Practical Runtime Software Updates with Kitsune

Chris Hayden

Work done with: Mike Hicks, Jeff Foster, Ted Smith, and Michail Denchev
Apache 2.2.11 Released

The Apache HTTP Server Project is proud to announce the release of version 2.2.11 of the Apache HTTP Server ("Apache"). This version is principally a bugfix release.

This version of Apache is a major release and the start of a new stable branch, and represents the best available version of Apache HTTP Server. New features include Smart Filtering, Improved Caching, AJP Proxy, Proxy Load Balancing, Graceful Shutdown support, Large File Support, the Event MPM, and refactored Authentication/Authorization.
Oracle Critical Patch Update Advisory - April 2009

Description
A Critical Patch Update is a collection of patches for multiple security vulnerabilities. ...

Due to the threat posed by a successful attack, Oracle strongly recommends that customers apply fixes as soon as possible. This Critical Patch Update contains 43 new security fixes across all products.
**Oracle Critical Patch Update Advisory - April 2009**

**Description**
A Critical Patch Update is a collection of patches for multiple security vulnerabilities. ...

**Due to the threat posed by a successful attack, Oracle strongly recommends that customers apply fixes as soon as possible.** This Critical Patch Update contains 43 new security fixes across all products.
SW Upgrades: Disruptive

Oracle Critical Patch Update Advisory - April 2009

Description
A Critical Patch Update is a collection of patches for multiple security vulnerabilities. ... 

Due to the threat posed by a successful attack, Oracle strongly recommends that customers apply fixes as soon as possible. This update contains 43 new security fixes across all products.
Oracle Critical Patch Update Advisory - April 2009

Description
A Critical Patch Update is a collection of patches for multiple security vulnerabilities. ...

Due to the threat posed by a successful attack, Oracle strongly recommends that customers apply fixes as soon as possible. This update contains 43 new security fixes across all products.
SW Upgrades: Disruptive

Oracle Critical Patch Update Advisory - April 2009

Description
A Critical Patch Update is a collection of patches for multiple security vulnerabilities. ...

Due to the threat posed by a successful attack, Oracle strongly recommends that customers apply fixes as soon as possible. This Critical Patch Update contains 43 new security fixes across all products.
Dynamic Software Updating (DSU)

• Goal: Update programs *while they run*

• Why not stop and restart?
  ■ Preserve critical **Program State**
  ■ Make security patches, bug fixes, and new features available as soon as possible

• Useful for many applications:
  ■ Non-stop services: financial processing, air traffic control, network infrastructure
  ■ Long-running programs keeping in-memory state
    - operating systems, caching servers, in-memory database
Dynamic Software Updating (DSU)

- Run program at the old version
- At some point update to the new version, preserving and updating existing program state
- existing connections, important data on the stack and heap, program counter, ...
Dynamic Software Updating (DSU)

- Run program at the old version
- At some point update to the new version, preserving and updating existing **program state**
- existing connections, important data on the stack and heap, program counter, ...
Dynamic Software Updating (DSU)

- Run program at the old version
- At some point update to the new version, preserving and updating existing program state
- existing connections, important data on the stack and heap, program counter, ...
Dynamic Software Updating (DSU)

• Run program at the old version

• At some point update to the new version, preserving and updating existing **program state**

• existing connections, important data on the stack and heap, program counter, ...
Dynamic Software Updating (DSU)

- Run program at the old version
- At some point update to the new version, preserving and updating existing **program state**
  - existing connections, important data on the stack and heap, program counter, ...
Dynamic Software Updating (DSU)

- Run program at the old version
- At some point update to the new version, preserving and updating existing **program state**
  - existing connections, important data on the stack and heap, program counter, ...
Dynamic Software Updating (DSU)

- Run program at the old version
- At some point update to the new version, preserving and updating existing **program state**
  - existing connections, important data on the stack and heap, program counter, ...
Research DSU systems

• C/C++ application support
  ▪ PODUS, Ginseng, UpStare, POLUS, **Ekiden**, Gupta et al

