Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction

Kristis Makris   Rida A. Bazzi
{makristis,bazzi}@asu.edu

Ira A. Fulton School of Engineering
ARIZONA STATE UNIVERSITY

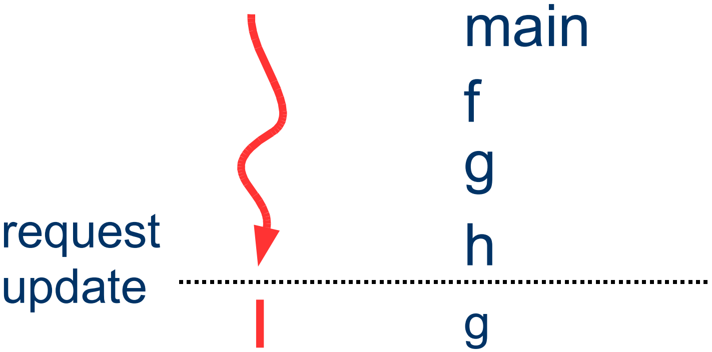
Motivation

- Software update problem: replace old version with new version

- Traditional approach is static:
  - stop, update, restart
  - Impairs high-availability

- Dynamic software update (DSU) can help minimize downtime

Execution trace



main

f

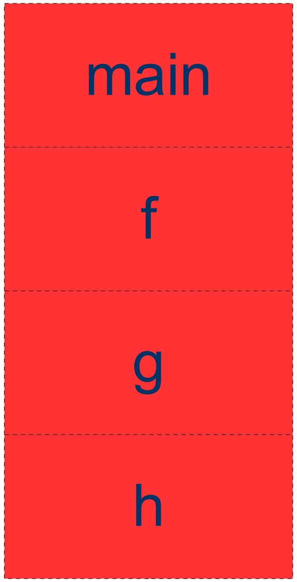
g

h

request update

g

Stack



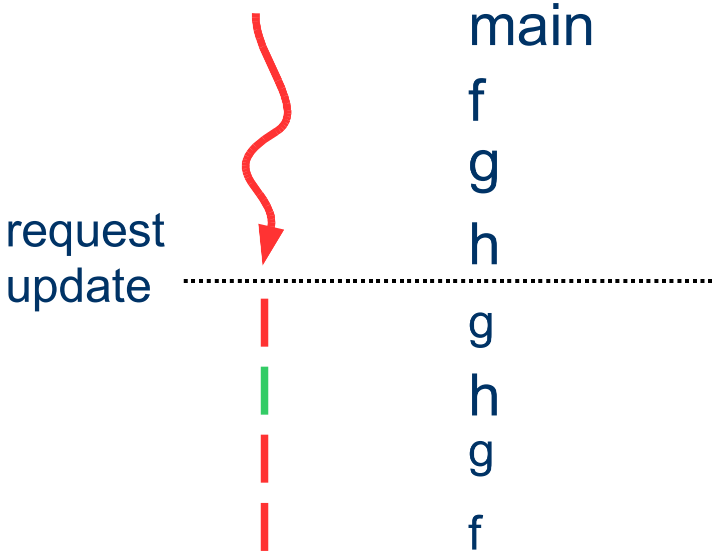
main

f

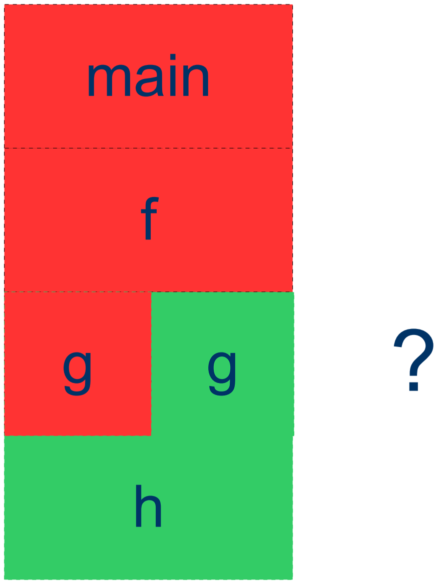
g

h

Execution trace



Stack



- Type-safety: No old code executed on new data; and vice versa

old version

```
typedef struct {
   char name[64];
   float balance;
 } customer_record_v1_t;
```

new version

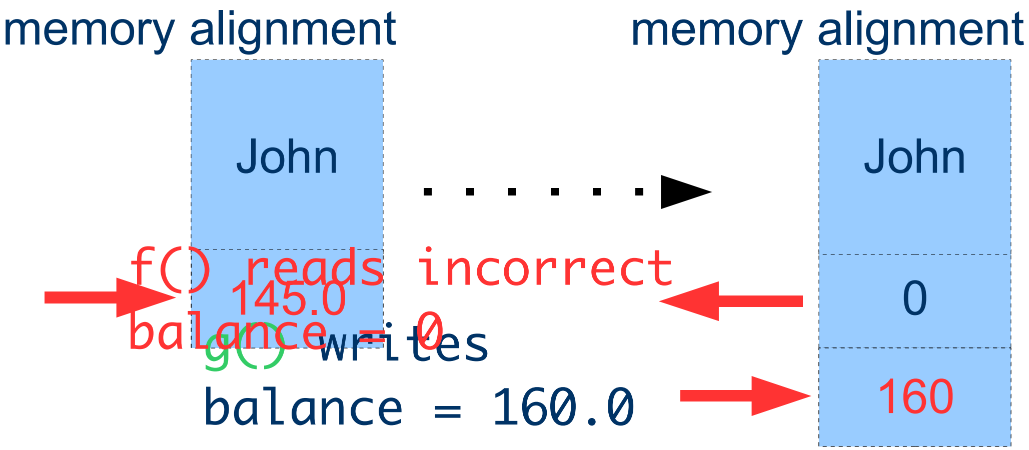
```
typedef struct {
   char name[64];
   int number_of_accesses;
   float balance;
} customer_record_v2_t;
```

