“An architecture for Scalable Concurrent Embedded Software"

No more communication in your program, the key to multi-core and distributed programming.

Eric.Verhulst@altreonic.com

Content

- Who’s Altreonic ?
- General context: Systems and Software Engineering
- About Moore’s imperfect law
- The von Neuman syndrome
- Why multicore, it is new?
- Where’s the programming model?
- The OpenComRTOS approach:
  - Formally modeled
  - Hubs and packet switching
  - Small code size
  - Virtual Single Processor model
  - Scalability, portability, ...
  - Visual Programming

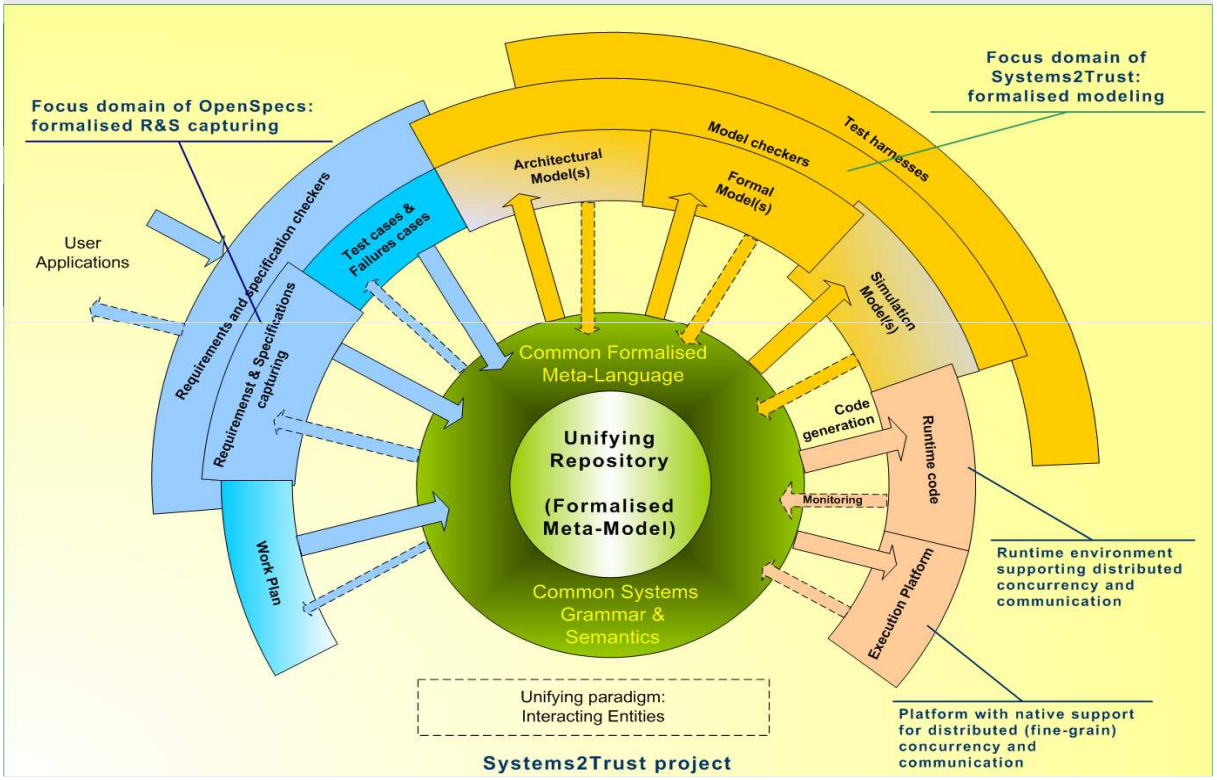
Who's Altreonic?

