Nabil S. Al Ramli

TAME

Event Based Concurrency System
What is TAME?

Event based concurrency system

- C++ libraries and source-to-source translation
- For network applications
- No “stack-ripping”
- Type safe (templates)
- Performance of events
- Programmability of threads
# Events Vs. Threads

<table>
<thead>
<tr>
<th>Measure</th>
<th>Threads</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for extremely high concurrency</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Portability</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Better performance, Less use of memory</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>No stack-ripping</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>
Events Vs. Threads

**Threads**

- Function A
- Function B
- Blocking operation

**Events**

- Function A
- Function B
- Non-blocking operation
Semantics Example

// Threads
void wait_then_print_threads() {
    sleep(10); // blocks this function and all callers
    printf("Done!");
}

// Tame primitives
tamed wait_then_print_tame() {
    tvars { rendezvous<> r; }
    event<> e = mkevent(r); // allocate event on r
    timer(10, e); // cause e to be triggered after 10 sec
    twait(r); // block until an event on r is triggered
    // only blocks this function, not its callers!
    printf("Done!");
}

// Tame syntactic sugar
tamed wait_then_print_simple_tame() {
    twait { timer(10, mkevent()); }
    printf("Done!");
}
## TAME Primitives

<table>
<thead>
<tr>
<th>Classes</th>
<th>Keywords &amp; Language Extensions</th>
<th>Functions &amp; Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>event&lt;&gt;</td>
<td>twait($r[i]$);</td>
<td>mkevent($r,i,s$);</td>
</tr>
<tr>
<td></td>
<td>- A basic event.</td>
<td>- Allocate a new event with event ID $i$.</td>
</tr>
<tr>
<td>event&lt;$T&gt;$</td>
<td>twait($r[i,i]$);</td>
<td>- When triggered, it will awake rendezvous $r$ and store trigger value in slot $s$.</td>
</tr>
<tr>
<td></td>
<td>- An event with a single trigger value of type $T$. This value is set when the event occurs; an example might be a character read from a file descriptor. Events may also have multiple trigger values of types $T_1 \ldots T_n$.</td>
<td>mkevent($s$);</td>
</tr>
<tr>
<td>rendezvous&lt;$I&gt;$</td>
<td>twait {};</td>
<td>- Allocate a new event for an implicit rendezvous. When triggered, store trigger value in slot $s$.</td>
</tr>
<tr>
<td></td>
<td>- Marks safe local variables.</td>
<td>e.trigger($v$);</td>
</tr>
<tr>
<td></td>
<td>twait {} statements;</td>
<td>- Trigger event $e$, with trigger value $v$.</td>
</tr>
<tr>
<td></td>
<td>- Wait point syntactic sugar: block on an implicit rendezvous until all events created in statements have triggered.</td>
<td>timer($to,e$); wait_on_fd($fd, rw, e$);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Primitive event interface for timeouts and file descriptor events, respectively.</td>
</tr>
</tbody>
</table>

**Figure 2:** Tame primitives for event programming in C++. 
Wait Points

- blocks calling function until 1+ events triggered
- function returns to its caller
- but the function does not complete
- execution point and local vars are preserved
- When an event occurs, the function “unblocks” and resumes processing at the wait point

```c
wait { statements; }

{ rendezvous<> __r;
  statements; // where mkevent calls create events on __r
  while (not all __r events have completed)
  wait(__r); }
```
Safe Local Variables

• Their values are preserved across wait points
• Preserved in a heap-allocated closure
• \texttt{tvars \{\}}

• Unsafe local variables have indeterminate values after a wait point
• Compile warnings when a local variable should be made safe
Closures

• Internally, Tame writes a new C++ structure for each tamed function

• Each tamed function has one closure, contains:
  – **Next statement to execute**
  – **Function parameters**
  – **tvars variables**
  – **rendezvous**
Closures Translation

tamed A::f(int x) {
tvars { rendezvous<> r; }
a(mkevent(r)); twait(r); b(); }

void A::f(int __tame_x, A_f_closure *__cls) {
if (__cls == 0)
__cls = new A__f__closure(this, &A::fn, __tame_x);
assert(this == __cls->this_A);
int &x = __cls->x;
rendezvous<> &r = __cls->r;
switch (__cls->entry_point) {
case 0: // original entry
goto __A__f__entry__0;
case 1: // reentry after first twait
goto __A__f__entry__1; }
__A__f__entry__0:
a(_mkevent(__cls,r));
if (!r.has_queued_trigger()) {
__cls->entry_point = 1;
r.set_reenter_closure(__cls);
return; }
__A__f__entry__1:
b();
}
A “Tame” solution

(slot) <T> a;

e(a):<T>

r: <I>

a <- data

e.trigger(data)

handler

event<T>

(wait point)

(rendezvous<I>)

(non-blocking, asynchronous operation, e.g., I/O)
Composeability

```

timer
  event 1

gethost_ev
event 2
  trigger
  ID true
  slot a

r
  ID false
  twait
  ok

```

```
timer
  event 1

gethost_ev
e_base
  trigger
  ID true
  slot result

r
  ID false
  twait
  rok;

```

```
e
  trigger
  slots ok, a

```

e.trigger(rok, result)

Private to __add_timeout

wake caller
tamed broken(dnsname nm) {
    tvars { rendezvous<> r; ipaddr res; }
    gethost_ev(nm, mkevent(r, res));
    // Whoops: forgot to twait(r)!
}

- **Solution is reference-counting**
  - *Keep track of events and closures*
  - *C++ “smart pointer” classes*

- **Two types of reference counts**
  - **Strong references**
    - are conventional reference counts
  - **Weak references**
    - Allow access to the object only if not deallocated

(a) After the event allocation on line 3.

(b) After control exits broken on line 5.
## Related Work

<table>
<thead>
<tr>
<th>System</th>
<th>Similarities</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capriccio</td>
<td>• Uses events</td>
<td>• Thread interface</td>
</tr>
<tr>
<td></td>
<td>• Automatic stack management</td>
<td></td>
</tr>
<tr>
<td>Adya et al.</td>
<td>• Uses events</td>
<td>• Some manual stack management</td>
</tr>
<tr>
<td>SEDA</td>
<td>• Can be used together with TAME</td>
<td>• Uses threads and events</td>
</tr>
</tbody>
</table>