup[leaderId])
15       SendMessage(vid, msg);
16   } ...
```

(b) Partition oriented implementation.

Figure 7: Optimization for *HierarchicalAggregate*.

Figure 7 (b) shows a partition-oriented implementation, where partition-level aggregated states are maintained (lines 1-3) and initialized (lines 5-6). These data structures essentially aggregate local continuous segments with the same labels and update them as a single unit during traversal, rather than going through each vertex repeatedly: each local continuous segment with the same label is assigned a leader. Rather than having each vertex maintaining an *aggId*, the partition maintains a mapping from leaders to *aggId*s in *LeaderAggIds.* Because vertices with the same leader always have the same *aggId*, a partition can simply update one entry in *LeaderAggIds* for all those vertices when the *aggId* changes for any of the vertices. Similarly, destination vertices of cross-partition edges from this segment of vertices are recorded in *RemoteVertexGroup* and can be identified without following the edges within this segment repeatedly. Those data structures are populated during initialization. When a message arrives at a partition with a boundary vertex as the destination vertex, the worker checks its label (line 9), and updates the *AggId* of the corresponding leader vertex if it receives a smaller *AggId* (lines 10-13). Finally, it broadcasts the new *AggId*

Component Name	Language	LOC(K)
Annotation Library	C++, C	3.4
Binary Rewriter	C++/CLI	1.6
Transformer	C++	1.5
Engine(JobMgr,Daemon,MetaSvr)	C++	27.5
FrontEnd(Compiler, JobClient)	C#	17.3
VS AddIn(Wizards, UI)	C#	57.8
Total	-	109.1

Table 1: Components in G^2.

to its cross-partition neighbors (lines 14-15).

4.4 Failure Handling

G^2 supports iterative graph computation, which renders inapplicable the map/reduce type of failure recovery using re-computation. In a synchronous graph computation model, where a global barrier is established at each round, a globally consistent checkpoint can be taken at each barrier. When failures happen, computation can be rolled back to the most recent checkpoint. G^2 allows each partition to maintain states. A checkpoint therefore covers such per-partition state, as well as the remote messages that each partition generates at the end of a round for other partitions. Appropriate levels of redundancies might be needed for checkpoints in order to recover from permanent machine failures.

With an asynchronous graph computation model, G^2 can choose to insert a global barrier at an appropriate interval for consistent checkpointing. We can also resort to the standard Chandy-Lamport algorithm for taking a consistent snapshot, where a global barrier is a special simple (yet less efficient) implementation for this algorithm. In the worst case, G^2 can always roll back to re-execute a graph-traversal phase (assuming that failures do not lead to data loss in the original graph information). Our current implementation uses global barriers for consistent checkpointing, but do not replicate the checkpoint to tolerate permanent failures.

5 IMPLEMENTATION

G^2 provides a complete tool set to help developers diagnose systems. Table 1 shows the programming language and lines of code for components of G^2: annotation library and binary rewriter are used to capture system execution graphs. Transformer is used to store a graph. Engine, Front-End, and Visual Studio AddIn are for processing and visualizing a graph.

Capture Graph. G^2 can use existing traces from previous work, e.g., those on path-based analysis, to build execution graphs. It also provides its own tool chain for developers to instrument target systems for gathering information of interest. Whether instrumentation requires manual code change depends on the types of edges to be captured. We have developed a Phoenix [5] based binary rewriter tool to annotate *synchronous use* edges (i.e., call) and their corresponding causal scopes (i.e., the call boundary) automatically. A user can choose what to instrument with a configuration file, reflecting her choice to balance between cost and coverage. G^2 also captures *sync* edges automatically at the Win32 layer by instrumenting Windows synchronization APIs. For *asynchronous use* edges, G^2 provides an annotation library; the following code illustrates how to track network messages and their corresponding handlers using this library. Users first annotate a context(*NetworkMsg*) by making it inherit a *G2::CausalCtx* object. Users then add a call to *LogUseEdgeBegin* and *LogCausalScope* respectively, when this context is about to be delivered and used. The macro call of *LogCausalScope* is a C++ object, whose lifetime defines a causal scope.

