Events Can Make Sense

Maxwell Krohn (*MIT CSAIL*)
Eddie Kohler (*UCLA*)
M. Frans Kaashoek (*MIT CSAIL*)

*Presented by Nabeel*
Agenda

- Event Vs Threads
- Tame Abstraction
- Implementation
- Methodology
- Limitation
- Performance
Events

- Uses event loop and event handlers

Advantages
- More expressive
- Uses less memory
- Easily portable

Disadvantages
- Difficult to maintain and debug
- Manual memory management
- Stack ripping
Threads

- Uses different execution contexts for concurrency
- Advantages
  - Standard control flow
  - Automatically managed local variables
  - Easy to maintain
- Disadvantages
  - Synchronization bottleneck
  - Consumes memory
  - Context switch overhead
Challenge

- A combined model

1. the flexibility and performance of events

2. the programmability of threads
TAME

- System for managing concurrency in network applications
- API for event based programming
- No stack ripping
- Automatic memory management
- Standard control flow
```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <tamer/tamer.hh>

using namespace tamer;

tamed void tame_print()
{
    printf("tame_print - Entering tame_print and sleep for 3 sec\n");
    twait { tamer::at_delay_sec(3, make_event()); }
    printf("tame_print - Exiting tame_print \n");
}

int main(int, char **[]) {
    tamer::initialize();
    printf("main - Calling the tamed function tame_print \n");
    tame_print();
    printf("main - Exiting main \n");
    while (!tamer::driver_empty())
        tamer::once();

    /* OUTPUT
    * main - Calling the tamed function tame_print
    * tame_print - Entering tame_print and sleep for 3 sec
    * main - Exiting main
    * tame_print - Exiting tame_print
    */

    "basic.ttf" 30L, 700C written
```
Tame Abstractions

- Events
  - future occurrence
- Wait Points
  - blocking point
- Rendezvous
  - flexible wait point
- Safe local variables
  - preserved across wait points
Events

- Represents the future occurrence
- Event triggered via it's trigger method

Terminology
- Event object
- Trigger slots
- Trigger values
Event Primitive

- To create a new event
  
  ```
  event<T*> = make_event(T &)
  ```

- Trigger method marks the event's occurrence
  
  ```
  void trigger(T)
  ```

- class event <T*> {
  
  ```
  public:

  event();

  void trigger(T*);
  ```
}
Wait Points

- Blocks until events inside `twait {..}` are triggered

- Functions having `twait{..}`
  - Marked with tamed keyword
  - Blocks till the event inside `{..}` triggers
  - Caller of the function returns

- Execution point and local variables preserved in memory

- Wait for all primitive
twait { statements; }

Example:

twait { at_delay_sec(5, make_event()); }
#include <tamer/tamer.hh>

using namespace tamer;

tamed void func(tamer::event<int> e)
{
    printf("func - Entering func and returning 0 as trigger value\n");
    e.trigger(0);
}

tamed void tame_print()
{
    tvars {
        int val = 100;
    }
    printf("tame_print - Entering tame_print and val is %d\n", val);
    twait {
        func(make_event(val));
        tamer::at_delay_sec(5, make_event());
    }
    printf("tame_print - After calling func val is %d\n", val);
    printf("tame_print - Exiting tame_print \n");
}

int main(int, char *[])
{
    tamer::initialize();
    printf("main - Calling the tamed function tame_print \n");
    tame_print();
    printf("main - Exiting main \n");
    while (!tamer::driver_empty())
    {
        tamer::once();
    }
}
Events & Waitpoints

main - Calling the tamed function tame_print
tame_print - Entering tame_print and val is 100
func - Entering func and returning 0 as trigger value
main - Exiting main
tame_print - After calling func val is 0
tame_print - Exiting tame_print
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS#
Rendezvous

