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

Richard West and Gary Wong

Boston University
Introduction

- Recent emphasis on COTS systems for diverse applications
  - e.g., Linux for real-time
- Desirable to customize system for application-specific needs → 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
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

```c
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 Reallocation

```c
int *p;
char **q;
p = new(int);
    // heap-allocate 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

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

Times in seconds (2.8GHz Pentium 4); sizes in bytes
## Exec. Times (Parallel Subset-sum)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Parallel time (real)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuckoo</td>
<td>9.45</td>
</tr>
<tr>
<td>gcc –O2</td>
<td>4.59</td>
</tr>
<tr>
<td>gcc</td>
<td>7.40</td>
</tr>
</tbody>
</table>

Execution times for 4-threaded subset sum problem on 27 integers (4x2.2GHz Opteron)
Example: Unaligned Address Problem

```c
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

<table>
<thead>
<tr>
<th>System</th>
<th>C</th>
<th>Cyclone</th>
<th>Java</th>
<th>SFI</th>
<th>Cuckoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient memory usage</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memory safe</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Y/N</td>
<td>✓</td>
</tr>
<tr>
<td>Stack overflow checking</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multithreaded memory safe</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Operate without garbage collection</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Unrestricted allocation w/o garbage collection</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

- 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.