- Eonic (Eric Verhulst) : 1989 – 2001
  - Parallel RTOS Virtuoso (=> Wind River Systems)
  - Formally rooted in CSP (Hoare): "pragmatic superset of CSP"
- Open License Society: 2004 – now:
  - R&D on Systems and Software Engineering
  - Unified Semantics & Interacting Entities
  - Formally developed OpenComRTOS
- Altreonic: 2008 – now
  - Commercialises and develops OLS results

General context: SE

- Systems/software engineering requires common language in all stages for all activities
  - => **"Unified semantics"**
- Conquer complexity: => "**Interacting Entities"**
- Different views on same system, same semantics:
  - Requirements, specifications
  - Checkpoints, issues, change request
  - Modeling & Simulation
  - Failure view, testing view
  - Verification view, validation view
  - Workplan view

Unified Systems/Software engineering



Moore's law

- Moore's law:
  - Shrinking semicon features => more functionality and more performance
- Rationale: clock speed can go up
- The catch is at system level:
  - Datarates must follow
  - Memory access speed must follow
  - I/O speeds must follow
  - Througput (peak performance vs. latency (real-time behaviour)
  - Power consumption goes up as well (F^2, Vcc)
- => Moore's law is not perfect

The von Neuman syndrome

- Von Neuman's CPU:
  - First general purpose reconfigurable logic
    - Saves a lot of silicon (space vs. time)
    - Separate silicon architecture from configuration
      - "program" in memory => "reprogrammable
      - CPU state machine steps sequentially through program
  - The catch:
    - Programming language reflects the sequential nature of the von Neuman CPU
    - Underlying hardware is visible (C is abstract asm)
    - Memory is much slower than CPU clock (PC: > 100 times!)
      - Ignores real-world I/O
    - Ignores that software are models of some (real) world
    - Real world is concurrent with communication and synchronisation

Why Multi-Core?

- System-level:
  - Trade space back for time and power:
    - 2 x F > 2*F, when memory is considered
  - Lower frequency => less power (~1/4)
  - Embedded applications are heterogenous:
    - Use function optimised cores
- The catch:
  - Von Neuman programming model incomplete
  - Distributed memory is faster but
    - requires "Network-On- and Off-Chip"

Multi-Core is not new

- Most embedded devices have multi-core chips:
  - GSM, set-up boxes: from RISC+DSP to RISCs+DSPs+ASSP+... = MIMD
  - Not to be confused with SMP and SIMD
- Multi-core = manycore = parallel processing (board or cabinet level) on a single chip
- Distributed processing widely used in control and cluster farms
- The new kid in town = communication
  - (on the chip)

Where's the (new) programming model?

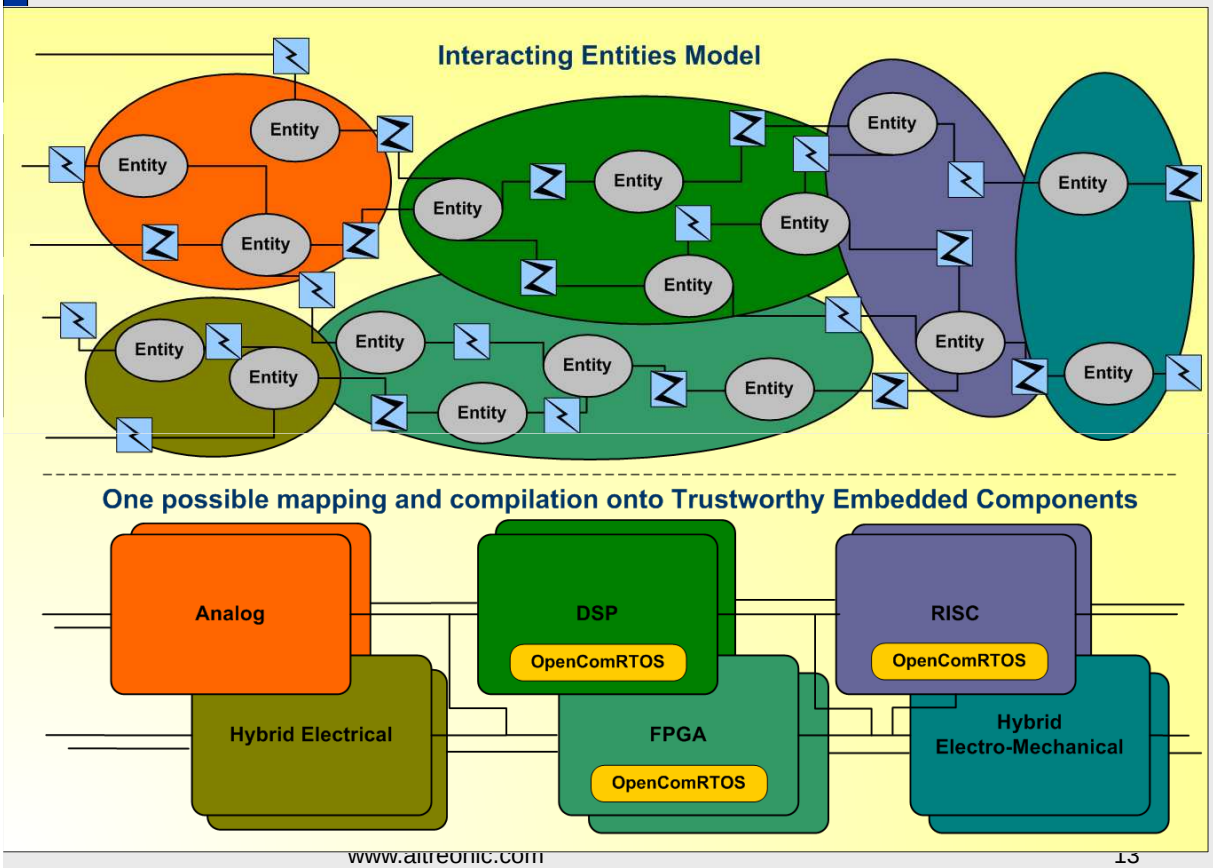
- Issue: what about the "old" software?
  - => von neuman => shared memory syndrome
  - But: issue is not access to memory but integrity of memory
  - But: issue is not bandwidth to memory, but latency
  - Sequential programs have lost the information of the inherent (often async) parallelism in the problem domain
- Most attempts (MPI, ...) just add a large communication library:
  - Issue: underlying hardware still visible
  - Difficult for:
    - Porting and Scalability
    - Often application domain specific

