Capturing Flight Software Architecture
With a Domain-Specific Language

Kim P. Gostelow

Jet Propulsion Laboratory, California Institute of Technology
July 30 – August 3, 2012

© 2012 California Institute of Technology. Government sponsorship acknowledged
Multicore

Tilera TILE64 Multicore Chip

Independently powered tile

A general-purpose CPU

Cache Engine (L1/L2, TLB, DMA)

Switch Engine (5 networks)

Processor Engine (64-bit VLIW)
The Vision

• Look to when there are thousands of cores on a spacecraft
  – Expectation: Power = Speed x Reliability
    • Faulty core=> computations move to another core
    • Reduce power => performance slows, but does not quit
  – Computations reorganize in real-time
  – Introspective
  – Little or no consideration needed by the programmer
The Problem

• The above can be achieved now, but only on a small scale by costly, special-case programming

• Programmers should not spend their time orchestrating intricate (and brittle) data arrangements and code
  – It breaks when processors fail
  – It should not be part of the job

• We want the machine, without intervention, without programmer’s special attention, to re-organize its work automatically in the face of cores and links failing/re-appearing at random, in real-time.
## Towards A Solution

<table>
<thead>
<tr>
<th>Von Neumann</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>(~ Clocked sequential circuit)</td>
<td>(~ Asynchronous circuit)</td>
</tr>
<tr>
<td>An instruction executes when the program counter reaches it.</td>
<td>The function executes when the required data arrives.</td>
</tr>
<tr>
<td>Instructions manipulate the contents of memory cells.</td>
<td>Variables are mathematical variables, not memory cells</td>
</tr>
</tbody>
</table>
What is a Functional Language?

The relation \( f: A \rightarrow B \) is a **function** if:

For-all a **in** A there is a **unique** b **in** B such that \( f(a) = b \)

- For the programmer, the consequences of the above are:
  - Immutable values
    - Can define a value only once
    - A variable has only one meaning
    - Single-assignment
  - No shared memory

| Synonyms |
The relation $f: A \rightarrow B$ is a **function** if:

For-all $a$ in $A$ there is a unique $b$ in $B$ such that $f(a) = b$
**Example: generate-map-reduce**

```
function gmr(a, b, f, g) =
    spread = (b-a)/2
    if split_is_efficient(f, spread) then
        g( gmr(a,a+spread,f,g),
            gmr(a+spread,b,f,g) )
    else g(map(f,gen(a,b)))
```

- **f, g** are functions, and their composition is a function.

---

**Diagram:**

- One thread executes k calls.
- Actor diagram showing the process of generating map-reduce operations.
Beginning the DSL

- Multi-chips of multi-cores
  - 1000s of cores on a spacecraft
  - Power on/off
  - Power = Speed x Reliability
- Auto-redundancy / auto-restart
  - Threads must be able to:
    start/stop/re-start, move, be copied, replicated, ... at any time, in real-time
- Auto-concurrency =>

  DSL is a Functional language => no state

But, a system really does have state.
What Is State?

- A variable in the application domain
- Retained over more than one cycle
- Influences subsequent cycles

**Examples:**
- Spacecraft attitude
- Number of bytes in the downlink buffer

We do *not* mean the “state of the computer’s memory” (which may be a state in some lower-level domain).
A DSL Recognizing State

• Define a **module** as the *context* for state
  – Message-passing
  – Actors
• C-like syntax (today)
• Keywords: **state** and **module**
  – **static** is not allowed
  – Pointers to state not allowed
  – No other way to define state

```c
module gnc {
    state GncVector x;
    state ControlState y;
    param GncParms z;

    function gnc_64_hz(int z);
    function init(void);
};
```
A Module Interface Function

Example:

```
module gnc {
  state GncVector x;
  state GncState y;
  param GncParams z;

  function gnc_64_hz(int z);
  function init(void);
}

function gnc_64_hz (int z) {
  using state GncVector x;
  ...
  next x.a = x.a + r(z);
  ...
}

Current x and next x are distinct.
```
Atomic Updates to State

• Current practice: Change state incrementally throughout a message-processing cycle
  – Is the current value of x the old state value, or the new one?
  – Easy to lose track

• Proper practice: State update automatic and atomic at the end of a message processing cycle
  – Computed next state distinct from current state
  – Current state does not change during message processing
Benefits

• Mathematically appropriate and safe
  – PDEs, estimation, finite-state machines... are of the form
    \[
    x_{t+1} = f(x_t, u) + v
    \]
    \[
    y_t = g(x_t, u) + w
    \]
    where \( x \) is a vector.

• Easier to write functional programs
  – Computed next state distinct from current state
  – Current state does not change during message processing
Graceful Degradation Fault Tolerance

The nature of the processing in this application allows easy implementation.

Lost cores means loss of performance only. Failed core Replacement core

Supervisor recovers full processing capability

Tilera

Devoted to Image Processing

Many cores dedicated to parallel processing of image data
Fail-operational Fault Tolerance

Fail Operational

Fault: Voter masks error

Detection and voting

Recover TMR

Recovering full TMR

Replacement core
Policy-based Computing

Supervisor
1. Initially creates the application
2. Monitors health and performs fault recovery
3. Carries out policies: power/cores/reliability

Rewire for TMR in real-time

Thread must be “repeatable”, e.g., no shared memory – a function.

Policy change:
Increase resilience
Automatic Telemetry

• New keywords
  – Channelized state telemetry
    • Keyword: eha (engineering, housekeeping, and accountability at JPL)
    • Downlink significant state variables
  – Event report
    • Keyword: evr (event reporting at JPL)
    • State change => an event
    • Should it be: event => state change ?

Translator handles all of the details.
State Checking

• Goal: Never reboot
• Collect all state in a *state dictionary*
• Automatically produce
  – Inventory
  – Spreadsheets for system engineers to specify desired/required states prior to each critical event
  – On-board checking, reporting programs
  – Ground display and analysis tools
State Details

• Static analysis: Verify that each variable declared to be state is indeed state

Let \( x \) be declared a **state** variable in module \( M \).
Define

\[
is_{-}\text{state}(x, M) =
\]

There is a module interface function \( F \) in \( M \):

There is a path \( P \) starting with \( F \) (possibly through calls to other functions):

The first access to \( x \) in \( P \) is a read (not a write).\]
Execution Models: The Bottom-line

- **Functional semantics**: Two functions are *concurrent* unless the output of one is an input to the other

- **Sequential semantics**: Two functions are *sequential* unless proven they can be made concurrent
Summary

• Hardware drives what we can do
  – A sea of cores
  – Power = Speed x Reliability
    • Computations that migrate, replicate, start/stop/repeat without concern
    • Policy-based computing
• Above suggests a functional language
• State in a functional setting => Language recognizes state
  – Separate current state from next state
    • Atomic state updates
  – Know entire state: no reboots
  – Automated telemetry


5. Fortress Programming Language http://projectfortress.java.net/