g() is called

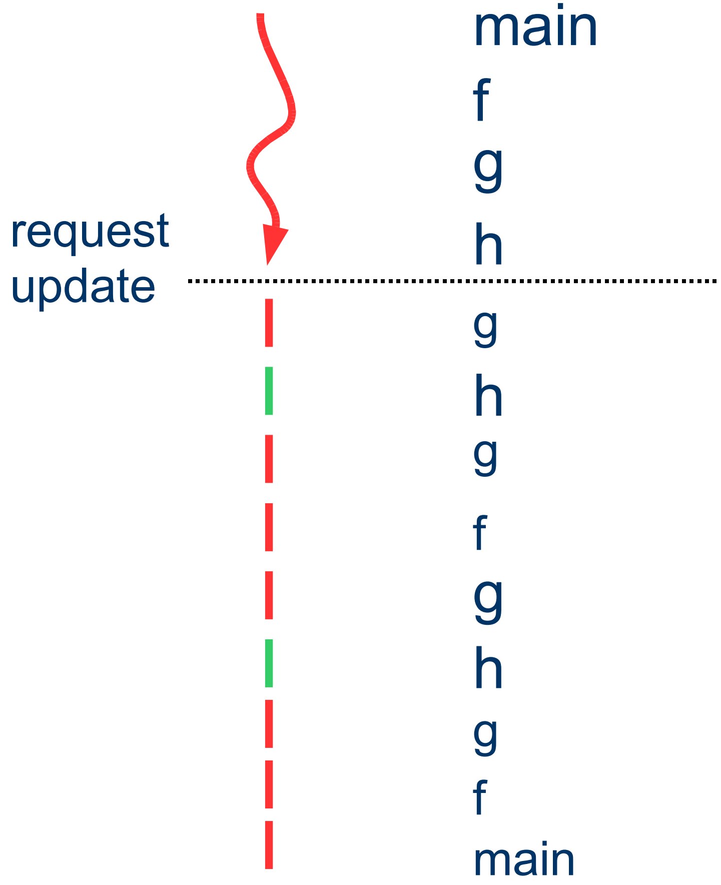
data are transformed

g() returns; f() executes

```
f() writes
balance = 145.0
```



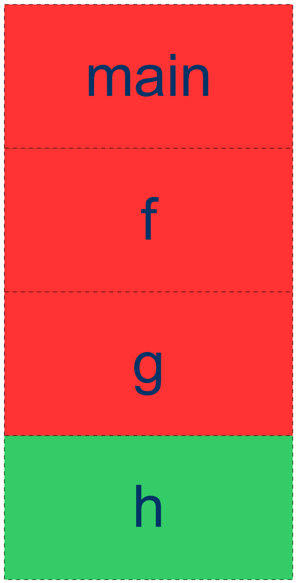
Execution trace



request update

main
f
g
h
g
h
g
f
g
h
g
f
main

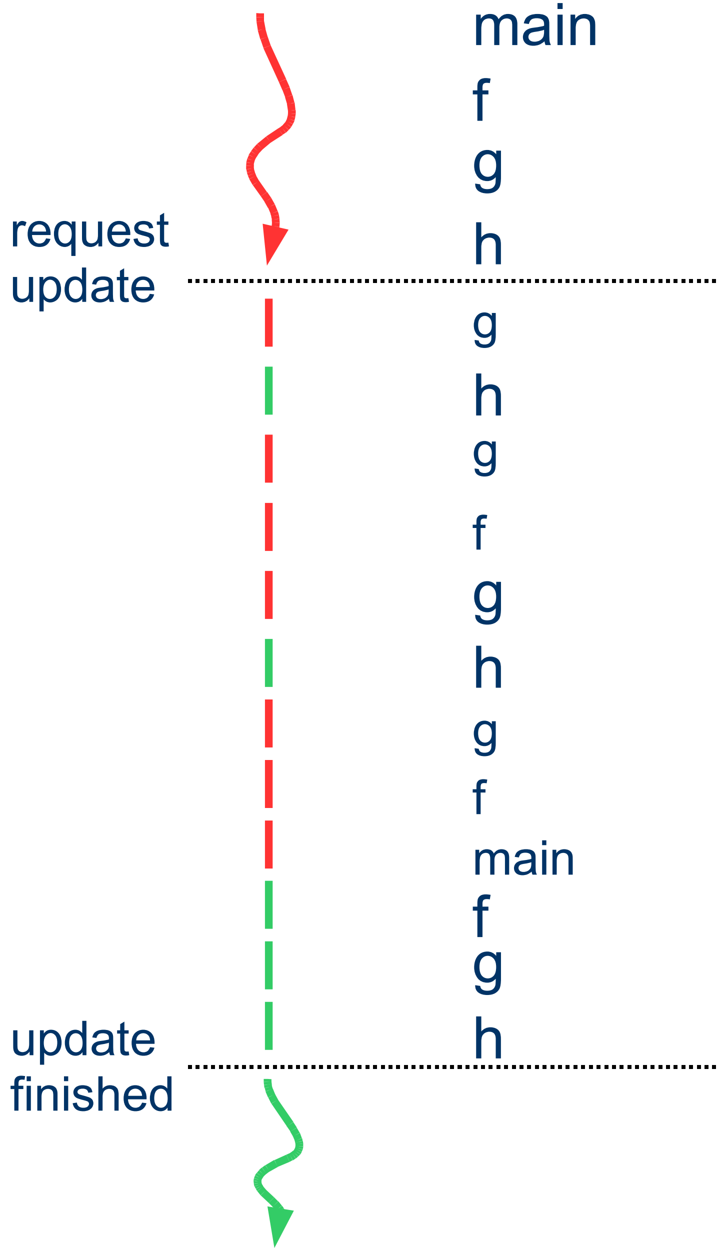
Stack



main
f
g
h

- Is old version in valid state ?
- Is there a valid mapping ?
- Undecidable problem
  - Need user input
- Should provide useful safety guarantees

Execution trace



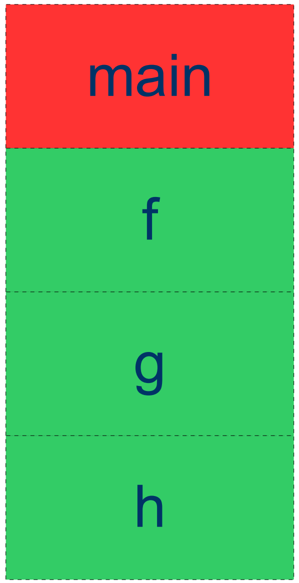
main
f
g
h

request update

g
h
g
f
g
h
g
f
main
f
g
h

update finished

Stack



main
f
g
h

- Undecidable problem
  - Need user input
- Should provide useful safety guarantees

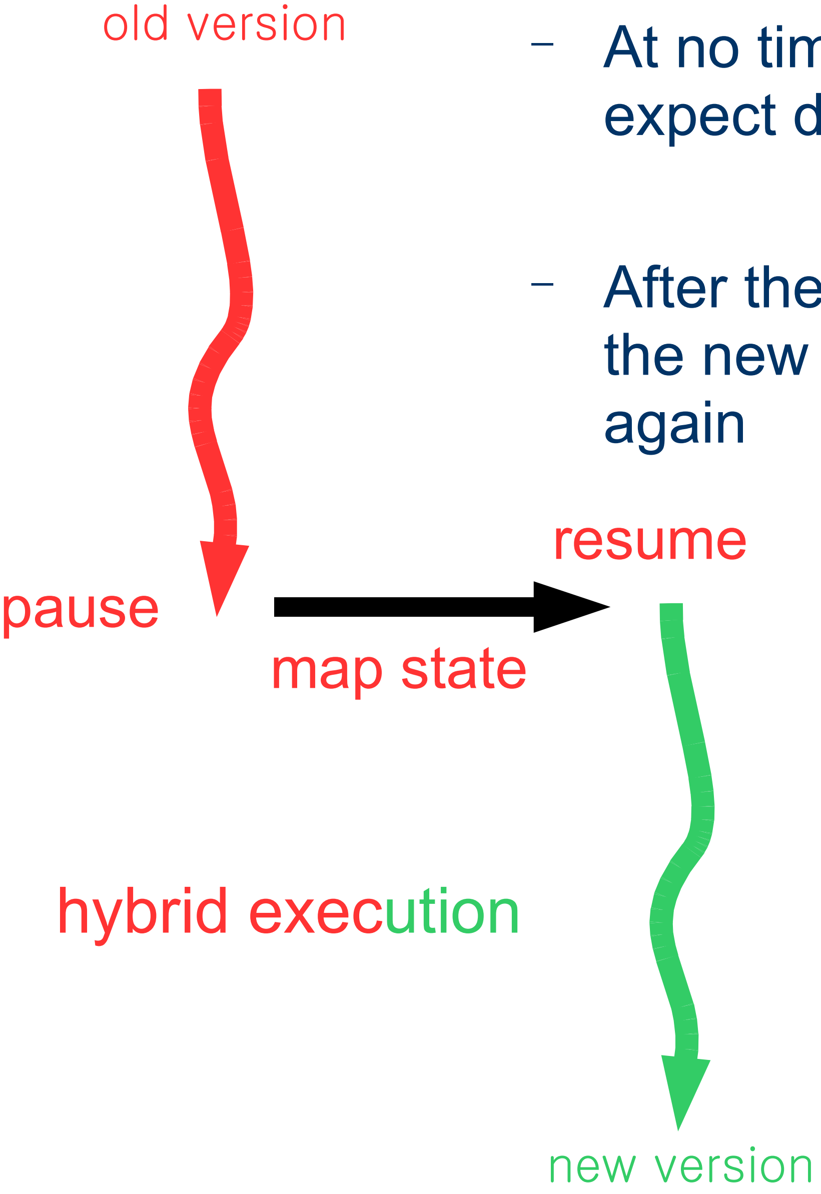
Useful DSU Safety Guarantees

- Atomic update (subsumes type-safety)
- Transaction-safety
- Thread-safety

Atomic Update

- At no time does the executing application expect different representations of state

