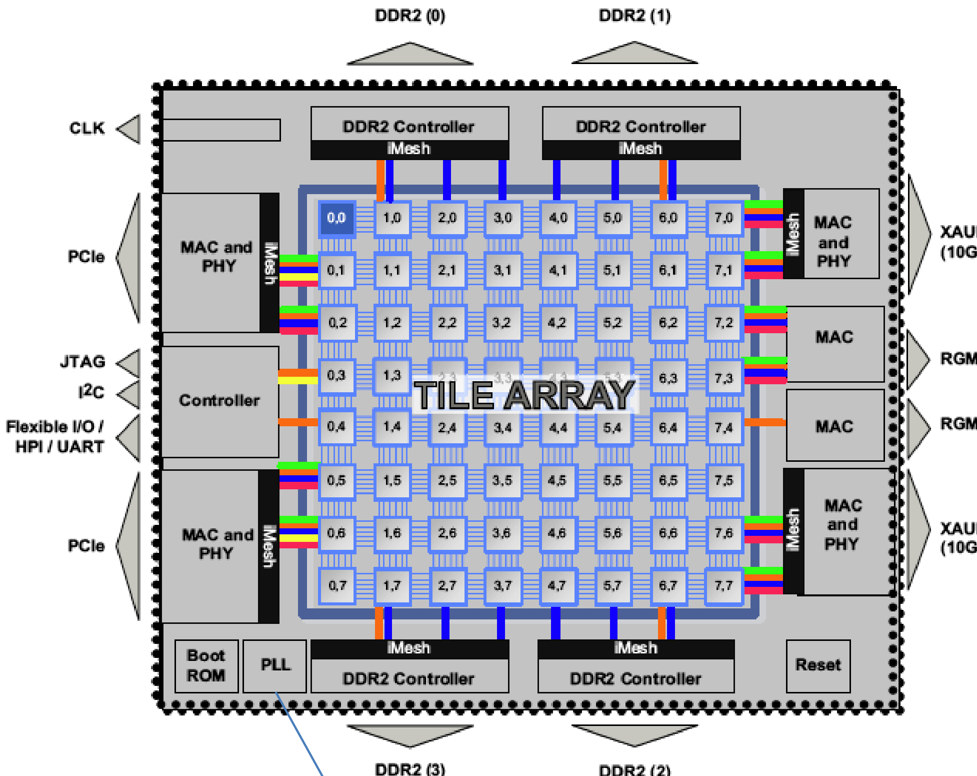
Capturing Flight Software Architecture With a Domain-Specific Language

Kim P. Gostelow

Jet Propulsion Laboratory, California Institute of Technology

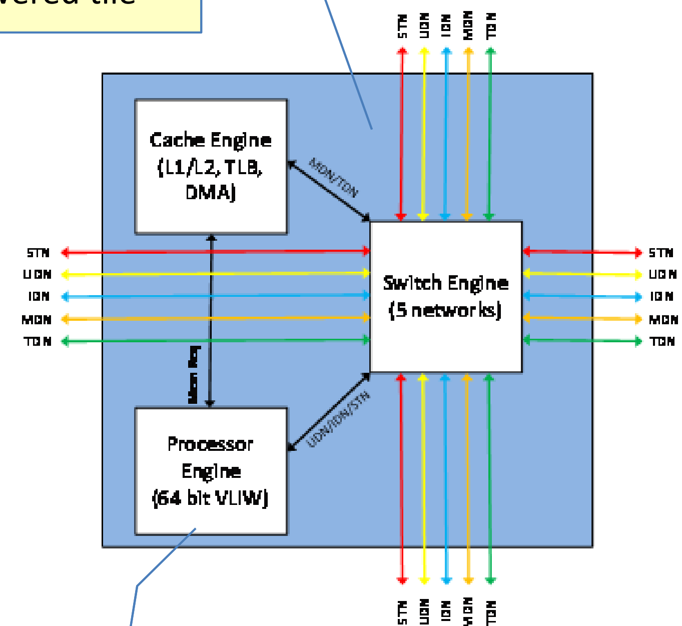
July 30 – August 3, 2012

Multicore



Tilera TILE64 Multicore Chip

Independently powered tile



A general-purpose CPU

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

Von Neumann (~ Clocked sequential circuit)	Functional (~ Asynchronous circuit)
An instruction executes when the program counter reaches it.	The function executes when the required data arrives.
Instructions manipulate the contents of memory cells.	Variables are mathematical variables, not memory cells - Contents cannot change once computed - No side-effects, no shared memory.

What is a Functional Language?