• C/C++ operating systems
  ▪ K42, LUCOS, DynaMOS

• Java applications
  ▪ Jvolve, JDRUMS, Prose

• Other languages
  ▪ *Mod, DLpop, Dynamic ML
Many forms of DSU now mainstream
Many forms of DSU now mainstream

language run-times
Many forms of DSU now mainstream

language run-times

app. tools
Many forms of DSU now mainstream

- Erlang
- Java
- Smalltalk
- .NET
- APPLICATION ENHANCER
- LiveRebel
- Ksplice

language run-times, app. tools, OSes

Bought by Oracle in 2011
Practical DSU system features

- **Ease of Use**
  - Update behavior should be easy to reason about
  - Should be easy to retrofit existing programs
  - Minimize per-update programmer work

- **Flexibility**
  - Support natural program evolution, on-the-fly

- **Efficiency**
  - Quick update times
  - Little or no overhead on normal execution

- **Scalability**
  - Support large, real-world programs
Kitsune, a practical DSU system
Kitsune, a practical DSU system

• Updates the entire program at once
  ▪ A dynamic update “restarts” the program with the new code and the existing state
  ▪ Only requires simple source-to-source translator
    - Employs entirely standard compilation and tools
Kitsune, a practical DSU system

- Updates the entire program at once
  - A dynamic update “restarts” the program with the new code and the existing state
  - Only requires simple source-to-source translator
    - Employs entirely standard compilation and tools
- Update orchestration controlled by developer
  - Extra manual effort is worth it: easier to reason about
Kitsune, a practical DSU system

- Updates the entire program at once
  - A dynamic update “restarts” the program with the new code and the existing state
  - Only requires simple source-to-source translator
    - Employs entirely standard compilation and tools

- Update orchestration controlled by developer
  - Extra manual effort is worth it: easier to reason about

- New tool called xfgen generates code to transform the existing state
  - Based on simple developer-written specifications
Kitsune: Results

• Applied Kitsune to five open-source programs
  ▪ memcached, redis, icecast: 3-6 mos. of releases
  ▪ vsftpd, Tor: 2-4 years of releases

• Performance overhead in the noise

• Update times typically less than 40ms

• Programmer effort small
  ▪ < 100 LOC per program (largely one-time effort)
  ▪ ~ 100 LOC of xfgen specs across all releases
  ▪ All comparable to or less than prior systems
Kitsune build process

Summary:
• For each source file
  • replace gcc -c with composition of kitc and gcc
  • Add -shared flag to linker and include kit-rt.a
Kitsune updating model

- driver
Kitsune updating model

1. Load first version
Kitsune updating model

1. Load first version
2. Run it
Kitsune updating model

1. Load first version
2. Run it
Kitsune updating model

1. Load first version
2. Run it
3. Call back to driver when update ready
Kitsune updating model

1. Load first version
2. Run it
3. Call back to driver when update ready
4. Load second version
Kitsune updating model

1. Load first version
2. Run it
3. Call back to driver when update ready
4. Load second version
Kitsune updating model

1. Load first version
2. Run it
3. Call back to driver when update ready
4. Load second version
5. Migrate and transform state
Kitsune updating model

1. Load first version
2. Run it
3. Call back to driver when update ready
4. Load second version
5. Migrate and transform state
6. Free up old resources
Kitsune updating model

1. Load first version
2. Run it
3. Call back to driver when update ready
4. Load second version
5. Migrate and transform state
6. Free up old resources
7. Continue with new version
Programmer obligations
Programmer obligations

• Kitsune DSU requires the programmer to
  ▪ Identify where updates may take place
Programmer obligations

• Kitsune DSU requires the programmer to
  ▪ Identify where updates may take place
  ▪ Identify the state to be transferred, and where it should be received in the new code
Programmer obligations