- After the update only new code executes over the new state; no old code ever executes again



old version

pause

map state

resume

hybrid execution

new version

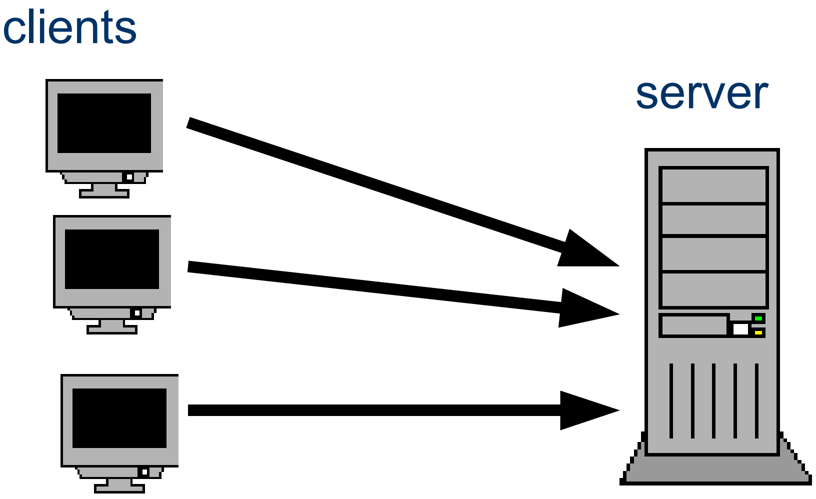
Transaction-safety

- Some code executes only in old or only in new version
- Requires user annotations

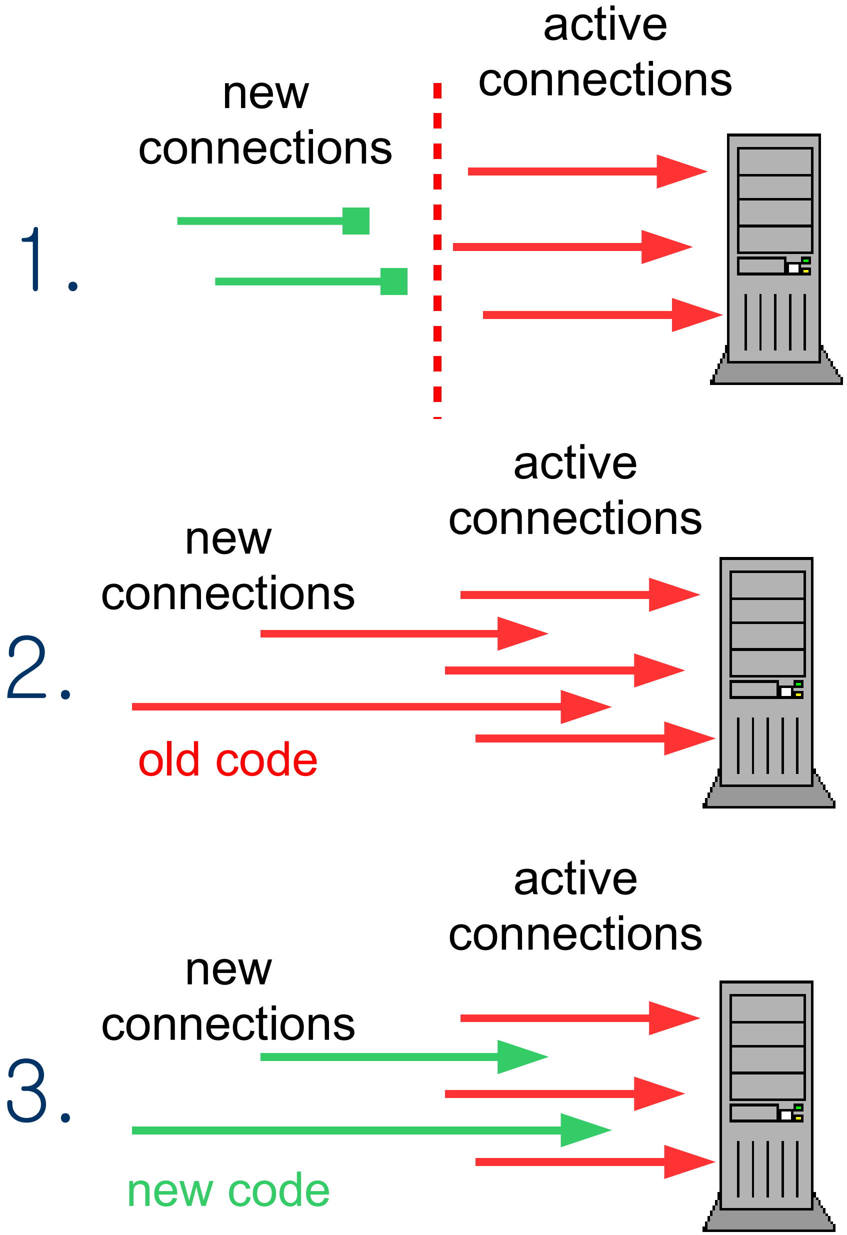
```
f() {
  ...
  while(condition) {
   i();
   j();
   k();
  }
  ...
}
```

do not update inside region

Thread-safety



```
while (condition) {
  recv(&data);
  process(&data);
}
```



Providing thread-safety requires immediate updates

- Atomic update
- Bounded delay

Existing DSU mechanisms do not provide support for immediate updates

Our Results

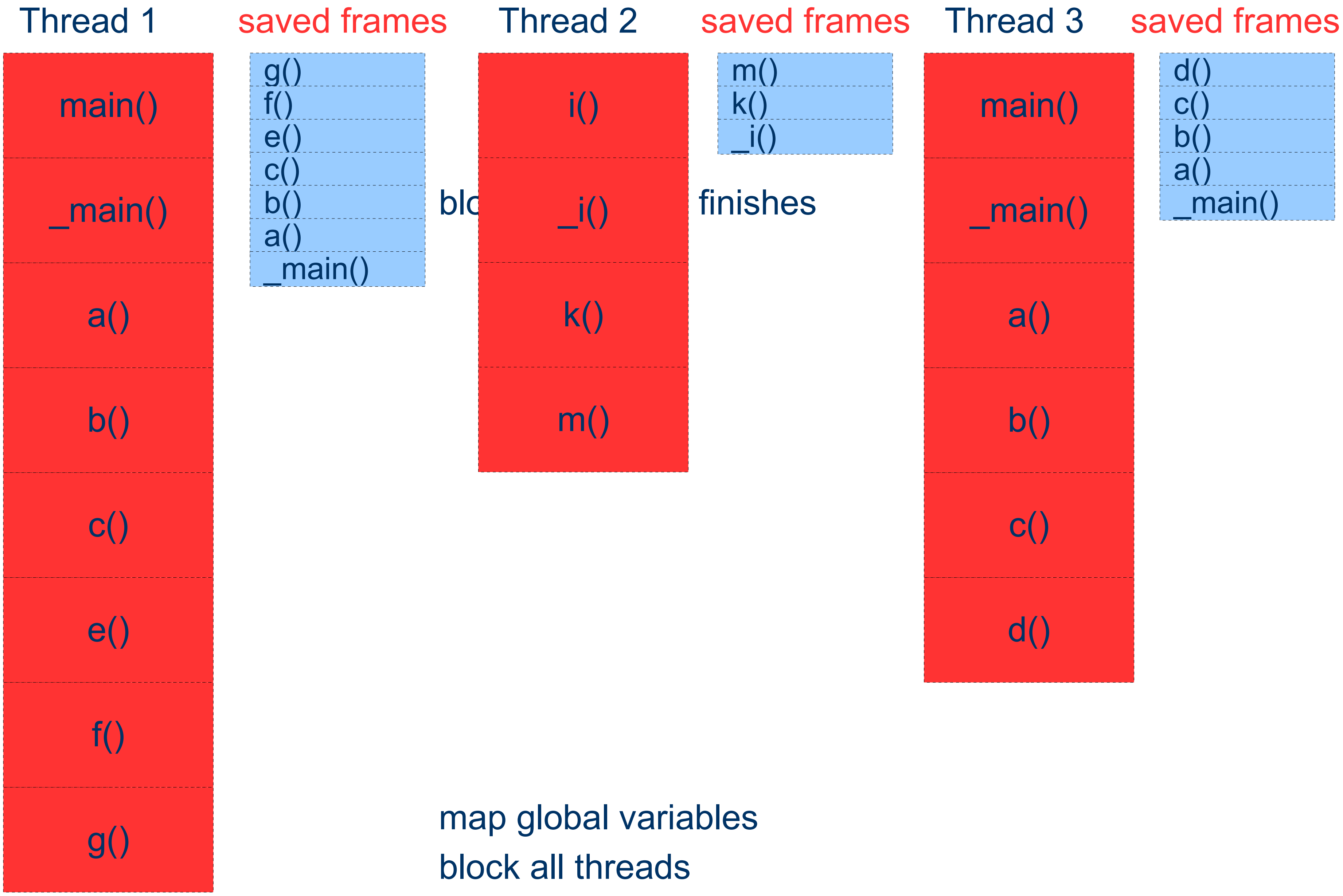
- First general DSU mechanism that supports
  - Immediate updates
    - Atomic update
    - Bounded delay
    - Multi-threaded
  - Update active code and data
  - Low data-access overhead