```
class NetworkMsg : public G2::CausalCtx {
  int Send(...) {
    LogUseEdgeBegin(this, ...);
    ...
  } ...
void OnRecvNetworkMessage (NetworkMsg& msg, ...) {
  LogCausalScope(msg);
  msg.Execute(...);
  ... }
```

Store Graph. We developed a *Transformer* that converts raw runtime-event streams to database tables. G^2 stores a system execution graph in four relational tables. Each object has a unique key, which we use as reference keys across tables and partitions. The *CodeLocation* provides the context for each event and in particular the component, class, function, file, and line number of the statement that generates the event. The *ProcessInfo* table covers the runtime process information, such as process id, machine name, process start time, and so on. The *Event* table contains information about each event, including its type, a reference to an edge if it is an endpoint of that edge, a physical timestamp, references to *CodeLocation* and *ProcessInfo*, and payload (e.g., *printf* log content). The *Edge* table contains the edge type, a unique edge ID, and references to the source and destination events. The *Edge* table defines the *structure plane* of a system execution graph, while the other three form the *data plane*. A slicing operation can be done purely on the *Edge* table, but for event aggregation it is often necessary to query the *CodeLocation* and *ProcessInfo* tables. The *Edge* table is frequently accessed during graph traversal and is therefore cached in memory for fast access.

Process Graph. Queries submitted to G^2 are compiled into a distributed query plan, with appropriate resource files dispatched to workers running on machines managing partitions of a graph. Execution of a query plan is done through coordination between the job manager and the workers, as described in Section 4. The job manager

Systems	Acs#	Ace#	Func#	Rule#
G^2	9	11	197	10
SCOPE/Dryad	17	13	730	5
BerkeleyDB	2	2	1,542	23

Table 2: Instrumentation statistics. *Acs#*, *Ace#*, *Func#*, and *Rule#* refer to the number of manually annotated causal scopes, manually annotated edges, instrumented functions, and rules in the configuration files for the binary rewriter, respectively.

monitors progress of graph traversal and assists in message exchanges between workers. After a round of local processing ends on a partition *A*, the worker for *A* groups messages based on their destinations and notifies the job manager of the list of partitions with data from *A*. The job manager piggybacks the list partitions with data ready for *A*. The worker for *A* will then fetch those from the corresponding workers. To make message exchange efficient, workers cache generated message groups in memory and discard them after they are fetched. The job manager is also responsible for enforcing global barriers upon completion of graph traversal, as well as to create consistent checkpoints.

6 EXPERIMENTS AND EXPERIENCE

We have applied G^2 to SCOPE/Dryad, G^2 itself, and BerkeleyDB. Our evaluation attempts to answer the following questions: a) what is the cost of applying G^2? b) how does the G^2 engine perform on real execution graphs? c) does G^2 help developers diagnose complicated distributed system problems?

Target systems are co-deployed with G^2 on a cluster of 60 machines; each has a dual 2GHz Intel Xeon CPU, 8 GB memory, two 1TB SATA disks, and are connected with 1 Gb Ethernet.

6.1 Cost of Applying G^2

Human effort. Table 2 reports the statistics about the annotation effort to apply G^2 on these systems. For instrumenting functions, users write only a configuration file for the binary rewriter to specify names of functions they are interested in. For all three cases, the configuration files are less than 25 lines, as shown in Table 2.

The *asynchronous use* edges and correspondent causal scopes require manual annotation on source code. Our experiences show that most asynchronous messages and events are handled by a small number of components or by a middleware library, making annotation easy. We annotated fewer than 20 places for each benchmark. Annotations on G^2 took us less than one hour. Interestingly, in our SCOPE/Dryad experiment, we did forget to annotate a place where the code directly uses the *CreateProcess* function to create a new process, bypassing the middleware component that we annotated. Such cases are rare and can often be discovered during a diagnosis process. Developers can optionally capture other dependencies: we do not model those.

