MUTABLE CHECKPOINT-RESTART
Automating Live Update For Generic Server Programs

Cristiano Giuffrida (VU), Călin Iorgulescu (EPFL)
Andrew S. Tanenbaum (VU)
MCR in a nutshell

- Generic live update framework
- Can handle whole-program updates
- Low engineering effort to adapt updates (~30 LOC / 1 KLOC)
- Low runtime performance overhead (~2%)
- Realistic update times (≤ 1 s) under load (with 100 connections)

- Combines checkpoint-restart with native initialization
Outline

The Live Update Problem

Mutable Checkpoint-Restart

  Overview

  Architecture

  Evaluation

Conclusion
Outline

The Live Update Problem

Mutable Checkpoint-Restart

Overview

Architecture

Evaluation

Conclusion
Update without downtime

• Don’t stop running services!
• Don’t disrupt clients!

“If you think database patching is onerous, then try patching a SCADA system that’s running a power plant.”

-- Kelly Jackson Higgins on the SCADA patch problem, 2013
What are the existing options?

- Rolling upgrades
  - require redundant hardware
  - may lead to inconsistencies between machines

- In-place live update
  - replaces part of the running code
  - limited in terms of update complexity

- Whole-program live update
  - requires considerable annotation effort

Servers protected with Ksplice Uptrack: 100,000+ at more than 700 companies

Source: www.ksplice.com
What are the existing options?

- Rolling upgrades
  - require redundant hardware
  - may lead to inconsistencies between machines

- In-place live update
  - replaces part of the running code
  - limited in terms of update complexity

- Whole-program live update
  - requires considerable annotation effort
  - required
What We Did: Mutable Checkpoint-Restart

- Focused on servers: **high availability** requirements & well-defined structure
- **Low engineering effort** to adapt new programs
- Updating recreates **entire process hierarchy**
  - individual processes natively initialize their state
  - process state changes are transferred to the new version

- Tested on 4 real-world servers: *Apache httpd, nginx, OpenSSH, vsftpd*
Outline

The Live Update Problem

Mutable Checkpoint-Restart

Overview

Architecture

Evaluation

Conclusion
A generic live update framework

• Support complex updates

• Don’t break OS invariants
  • maintain consistent server state
  • e.g., open connections must not break

• Support generic C servers
  • weakly typed & types change
  • no object tracing
A generic live update framework

- Support complex updates
- Don’t break OS invariants
  - maintain consistent server state
  - e.g., open connections must not break
- Support generic C servers
  - weakly typed & types change
  - no object tracing

- Quiescence Detection
  - apply updates safely
A generic live update framework

- Support complex updates
- Don’t break OS invariants
  - maintain consistent server state
  - e.g., open connections must not break
- Support generic C servers
  - weakly typed & types change
  - no object tracing

- Quiescence Detection
  - apply updates safely

- Mutable Reinitialization
  - populate long-lived objects
A generic live update framework

- Support complex updates
- Don’t break OS invariants
  - maintain consistent server state
  - e.g., open connections must not break
- Support generic C servers
  - weakly typed & types change
  - no object tracing

- Quiescence Detection
  - apply updates safely
- Mutable Reinitialization
  - populate long-lived objects
- State Transfer
  - hybrid garbage collector approach
The update process

1. An update request is received
The update process

Quiescence

2. All tasks reach a quiescent point & suspend
The update process

3. The new version initializes
The update process

4. Modified process state is **transferred**

Quiescent (V1)  Memory object pairing  Initialized (V2)

State transfer
The update process

5. Control is transferred to V2
Preparing a program for MCR

Offline Profiling

server.c

Quiescence Profiler

Quiescence Points

Build time instrumentation

server.bc

LLVM Pass

(Static Type Analysis)

GOLD Linker

libmcr.a

Runtime instrumentation

server

libmcr.so
Outline

The Live Update Problem

Mutable Checkpoint-Restart

Overview

Architecture

Evaluation

Conclusion
A simple server example

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}
A simple server example

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}
A simple server example

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t;
l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}
A simple server example

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}
Quiescence Detection

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}

- Updates occur only at safe points (quiescence points)
- Allows stopping/restarting tasks safely
- Reduces expensive call stack instrumentation

Updates occur only at safe points (quiescence points)

Quiescence point
- long lived call stack
- underlying blocking call
- initialization already done
Mutable Reinitialization 1/2

- Rebuilding the entire execution state is notoriously **hard**!
  - ... *but* ...
- **Most long-lived state is populated during initialization**

- **Record** immutable object creation events (**startup log**)
  - e.g., creating a socket, opening a file

