Scalable Error Isolation for Distributed Systems

Diogo Behrens, Sergei Arnautov, Christof Fetzer (TU Dresden)
Marco Serafini (Qatar Computing Research Institute)
Flavio P. Junqueira (Microsoft Research, Cambridge)

May 6, 2015
Motivation

Amazon Web Services » Service Health Dashboard » Amazon S3 Availability Event: July 20, 2008

Amazon S3 Availability Event: July 20, 2008

We wanted to provide some additional detail about the problem we experienced on Sunday, July 20th.

At 8:40am PDT, error rates in all Amazon S3 datacenters began to quickly climb and our alarms went off. By 8:50am PDT, error rates were significantly elevated and very few requests were completing successfully. By 8:55am PDT, we had multiple engineers engaged and investigating the issue. Our alarms pointed at problems processing customer requests in multiple places within the system and across multiple data centers. While we began investigating several possible causes, we tried to restore system health by taking several actions to reduce system load. We reduced system load in several stages, but it had no impact on restoring system health.
**ABSTRACT**

Mesa is a highly scalable analytic data warehousing system that stores critical measurement data related to Google’s Internet advertising business. Mesa is designed to satisfy the critical nature of this data result in unique technical and operational challenges for processing, storing, and querying. The requirements for such a data store are:
Motivation

Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan
Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal
Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqui, David Jones
Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, Divyakant Agrawal

ABSTRACT

Mesa is a highly scalable analytics database that stores critical measurements for Internet advertising business. Mesa tables go through schema changes every month. Mesa users frequently need to modify schemas associated with existing tables, necessitating a re-sorting of the existing data. Mesa handles this case by treating the old and new schema versions as one for update/compaction. Specifically, Mesa makes the schema change visible to new queries immediately, handles computation in the query path until the next base compaction.

4.4 Mitigating Data Corruption Problems

Mesa uses tens of thousands of machines in the cloud that are administered independently and are shared among many services at Google to host and process data. For any computation, there is a non-negligible probability that faulty hardware or software will cause incorrect data to be generated and/or stored. Simple file level checksums are not sufficient to defend against such events because the corruption can occur transiently in CPU or RAM. At Mesa’s scale, these seemingly rare events are common. Guarding against such corruptions is an important goal in Mesa’s overall design.
Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqui, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, Divyakant Agrawal

4.4 Mitigating Data Corruption Problems

Mesa uses tens of thousands of machines in the cloud that are administered independently and are shared among many services at Google to host and process data. For any computation, there is a non-negligible probability that faulty hardware or software will cause incorrect data to be generated and/or stored. Simple file level checksums are not sufficient to defend against such events because the corruption can occur transiently in CPU or RAM. At Mesa’s scale, these seemingly rare events are common. Guarding against such corruptions is an important goal in Mesa’s overall design.

CRC for messages!  ECC for memory!  CPU??
From data corruption to service disruption
From data corruption to service disruption

process $p_i$

buffer

variables

process $p_j$

buffer

variables

3 / 28
From data corruption to service disruption

process $p_i$

process $p_j$
From data corruption to service disruption

process $p_i$

error propagation

process $p_j$
From data corruption to service disruption

ECC?

error propagation

buffer

variables

process $p_i$

variables

process $p_j$
From data corruption to service disruption

ECC? CRC?

