Making Legacy Code Safe and Scalable

- Legacy C software is not designed for multi-core architectures.
- Safety flaws in C make it hard to transform programs to exploit multi-core architectures, and cause vulnerabilities that limit adoption
- However, it is not practical to rewrite legacy systems in a safer language.

Incrementally add safety and concurrency to legacy C code

Principles

- Practical for use on real code
- Minimal changes to source
- Port one file at a time
- No changes to program behaviour
- No lock-in
- Dynamic checks where needed
- Automatic refactoring of code

Concurrency Support

Typical programs use locks to synchronize

We provide atomic sections instead

```
mutex m;                          atomic {
acquire(&m);          ⇨             ... operations
... operations ...                  ...
release(&m);                      }
```

Traditional way — New way

The programmer must **declare** locking rules

```
mutex m;
int shared_var protected_by(m);

atomic {
  ... x = shared_var; ...
}
```

this access causes m to be acquired automatically

Compiler picks lock order, inserts locking code

Concurrency Results

- Applied to 50'000 line web server
- Replicated most of its locking strategy
  - Locks were coarsened in four modules
- Appproximately 1% of source lines modified
- 2-5% overhead

Declarative Locking

Code is safer, easier to write:

- Deadlocks are prevented
- Data races are less likely

Supports gradual migration

- Interoperates with explicitly locked code
- Switch to annotations + atomic sections one data structure at a time

Researchers: Eric Brewer, David Gay, Rob Ennals

UC Berkeley Collaborators: George Necula, Zachary Anderson, Jeremy Condit, Matt Harren, Bill McCloskey, Feng Zhou

Research at Intel

www.intel.com/research

Making Legacy Code Safe and Scalable
Type and Memory Safety for C

Add annotations to check the correctness of an existing application's memory usage using compile and runtime checks

Program behaviour is unchanged

Type Safety

Annotations provide information on array bounds, object types:

```
struct thing{
    char *count(buflen) buf;
    int buflen;
    char *nullterm name;
    int tag;
    union {
        struct x_t x when(tag == X);
        struct y_t y when(tag == Y);
    } u;
};
```

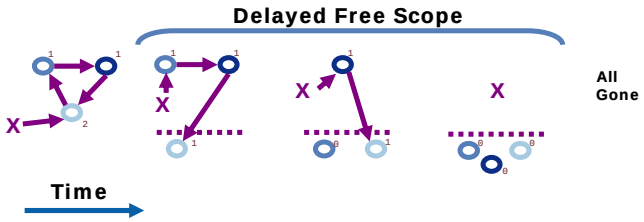
Compile and runtime checks enforce safety.
Trusted code provides escape hatch.

Memory Safety

Reference counting used for runtime check that freed memory blocks have no references.

C code often contains harmless short-lived dangling references to freed memory blocks.

"Delayed Free Scopes" allow one to briefly permit such references, delaying all frees until the end of the scope.



Results

- Successfully applied to a collection of C benchmarks (some from SPEC 2000/2006)
- Successfully applied to a fully bootable Linux kernel
- Costs:
  - Approximately 1% of source lines modified
  - Typical overheads below 50% (type safety) and 35% (memory safety)

Researchers: Eric Brewer, David Gay, Rob Ennals

UC Berkeley Collaborators: George Necula, Zachary Anderson, Jeremy Condit, Matt Harren, Bill McCloskey, Feng Zhou

Research at Intel
www.intel.com/research