Our Approach: UpStare

- Compiler, patch-generator, runtime
  - Insert update points
  - Source-to-source transformations of C programs
  - Architecture and OS independent

- Immediate multi-threaded updates
  - Atomic update: using stack reconstruction
  - Bounded delay: converting blocking calls to non-blocking
  - Multithreaded: safely blocking all threads

Stack Reconstruction: unrolling



Thread 1

main()
_main()
a()
b()
c()
e()
f()
g()

saved frames

g()
f()
e()
c()
b()
a()
_main()

blc

Thread 2

i()
_i()
k()
m()

saved frames

m()
k()
_i()

finishes

Thread 3

main()
_main()
a()
b()
c()
d()

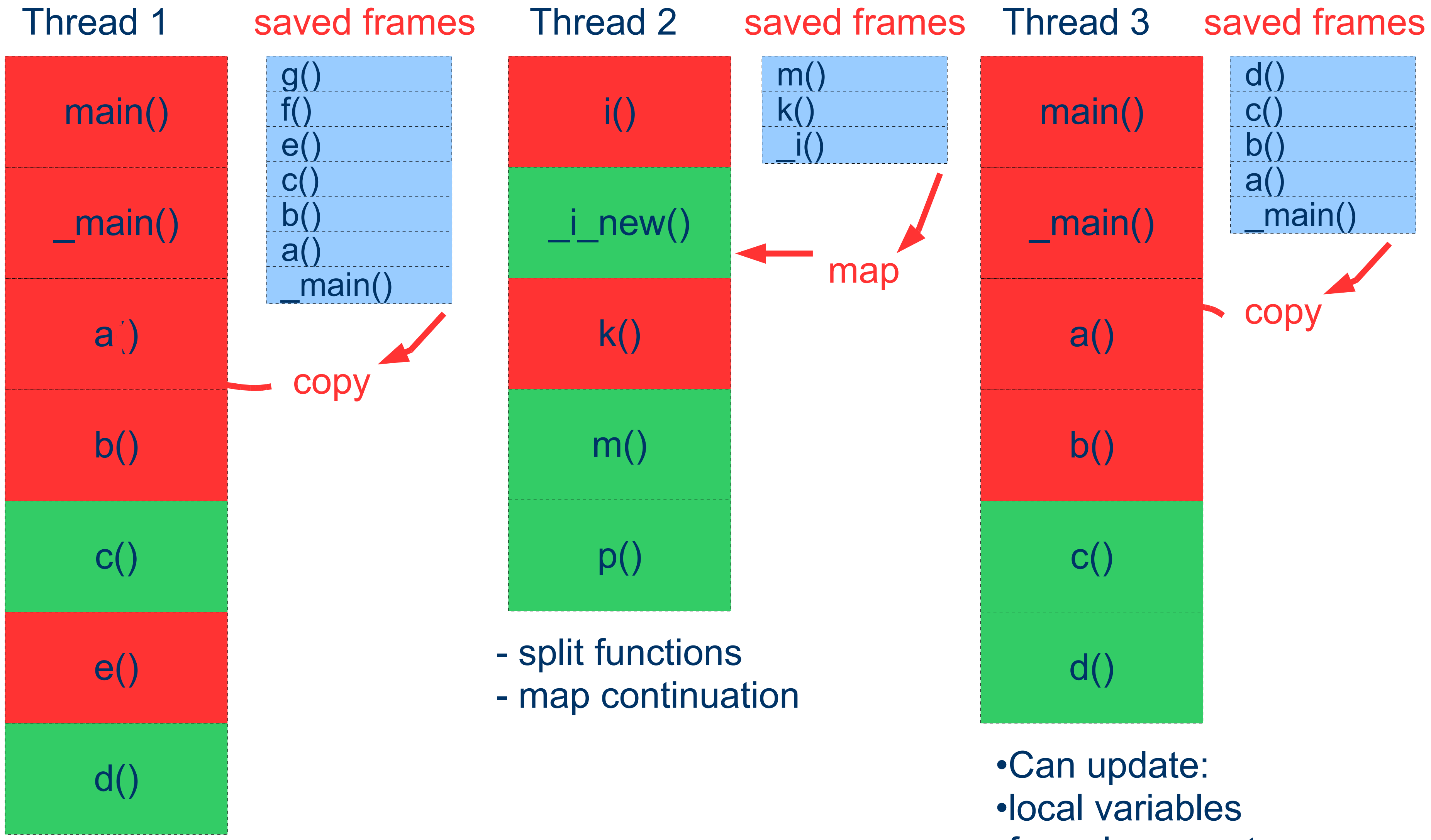
saved frames

d()
c()
b()
a()
_main()

map global variables

block all threads

Stack Reconstruction: restoring



Thread 1

main()
_main()
a()
b()
c()
e()
d()

saved frames

g()
f()
e()
c()
b()
a()
_main()

copy

Thread 2

i()
_i_new()
k()
m()
p()

saved frames

m()
k()
_i()

map

Thread 3

main()
_main()
a()
b()
c()
d()

saved frames

d()
c()
b()
a()
_main()

copy

- merge functions together

- split functions
- map continuation

apply stack transformers

- Can update:
- local variables
- formal parameters
- return addresses
- Program Counter

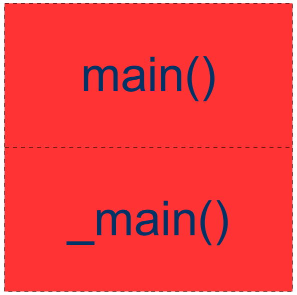
Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
```

old version

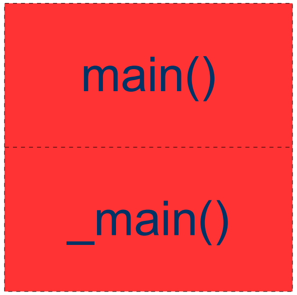


Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
  }
```



main()

_main()

old version

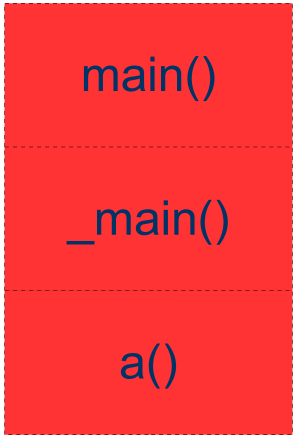
Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
```

old version



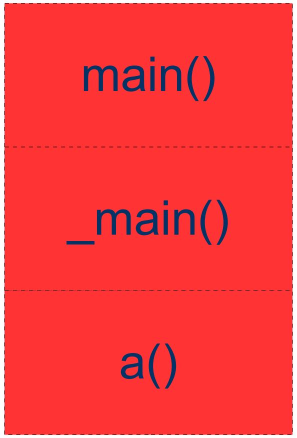
Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
```

old version



main()

_main()

a()