process $p_i$

error propagation

process $p_j$
This talk is about...
This talk is about...
Goal: error isolation

process $p_i$

process $p_j$

variables

buffer

+ $\rightarrow$

discard!
Goal: error isolation
How to deal with data corruptions?
while(1) {
    switch(state) {
    case INIT:
        // create socket, bind and listen
        fd = socket(AF_INET, SOCK_STREAM, 0);
        if (fd < 0) {
            perror("socket");
            return EXIT_FAILURE;
        }
        // this is important, so that if a process restarts, it can
        // quickly reuse the same port
        int on = 1;
        if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {
            perror("setsockopt");
            return EXIT_FAILURE;
        }
        bzero(&addr, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr = htonl(INADDR_ANY);
        addr.sin_port = htons(port);
        if (bind(fd, (struct sockaddr*) &addr, sizeof(addr)) < 0) {
            perror("bind");
            return EXIT_FAILURE;
        }
        if (listen(fd, 2) < 0) {
            perror("listen");
            return EXIT_FAILURE;
        }
        // initialize ukv service
        ukv = ukv_init();
        state = ACCT;
        break;
    case RECV:
        // once a connection is accepted, read from the connection
        // until it is closed
        read = recvfrom(cfd, buffer, BUFSIZE, 0, (struct sockaddr*) &caddr, &len);
        if (read <= 0) {
            perror("recvfrom");
            close(cfd);
            state = ACCT;
            break;
        }
        buffer[read] = '\0';
        msg = buffer;
        state = PROC;
        break;
    case PROC:
        r = ukv_recv(ukv, msg);
        if (!r) state = FINI;
        else state = SEND;
        break;
    case SEND:
        sendto(cfd, r, strlen(r), 0, (struct sockaddr*) &caddr, sizeof(caddr));
        ukv_done(ukv, r);
        state = RECV;
        break;
    }
DIY: Ad hoc software checks

- Complex, time consuming
- Which errors to consider?
- No principled approach

```c
while(1) {
    switch(state) {
    case INIT: {
        // create socket, bind and listen
        fd = socket(AF_INET, SOCK_STREAM, 0);
        if (fd < 0) {
            perror("socket");
            return EXIT_FAILURE;
        }
        // this is important, so that if a process restarts, it can
        // quickly reuse the same port
        int on = 1;
        if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {
            perror("setsockopt");
            return EXIT_FAILURE;
        }
        bzero(&addr, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr = htonl(INADDR_ANY);
        addr.sin_port = htons(port);
        if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
            perror("bind");
            return EXIT_FAILURE;
        }
        if (listen(fd, 2) < 0) {
            perror("listen");
            return EXIT_FAILURE;
        }
        // initialize ukv service
        ukv = ukv_init();
        state = ACCT;
        break;
    }
    case RECV: {
        // once a connection is accepted, read from the connection
        // until it is closed
        read = recvfrom(cfd, buffer, BUFSIZE, 0, (struct sockaddr*) &caddr, &len);
        if (read <= 0) {
            perror("recvfrom");
            close(cfd);
            state = ACCT;
            break;
        }
        buffer[read] = '\0';
        msg = buffer;
        state = PROC;
        break;
    }
    case PROC: {
        r = ukv_recv(ukv, msg);
        if (!r) state = FINI;
        else state = SEND;
        break;
    }
    case SEND: {
        sendto(cfd, r, strlen(r), 0, (struct sockaddr*) &caddr, sizeof(caddr));
        ukv_done(ukv, r);
        state = RECV;
        break;
    }
    }
    if (!r) state = FINI;
    else state = SEND;
}
```
Principled approach: Byzantine fault tolerance

agreement protocol

servers

client
Principled approach: Byzantine fault tolerance

- Expensive replication
- Performance degradation
- Multithreading?
Principled approach: Local hardening
Principled approach: Local hardening

- **Instruction duplication (SWIFT, CGO’05)**
  - compiler technique
  - not designed for distributed systems
  - last-mile faults (no error isolation)
Principled approach: Local hardening

- **Instruction duplication (SWIFT, CGO’05)**
  - compiler technique
  - not designed for distributed systems
  - last-mile faults (no error isolation)

- **PASC (ATC’12)**
  - achieves error isolation
  - large memory overhead (2x)
  - no support for multithreading
Scalable Error Isolation
A new approach
Local hardening

No additional messages exchanged;
Local redundancy in space and time
Scalable Error Isolation (SEI)

**Local hardening**
No additional messages exchanged; Local redundancy in space and time

**End-to-end**
CRCs for communication and **computation**
Scalable Error Isolation (SEI)

Local hardening
No additional messages exchanged;
Local redundancy in space and time

End-to-end
CRCs for communication and computation

Formal guarantees
Fault model and correctness proof
Scalability dimensions
Memory scalability
Small footprint (ECC or other error codes)
Scalability dimensions

Memory scalability
Small footprint (ECC or other error codes)