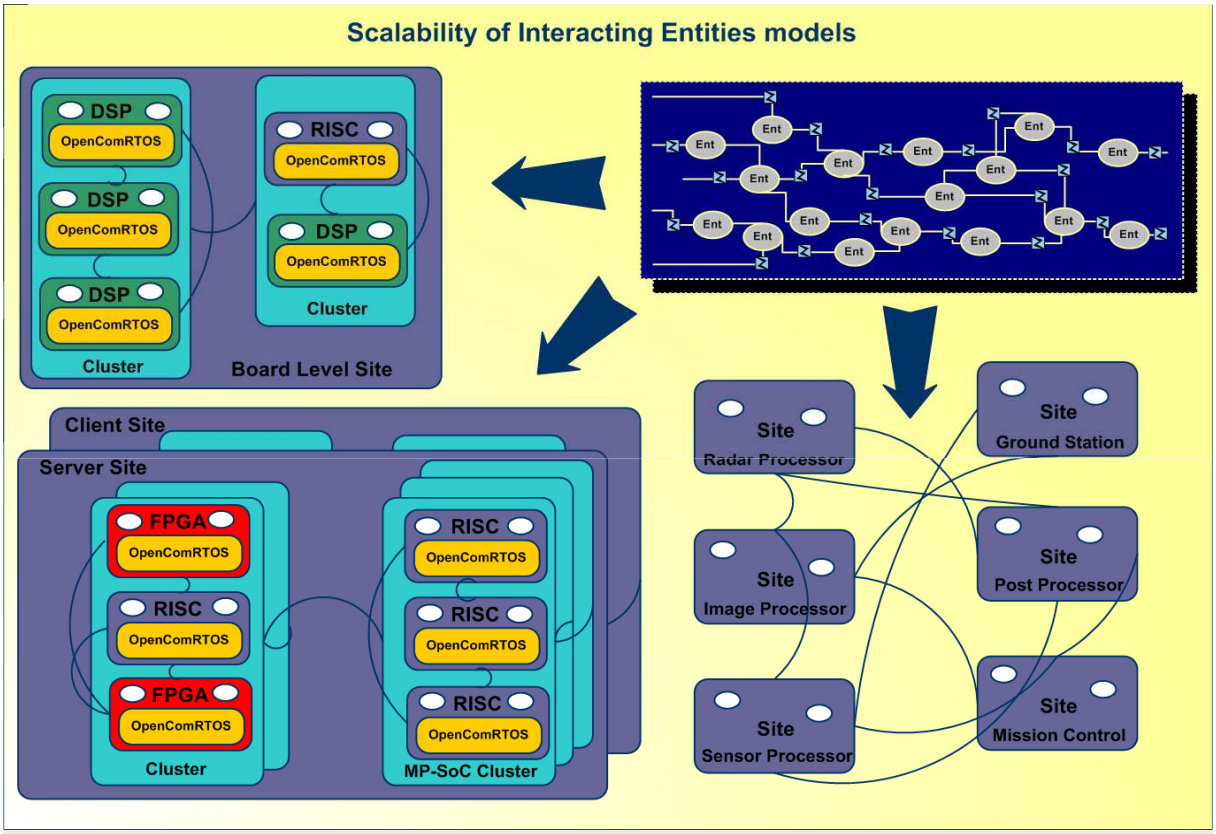
The OpenComRTOS approach

- Derived from a unified systems engineering methodology
- Two keywords:
  - **Unified Semantics**
    - use of common “systems grammar”
    - covers requirements, specifications, architecture, runtime, ...
  - **Interacting Entities** (models almost any system)
- RTOS and embedded systems:
  - Map very well on “interacting entities”
  - Time and architecture mostly orthogonal
  - Logical model is not communication but “interaction”

The OpenComRTOS project

- Target systems:
  - Multicore, parallel processors, networked systems, include “legacy” processing **nodes** running old (RT)OS
- Methodology:
  - Formal modeling and formal verification
- Architecture:
  - Target is multi-node, hence communication is system-level issue, not a programmer’s concern