- Associate relevant events to the wait point
- Every event object associates with one rendezvous (r)
- twait(r) unblocks for the first trigger
- Consumes event and restarts the blocked function
- Event ID identifies events
Rendezvous Primitive

rendezvous <l> r
rendezvous<> r
...
make_event(r, l, T*)
make_event(r, l)
make_event(r)
...
twait(r, l)
twait(r)
Safe Local Variables

- Values are preserved across wait points

- Allocates the variables from the heap

- tvars {....}
```c
int main(int, char **[]) {
    tamer::initialize();
    printf("main - Calling the tamed function tame_print \n");
    tame_print();
    printf("main - Exiting main \n");
    while (!tamer::driver_empty())
        tamer::once();
}
```
Rendezvous & Safe local vars

```
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS# ./rendezvous
main - Calling the tamed function tame_print
tame_print - Entering tame_print
func1 - Entering func1
func2 - Entering func2
tame_print - returned event id is 1
tame_print - returned event id is 2
tame_print - Exiting tame_print
main - Exiting main
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS# 
```
Control Flow Example

tamed void gethostbyname_tame(char *host, tamer::event<char *> * e)
{
    struct hostent *hp = gethostbyname(host);
    if (hp == NULL) {
        printf("gethostbyname() failed\n");
        e.trigger(NULL);
    } else {
        unsigned int i=0;
        e.trigger(strdup(inet_ntoa(*( struct in_addr*)( hp -> h_addr_list[i]))));
    }
}

tamed void tame_print()
{
    tvars {
        int i;
        char *ip[20];
    }
    printf("tame_print - Entering tame_print\n");
    /* Sequential Version */
    /*for (i=0; i<20; i++) {
        twait { gethostbyname_tame("vt.edu", make_event(ip[i])); }
        printf("%d is %d and Ip address is %s\n", i, ip[i]);
    }*/
    /* Parallel Version */
    twait {
        for (i=0; i<20; i++) {
            gethostbyname_tame("vt.edu", make_event(ip[i]));
            printf("%d is %d and Ip address is %s\n", i, ip[i]);
        }
    }
    printf("tame_print - Exiting tame_print \n");
}

int main(int, char *[]) {
    tamer::initialize();
    printf("main - calling the tamed function tame_print \n");
    tame_print();
}
Control Flow Example

```bash
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS# time ./parallel
main - Calling the tamed function tame_print
  i is 0 and Ip address is 198.82.183.9
  i is 1 and Ip address is 198.82.183.9
  i is 2 and Ip address is 198.82.183.9
  i is 3 and Ip address is 198.82.183.9
  i is 4 and Ip address is 198.82.183.9
  i is 5 and Ip address is 198.82.183.9
  i is 6 and Ip address is 198.82.183.9
  i is 7 and Ip address is 198.82.183.9
  i is 8 and Ip address is 198.82.183.9
  i is 9 and Ip address is 198.82.183.9
  i is 10 and Ip address is 198.82.183.9
  i is 11 and Ip address is 198.82.183.9
  i is 12 and Ip address is 198.82.183.9
  i is 13 and Ip address is 198.82.183.9
  i is 14 and Ip address is 198.82.183.9
  i is 15 and Ip address is 198.82.183.9
  i is 16 and Ip address is 198.82.183.9
  i is 17 and Ip address is 198.82.183.9
  i is 18 and Ip address is 198.82.183.9
  i is 19 and Ip address is 198.82.183.9

main - Exiting main

real    0m0.231s
user    0m0.016s
sys     0m0.016s
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS# 
```
Types and Composability

- Event ID
  - Identify events
  - Known during event registration
  - All events on the same rendezvous must have the same event ID type