Thread scalability
Support for multithreaded applications
Scalability dimensions

Memory scalability
Small footprint (ECC or other error codes)

Thread scalability
Support for multithreaded applications

Codebase scalability
Compiler technique reduces developer work
Outline

Overview of SEI
Requirements, fault model, and algorithm

Challenges
Support for multithreaded applications

Implementation: libsei
Library for C-based programs

Evaluation
Fault coverage and performance overhead
Outline

**Overview of SEI**
Requirements, fault model, and algorithm

**Challenges**
Support for multithreaded applications

**Implementation:** libsei
Library for C-based programs

**Evaluation**
Fault coverage and performance overhead
Target applications

- Event based – message passing

Event handler

State
Target applications

- **Event based – message passing**

  ![Diagram of event-based messaging]

- **Multithreaded applications**
  - Critical sections to access shared variables
  - Hierarchical locking (consistent order) to avoid deadlocks
Faults corrupt any number of variables

Assumptions:
▶ Fault frequency: at most one fault per event handler
▶ Corruption coverage: detection codes work, e.g., ECC, CRC

Arbitrary State Corruption (ASC) Model
Arbitrary State Corruption (ASC) Model

Faults corrupt any number of variables

Assumptions:

- **Fault frequency:**
  at most one 1 fault per event handler
Arbitrary State Corruption (ASC) Model

Faults corrupt any number of variables

Assumptions:

- **Fault frequency:** at most one 1 fault per event handler
- **Corruption coverage:** detection codes work, e.g., ECC, CRC
Hardening with SEI

Event handler

transformation

Check input
First execution
Reset
Second execution
Validation
Hardening with SEI

- Check input
- First execution
- Reset
- Second execution
- Validation

discard message if CRC invalid
Hardening with SEI

Check input

First execution

discard message if CRC invalid

check read variables

Reset

Second execution

Validation
Hardening with SEI

Check input

First execution

discard message if CRC invalid

check read variables

Reset

Second execution

Validation
Hardening with SEI

Check input
- discard message if CRC invalid

First execution
- check read variables, snapshot original state

Reset

Second execution

Validation
Hardening with SEI

Check input
- discard message if CRC invalid

First execution
- check read variables, snapshot original state

Reset
- restore snapshot, record state modifications

Second execution

Validation
Hardening with SEI

- **Check input**
  - discard message if CRC invalid

- **First execution**
  - check read variables, snapshot original state

- **Reset**
  - restore snapshot, record state modifications

- **Second execution**
  - check read variables, record modified variables

- **Validation**
Hardening with SEI

- **Check input**
  - discard message if CRC invalid

- **First execution**
  - check read variables, snapshot original state

- **Reset**
  - restore snapshot, record state modifications

- **Second execution**
  - check read variables, record modified variables

- **Validation**
  - compare state modifications
Hardening with SEI

Check input
- discard message if CRC invalid

First execution
- check read variables, snapshot original state

Reset
- restore snapshot, record state modifications

Second execution
- check read variables, record modified variables

Validation
- compare state modifications
Hardening with SEI

Check input:
- discard message if CRC invalid

First execution:
- check read variables, snapshot original state

Reset:
- restore snapshot, record state modifications

Second execution:
- check read variables, record modified variables, and calculates CRC
- compare state modifications

Validation:
- invalid CRC, discard!
Outline

Overview of SEI
Requirements, fault model, and algorithm

Challenges
Support for multithreaded applications

Implementation: libsei
Library for C-based programs

Evaluation
Fault coverage and performance overhead
Deterministic event handling with multithreading

First execution

Reset

Second execution

Validation

but no faults!

read v=A

lock

unlock

read v=B

lock

unlock

write v=B

lock

unlock

T1

C

m1

T2

C

m2

First execution

...
Deterministic event handling with multithreading

$m_1$

T1

First execution

Reset

Second execution

V

S

read v=A

lock

read v=A

unlock

block...