• Kitsune DSU requires the programmer to
  ▪ Identify where updates may take place
  ▪ Identify the state to be transferred, and where it should be received in the new code
  ▪ Move the program counter to the right event loop when the new version restarts
Example single-threaded server

typedef int data;
data *mapping;

void client_loop(int fd) {
    while (1) {
        // ... process client requests
    }
}

int main() {
    int l_fd, cl_fd;
    mapping = malloc(...);
    l_fd = setup_conn();
    while (1) {
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
Example single-threaded server

typedef int data;
data *mapping;

void client_loop(int fd) {
    while (1) {
        // ... process client requests
    }
}

int main() {
    int l_fd, cl_fd;
    mapping = malloc(...);
    l_fd = setup_conn();
    while (1) {
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}

modification for Kitsune
Example single-threaded server

typedef int data;
data *mapping;

void client_loop(int fd) {
    while (1) {
        // ... process client requests
    }
}

int main () {
    int l_fd, cl_fd;

    mapping = malloc(...);

    l_fd = setup_conn();

    l_fd = setup_conn();

    while (1) {
        kitsune_update("main");
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }

    kitsune_update("main");
    cl_fd = get_conn(l_fd);
    client_loop(cl_fd);
}
typedef int data;
data *mapping;

void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main () {
    int l_fd, cl_fd;
    mapping = malloc(...);
    l_fd = setup_conn();

    while (1) {
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
typedef int data;
data *mapping; // automigrated

void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main() {
    int l_fd, cl_fd;
    kitsune_do_automigrate();
    mapping = malloc(...);
    l_fd = setup_conn();
    l_fd = setup_conn();
    while (1) {
        kitsune_update("main");
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
typedef int data;
data *mapping;  // automigrated
void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main () {
    int l_fd, cl_fd;
    kitsune_do_automigrate();
    if (!kitsune_is_updating()) {
        mapping = malloc(...);
    }

    l_fd = setup_conn();

    while (1) {
        kitsune_update("main");
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
typedef int data;
data *mapping; // automigrated

void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main NOTE_LOCALS () {
    int l_fd, cl_fd;
    kitsune_do_automigrate();
    if (!kitsune_is_updating()) {
        mapping = malloc(...);
    }

    l_fd = setup_conn();
}

while (1) {
    kitsune_update("main");
    cl_fd = get_conn(l_fd);
    client_loop(cl_fd);
}

modification for Kitsune
Example single-threaded server

typedef int data;
data *mapping; // automigrated

void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main NOTE_LOCALS () {
    int l_fd, cl_fd;
    kitsune_do_automigrate();
    if (!kitsune_is_updating()) {
        mapping = malloc(...);
    }
    if (!MIGRATE_LOCAL(l_fd)) {
        l_fd = setup_conn();
    }
    while (1) {
        kitsune_update("main");
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
typedef int data;
data *mapping; // automigrated

void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main NOTE_LOCALS () {
    int l_fd, cl_fd;
    kitsune_do_automigrate();
    if (!kitsune_is_updating()) {
        mapping = malloc(...);
    }
    if (!MIGRATE_LOCAL(l_fd)) {
        l_fd = setup_conn();
    }
    if (!kitsune_is_updating_from("client")) {
        MIGRATE_LOCAL(cl_fd);
    }
    while (1) {
        kitsune_update("main");
        cl_fd = get_conn(l_fd);
        client_loop(cl_fd);
    }
}
typedef int data;
data *mapping;  // automigrated

void client_loop(int fd) {
    while (1) {
        kitsune_update("client");
        // ... process client requests
    }
}

int main NOTE_LOCALS () {
    int l_fd, cl_fd;
    kitsune_do_automigrate();
    if (!kitsune_is_updating()) {
        mapping = malloc(...);
    }
    if (!MIGRATE_LOCAL(l_fd)) {
        l_fd = setup_conn();
    }
}

if (!kitsune_is_updating_from ("client") {  
    MIGRATE_LOCAL(cl_fd);
    client_loop(cl_fd);
}
while (1) {
    kitsune_update("main");
    cl_fd = get_conn(l_fd);
    client_loop(cl_fd);
}
Multithreading support

• All threads synchronize at update points
  ▪ Assumes threads will reach update points eventually
  ▪ Assumes they hold no resources (e.g., locks)
• Main thread restarts (as in the ST case)
• Restarts child threads at update points
  ▪ Hijack threads library to track pthread function, args
• Done when all threads reach their update points after restarting
Migrating and transforming state

• State may need to be transformed to work with the new program
  • Transformation piggybacks on top of migration
Migrating and transforming state