The relation f: A->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

    } Synonyms
  - No shared memory

Functions

The relation f: A->B is a ***function*** if:

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

Not a function

```
extern int sum;

int A(int a) {
  sum = get_time_of_day();
  for (int i=0; i<a; i++)
    sum += f(i);
  return sum;
}
```

A Function

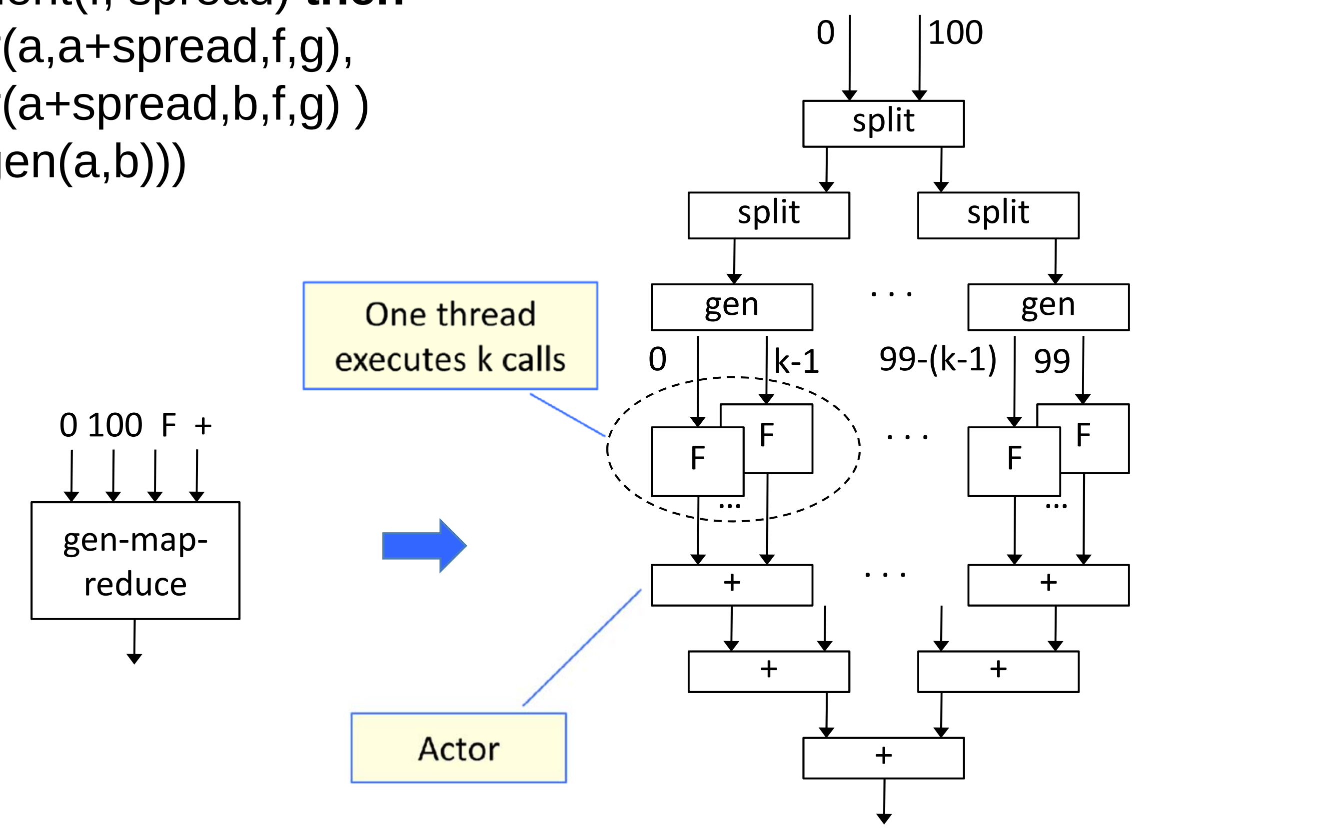
```
int B(int a, int time) {
  for i=0 .. a
    v[i] = f(i);
  return accum(v) + time;
}
```