  - Scheduling is orthogonal issue
  - An application function = a “task” or a set of “tasks”
    - Composed of sequential “segments”
    - Tasks synchronise and pass data (“interaction”)

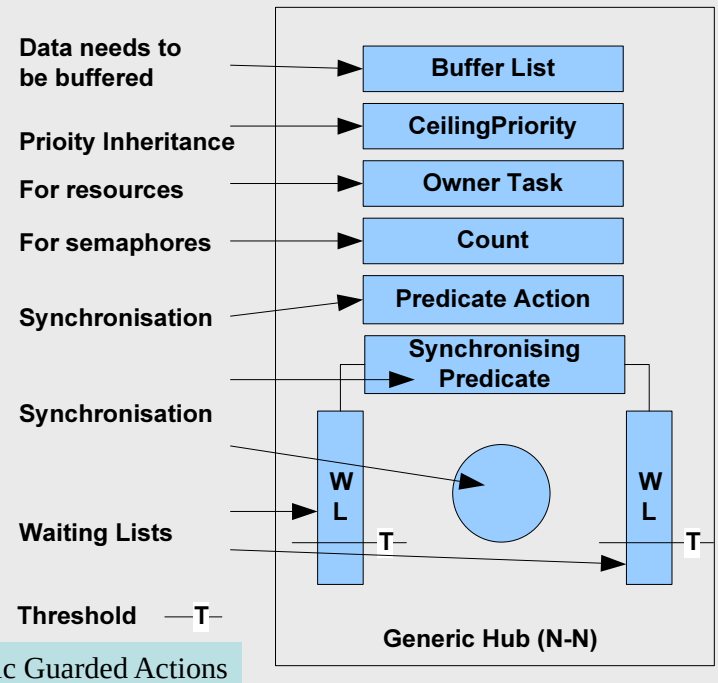

Interacting Entities Model
Entity
One possible mapping and compilation onto Trustworthy Embedded Components
Analog
Hybrid Electrical
DSP
OpenComRTOS
FPGA
OpenComRTOS
RISC
OpenComRTOS
Hybrid
Electro-Mechanical


Scalability of Interacting Entities models
DSP
OpenComRTOS
RISC
OpenComRTOS
Cluster
Board Level Site
Ent
Client Site
Server Site
FPGA
OpenComRTOS
Cluster
MP-SoC Cluster
Site
Radar Processor
Site
Ground Station
Site
Image Processor
Site
Post Processor
Site
Sensor Processor
Site
Mission Control

The OpencomRTOS “HUB”

