Cuckoo: a Language for Implementing Memory- and Thread-safe System Services

Richard West and Gary Wong

Boston University

Computer Science

Introduction

- Recent emphasis on COTS systems for diverse applications
  - e.g., Linux for real-time
- Desirable to customize system for application-specific needs $\rightarrow$ system extensibility
  - Dangers with customizing kernels with untrusted code
  - Can use type/memory-safe languages, hardware protection, software-fault isolation, proof-carrying codes etc
- Here, we focus on language support for memory- *and* thread-safety

Why Thread Safety?

- Languages such as Cyclone support memory-safety using “fat pointers” but these are not atomically updated
  - Asynchronous control flow can lead to memory violations

- Asynchronous control flow fundamental to system design!
  - Support for interrupts, signals etc
  - Multi-threaded address spaces

Memory Safety

- We define a program as *memory safe* if it satisfies the following conditions:
  - It cannot read/write memory which is not reserved by a trusted runtime system;
  - It cannot jump to any location which is not the address of a trusted instruction.
- We enforce type safety only in so far as required to enforce memory safety
- Memory safety in Cuckoo does not guarantee program correctness

Memory Safety Issues

- Stack safety
  - We do not assume hardware detection of stack overflows
- Pointers and array bounds
  - We assume that bound information is associated with the array itself, and is immutable; bounds are *not* associated with (mutable) pointers
  - Pointer arithmetic is ruled out
    - Instead, arithmetic on indices into arrays referenced by pointers
- Dangling pointers
  - We rely on the type system to rule out dangling pointers to automatic storage
- Type homogeneity
  - Dynamic memory allocator is type-aware
  - Memory reuse is permitted only between compatible types

Example: Stack Checking

Computer Science

```
extern int a(…) { // suppose stack usage is small
                  // in this block
    char a_local;
    if (…) b();
}
static void b (…) { // again, minimal stack usage
    if (…) c();
}
static int c() {
    char c_local[65536]; // stack-allocate lots of memory
    …
}
```

Thread Safety

- Memory-safe checks must be atomic with respect to multiple threads of control
- Null pointer checks:
  - Made atomic by loading pointer value into a register, R
  - R is guaranteed to be used for both the checking and dereferencing of any pointer
- Array bound checks:
  - Made atomic by associating array bound info not with pointer BUT array
  - Since array sizes are immutable bound checks can never involve race conditions

Array Types in C versus Cuckoo

- Char a[5];
- Char c1=*a; // valid in C but not Cuckoo
- Char c2=a[0]; // valid in Cuckoo, s.t. c2=c1 as in C
- Char c3=(*a)[0]; // also valid in Cuckoo

Example Casts in C and Cuckoo

Computer Science

```
struct foo {
int a[5];
char *s;
}
struct foo *p;
int x=*((int *)p);          // legal in C but not Cuckoo
int y=*((int (*)[5])p);     // also illegal in Cuckoo
int z=((int (*)[5])p)[0];   // now legal in Cuckoo
                            // assigns z 1st element of array
```

Potentially unsafe Memory Realloc

Computer Science

```
int *p;
char **q;
p=new(int);              // heap-alloc an integer
…delete(p);              // release memory ref'd by p
q=new(char *);           // reuse memory freed at addr p
*p=123;                  // assign values after p is freed
…**q=45;                 // memory[123]=45 -> dangerous!
```

Type-homogeneous dynamic memory allocation needed to avoid reallocating memory to incompatible types

Experimental Results

Computer Science

Compiler	Time (user)	Time (system)	Size (code)	Size (data)	Size (BSS)
SUBSET SUM					
Cuckoo	30.96	n/a	2377	288	152
gcc –O2	17.86	n/a	1833	280	192
gcc	24.75	n/a	1945	280	192
PRODUCER-CONSUMER					
Cuckoo	2.50	5.13	2527	308	428
gcc –O2	2.46	5.10	2001	300	480
gcc	2.50	5.14	2093	300	480
FIND-PRIMES					
Cuckoo	10.17	n/a	1301	260	10016
Cuckoo (OPT)	6.78	n/a	1285	260	10016
gcc	9.56	n/a	874	252	10032
gcc –O2	3.57	n/a	814	252	10032
Cyclone	12.43	n/a	91721	3340	59996
SFI	10.79	n/a	970	252	10032

Times in seconds (2.8GHz Pentium 4); sizes in bytes

Exec. Times (Parallel Subset-sum)

Compiler	Parallel time (real)
Cuckoo	9.45
gcc –O2	4.59
gcc	7.40

Execution times for 4-threaded subset sum problem on 27 integers (4x2.2GHz Opteron)

Example: Unaligned Address Problem

Computer Science

```
static void bad(void) {
    volatile int x=0xBADC0DE;
}
extern int main(void) {
    union foo {
        char *data;
        void (*code)(void);
    } bar;

    bar.code=bad;
    bar.data+=10; // whatever is offset to 0xBADC0DE
    bar.code();
    return 0;
}
```

Cuckoo versus Alternatives

System	C	Cyclone	Java	SFI	Cuckoo
Efficient memory usage	✓	✓		✓	✓
Memory safe		✓	✓	Y/N	✓
Stack overflow checking			✓	✓	✓
Multithreaded memory safe			✓	✓	✓
Operate without garbage collection	✓	✓		✓	✓
Unrestricted allocation w/o garbage collection	✓			✓	✓

Conclusions and Future Work

Computer Science

- Multithreaded memory safety can be a key issue in certain domains e.g., extensible systems
- Safety can be enforced for single- and multi-threaded programs with relatively low overhead
- Future work:
  - Further investigating and optimising the cost of dynamic memory allocation
  - Tradeoffs between permissive type systems and overheads of runtime checks
  - Implementation and analysis of a trusted runtime system