Runtime overhead. Runtime overhead for emitting events and edges is comparable to those in previous work on capturing causal dependencies [14, 7, 13, 8, 19, 26, 18, 27]. There are several categories of events/edges. The first category are asynchronous use edges and the corresponding causal-scope events (e.g., message send/receive), which are always captured. The second are legacy *printf* logs. These two parts do not introduce noticeable system slowdown compared to previous systems (with the same *printf* logs). The third are events from instrumented functions, and the cost is proportional to the numbers of function invocations that are captured. Usually invocations of interface functions of each component that are associated with error and failure handling are enough for diagnosis. For our experiments on the three distributed systems, only less than 0.1% of overall function invocations are captured in system execution graphs, and no noticeable overhead was observed. If more functions need to be instrumented, and we cannot afford to instrument all, we can turn to dynamic instrumentation techniques as done in our previous work [23].

Table 3 reports the statistics on sample execution graphs from our target systems. The G^2 data include events from executing tens of diagnosis queries against a SCOPE/Dryad snapshot. The SCOPE/Dryad data came from ten SCOPE queries for calculating different statistics of web data. The BerkeleyDB data was collected during approximately one hundred instances of system initialization guided by a model checker; the goal is to use G^2 to assist model checking research in another project. All numbers reported are per-machine averages. For example, for SCOPE/Dryad, a 120-minute trace generates about 1.2GB of G^2 data on each of the 60 machines. On average, the imposed I/O bandwidth ranges from 85.3 KB/s (G^2) to 174 KB/s (SCOPE/Dryad) on average, which did not cause noticeable runtime interference to the host systems. Not reported in this figure, in our sample SCOPE/Dryad execution graph, about 28% of the events recorded are legacy logs (category 2), which account for 64% of total sizes. Category 1 accounts for 33% by count and 16% by size, while category 3 takes the rest.

6.2 Performance Evaluation

This section evaluates the performance of G^2 engine.

Graph statistics. Table 3 shows the numbers of edges and vertices for the sample execution graphs from our target systems. The number of edges is far fewer than the number of events because all fall-through edges are implicit. The func# is the number of invocations for instru-

Systems	LOC(K)	Func#	Edge#	Event#	Raw(MB)	DB(MB)	Time(min)	node#
`G`^{2}	27	267,728	634,704	1,212,778	85	231	17	60
`SCOPE/Dryad`	1,577	3,128,105	8,964,168	20,106,457	1,226	3,269	120	60
`BerkeleyDB`	172	46,164	92,502	186,597	14	29	2	3

Table 3: Execution graph statistics about a snapshot for the target systems.

mented functions as specified in Table 2. System execution graphs can be large: the SCOPE/Dryad snapshot has on average more than 20 million events on each machine. The database size (DB) is approximately three times the raw event stream size (Raw), due to verbose DB data format (factor of 1.5) and associated indices. The *edge* table (including its indices) counts for only 30% of the total database size. Because it is frequently accessed, caching it in memory makes sense.

End to end performance. We evaluate the end to end performance of G^2 with 5770 random queries on the SCOPE/Dryad graph. Each query calculates a forward slice from a randomly selected root event and then computes a process-level aggregation on the slice. Figure 8 shows the overall running time of these queries with all optimizations turned-on. The G^2 engine is generally fast on these queries: 94.5% queries finished within 5 seconds, with only three queries (in the upper-right circle) taking more than 100 seconds. Our investigation shows that the randomly chosen root events for those three queries are close to the entry point of the Dryad job, yielding huge slices with up to 130 millions of events. Running time depends not only on the sizes of resulting slices, but also on properties of the graph (and its partitions), those properties dictating concurrency of query processing. For example, some queries (in the circle on the left) take more than 50 seconds, even though the corresponding slices are relatively small. This is because they experience a period of time with low concurrency.

