The Aesop Language

Dries Kimpe (Argonne National Laboratory)

June, 2012
NAS 2012, Xiamen, China
Aesop - In Short

- New **programming language** (+ support libraries)
  - Based on C language with added concurrency and other extensions
  - Designed for implementing **distributed network services**
  - Aims to maintain **sequential flow while programming** without requiring sequential execution.
  - Aims to be highly productive.

- Implemented as **Source-To-Source translator**
  - Translator written in Haskell, injects macro calls into the source.
  - Outputs plain C

- Also provides RPC helper
  - Generates local and remote network and encoding/decoding functions

- Stand-alone distribution
  - Git repository at git://git.mcs.anl.gov/aesop
  - Trac (wiki & bug reports) at http://trac.mcs.anl.gov/project/aesop

NAS 2012 - Xiamen, China
Aesop - Motivation

- Most people prefer writing sequential code (reasoning, algorithms, ...)
- Easiest way to have sequential code in a network server is using threads. 
  **However**
  - Threads can have high overhead (thread stack, context switch, thread creation, locking)
  - Not all device APIs map to a thread model
    (or hard to drive efficiently from multiple threads)
    - Poll/epoll/select
    - MPI_Waitsome/MPI_Waitany

- Consequently, many high-concurrency network services are written in an event-driven manner (memcached, apache, ...).

- Unfortunately, writing event-driven code is hard
  - **Manual stack** management (‘stack ripping’)
  - Difficult to follow **control flow** (callback to callback) [debugging!]
  - Cannot call functions that take a significant time to complete (‘inversion of control’)
  - Running **multiple event loops** (for multi-core processors)
Example: Echo Server (7/tcp)
(with some processing added in)

Open TCP connection to port 7, server writes back uppercase of data received

```c
void handleClient (fd) {
    char buf[];
    read (fd, buf);
    uppercase (buf);
    write (fd, buf);
    close (fd);
}

int main (int argc, char ** args) {
    [...]  
    while (true) {
        int fd = accept (sock);
        handleClient (fd); // or thread  
    }
}
```

Wait for incoming connection

Read msg from socket

Write msg to socket

Close socket

Automatic variables

Linear control flow

Concurrency granularity: thread
enum { STATE_READ, STATE_WRITE, STATE_CLOSE };

void handleRequest (request * req) {
    switch (req->state) {
    case STATE_READ:
        read (req->fd, req->buf); state = STATE_WRITE; break;
    case STATE_WRITE:
        uppercase (req->buf);
        write (req->fd, req->buf); state = STATE_CLOSE; break;
    }
}

int main (int argc, char ** args) {
    while (true) {
        if (can_accept (socket)) {
            int fd = accept;
            req = malloc (sizeof (request));
            req->fd = fd; req->state = STATE_READ;
            active_requests_add (req);
        }
        req = wait_for_req_ready ();
        if (!handleRequest (req))
            { active_requests_remove (req); close (req->fd); free (req); } 
    }
}
PVFS2 State Machine Compiler

- Similar to flex/bison: takes blocks of C code and adds glue that can be automated.
- Simple parser (C blocks are opaque)

```
machine echo_request {
    state start_read {
        run request_read;       // C function
        default => start_write; // next state
    }
    state start_write {
        run request_write;
        default => close;
    }
    state close {
        run request_cleanup;
        default => terminate;
    }
}
```

- Unified interface required (start, test)
- Restores some of the control flow

Note: accept code & actual read/write/cleanup code not shown

Unified interface required (start, test)  Help with state management

Restores some of the control flow
Aesop
Automatic State Machines

void handleRequest (int fd) {
    char buf[];
    read (fd, buf);
    uppercase (buf);
    write (fd, buf);
    cleanup ();
}

void handleRequest (int fd) {
    char buf[];
    read_completed
    uppercase (buf);
    start_write (fd, buf);
    write_completed
    cleanup ();
}

Concurrency: none or event loop
Why invent our own programming language?

- Existing Parallel Programming models
  - Focus on **optimizing CPU usage**
  - Do not offer assistance for handling devices such as network and storage devices.
  - Do not support **cancellation**
  - Problem partitioning mapped directly to threads
  - **Portability** might be an issue

- Event driven programming libraries
  - Unify handling of asynchronous operations
  - Require algorithms to be casted into an event-driven form
  - Generally only support single-threaded event loops
Blocking vs Non-Blocking

- **Non-blocking code** is cpu bound.
- **Blocking code** is not cpu bound, meaning that the completion typically depends on some external event.
- Aesop does not enforce correct usage.
- **Blocking or non-blocking** is a property of a C function (or function pointer)
  - indicated by the `__blocking` keyword
  - Visible when declaring function, not when calling function.

**Examples:**

- Calculating a checksum is **not blocking**.
- **Sleeping** for 6 seconds or waiting for an alarm time is **blocking**.
- **Reading** or **writing** from network or disk is **blocking**.

- Any **function** calling a **blocking function** is also **blocking**.

```c
__blocking int aesop_main (int argc, char ** args) { ... }
```
The *pbranch* keyword

- Basic concurrency construct in aesop is the *pbranch*
- *Pbranch* creates a C scope
- *Pbranches* can be executed concurrently with other code
  - Within the *pbranch*, execution is sequential

**Example 1:**
```c
pprivate int i;
for (i=0; i<100; ++i)
{
    pbranch {
        do_something (i);
        do_something_else ();
    }
}
```

**Example 2:**
```c
{
    pbranch { call1 (); }
    pbranch { call2 (); }
}
```
Private pbranch variables

- Pbranches share variables from the enclosing scope by default
- Use the pprivate variable modifier to give each pbranch a private copy
- Private copy is initialized when entering the pbranch.