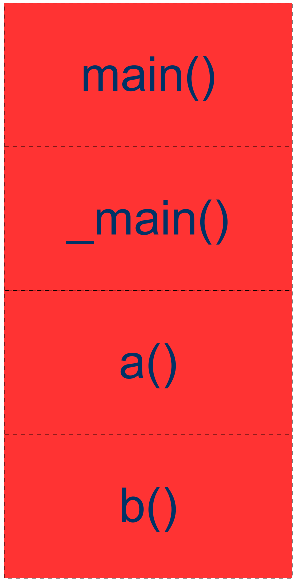
Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
```

old version


main()

_main()

a()

b()

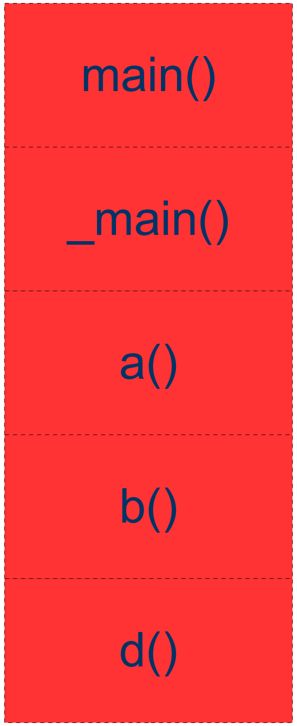
Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
```

old version



main()

_main()

a()

b()

d()

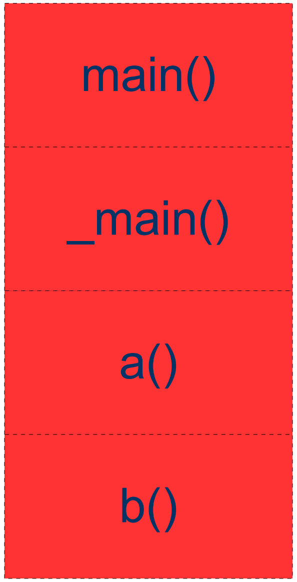
Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();
  d();
  while(condition) {
   UPDATE_POINT();
   e();
```

old version


main()

_main()

a()

b()

Continuation Points

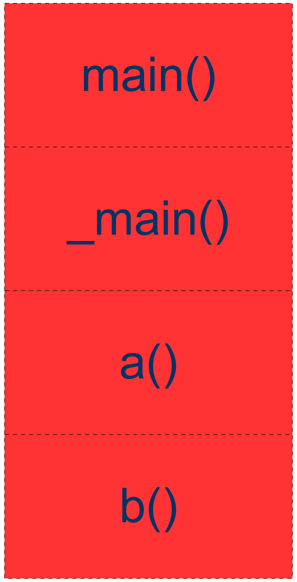
```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}

b() {
  UPDATE_POINT();  // CP 1
  d();             // CP 2
  while(condition) {
   UPDATE_POINT(); // CP 3
   e();            // CP 4
```

initiate an update

old version



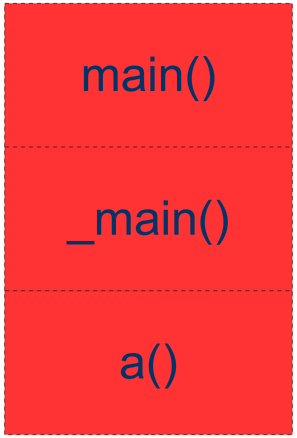
Saved continuation points:

b_CP_3

Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();  // CP 1
 b();             // CP 2
}
```

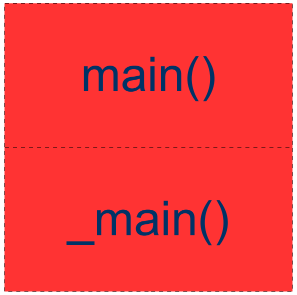

main()
_main()
a()

Saved continuation points:

b_CP_3
a_CP_2

Continuation Points

```
main() {
  UPDATE_POINT(); // CP 1
  a();            // CP 2
  c();            // CP 3
  g();            // CP 4
}
```



main()

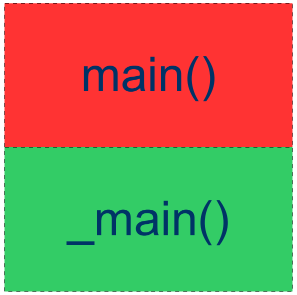
_main()

Saved continuation points:

b_CP_3
a_CP_2
_main_CP_2

Continuation Points

```
main() {
  UPDATE_POINT();
  a();              // CP 2
  c();
  g();
}
```



main()

_main()

<u>Saved continuation points:</u>

b_CP_3
a_CP_2
_main_CP_2

<u>Restored continuation points:</u>

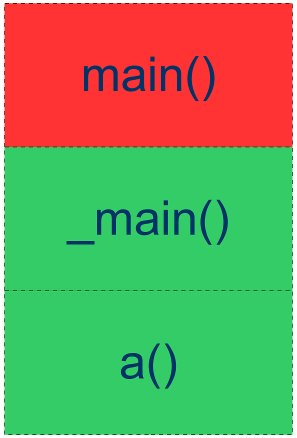
_main_CP_2

new version

Continuation Points

```
main() {
  UPDATE_POINT();
  a();            // CP 2
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();             // CP 2
}
```


main()

_main()

a()

Saved continuation points:

b_CP_3
a_CP_2
_main_CP_2

Restored continuation points:

_main_CP_2
a_CP_2

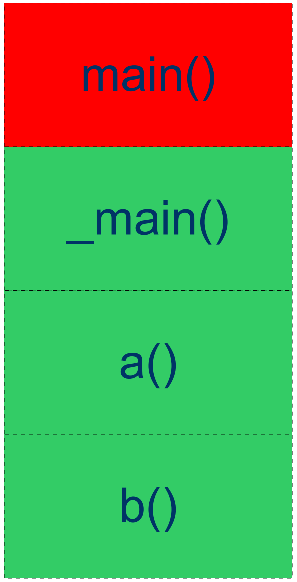
new version

Continuation Points

```
main() {
  UPDATE_POINT();
  a();
  c();
  g();
}

a() {
 UPDATE_POINT();
 b();
}
b() {
  UPDATE_POINT();  // CP 1
  d();             // CP 2
  f();             // CP 3
  while(condition) {
   UPDATE_POINT(); // CP 4
   e();            // CP 5
```

new version



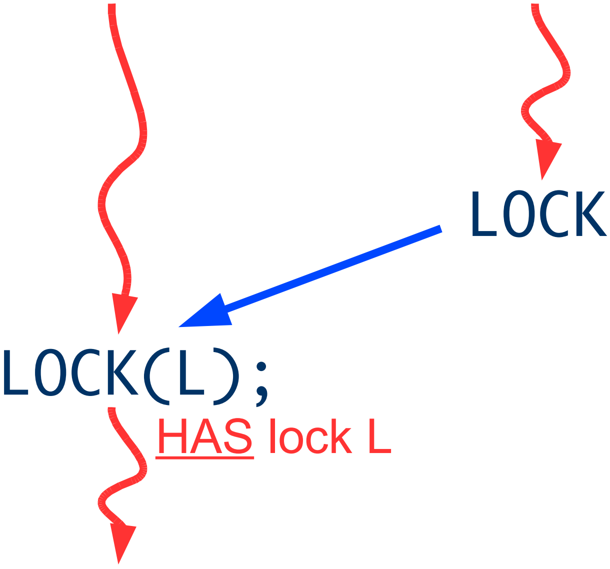
Saved continuation points:

b_CP_3
a_CP_2
_main_CP_2

Restored continuation points:

_main_CP_2
a_CP_2
b_CP_4

Multi–Threaded Updates



Thread 1

Thread 2

LOCK(L);

WANTS lock L

LOCK(L);