We also inspected the result of hierarchical aggregation. The result shows that it is effective in simplifying graphs. All resulting process-level graphs contain less than 85 vertices, except the three queries in the upper right circle: the aggregation yields graphs with only 0.01% of vertices.

Graph and graph computation characteristics. We recorded and examined a large forward-slice computation on the SCOPE/Dryad graph. Figure 9 shows how the numbers of events, local edges, and remote edges vary over time. The SCOPE/Dryad job has a bootstrap phase, during which a job scheduler copies resource files between machines. In the actual execution, this phase takes little time. However, because this phase involves a series of communication, slicing at this segment of the execution graph has little concurrency. It takes a relatively long time to process even though the number of events and the total I/O are small. This is reflected in the flat start in the figure. Graph traversal experiences respectable concurrency in the middle range when traversing the portion of the graph for the real Dryad execution. Then after around time 00:40 it starts to process the portion of the graph corresponding to the final phase, in which a job manager again has to talk to many machines to fetch statistics and to write them into a distributed file. The overall concurrency level on 60 machines is 7.23.

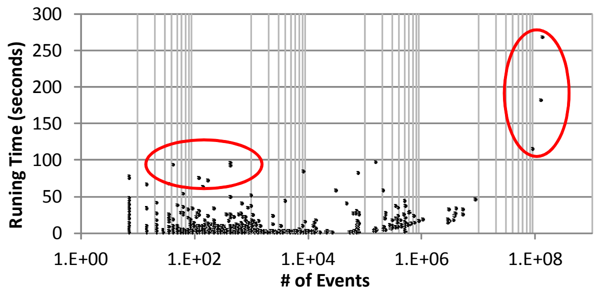

Figure 8: Process level aggregation performance.

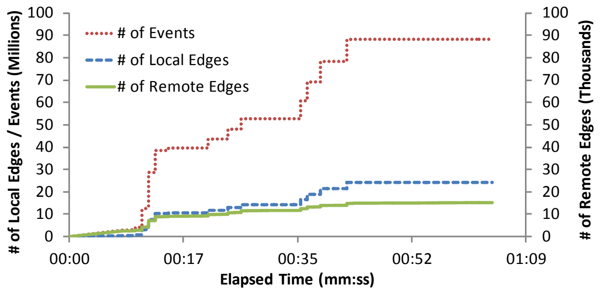

Figure 9: How the # of events, local edges, and remote edges vary along the execution time.

Effectiveness of graph engine optimizations. To evaluate the effectiveness of batched asynchronous iteration and partition-oriented interface design, we measured both *Slicing* and *HierarchicalAggregate* performance with nine different configurations, which are the combinations of two dimensions of configurations. The first is whether to enable Barriers among workers and Checkpointing after each round (*None*, *B*, or *B/C*). The second is to choose which local graph traversal policy (*OneHop*, *Batched*, or *Partition*). *OneHop* is to allow one hop traversal only in each round, a typical setting for other graph engines such as Pregel; *Batched* is to tra-

Phase	OneHop	Batched	Partition
Slicing(None)	1.49	0.99	1.00
Slicing(B)	2.70	1.19	1.21
Slicing(B/C)	2.86	1.27	1.27
Aggregation(None)	1.80	1.48	1.00
Aggregation(B)	2.67	1.91	1.01
Aggregation(B/C)	3.40	2.42	1.09

Table 4: Relative execution time for the component level aggregation analysis with nine different configurations. Abbreviations: B - barriers are enabled among workers; B/C - barriers are enabled among workers, and partition state is checkpointed.

verse until no further local vertices to be visited during this round; and *Partition* is similar to the second, but with the partition-state optimization discussed in Section 4.3, enabled by G^2's partition-oriented interface. Table 4 shows the average relative execution time for the component level aggregation analysis for a Dryad job with the nine configurations; each runs ten times. To be fair, with $\langle OneHop, B/C \rangle$, we checkpoint every 7 and 18 rounds for slicing and aggregation, respectively, so that they take approximately the same number of checkpoints as in other configurations.