write v=B

lock

T2

C

First execution

...
Outline

Overview of SEI
Requirements, fault model, and algorithm

Challenges
Support for multithreaded applications

Implementation: libsei
Library for C-based programs

Evaluation
Fault coverage and performance overhead
while(1) {
    ilen = recv_msg(imsg, &crc);
    do_something_here(msg);
    omsg = create_message(&olen);
    __output_append(omsg, olen);
    __output_done();
    __end(); // end of handler
    if (__begin(imsg, ilen, crc)) {
        do_something_here(msg);
        omsg = create_message(&olen);
        __output_append(omsg, olen);
        __output_done();
        __end(); // end of handler
        send_msg(omsg, olen, CRC(omsg, olen));
    } else continue; // discard invalid
}

- Annotate event handler with __begin and __end
- Annotate messages __output* for annotation __crc_pop for next CRC

libsei: SEI implementation in C
bitbucket.org/db7/libsei
while(1) {
    ilen = recv_msg(imsg, &crc);
    if (__begin(imsg, ilen, crc)) {
        do_something_here(msg);
        omsg = create_message(&olen);

        __end(); // end of handler
    } else continue; // discard invalid

    send_msg(omsg, olen, CRC(omsg, olen));
}

- **Annotate event handler**
  - with __begin and __end

- **Annotate messages**
  - __output* for annotation
  - __crc_pop for next CRC
while(1) {
    ilen = recv_msg(imsg, &crc);
    if (__begin(imsg, ilen, crc)) {
        do_something_here(msg);
        omsg = create_message(&olen);
        _output_append(omsg, olen);
        _output_done();
        _end();       // end of handler
    } else continue; // discard invalid

    send_msg(omsg, olen, CRC(omsg, olen));
}
libsei: SEI implementation in C
bitbucket.org/db7/libsei

while(1) {
    ilen = recv_msg(imsg, &crc);
    if (__begin(imsg, ilen, crc)) {
        do_something_here(msg);
        omsg = create_message(&olen);
        __output_append(omsg, olen);
        __output_done();
        __end(); // end of handler
    } else continue; // discard invalid

    send_msg(omsg, olen, __crc_pop());
}
while(1) {
    ilen = recv_msg(imsg, &crc);
    if (__begin(imsg, ilen, crc)) {
        do_something_here(msg);
        omsg = create_message(&olen);
        __output_append(omsg, olen);
        __output_done();
        __end();        // end of handler
    } else continue;  // discard invalid

    send_msg(omsg, olen, __crc_pop());
}
```c
while(1) {
    ilen = recv_msg(imsg, &crc);
    if (__begin(imsg, ilen, crc)) {
        do_something_here(msg);
        omsg = create_message(&olen);
        __output_append(omsg, olen);
        __output_done();
        __end();  // end of handler
    } else continue;  // discard invalid

    send_msg(omsg, olen, __crc_pop());
}
```

- **Annotate event handler** with `__begin` and `__end`
- **Annotate messages** `__output*` for annotation, `__crc_pop` for next CRC
- **Compile with GCC ≥ 4.7**
  - state updates within handler instrumented with TM pass
- **libsei does the rest**
  - executes handler twice
  - calculates CRCs
  - validates state updates
Outline

Overview of SEI
Requirements, fault model, and algorithm

Challenges
Support for multithreaded applications

Implementation: libsei
Library for C-based programs

Evaluation
Fault coverage and performance overhead
Use cases

- **memcached**
  - Key-value store
  - Widely used as cache for databases
  - Internally, a huge hash table with eviction queues
  - Multithreaded
Use cases

- **memcached**
  - Key-value store
  - Widely used as cache for databases
  - Internally, a huge hash table with eviction queues
  - Multithreaded

- **Deadwood**
  - DNS recursive server
  - Single-threaded
  - See paper for results
Fault coverage: targeted software fault injection

<table>
<thead>
<tr>
<th>Control flownative</th>
<th>SEI</th>
<th>Crash/other</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.66% - 90.34%</td>
<td>0.06%</td>
<td>14.70%</td>
</tr>
<tr>
<td>44.18% - 55.82%</td>
<td>0.15%</td>
<td>57.55%</td>
</tr>
</tbody>
</table>

SEI: two orders of magnitude fewer undetected errors
## Fault coverage: targeted software fault injection