HAS lock L

UPDATE_POINT();

voluntarily blocked

blocked

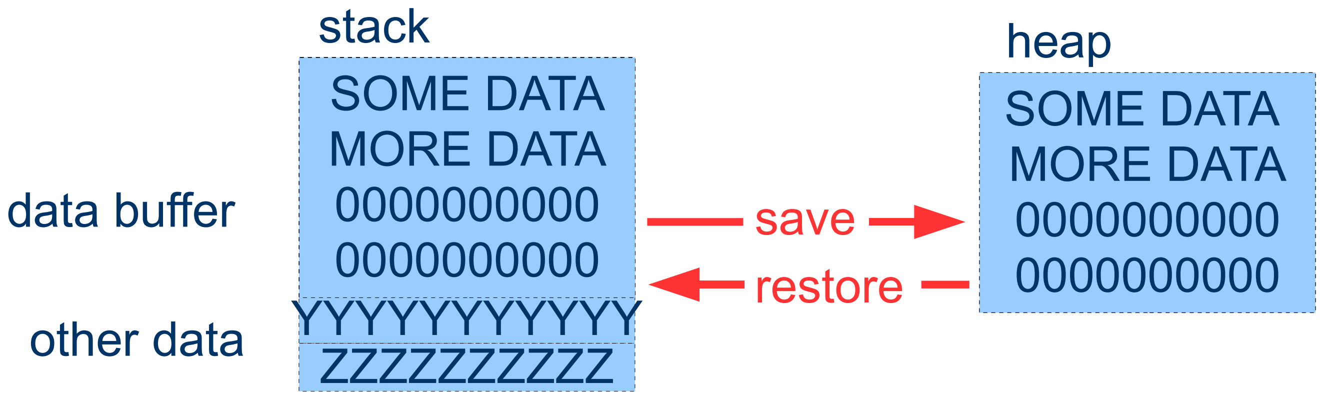
- Detect if all threads are blocked
- Treat locks as update points

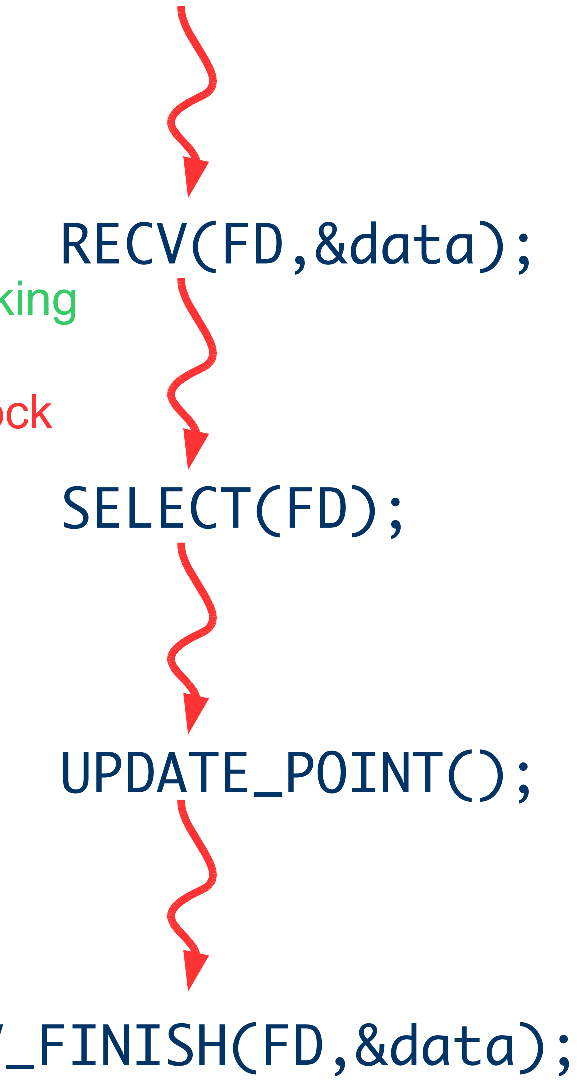
Multi-Process Updates

- wrap fork(), wait(), waitpid()
- coordinate atomic reconstruction

Bounded Delay

```
functionA() {
 char data[SIZE];
 ...
 recv(FD, &data);
 ...
}
```



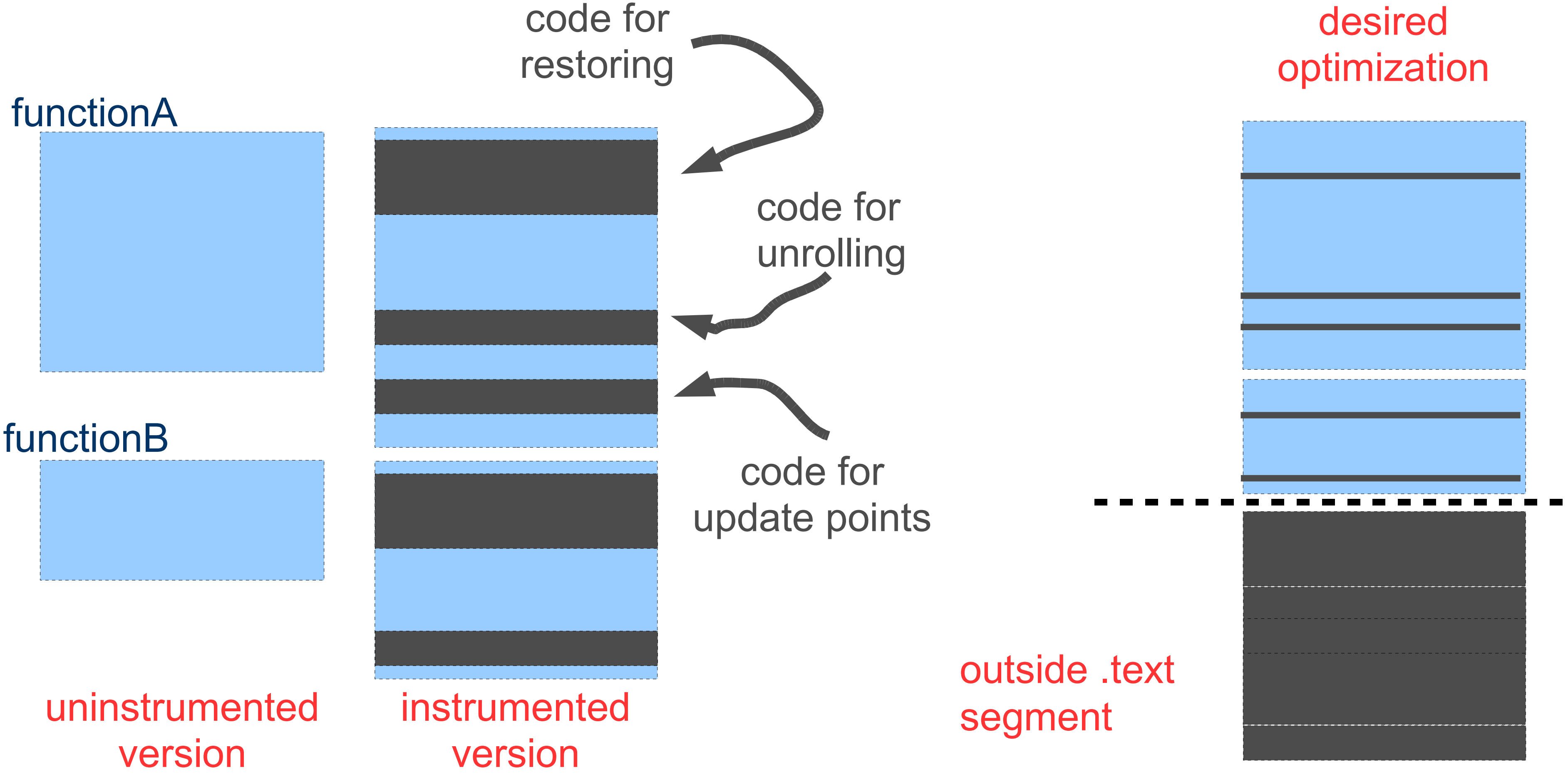


Evaluation

- KissFFT
  - Small, data-intensive application (2,000 LoC)
- Very Secure FTP Daemon
  - Medium-sized application (12,000 LoC)
  - Forks non-communicating connection handlers
- PostgreSQL DBMS
  - Large application (postmaster: 225,000 LoC)
  - Forks communicating connection handlers
    - Shared Memory

KissFFT v1.2.0

- Overhead
  - Execution time: 38%; faster than Ginseng (150%)