Overall, batched asynchronous iterations and partition-oriented interface are effective: without checkpointing, we see a 62-63% reduction in latency for both slicing (2.70 vs. 1) and hierarchical aggregation (2.67 vs. 1). The data also reveal the following: (i) Batched asynchronous iterations bring benefits in two ways: First, it allows local traversal to proceed (as in *Batched*) and significantly reduces the number of global rounds (from 208 rounds to 28 rounds for Slicing and from 111 rounds to 6 rounds for hierarchical aggregation). Second, it removes the need for global barriers. This is particularly effective when there are many rounds and significant variations across machines in each round. It is noticeably ineffective for Aggregation ($\langle Partition, B \rangle$, $\langle Partition, None \rangle$) because our partition-oriented optimization makes process-time variations between partitions negligible. (ii) Partition-oriented interface and data structures are effective for Aggregation (with 32% reduction) because we are seeing large local islands (e.g., one island with 7.7 million internal edges and only 2,895 remote edges): those local islands do not have to be visited repeatedly with our optimization. (iii) Overhead of checkpointing depends on how frequent we checkpoint and how much data we checkpoint. OneHop introduces lower overhead (5.11 MB/s) because it checkpoints the same amount of state in a longer time period compared to Batched (7.18 MB/s), and Batched has higher overhead compared to Partition because its state size is larger than that under Partition (5.96 MB/s).

Scaling performance. We evaluated scaling performance from two perspectives. The first is to measure scaling in terms of the number of machines. We do not show figures due to space constraints. We observe that the job latency decreases almost linearly initially when more machines are used. But after we have more than 16 machines, the speedup slows down due to inherent limit on concurrency. With 60 machines, the average latency is reduced to under 3 minutes from over 14 minutes on 8 machines.

The second is to measure scaling in terms of the number of concurrent queries. We use two different sets of slices for those queries. The first set has several large slices, which involves 5 to 8 graph partitions and contains approximately 0.4 to 1 million events. The second set has a set of randomly selected slices, which typically involves 1 to 2 machines and contains several thousands of events. For large slices, latency increases dramatically when we reach 100 queries. For small slices, the system can support almost 500 queries simultaneously without affecting query latencies and can handle 5,000 queries with an average latency under 3 minutes.

6.3 Experience, Limitations and Future Work

To make G^2 accessible to developers, we have built a set of templates to guide the use of the system and integrated the tool into Visual Studio for a seamless debugging experience. The *Visual Studio AddIn* includes a set of common diagnosis tools based on G^2, including event navigation along edges in all levels of graphs (called *gwalker*), as well as a set of wizards focusing on specific visualized diagnosis tasks, such as error log analysis, critical path analysis, and performance regression analysis, as discussed in Section 3.2. Following are two showcases from our experiences.

Slower job. When developing G^2, we found that query processing time became 60 times slower after minor code changes. To investigate, we applied G^2 for a machine-level aggregation with critical-path analysis on a forward slice of a query job. The result showed that most of the processing time was spent on machine *srgsi*-10. We then zoomed into a thread-level execution graph, and performed a graph diff with its counterpart from the same run in the previous day (before code changes were applied). Figure 10 depicts the diff result, which shows that thread 8772 has the largest deviation between the two jobs in terms of execution time. We further zoomed into the component level and then function level execution graphs in that thread, and found that the largest deviation happened in the *AcquireRowById* function which reads a row from a local table using the primary key. Our further investigation revealed that the table did not have a proper index, the result of a bug we introduced in Transformer that caused index creation to fail. This kind of perfor-

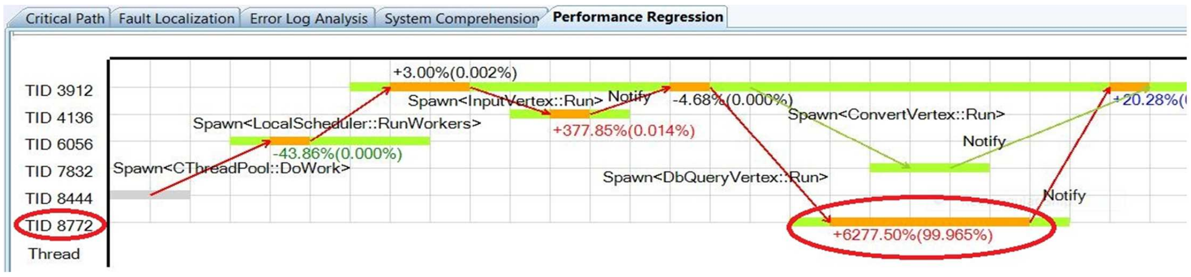

Figure 10: Thread level performance regression diff.

