MUTABLE CHECKPOINT-RESTART

Automating Live Update For Generic Server Programs

Cristiano Giuffrida (VU), **Călin Iorgulescu** (EPFL)
Andrew S. Tanenbaum (VU)

VU VRIJE UNIVERSITEIT AMSTERDAM

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

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

Outline

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

ORACLE | Ksplice

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

Source: www.ksplice.com

- Whole-program live update
  - requires considerable annotation effort

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

ORACLE | Ksplice

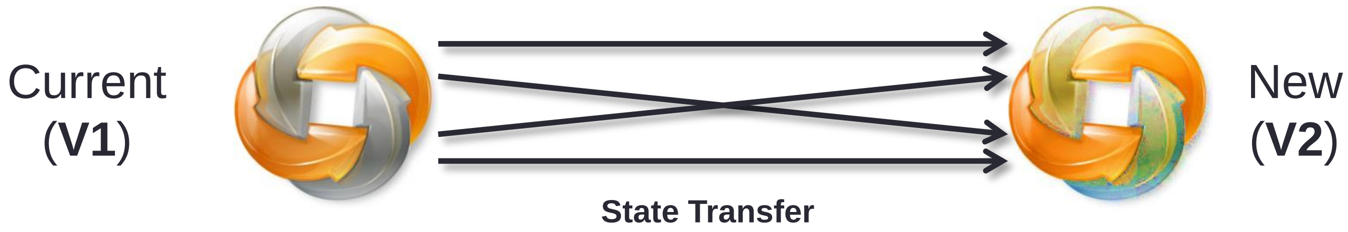
Servers protected with **Ksplice Uptrack**:
100,000+ at more than **700 companies**

Source: www.ksplice.com

- 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

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

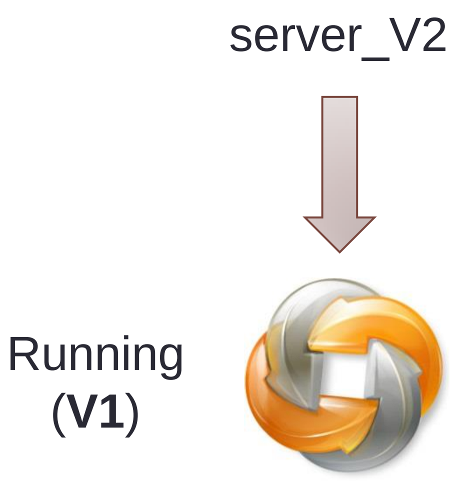
- **Quiescence Detection**
  - apply updates safely
- **Mutable Reinitialization**
  - populate long-lived objects

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



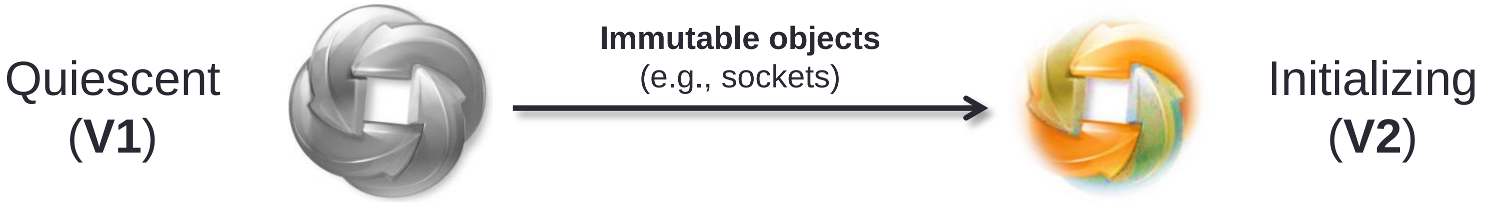
1. An update request is received

The update process Quiescence

Quiescent
(**V1**)

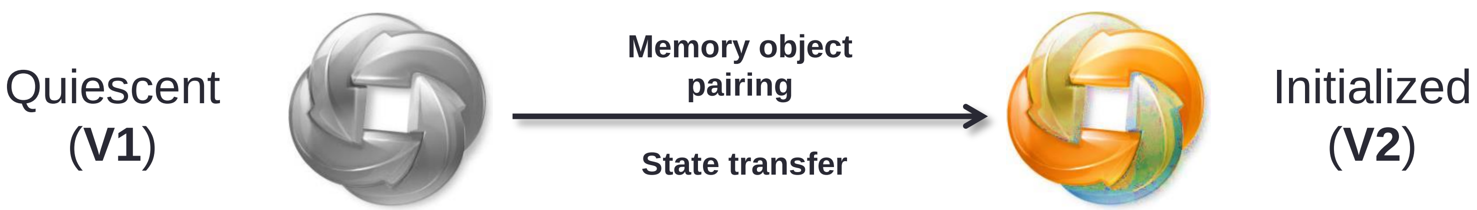
2. All *tasks* reach a **quiescent** point & suspend