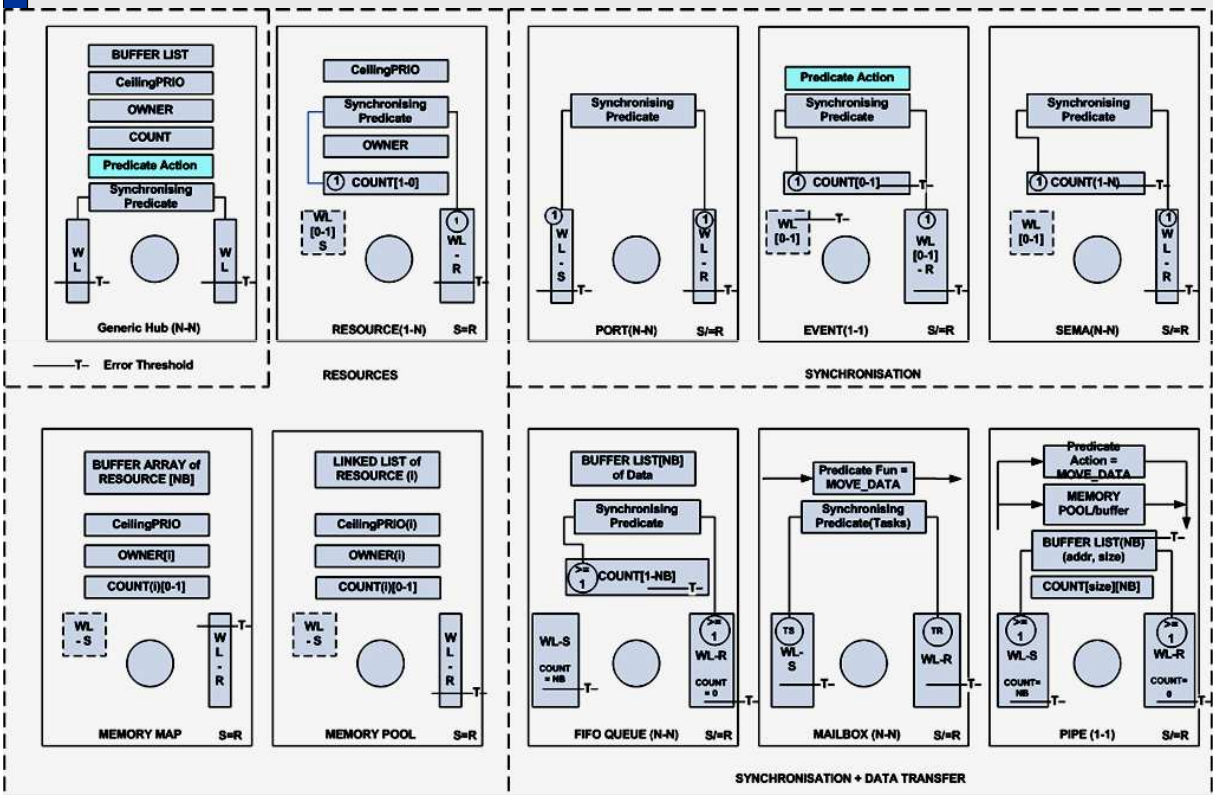
- Result of formal modeling
- Events, semaphores, FIFOs, Ports, resources, mailbox, memory pools, etc. are all variants of a generic HUB
- A HUB has 4 functional parts:
  - Synchronisation point between Tasks
  - Stores task’s waiting state if needed
  - Predicate function: defines synchronisation conditions and lifts waiting state of tasks
  - Synchronisation function: functional behavior after synchronisation: can be anything, including passing data
- All HUBs operate system-wide, but transparently: Virtual Single Processor programming model
- Possibility to create application specific hubs & services! => a new concurrent programming model