<table>
<thead>
<tr>
<th>Fault group</th>
<th>Variant</th>
<th>Undetected</th>
<th>SEI-detected</th>
<th>Crash/other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>native</td>
<td>9.66%</td>
<td>-</td>
<td>90.34%</td>
</tr>
<tr>
<td></td>
<td>SEI</td>
<td>0.06%</td>
<td>14.70%</td>
<td>85.23%</td>
</tr>
<tr>
<td>Data flow</td>
<td>native</td>
<td>44.18%</td>
<td>-</td>
<td>55.82%</td>
</tr>
<tr>
<td></td>
<td>SEI</td>
<td>0.15%</td>
<td>57.55%</td>
<td>42.29%</td>
</tr>
</tbody>
</table>

SEI: two orders of magnitude fewer undetected errors
Fault coverage: targeted software fault injection

<table>
<thead>
<tr>
<th>Fault group</th>
<th>Variant</th>
<th>Undetected</th>
<th>SEI-detected</th>
<th>Crash/other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>native</td>
<td>9.66%</td>
<td>-</td>
<td>90.34%</td>
</tr>
<tr>
<td></td>
<td>SEI</td>
<td>0.06%</td>
<td>14.70%</td>
<td>85.23%</td>
</tr>
<tr>
<td>Data flow</td>
<td>native</td>
<td>44.18%</td>
<td>-</td>
<td>55.82%</td>
</tr>
<tr>
<td></td>
<td>SEI</td>
<td>0.15%</td>
<td>57.55%</td>
<td>42.29%</td>
</tr>
</tbody>
</table>

SEI: two orders of magnitude fewer undetected errors.
### Fault coverage: targeted software fault injection

<table>
<thead>
<tr>
<th>Fault group</th>
<th>Variant</th>
<th>Undetected</th>
<th>SEI-detected</th>
<th>Crash/other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>native</td>
<td>9.66%</td>
<td>-</td>
<td>90.34%</td>
</tr>
<tr>
<td></td>
<td>SEI</td>
<td>0.06%</td>
<td>14.70%</td>
<td>85.23%</td>
</tr>
<tr>
<td>Data flow</td>
<td>native</td>
<td>44.18%</td>
<td>-</td>
<td>55.82%</td>
</tr>
<tr>
<td></td>
<td>SEI</td>
<td>0.15%</td>
<td>57.55%</td>
<td>42.29%</td>
</tr>
</tbody>
</table>

SEI: two orders of magnitude fewer undetected errors
Fault coverage: CPU undervolting
Fault coverage: CPU undervolting

- Detected:
  - Native: 99.15%
  - SEI: 100%

- Undetected:
  - Native: 0.85%
  - SEI: 0%
Fault coverage: CPU undervolting

![Bar chart showing fault coverage for CPU undervolting. The chart compares detected and undetected manifested errors. The native detected error rate is 99.15%, while the SEI detected error rate is 100%. The undetected error rates are 0.85% for native and 0.0% for SEI.](chart.png)
Fault coverage: CPU undervolting

- 99.15% detected errors
- 0.85% undetected errors
- 0.85% native errors
- 0% SEI errors

Error isolation:

SEI: no undetected errors
memcached performance: threads and key range

SEI: little overhead with ≥ 128 B and ranges of ≥ 100 keys.
memcached performance: threads and key range

![Graph showing throughput (k.req/s) vs. threads for key ranges of 8 B and 128 B. The graph compares SEI and native performance. SEI shows little overhead with keys ≥ 128 B and ranges of ≥ 100 keys.]
memcached performance: threads and key range

- SEI: little overhead with ≥128 B and ranges of ≥100 keys

![Graphs showing throughput vs. threads and key range for 8 B and 128 B key sizes.](image_url)
memcached performance: threads and key range

SEI: little overhead with $\geq 128\text{ B}$ and ranges of $\geq 100$ keys
Conclusion

- **Algorithm: Scalable Error Isolation (SEI)**
  - Local and end-to-end
  - Effective against data corruptions

- **Implementation: libsei**
  - No memory overhead with ECC
  - Little performance overhead with non-CPU intensive applications
  - Implementation is open source
Thank you! Questions?

Source code and technical report:
http://bitbucket.org/db7/libsei