mance regression problem is common in our experiences and a hierarchical diff analysis between two similar tasks is often effective in identifying root causes.

Failed write. We have used G^2 to investigate root causes of error logs in SCOPE/Dryad. One case is shown in Figure 11. The error log in the underlying distributed storage system reported that a write request to a chunk server *c1* at time 21:27 failed. Using G^2, we managed to find its root cause through several steps, marked in Figure 11.

1. We computed a backward slice starting from the error log entry and listed the warning and error entries ordered by time (*OrderBy*). A warning log entry showed up in the disk IO module, indicating that a chunk *x* was marked as deleted. However, we were not able to figure out why this happened based on the information in this backward slice.

2. We wrote a *Where* query for the most recent logs on the same machine who contain keyword "chunk *x*", trying to connect the missing edges which may tell why chunk *x* was marked as deleted. The query returned an event *B* indicating that at time 21:17 a background thread marked this chunk as deleted.

3. We use *gwalker* to navigate the logs on the graph from event *B*, and found an event *C* indicating that the meta server *m1* sent a delete request to *c1* because it found that chunk *x* no longer belonged to any file stream.

4. To locate the origin of the write request to *c1* on presumably deleted chunk *x*, we aggregated the backward slice rooted at event *A* at process level and found the write request came from *c3*, and propagated by *c2*, where c1, c2, and c3 formed a replication group for *x*. A further drill-down of the logs on *c3* showed that *c3* restarted at 21:27. During its replication log replay, it found an incomplete local chunk *x* and issued an empty write request to sync data from other replicas (*c1* and *c2*).

5. Trying to understand why *c3* did not receive the delete request from meta server for chunk *x*, we ran a process level aggregation on the forward slice from event *C*, and found that *m1* sent a delete request to *c3* at time 21:25, but *c3* was not online at that time. This revealed the root cause.

This interactive diagnosis process involved gwalker, slicing (at a specific layer), aggregation, and relational queries in G^2, and is guided with human expert knowledge. It is worth pointing out that G^2, as any diagnosis tool, is not intended to replace human completely. Rather, its value lies in its ability to allow users to find the right information efficiently.

Implicit dependencies. The backward slice in step 1 of the Failed Write diagnosis did not contain all the interesting events for us to find the root cause, due to implicit dependencies (through chunk *x*), which are not captured by G^2. This is a common limitation in causality-based approaches. We managed to connect the dots through a relational query in this case. In our performance diagnosis experience, we also found a lot of problems caused by resource contention or interference, and again such causal relations are not modeled by G^2. In the future, we plan to incorporate some interference analysis techniques (e.g., [25]) to introduce *interference edges* into our model.

Customized slicing computation. We found some slices were fairly large and took a long time to compute. In many cases, users do not need all the information in a slice. To better control the cost, G^2 provides three additional parameters to the *Slicing* operator: maximum slice radius (hops starting from the root event), maximum network hops, and a customizable edge filter which decides whether the computation should continue following this edge for a bigger slice. Our experience shows these parameters can greatly improve the productivity, especially when people are familiar with their target systems.