The update process Mutable Reinitialization



3. The new version **initializes**

The update process State Transfer



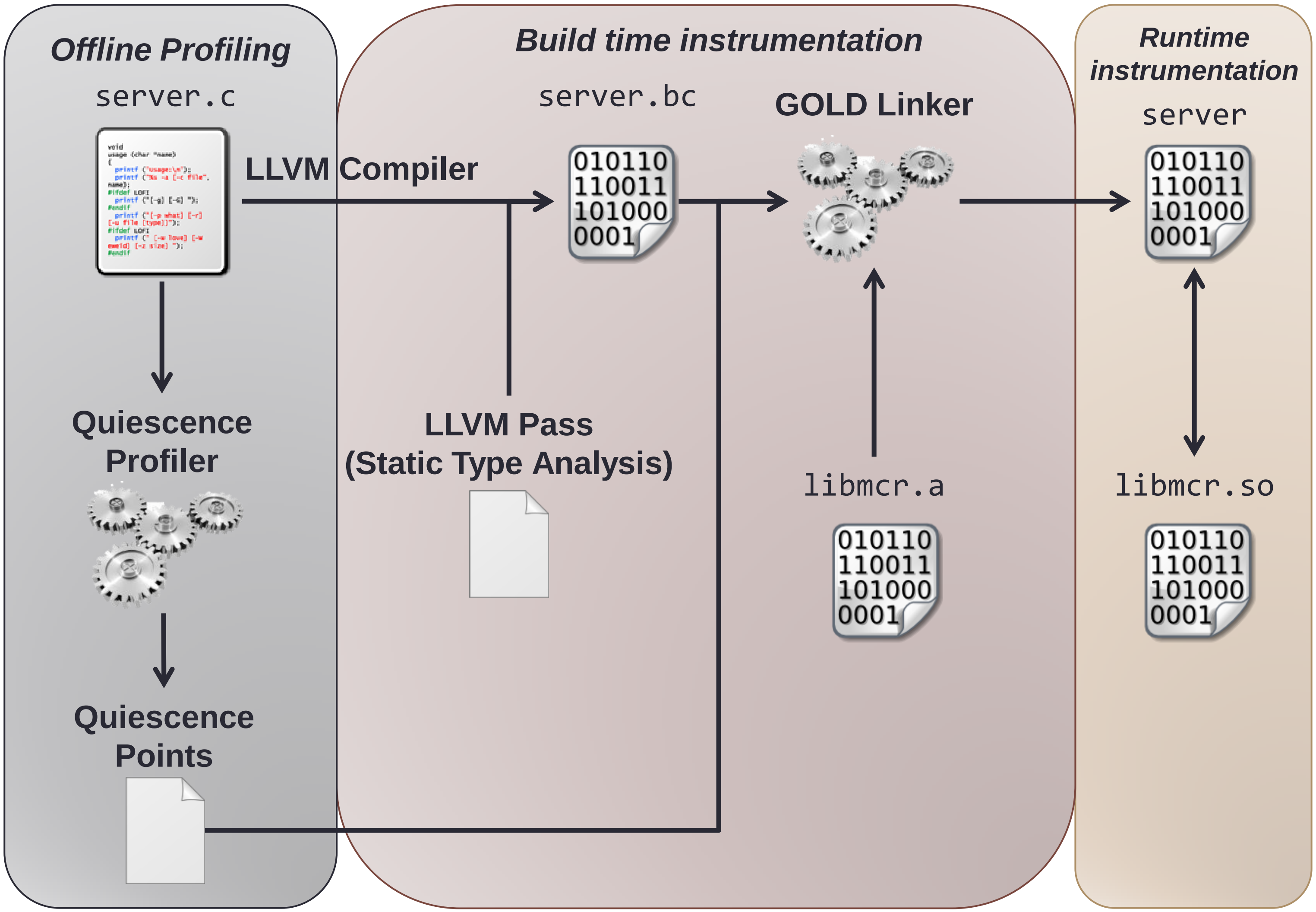
4. Modified process state is **transferred**