• State may need to be transformed to work with the new program
  ▪ Transformation piggybacks on top of migration

```c
typedef int data;
data *mapping;
```
Migrating and transforming state

- State may need to be transformed to work with the new program
  - Transformation piggybacks on top of migration

```c
typedef int data;
data *mapping;
```

```c
typedef char *data;
data *mapping;
```
Migrating and transforming state

• State may need to be transformed to work with the new program
  - Transformation piggybacks on top of migration

Migration

Xform

For each value \( x \) of type data in the running program and its corresponding location \( p \) in the new program

do
  \[ *p = malloc(N); \]
  \[ sprintf(*p,N,"%d",x); \]
end
Migrating and transforming state

- State may need to be transformed to work with the new program
  - Transformation piggybacks on top of migration

```c
typedef int data;
data *mapping;

typedef char *data;
data *mapping;
```

Migration
```c
new::mapsz = old::mapsz;
new::mapping = malloc(new::mapsz*sizeof(char*));
for (int i=0;i<new::mapsz;i++) {
    old::data x = old::mapping[i];
    new::data *p = &new::mapping[i];
    *p = malloc(N);
    snprintf(*p,N,"%d",x);
}
```

Xform
Migrating and transforming state

• State may need to be transformed to work with the new program
  ▪ Transformation piggybacks on top of migration

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef int data; data *mapping;</td>
<td>typedef char *data; data *mapping;</td>
</tr>
</tbody>
</table>

**Xfgen tool**
• Automate migration code
• Require programmer to write *relevant* xform code using high-level specs
xfgen specs
**xfgen specs**

- **INIT tgt: { action }**
  - Initializes *new* variable or values of a new type
    - `tgt` is a global/local variable, or a C type or a struct field
    - `action` is C-like code for initializing it
**xfgens specs**

- **INIT** `tgt: { action }`
  - Initializes *new variable or values of a new type*
    - `tgt` is a global/local variable, or a C type or a struct field
    - `action` is C-like code for initializing it
- `tgt → tgt: { action }`
  - Copies and transforms *changed variables/values*
    - `action` may refer to old value when initializing the new one
**xfgen specs**

- **INIT** \( tgt: \{ \text{action} \} \)
  - Initializes *new* variable or values of a new type
    - \( tgt \) is a global/local variable, or a C type or a struct field
    - \( \text{action} \) is C-like code for initializing it

- \( tgt \rightarrow tgt: \{ \text{action} \} \)
  - Copies and transforms *changed* variables/values
    - \( \text{action} \) may refer to old value when initializing the new one

- xfgen generates C code from these specs and integrates it with generated migration code
Example 1

old

int op_count;
int set_count;
int get_count

new
Example 1

\begin{itemize}
\item \textbf{old} \hspace{1cm} \textbf{new}
\item int \texttt{op\_count};
\item int \texttt{set\_count};
\item int \texttt{get\_count}
\end{itemize}

\begin{itemize}
\item INIT \texttt{get\_count}: \{ $\text{out} = \text{floor}(\text{oldsym}(\text{op\_count}) / 2); \} $
\item INIT \texttt{set\_count}: \{ $\text{out} = \text{ceiling}(\text{oldsym}(\text{op\_count}) / 2); \} $
\end{itemize}

\textit{xfgen spec}
Example 1

**old**

- `int op_count;`
- `int set_count;`
- `int get_count`

**new**

- `int op_count;`
- `int set_count;`
- `int get_count`

**INIT get_count:**

```c
$out = floor($oldsym(op_count) / 2);
```

**INIT set_count:**

```c
$out = ceiling($oldsym(op_count) / 2);
```

**xfgen spec**

```c
void _kitsune_transform_get_count() {
    int *old_op_count = (int*) kitsune_lookup_key_old("op_count");
    int *new_get_count = (int*) kitsune_lookup_key_new("get_count");
    *new_get_count = floor(*old_op_count / 2);
}
```

```c
void _kitsune_transform_set_count() { /* as above */ }
```