- Trigger Values
  - Are results
  - Not known until event triggers
  - Single rendezvous handles different typed trigger values
Types and Composability

```c
using namespace tamer;

tamed void sleepfunc(tamer::event<> e)
{
    twait {tamer::at_delay_sec(10, make_event());}
    e.trigger();
}

tamed void tame_print()
{
    tvars {
        tamer::rendezvous<bool> r;
        bool result;
    }
    printf("tame_print - Entering tame_print\n");
    tamer::at_delay_sec(5, make_event(r, false));
    sleepfunc(make_event(r, true));
    printf("tame_print - nevents : %d, nready : %d and nwaiting : %d \n", r.nevents(), r.nready(), r.nwaiting());
    twait(r, result);
    printf("tame_print - nevents : %d, nready : %d and nwaiting : %d \n", r.nevents(), r.nready(), r.nwaiting());
    if (!result)
    {
        printf("tame_print - Timeout Fired \n");
    //r.clear();
    twait(r, result);
    printf("tame_print - nevents : %d, nready : %d and nwaiting : %d \n", r.nevents(), r.nready(), r.nwaiting());
    if (result)
        printf("tame_print - Event Fired \n");
    printf("tame_print - Exiting tame_print \n");
    }
}

int main(int, char *[])
{
    tamer::initialize();
    printf("main - Calling the tamed function tame_print \n");
    tame_print();
    printf("main - Exiting main \n");
    while (!tamer::driver_empty())
    {
        tamer::once();
    }
}
Types and Composability

```bash
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS# ./types
main - Calling the tamed function tame_print
tame_print - Entering tame_print
tame_print - nevents : 2, nready : 0 and nwaiting : 2
main - Exiting main
tame_print - nevents : 1, nready : 0 and nwaiting : 1
tame_print - Timeout Fired
tame_print - nevents : 0, nready : 0 and nwaiting : 0
tame_print - Event Fired
tame_print - Exiting tame_print
root@nabeel:/home/nabeel/Downloads/tamer-1.2.2/ex/OS# 
```
Thread Support

- twait without tamed return type
- Yield and wakeup mechanism
- twait – to block the current thread
- tfork – to start a new thread
- Event blocking and joining on a thread unified
Memory Management

- Reference counting scheme to enforce invariants

I1 : A function's closure lives at least until control exits the function for the last time.

I2 : A function's closure live as least until events created in the function have triggered

I3 : Events associated with rendezvous r must trigger exactly once before r is deallocated
Reference Counting Scheme

- Runtime takes care of events and closure

R1 : Entering/exiting a tamed function adds/removes a strong reference to the corresponding closure (I1)

R2 : Each event created inside closure holds strong reference to the closure. The reference is dropped once the event is triggered (I2)

R3 : A rendezvous and its associated events keep weak references. Allows for event cancellation before rendezvous deallocation (I3)

R4 : Exiting a tamed function cancels any rendezvous allocated in that function
Implementation

- Function pointers tracks the wait points of events in each rendezvous

- The func parameters and safe local variables will be in a closure structure

- C++ libraries and source-to-source translation

- No platform specific support or compiler modification required.
Methodology

- OKWS – serial chains of asynchronous function calls
- OkCupid.com – User preferences
- NFS Server
Limitations

- Heavy usage of heap
- Heavy usage of synchronization primitives
- Involves signature changes to convert a C++ code into tame model
Performance

- Capriccio Knot server Vs Tamed version of Knot
- SpecWeb like benchmark – memory and CPU
- Server
  - 2 CPU 2.33 Ghz Xeon 5140 4GB RAM
  - Ubuntu kernel 2.6.17-10
- Clients
  - Array of six clients connected thru a gigabit switch
  - 200 simulatenous requests for 1 minute
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Capriccio</th>
<th>Tame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (conn / sec)</td>
<td>28318</td>
<td>28457</td>
</tr>
<tr>
<td>No. of Threads</td>
<td>350</td>
<td>1</td>
</tr>
<tr>
<td>Physical Memory (kB)</td>
<td>6560</td>
<td>2156</td>
</tr>
<tr>
<td>Virtual Memory (kB)</td>
<td>49517</td>
<td>10740</td>
</tr>
</tbody>
</table>
Questions