The update process

Running
(**V2**)

5. **Control is transferred** to V2

Preparing a program for MCR



Outline

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;
}
```

Server initialization code

- load config
- create sockets
- allocate memory
- spawn workers

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;
}
```

Event processing loop

- accept new connections
- delegate to workers

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;
}
```

Global memory objects

- opaque objects
- well-defined objects

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

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;
}
```

V1	V2
conf.sock = 5	conf.sock = **5**
Native syscall	Replayed event from V1

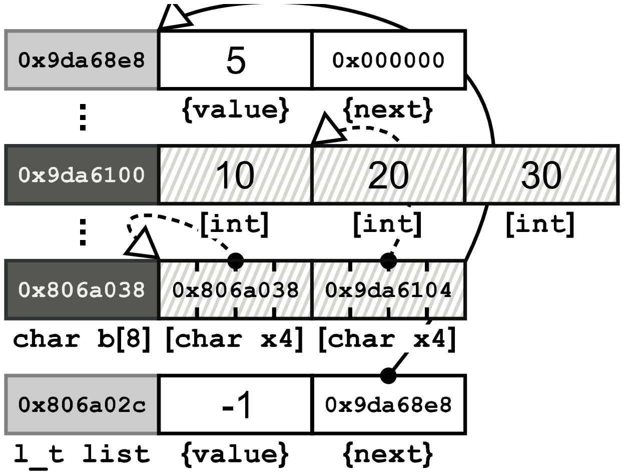
```
void server_init(struct conf_s *conf)
{
    ...
    conf.sock = socket(...);
    ...
}
```

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

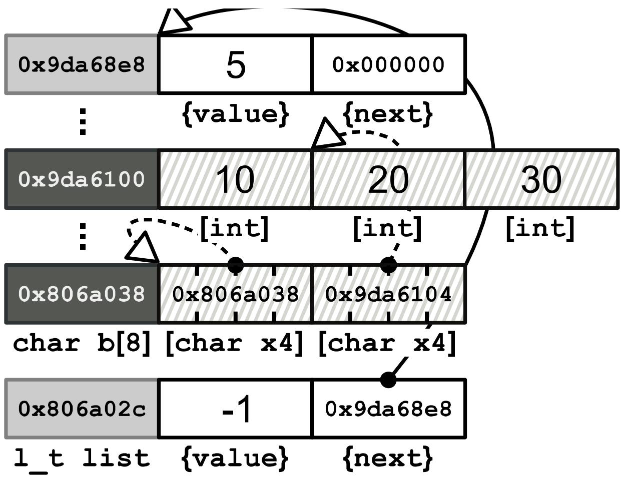
```
/* Auxiliary data structures. */
char b[8];
typedef struct list_s {
    int value;
    struct list_s *next;
} l_t; l_t list;

...
```



V1

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;
    int new;
} l_t; l_t list;
...
```

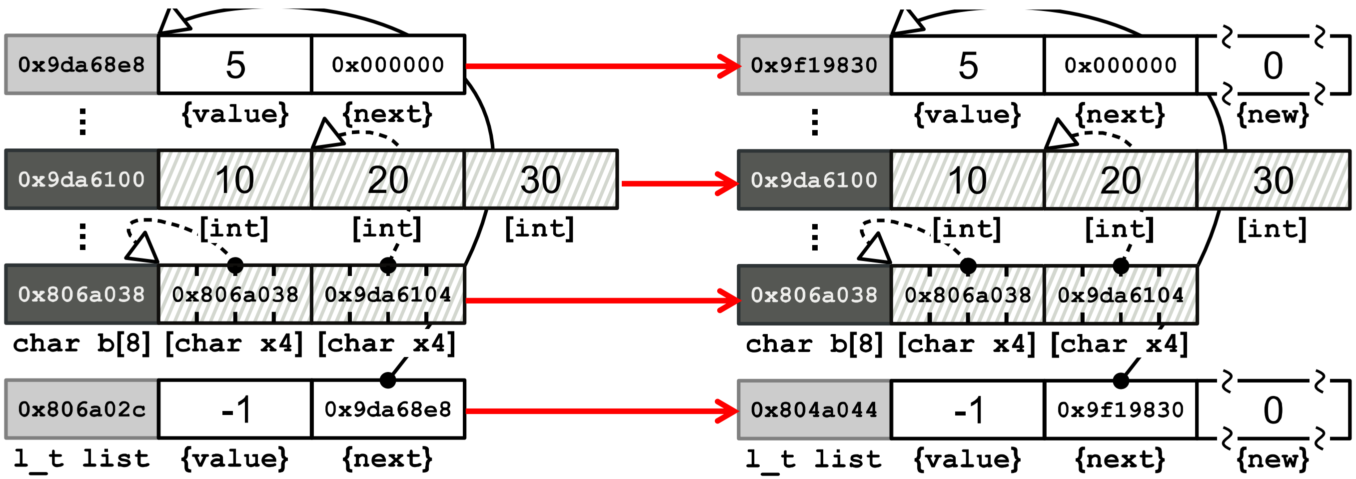
V1

V2

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;
    int new;
} l_t; l_t list;
...
```