- **Replay** event result during initialization

- Let the initialization complete
Mutable Reinitialization 2/2

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t, l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}

void server_init(struct conf_s *conf) {
    ...
    conf.sock = socket(...);
    ...
}
Mutable Reinitialization 2/2

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf.sock</td>
<td>conf.sock = 5</td>
<td>conf.sock = 5</td>
</tr>
<tr>
<td></td>
<td>Native syscall</td>
<td>Replayed event from V1</td>
</tr>
</tbody>
</table>

void server_init(struct conf_s *conf) {
    ...
    conf.sock = socket(...);
    ...
}
State Transfer

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t;
l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}
State Transfer

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t;
l_t list;

...
State Transfer

/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

...
/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

...
Violating Assumptions

- Unsupported immutable objects
  - e.g., process IDs stored in opaque data structures, pointer encoding

- Non-deterministic process model
  - e.g., workers are spawned on-demand

- Non-replayed operations that affect initialization
  - e.g., check if a PID file exists, then quit (Apache httpd)
/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

/* Startup configuration. */
struct conf_s *conf;

/* Server implementation. */
int main() {
    server_init(&conf);
    while (1) {
        void *e = server_get_event();
        server_handle_event(e, conf, b, &list);
    }
    return 0;
}

/* MCR annotations. */
MCR_ADD_OBJ_HANDLER(b, user_b_handler);
MCR_ADD_REINIT_HANDLER(user_reinit_handler);

User defined handlers
• opaque object handlers
• custom initialization handlers
Outline

The Live Update Problem

Mutable Checkpoint-Restart

Overview

Architecture

Evaluation

Conclusion
MCR with real world servers

- Tested on **Apache httpd** (2.2.23), **nginx** (0.8.54), **OpenSSH** (1.1.0), **vsftpd** (3.5p1)

- How much **engineering effort** is needed?

- Does MCR yield low **performance overhead**?

- Does MCR yield reasonable **update time**?

- How much **memory** does MCR use?
How much engineering effort is needed?

<table>
<thead>
<tr>
<th># patches</th>
<th>Changes LOC</th>
<th>Annotations LOC</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache httpd</td>
<td>5</td>
<td>10844</td>
<td>383</td>
</tr>
<tr>
<td>nginx</td>
<td>25</td>
<td>9681</td>
<td>357</td>
</tr>
<tr>
<td>vsftpd</td>
<td>5</td>
<td>5830</td>
<td>103</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>5</td>
<td>14370</td>
<td>184</td>
</tr>
</tbody>
</table>

Overview of custom handler code written to support updates.

- Ideally, annotations would be written directly by the maintainers
Does MCR yield low performance overhead?

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Static</th>
<th>Static + Dynamic</th>
<th>Static + Dynamic + Quiescence</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache httpd</td>
<td>1.040</td>
<td>1.043</td>
<td>1.047</td>
<td>~4.7%</td>
</tr>
<tr>
<td>nginx</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>~0%</td>
</tr>
<tr>
<td>nginx&lt;sub&gt;reg&lt;/sub&gt;</td>
<td>1.175</td>
<td>1.192</td>
<td>1.186</td>
<td>~19%</td>
</tr>
<tr>
<td>vsftpd</td>
<td>1.027</td>
<td>1.028</td>
<td>1.028</td>
<td>~2.8%</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>0.999</td>
<td>1.001</td>
<td>1.001</td>
<td>~0%</td>
</tr>
</tbody>
</table>

Normalized overhead of each instrumentation step.

- What about nginx<sub>reg</sub>?
- Static instrumentation for **custom region allocators**
  - reduces number of opaque objects
  - adds runtime overhead
Does MCR yield reasonable update time?

- Update = *Quiescence* + *Initialization* + *State Transfer*
  - *quiescence* time $\to < 100$ms
  - *initialization* time $\to < 50$ms (overhead)

Workload independent

State transfer time depending on the number of open connections
How much memory does MCR use?

- Larger memory footprints
- Binary size increased by: 18.7% - 135.2%
- Resident set size increased by: 10% - 383.6% (avg. 188.5%)

Why?
- memory object tracing metadata
- implementation is optimized for speed

Favor annotation-less support over memory overhead
Outline

The Live Update Problem

Mutable Checkpoint-Restart

Overview

Architecture

Evaluation

Conclusion
Conclusion

• MCR combines checkpoint-restart with native initialization

• Generic live update framework for whole-program updates

• Low engineering effort to adapt updates (~30 LOC / 1 KLOC)

• Low runtime performance overhead (~2%)}

• Realistic update times (≤ 1 s) under load (with 100 connections)
Thank you!

• Questions ?