**generated code**
Example 2

old

typedef int data;
struct list {
    int key;
data *val;
    struct list *next;
} *mapping;

new

typedef long data;
struct list {
    int key;
data *val;
    int cid;
    struct list *pnext;
} *mapping;
Example 2

typedef int data;
struct list {
    int key;
data *val;
    struct list *next;
} *mapping;

typedef long data;
struct list {
    int key;
data *val;
    int cid;
    struct list *pnext;
} *mapping;

typedef data -> typedef data: { $out = (long)$in; } 
INIT struct list.cid { $out = -1; }
struct list.next -> struct list.pnext

xfgen spec
Example 3

old

\textbf{data **mapping;}

\textbf{new}

\textbf{struct list \{}
  \textbf{int key;}
  \textbf{data *val;}
  \textbf{struct list *next;}
\textbf{\} *mapping;}

\textbf{Example 3}
Example 3

**old**

```c
data **mapping;

mapping -> mapping {
    int key;
    $out = NULL;
    for(key = 0; key < $oldsym(config_size); key++) {
        if ($in[key] != 0) {
            $newtype(struct list) *cur =
                malloc(sizeof($newtype(struct list)));
            cur->key = key;
            cur->val = $in[key];
            cur->next = $out;
        }
    }
}
```

**new**

```c
struct list {
    int key;
    data *val;
    struct list *next;
} *mapping;
```

*xfgen spec*
**xfgen C extensions**

- Convenient syntax to hide complexity of dealing with two versions:
  - `$out`, `$in` - address of input/output of rule
  - `$newsym(name)`, `$oldsym(name)` - return address of new/old version variable or function
  - `$newtype(type name)`, `$oldtype(type name)` - refer to the new or old types from xfgen code
  - `$xform(old_type, new_type)` - returns the address of a function to transform between two types
vsftpd transformation example

standalone.c/s_p_ip_count_hash -> standalone.c/s_p_ip_count_hash:

```c
$oldtype(struct hash) *in_hash = ($oldtype(struct hash) *)$in;
int i;
$out = ($newtype(struct hash) *)hash_alloc(256, sizeof($newtype(struct in6_addr)),
        sizeof(unsigned int), $newsym(hash_ip));

/* for each bucket */
for (i=0; i<in_hash->buckets; i++) {
    $oldtype(struct hash_node) *old_p_node = in_hash->p_nodes[i];
    /* for each bucket element, rehash */
    while(old_p_node != NULL) {
        void *ip = xform_ip_address(old_p_node->p_key);
        hash_add_entry(($newtype(struct hash) *)$out, ip, old_p_node->p_value);
        free(ip);
        old_p_node = old_p_node->p_next;
    }
}
```
Using Kitsune and xfgengen

-需要用 Kitsune 和 xfgengen

1. 使用 Kitsune 和 xfgengen
2. 用 Kitsune 和 xfgengen
Using Kitsune and xfgeng

- Transformation specs in per-update .xf file
- Linked in with new version and invoked by kitsune_do_automigrate() and MIGRATE_LOCAL()
Annotations

• To generate traversal code, we look at the types
Annotations

• To generate traversal code, we look at the types
• Add annotations to help code generator
  ▪ KS_PTRARRAY(S) – size of pointed-to array

```c
struct buffer {
  int n;
  data *KS_PTRARRAY(n) b;
};
```
Annotiations

- To generate traversal code, we look at the types
- Add annotations to help code generator
  - **KS_PTRARRAY(S)** – size of pointed-to array
  - **KS_ARRAY(S)** – size of array

```c
struct inlinebuf {
    int n;
    data KS_ARRAY(n) b[];
};
```
Annotations

• To generate traversal code, we look at the types
• Add annotations to help code generator
  ▪ KS_PTRARRAY(S) – size of pointed-to array
  ▪ KS_ARRAY(S) – size of array
  ▪ KS_OPAQUE – non-traversed pointer
Annotations