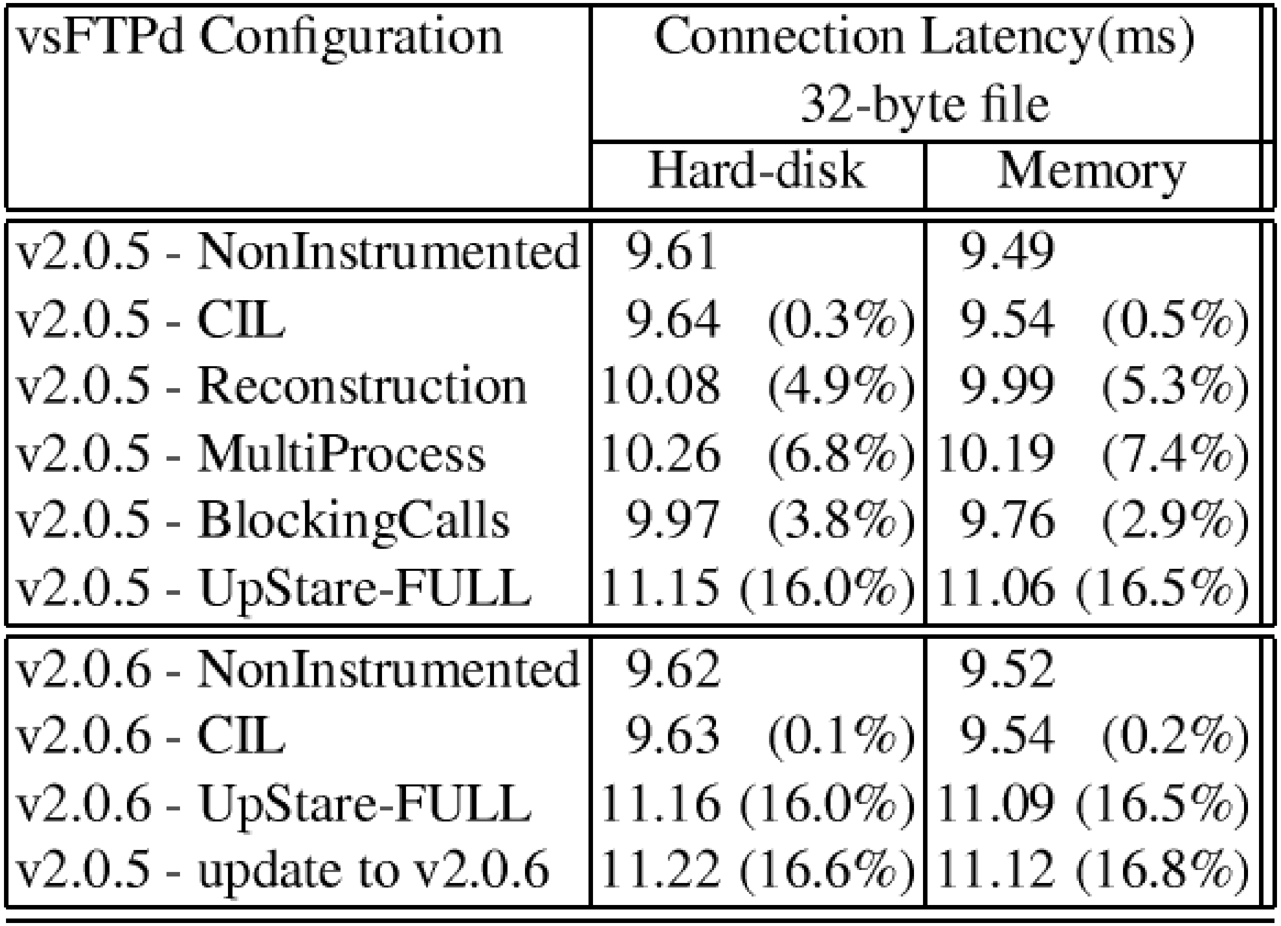


Very Secure FTP Daemon

- Updates
  - Under two use cases: idle client, file transfer
  - 13 updates (5.5 years-worth)
  - 11 manual continuation mappings
  - Latency 60ms (50ms to block all threads)

- Overhead
  - Latency: retrieve a 32-byte file 1000 times
  - Throughput: retrieve a 300MB file
  - In-memory and on-disk, over cross-over cable

vsFTPd Overhead



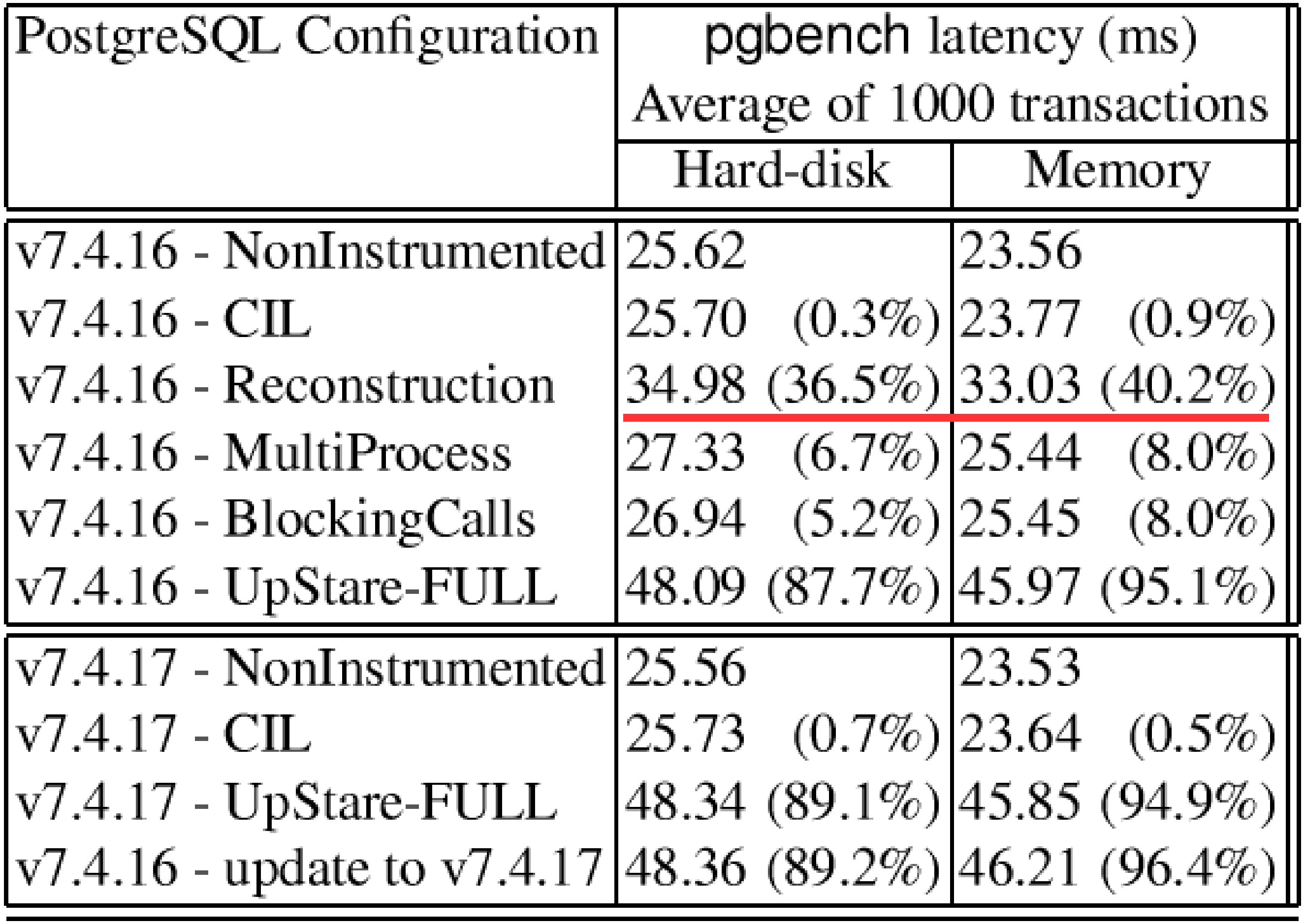
vsFTPd Configuration	Connection Latency(ms) 32-byte file	
	Hard-disk	Memory
v2.0.5 - NonInstrumented	9.61	9.49
v2.0.5 - CIL	9.64 (0.3%)	9.54 (0.5%)
v2.0.5 - Reconstruction	10.08 (4.9%)	9.99 (5.3%)
v2.0.5 - MultiProcess	10.26 (6.8%)	10.19 (7.4%)
v2.0.5 - BlockingCalls	9.97 (3.8%)	9.76 (2.9%)
v2.0.5 - UpStare-FULL	11.15 (16.0%)	11.06 (16.5%)
v2.0.6 - NonInstrumented	9.62	9.52
v2.0.6 - CIL	9.63 (0.1%)	9.54 (0.2%)
v2.0.6 - UpStare-FULL	11.16 (16.0%)	11.09 (16.5%)
v2.0.5 - update to v2.0.6	11.22 (16.6%)	11.12 (16.8%)