Deployment and interference. The data placement has three reasonable choices as we see: data on each machine, on one dedicated machine per pod (in which the machines share a same uplink), and on a single machine. We did not run the second option (in a real data center) yet, and our scalability study above touched on diagnosis performance vs. number of machines. Our belief is that the second option is more suitable for a real deployment to minimize interference, while the other two can be used in testing environments depending on the size of the setup.

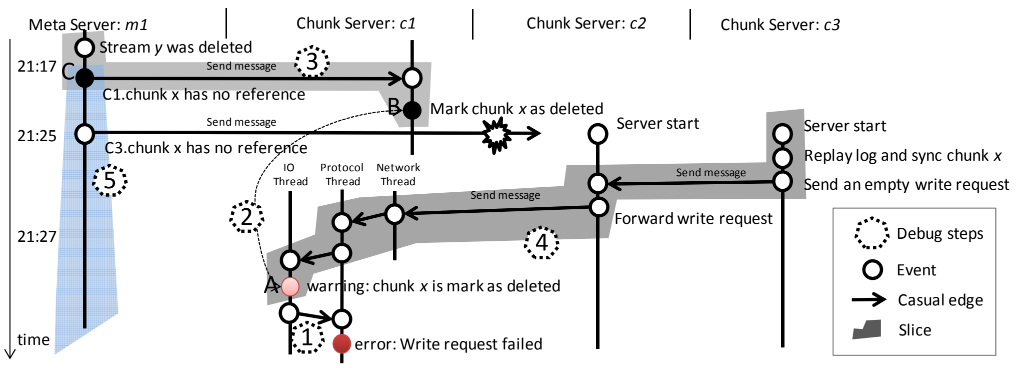

Figure 11: Diagnosis process for the failed-write scenario.

7 Related Work

The design of G^2 draw inspirations from a great number of previous works. We discuss those in three categories: execution modeling, distributed execution engine/storage, and diagnosis platform.

Execution Modeling. G^2 captures overall system behavior in a system execution graph with the help of annotation and instrumentation. It is clearly related to path-based analysis [14, 13, 26, 7, 8, 27], where a path is often defined as a sequence of events that is triggered by a client request. Path instances can either be collected through annotation [14, 13, 26] and schemas [8] provided by developers, or statistically inferred from inter-machine communications [7]. A variety of analysis can be enabled on path instances. For example, Magpie [8] aims to analyze workload models from path instances; PinPoint [14, 13] uses statistical methods to find components that are highly correlated to failed requests; Pip [26] checks these instances against specifications of expected system behavior defined by users. Path instances correspond to forward slices from the points where client requests are submitted in a system execution graph—G^2's model is general in that slicing can be in both directions from any point. Technically, path-based techniques [14, 13, 26, 7, 8, 19] could be applied on these forward slices and therefore integrated into G^2. We plan to investigate this feasibility in the future. X-Trace [19, 18] is similar to G^2 as it captures system behavior as task trees, and it tends to store the information in a service like OpenDHT to allow further distributed processing.

A large body of work focuses on diagnosing distributed systems using purely legacy logs. For example, Wei et al. [29] use machine learning to mine console logs to detect large-scale system problems, and SherLog [31] uses a constraint solver with information from a log to rebuild system execution flow that produces the same log. While no annotation or schema input from users are needed in those systems, there is usually a trade off: a machine-learning approach [29] involves sufficient art to ensure accuracy, while recovering information using constraint solving [31] faces scalability challenges.

Our slicing concept is inspired by program slicing [6] in program analysis. Hierarchical aggregation is related to hierarchical dynamic slicing [28], although the underlying techniques are different. Program slicing captures fine-grain data/control dependencies among variables/statements, and semantic hierarchy inside programs, while G^2 captures dependencies at a coarse granularity and resorts to approximation for scalability.