Can run in parallel if f is a function

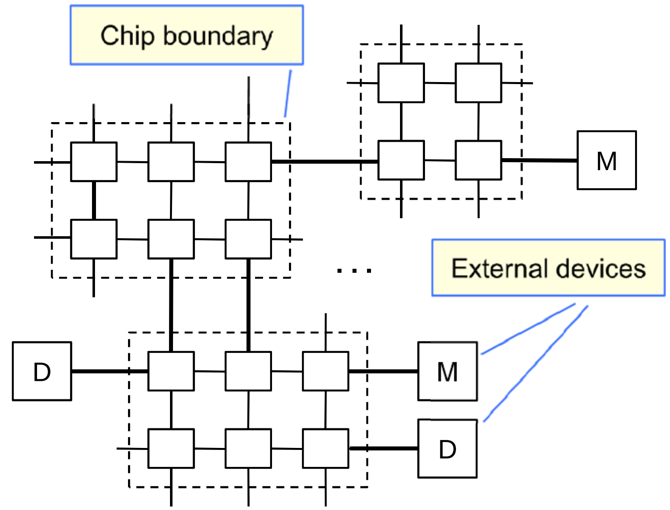
Example: generate-map-reduce

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



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

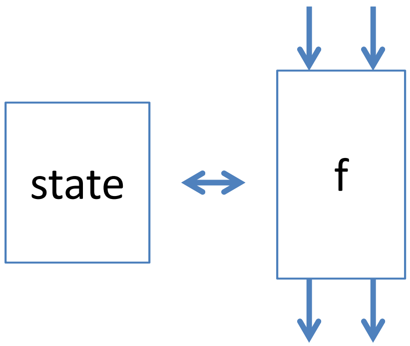
But, a system really *does* have state.

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

What's to be done?



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

```
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);
};
```

Module interface function:

Declaration

Definition

```
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

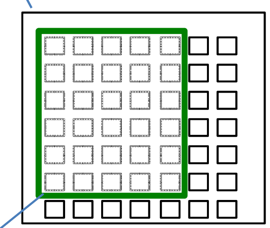
Graceful Degradation Fault Tolerance

Graceful Degradation

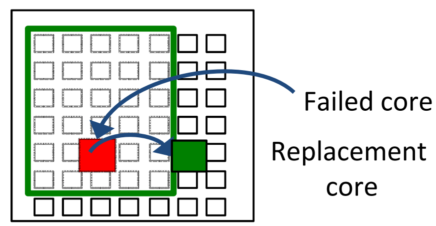
The nature of the processing in this application allows easy implementation

Tilera

Supervisor recovers full processing capability



Lost cores means loss of performance only.

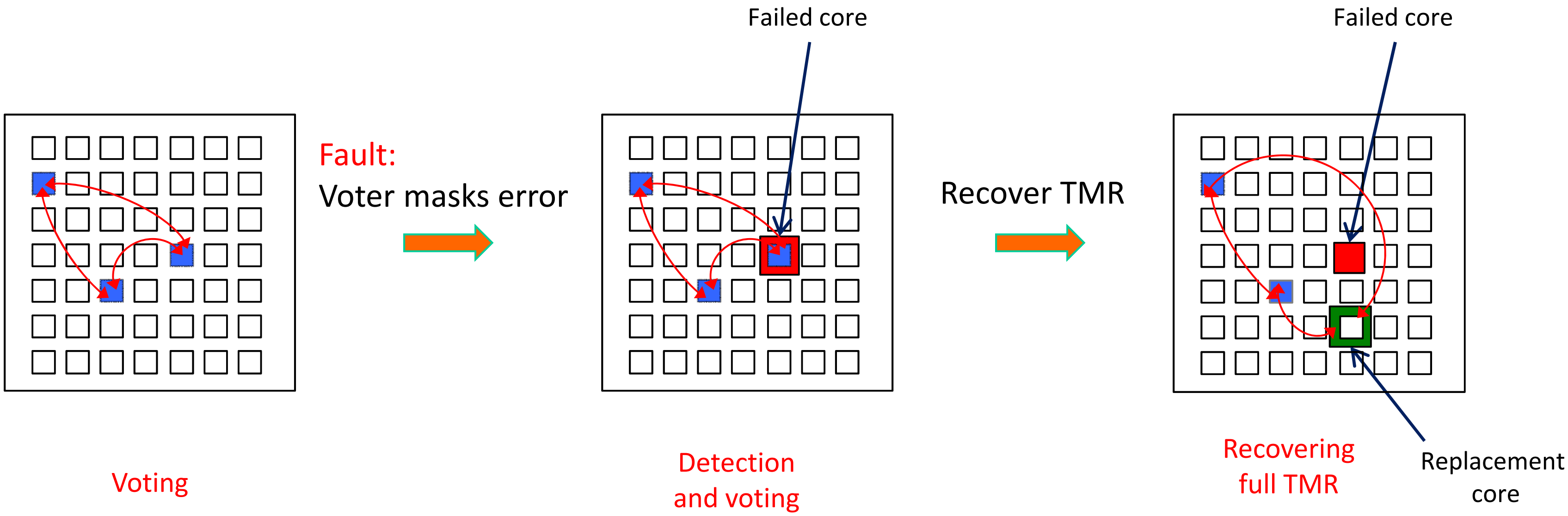


Devoted to Image Processing

Many cores dedicated to parallel processing of image data

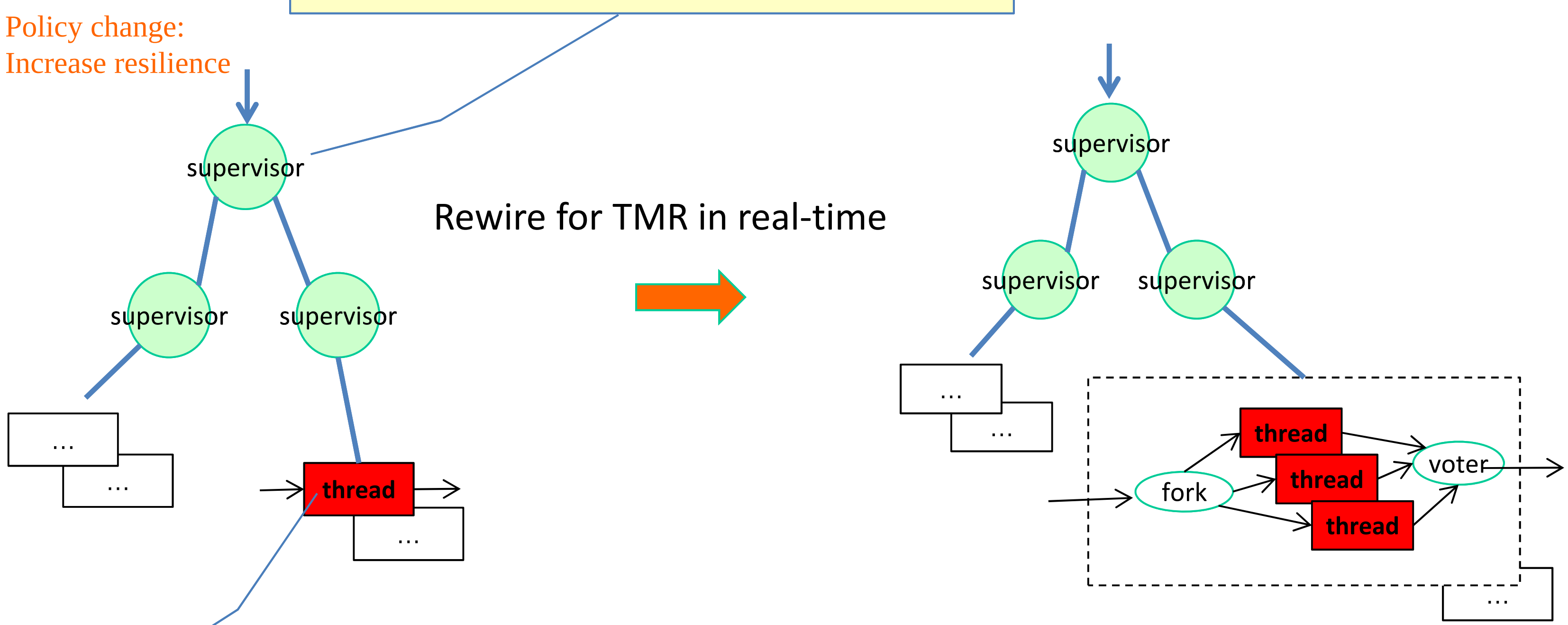
Fail-operational Fault Tolerance

Fail Operational



Policy-based Computing

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



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

Automatic Telemetry

Translator handles all of the details.

- New keywords
  - Channelized state telemetry
    - Keyword: **eha** (enginering, 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 ?

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_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

References

1. John Backus "Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs" Turing Award Lecture, Communications of the ACM, 21(8) August 1978, pgs. 614-641.
2. Joe Armstrong "Making reliable distributed systems in the presence of software errors" PhD Thesis, *Swedish Institute of Computer Science*, 2003.
3. M. Bennett, D. Dvorak, J. Hutcherson, M. Ingham, R. Rasmussen, D. Wagner "An Architectural Pattern for Goal-Based Control" IEEE Aerospace Conference. Big Sky, MT. March 2008 .
4. Kim P. Gostelow *Policy-based Computing and Extra-Functional Properties of Programs*. Presentation at the Software Working Group of the Fourth Workshop on Fault-Tolerant Spaceborne Computing Employing New Technologies 2011. Albuquerque, NM. May 22, 2011
5. Fortress Programming Language http://projectfortress.java.net/

References

6. Gostelow, Kim P. “The Design of a Fault-Tolerant, Real-Time, Multi-Core Computer System” IEEE Aerospace Conference, Big Sky, MT 2011
7. Dennis and Misunas “A Preliminary Architecture for a Basic Data-Flow Processor”. 1975 Sagamore Computer Conference on Parallel Processing
8. Arvind, Gostelow, and Plouffe “Indeterminacy, Monitors, and Dataflow” Proc 6th ACM Symposium on Operating Systems Principles
9. Arvind, KP Gostelow “The U-Interpreter” IEEE Computer 15(2): 42-49 (1982)
10. Ubiquitous High Performance Computing (UHPC) Solicitation Number: DARPA-BAA-10-37 (2010)