- To generate traversal code, we look at the types
- Add annotations to help code generator
  - **KS_PTRARRAY(S)** – size of pointed-to array
  - **KS_ARRAY(S)** – size of array
  - **KS_OPAQUE** – non-traversed pointer
  - **KS_FORALL(@t)** – polymorphism intro.
  - **KS_VAR(@t)** – refer to type var
  - **KS_INST(typ)** – instantiate poly. type
Polymorphic types

template<T>
struct list {
    T *val;
    list<T> *next;
} C++

struct list {
    void *val;
    struct list *next;
}; C
Polymorphic types

```
template<T>
struct list {
    T *val;
    list<T> *next;
} C++

struct list {
    void *val;
    struct list *next;
}; C

struct list {
    void KS_VAR(@t) *val;
    struct list KS_INST(@t) *next;
} KS_FORALL(@t);
```

annotated C
Polymorphic types

- Given such a definition, xfgen can generate transformers for instances of type struct list
  - Given struct list KS_INST(data) *l, can apply transformation to generic type:
    - data -> data { $out = ... $in ... }

```c
struct list {  
    void *val;  
    struct list *next;  
};
```

```c++
template<T>  
struct list {  
    T *val;  
    list<T> *next;  
};
```
Benchmark programs

• **Very secure FTP daemon** - file transfers, securely
• **Redis** - key/value server
• **Tor** - anonymous routing daemon
• **Memcached** - caching daemon
• **Icecast** - streaming music server
Steady state performance overhead

<table>
<thead>
<tr>
<th>Program</th>
<th>Orig (siqr)</th>
<th>Kitsune</th>
<th>Ginseng</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsftpd 2.0.6</td>
<td>6.55s (0.04s)</td>
<td>+0.75%</td>
<td>–</td>
</tr>
<tr>
<td>memcached 1.2.4</td>
<td>59.30s (3.25s)</td>
<td>+0.51%</td>
<td>–</td>
</tr>
<tr>
<td>redis 2.0.4</td>
<td>46.83s (0.40s)</td>
<td>-0.31%</td>
<td>–</td>
</tr>
<tr>
<td>icecast 2.3.1</td>
<td>10.11s (2.27s)</td>
<td>-2.18%</td>
<td>–</td>
</tr>
<tr>
<td><strong>32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsftpd 2.0.3</td>
<td>5.71s (0.01s)</td>
<td>+1.79%</td>
<td>+8.05%</td>
</tr>
<tr>
<td>memcached 1.2.4</td>
<td>101.40s (0.35s)</td>
<td>-0.49%</td>
<td>+18.44%</td>
</tr>
<tr>
<td>redis 2.0.4</td>
<td>43.88s (0.16s)</td>
<td>-1.21%</td>
<td>–</td>
</tr>
<tr>
<td>icecast 2.3.1</td>
<td>35.71s (0.68s)</td>
<td>+1.18%</td>
<td>-0.28%</td>
</tr>
</tbody>
</table>

- Overall: -2.18% to 1.79% overhead (in the noise)
- Also: UpStare (different platform): 4.9% and 7.4% for vsftpd
# Update times

<table>
<thead>
<tr>
<th>Program</th>
<th>Med. (siqr)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsftpd → 2.0.6</td>
<td>2.99ms (0.04ms)</td>
<td>2.62</td>
<td>3.09</td>
</tr>
<tr>
<td>memcached → 1.2.4</td>
<td>2.50ms (0.05ms)</td>
<td>2.27</td>
<td>2.68</td>
</tr>
<tr>
<td>redis → 2.0.4</td>
<td>39.70ms (0.98ms)</td>
<td>36.14</td>
<td>82.66</td>
</tr>
<tr>
<td>icecast → 2.3.1</td>
<td>990.89ms (0.95ms)</td>
<td>451.73</td>
<td>992.71</td>
</tr>
<tr>
<td>icecast-nsp → 2.3.1</td>
<td>187.89ms (1.77ms)</td>
<td>87.14</td>
<td>191.32</td>
</tr>
<tr>
<td>tor → 0.2.1.30</td>
<td>11.81ms (0.12ms)</td>
<td>11.65</td>
<td>13.83</td>
</tr>
</tbody>
</table>

- < 40ms in all cases but icecast
Update times, by state size

- Key difference is data representation: arrays vs. nested lists with pointers to static memory
Conclusions

• My research has aimed to achieve practical DSU
• Specification, testing, and verification work helps define/check DSU correctness
• Kitsune DSU system:
  ▪ Applied to 35 updates of 5 real applications
  ▪ No performance overhead on regular execution
  ▪ Little work for the programmer

Questions?