Distributed Execution Engine and Storage. G^2 hinges on its graph traversal engine to operate on huge execution graphs, often composed of millions even billions of vertices. Pregel [24] is a system for general large-scale graph processing. Tailored for execution graphs and graph traversal, G^2 adopts a batched asynchronous model rather than a bulk synchronous model in Pregel; it exposes a partition-oriented interface, rather than a vertex-oriented one in Pregel.

Other distributed computing engines have also been applied to specific computation on large graphs. Distributed execution engines such as MapReduce [16] and Dryad/DryadLINQ [22, 30] have been applied to compute PageRank [4] on a web graph. Recently, MapReduce Online [15] is also used for interactive big data analysis. G^2 also leverages *MapReduce* as the basic construction primitive to implement the diagnosis operators. Besides, it employs dedicated graph traversal primitive to reduce the query latency. G^2 partitions a graph into partitions and stores partitions on different machines. Distributed storage systems, such as key/value stores or table-based stores, have been studies extensively, although not for storing large graphs in particular. Recent examples include Cassandra [1], Dynamo [17], and BigTable [12]. G^2 adopts the similar approach, and it co-locates the execution to the partitions so as to reduce the storage access latency.

Diagnosis Platform. Several previous diagnosis tools have also leveraged the power of distributed systems.

Cloud9 [10] has pioneered the concept of *testing as a service*. In particular, it shows a symbolic execution testing engine that can be parallelized in a cloud. We share the same vision and believe G^2 can enable diagnosis as a service. Wei et al. [29] parallelized their algorithm for learning legacy logs on Amazon EC2 with Hadoop [21].

Dapper [27] is a tracing framework designed for low overhead, application transparency, and ubiquitous deployment. Trace data are organized in a Dapper trace tree, where each node represents a basic unit of work called a span. Each trace is stored in BigTable. It also offers a programmatic API, as well as an annotation API. Due to its more restricted trace-tree model, it does not support graph-traversal or any of the operators in G^2. DTrace [9] is another tracing framework that supports on-demand instrumentation of distributed systems. It allows customized predicates and aggregation functions via a scripting language. The aggregation is applied to a set of flat trace records, which is different from G^2 as the later applies aggregation to a graph.

8 Concluding Remarks

Execution graphs capture runtime behavior of distributed system executions. These graphs are unique in their value for distributed-system diagnosis and in their distinctly different characteristics compared to well-known social and web graphs. G^2 makes those graphs useful with new graph operators and with query support, and makes graph processing efficient with a distributed engine. By doing so, G^2 becomes an effective tool for distributed-system diagnosis and at the same time advances the state of art in distributed large-scale graph processing.

References

[1] The apache cassandra project. `http://cassandra.apache.org/`.

[2] Berkeley db. `http://www.oracle.com/database/berkeley-db/db/index.html`.

[3] The LINQ project. `http://msdn.microsoft.com/netframework/future/linq/`.

[4] Pagerank. `http://en.wikipedia.org/wiki/PageRank`.

[5] Phoenix compiler framework. `http://research.microsoft.com/phoenix/`.

[6] Program slicing. `http://en.wikipedia.org/wiki/Program_slicing`.

[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*. ACM, 2003.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[9] M. W. S. Bryan M. Cantrill and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, 2004.

[10] G. Candea, S. Bucur, and C. Zamfir. Automated Software Testing as a Service (TaaS). In *ACM SoCC*, 2010.

[11] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[13] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[15] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[18] R. Fonseca, M. J. Freedman, and G. Porter. Experiences with tracing causality in networked services. In *INM/WREN*, 2010.

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[21] HADOOP. *http://hadoop.apache.org/*.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[23] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: Debugging deployed distributed systems. In *NSDI*, 2008.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, New York, NY, USA, 2010. ACM.

[25] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *DSN*, 2010.

[26] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, 2010.

[28] T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *ISSTA*. ACM, 2007.

[29] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*. ACM, 2009.

[30] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*. USENIX Association, 2008.

[31] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*. ACM, 2010.