- Latency: 16–17% (1.6ms); throughput: 0%

PostgreSQL DBMS

- Updates
  - 1 update: v7.4.16 to v7.4.17
  - No manual continuation points; latency 60ms
- Overhead
  - Latency: run 1 transaction 1000 times
  - Throughput: “TPC-B like” pgbench; 100,000 txs
  - In-memory and on-disk, over cross-over cable

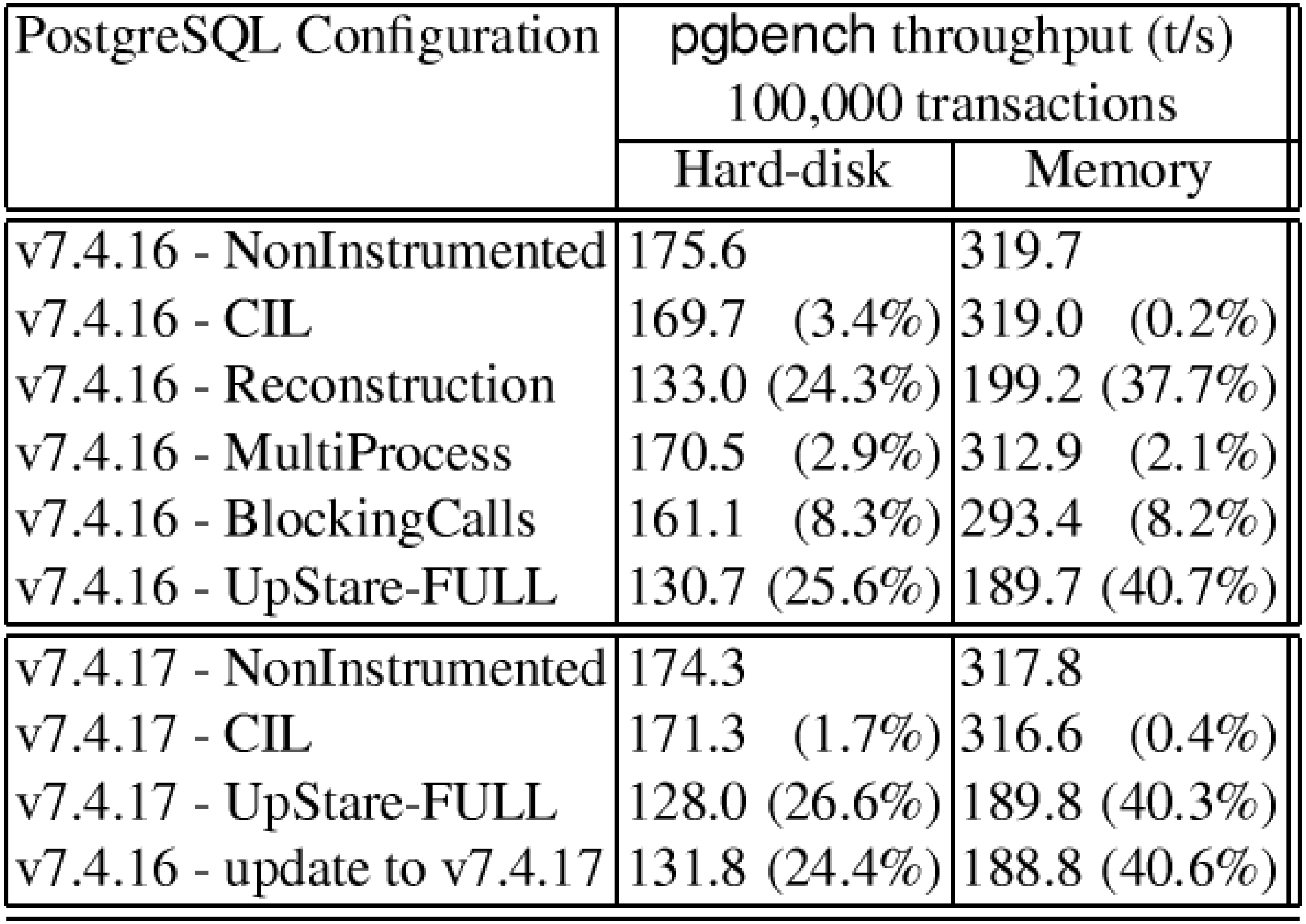
PostgreSQL Latency



PostgreSQL Configuration	pgbench latency (ms) Average of 1000 transactions	
	Hard-disk	Memory
v7.4.16 - NonInstrumented	25.62	23.56
v7.4.16 - CIL	25.70 (0.3%)	23.77 (0.9%)
v7.4.16 - Reconstruction	34.98 (36.5%)	33.03 (40.2%)
v7.4.16 - MultiProcess	27.33 (6.7%)	25.44 (8.0%)
v7.4.16 - BlockingCalls	26.94 (5.2%)	25.45 (8.0%)
v7.4.16 - UpStare-FULL	48.09 (87.7%)	45.97 (95.1%)
v7.4.17 - NonInstrumented	25.56	23.53
v7.4.17 - CIL	25.73 (0.7%)	23.64 (0.5%)
v7.4.17 - UpStare-FULL	48.34 (89.1%)	45.85 (94.9%)
v7.4.16 - update to v7.4.17	48.36 (89.2%)	46.21 (96.4%)

- Latency: 89–97% (22.5ms)

PostgreSQL Throughput



PostgreSQL Configuration	pgbench throughput (t/s) 100,000 transactions	
	Hard-disk	Memory
v7.4.16 - NonInstrumented	175.6	319.7
v7.4.16 - CIL	169.7 (3.4%)	319.0 (0.2%)
v7.4.16 - Reconstruction	133.0 (24.3%)	199.2 (37.7%)
v7.4.16 - MultiProcess	170.5 (2.9%)	312.9 (2.1%)
v7.4.16 - BlockingCalls	161.1 (8.3%)	293.4 (8.2%)
v7.4.16 - UpStare-FULL	130.7 (25.6%)	189.7 (40.7%)
v7.4.17 - NonInstrumented	174.3	317.8
v7.4.17 - CIL	171.3 (1.7%)	316.6 (0.4%)
v7.4.17 - UpStare-FULL	128.0 (26.6%)	189.8 (40.3%)
v7.4.16 - update to v7.4.17	131.8 (24.4%)	188.8 (40.6%)

- Throughput: 41% in-memory; 26% on-disk

Related Work

- Application DSU
  - OPUS: Small, isolated, feature-less updates
  - POLUS: Cannot update local variables
  - Ginseng: Data-access indirection

- Kernel DSU
  - DynAMOS,KSplice
    - Data-access indirection; limited safety
  - K42: Immediate updates; specially crafted

On-going and Future Work

- Move cold code at end of process image
- Automatically map pointers
  - Developed in previous work; not integrated yet
- Runtime safety checking
  - Transaction-safety through dynamic stack tracing
    - Temporary overhead 10-12%
- Semantic analysis
  - Identify nature of updates and prove correctness
  - Reduce user input
- Update in-transit data and files

Conclusion

- UpStare: Dynamic Software Updating
  - Source-to-source transformation of C applications
  - Useful safety guarantees
- Stack reconstruction
  - Update active code and data atomically
  - No data-access indirection
- Multi-threaded and multi-process updates
- Immediate updates
  - Convert blocking calls to non-blocking
- Demonstrated updates
  - vsFTPd: 13 updates (5.5 years-worth); 12,000 LOC
  - PostgreSQL v7.4.16; over 200,000 LOC
- Low overhead (0-26%); unoptimized

Questions ?