V1 **V2**

Violating Assumptions 1/2

- 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*)

Violating Assumptions 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;
}

/* 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 ?

	# patches	Changes LOC	Annotations LOC	Overhead
Apache httpd	5	10844	383	**3.4%**
nginx	25	9681	357	**3.5%**
vsftpd	5	5830	103	**1.7%**
OpenSSH	5	14370	184	**1.2%**

Overview of custom handler code written to support updates.

- Ideally, annotations would be written directly by the maintainers

Does MCR yield low performance overhead ?

	Static	Static + Dynamic	Static + Dynamic + Quiescence	Overhead
Apache httpd	1.040	1.043	1.047	**~4.7%**
nginx	1.000	1.000	1.000	**~0%**
nginx$_{reg}$	*1.175*	*1.192*	*1.186*	***~19%***
vsftpd	1.027	1.028	1.028	**~2.8%**
OpenSSH	0.999	1.001	1.001	**~0%**

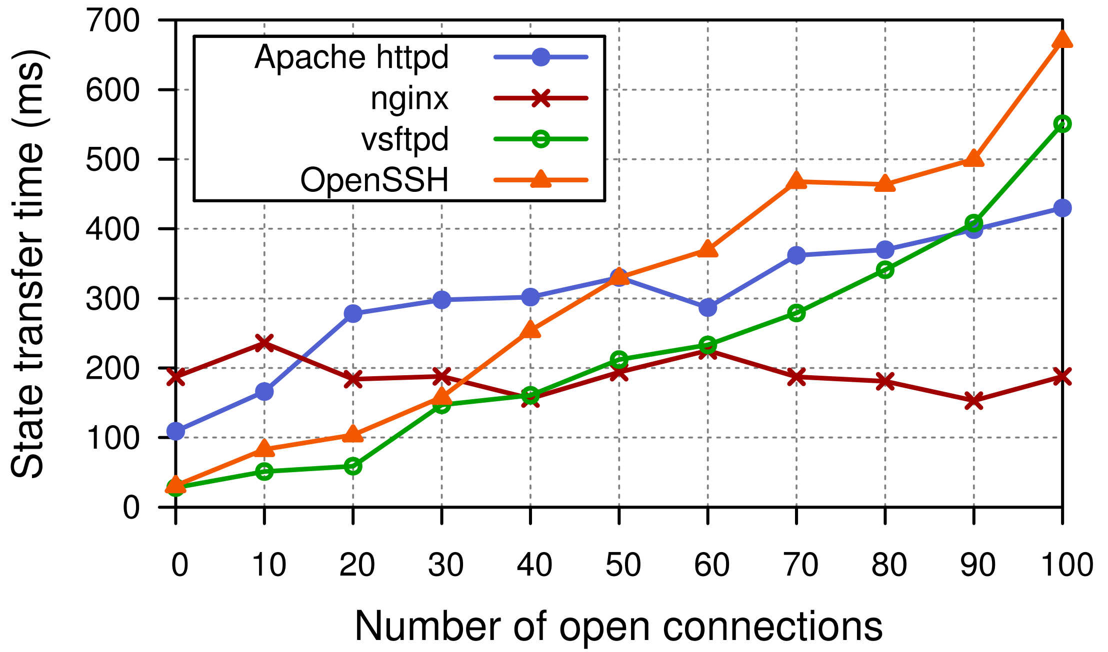
Normalized overhead of each instrumentation step.

- What about nginx$_{reg}$?
- 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 → <100ms
  - *initialization* time → < 50ms (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

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 ?