Code example

```c
pprivate tmp;
pbranch {
    // own copy of tmp
}
pbranch {
    // own copy of tmp
}
```
pbranch synchronization: \texttt{pwait}

The \texttt{pwait} keyword enables synchronizing with the enclosed \texttt{pbranches}.

Example

\begin{verbatim}
pwait {
  pbranch {
    func_1 ();
  }
  pbranch {
    func_2 ();
  }
}
func3 ();
\end{verbatim}

// \texttt{func_1()} might execute concurrently with \texttt{func_2()};
// \texttt{func_2()} will not wait for \texttt{func_1()} to complete if it blocks.
// \texttt{func_3()} will not execute until \texttt{func_1()} and \texttt{func_2()} completed.

Note: \texttt{pbranch} without enclosing \texttt{pwait} is possible: lonely \texttt{pbranch}. 
Cancelling blocking functions

- One of the major differences between aesop and other concurrency extensions (such as OpenMP) is aesop’s support for **cancellation**.
- **pbranches** can be cancelled.
  - Cancellation is `clean`: proper cancellation function is automatically called by aesop. (for example: MPI_Cancel for MPI_Recv, aio_cancel for aio_read, ...)

**Example:** cancelling operation after timeout (some error tracking omitted)

```c
pwait {
  pbranch {
    do_some_processing ();  // non-blocking function
    send_query ();          // blocking function
    receive_response ();    // blocking function
    aesop_cancel_branches ();
  }
  pbranch {
    aesop_timer (10);
    aesop_cancel_branches ();
  }
}
```
Resources

- So far, functions were **blocking** because they **called one or more blocking functions**.
- A **resource** is a collection of one or more (public) **blocking** functions, with the difference that those blocking functions do not call any other blocking functions.
  - Resource functions look and behave exactly like all other functions.
  - The innermost blocking function in a call-graph is **always a resource function**.
- Resources can (and typically do) also contain regular (non-blocking) functions.
- Resources also contain some special aesop-internal functions for testing, polling, context handling and **cancellation**.
- Resources are written in **plain C**

Example resources:
- Timer (in default aesop distribution)
- Signal
- socket
Some Notes

- Aesop does not require or create threads: it does not dictate concurrency model
  - Resources might use threads internally
  - Aesop is thread-safe (could use multiple threads to drive aesop)

- Resources can choose the most efficient driving model (threads, poll) \textit{(for a given system)} without affecting the use of the resource.

- There are no explicit state machines
  - Blocking functions are pulled apart into segments containing no blocking calls
    - Variables are moved from the stack to the heap (and references adjusted)
  - Non-blocking functions not modified

- Easiest mental model for aesop code: \textbf{lightweight cooperative threads}
  (with blocking function possible context switch point)
Status

- Stand-alone source code distribution of Aesop development environment for use in other storage projects. Documentation on how to install and how to use Aesop.

- Aesop is **ready** for use
  - Triton (storage system) is written completely in aesop
  - Collection of resources available
  - Own repo / bug tracker \[git://git.mcs.anl.gov/aesop\]
  - Passed performance requirements (more in a moment)

- Places where Aesop can be **improved**
  - Some minor language bugs
    - Workarounds generally possible
  - Compilation speed
  - Debugging
    - Using preprocessor directives to link to original source, but no debugger supports stepping through the logical control flow.
Productivity

Implemented simple server listening on a TCP socket and responding to one or more client requests, optionally storing or retrieving data from disk.

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>Mod. CC</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aesop</td>
<td>16</td>
<td>11</td>
<td>179</td>
</tr>
<tr>
<td>Thread per client</td>
<td>17</td>
<td>12</td>
<td>182</td>
</tr>
<tr>
<td>Thread per op</td>
<td>22</td>
<td>17</td>
<td>249</td>
</tr>
<tr>
<td>Threadpool</td>
<td>32</td>
<td>26</td>
<td>313</td>
</tr>
<tr>
<td>Event</td>
<td>28</td>
<td>23</td>
<td>341</td>
</tr>
</tbody>
</table>

**CC:** McCabe Cyclomatic Complexity (CC),  
**Mod. CC:** Modified McCabe Cyclomatic Complexity (mod. CC)  
**SLOC:** Source Lines of Code (SLOC).

Note: no error handling, ignoring shared definitions, **not using cancellation**
Performance (Write-Null)

operations per second

16 128 256 512 1024

aesop thread-per-client thread-per-client-nb thread-per-op thread-pool event
Performance (Read-Null)

operations per second

16 128 256 512 1024

aesop  thread-per-client  thread-per-client-nb  thread-per-op  thread-pool  event

NAS 2012 - Xiamen, China
Max Memory Usage (write)

- aesop
- thread-per-client
- thread-per-client-nb
- thread-per-op
- thread-pool
- event

memory usage (KB)

clients: 16, 128, 256, 512, 1024

NAS 2012 - Xiamen, China
Response Time (write, 1024 clients)
Aesop: Summary

- **Concurrency extension** to the C programming language
- Takes normal C code and detects *blocking* function calls
- **Current implementation** creates (implicit) state machine code and writes the boilerplate code; *Isolates algorithm* from concurrency model.
- Lowers the bar for writing high-performance network servers (for example, can convert sequential into event-driven)
- Fully C compatible: important for reusing existing code and interfacing with low-level OS layers

**... with some added features thrown in:**

- Concurrent & decoupled execution
- Cancellation
- RPC code generator
Conclusion

Many people are working on Aesop:
- Phil Carns, Kevin Harms, **Dries Kimpe**, Sam Lang, Rob Ross, Justin Wozniak

Further Reading:
- Code repository: [http://git.mcs.anl.gov/aesop](http://git.mcs.anl.gov/aesop)

Questions/Remarks?