The generic hub as metamodel

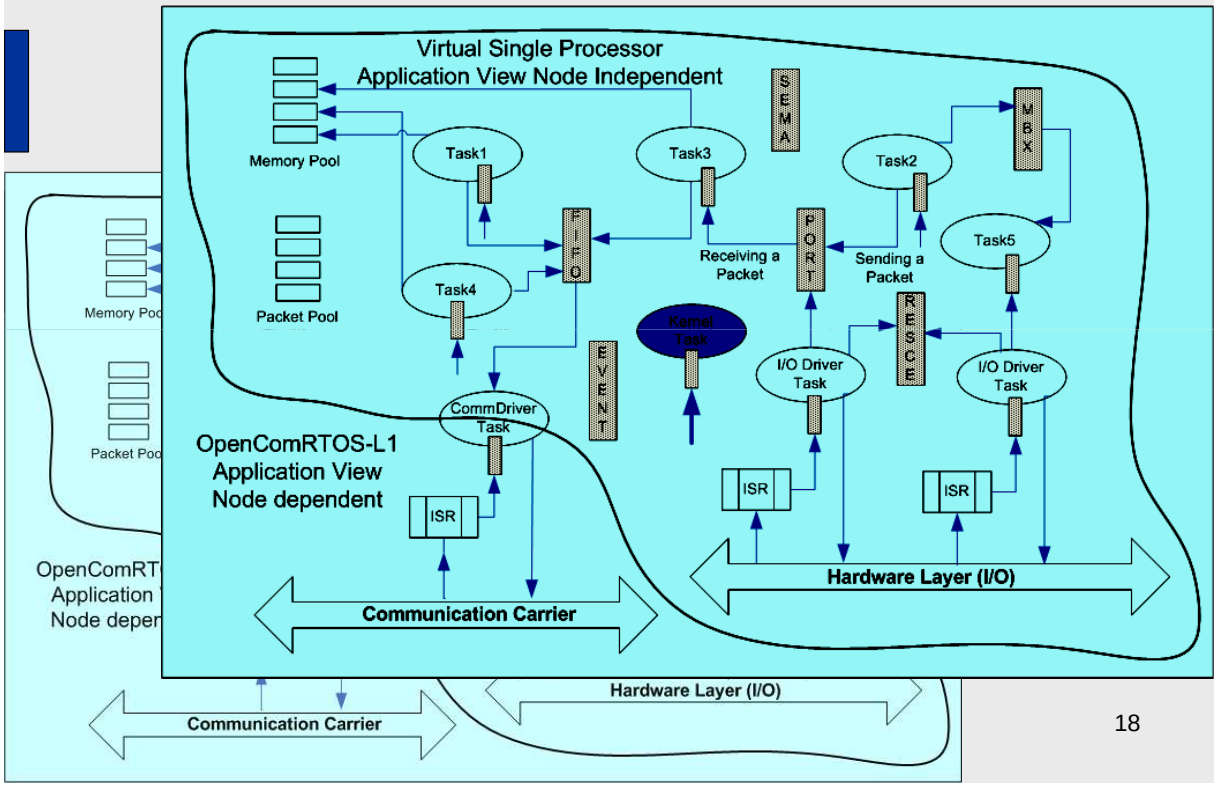


Similar to Atomic Guarded Actions
Or
A pragmatic superset of CSP

All RTOS entities are “HUBs”



Resulting programming model



Rich semantics: _NW|W|WT|Async

- L1_Start/Stop/Suspend/ResumeTask
- L1_SetPriority
- L1_SendTo/ReceiveFromHub
- L1_Raise/TestForEvent_(N)W(T)_Async
- L1_Signal/TestSemaphore_X
- L1_Send/ReceivePacket_X L1_WaitForAnyPacket_X
- L1_Enqueue/DequeueFIFO_X
- L1_Lock/UnlockResource_X
- L1_Allocate/DeallocatePacket_X
- L1_Get/ReleaseMemoryBlock_X
- L1_MoveData_X
- L1_SendMessageTo/ReceiveMessageFromMailbox_X
- L1_SetEventTimerList
- … => user can create his own service!

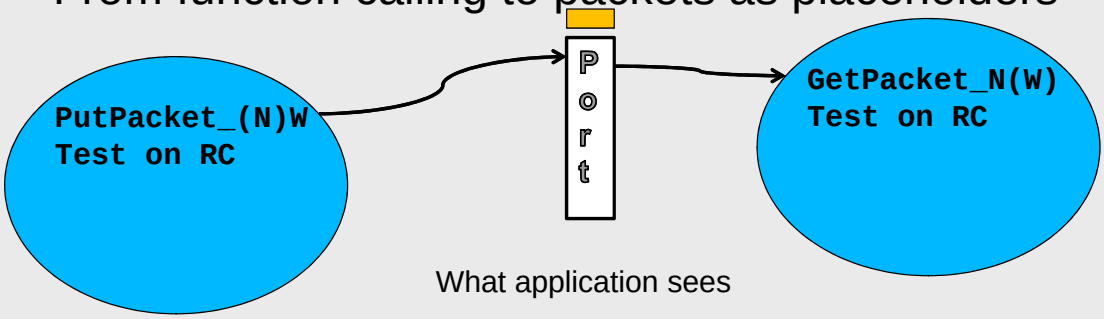
“Strange” semantics: _Async

- Example:

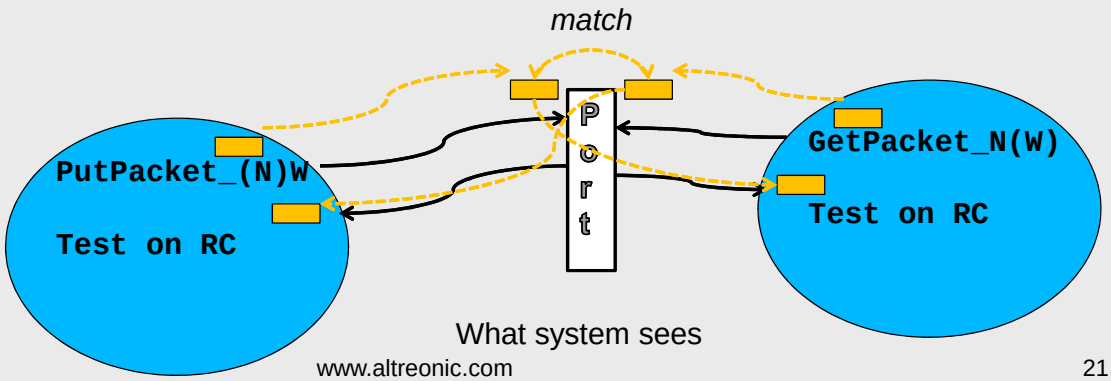
  L1_SendPacket_A (Port)
  L1_ReceivePacket_A (Port)
  ….
  L1_WaitForAnyPacket_W
- Asynchronous semantics are natural but tricky
- Sending async is easy to understand
- Receiving async: what does it mean?
- But: packets are a limited resource
  - => give each task a synchronisation credit = # packets
  - When credit is depleted => resynchronise
- Very handy for:
  - Drivers
  - Implementing “select” semantics (ALT ? and ALT ! In CSP)
  - But requires more control by programmer

Packets and Hubs

- From function calling to packets as placeholders



What application sees



What system sees

Unexpected: RTOS 5-10x smaller

- Reference is Virtuoso RTOS (ex-Eonic Systems)
- New architectures benefits:
  - Much easier to port
  - Same functionilaty (and more) in 10x less code
  - Smallest size SP: 1 KByte program, 200 byts of RAM
  - Smallest size MP: 2 KBytes
  - Full version MP: 5 KBytes
- Why is small better ?
  - Much better performance (less instructions)
  - Less power
  - Frees up more fast internal memory
  - Easier to verify and modify
- Architecture allows new services without changing the RTOS kernel task!

Clean architecture gives small code: fits in on-chip RAM

OpenComRTOS L1 code size figures (MLX16)				
	MP FULL		SP SMALL	
	L0	L1	L0	L1
L0 Port	162		132	
L1 Hub shared		574		400
L1 Port		4		4
L1 Event		68		70
L1 Semaphore		54		54
L1 Resource		104		104
L1 FIFO		232		232
L1 Resource List		184		184
Total L1 services		1220		1048
Grand Total	**3150**	**4532**	**996**	**2104**

Smallest application: 1048 bytes program code and 198 bytes RAM (data)
(SP, 2 tasks with 2 Ports sending/receiving Packets in a loop, ANSI-C)
Number of instructions : 605 instructions for one loop (= 2 x context switches,
2 x L0_SendPacket_W, 2 x L0_ReceivePacket_W)

Probably the smallest MP-demo in the world

	Code Size	Data Size
Platform firmware	**520**	**0**
- 2 application tasks **- 2 UART Driver tasks** **- Kernel task** **- Idle task** **- OpenComRTOS full MP (_NW, _W, _WT, _A)**	**230** **338** **3500**	**1002, of which** **- Kernel stack: 100** **- Task stack: 4*64** **- ISR stack: 64** **- Idle Stack: 50** **- 568**
Total	**4138 + 520**	**1002 + 568**

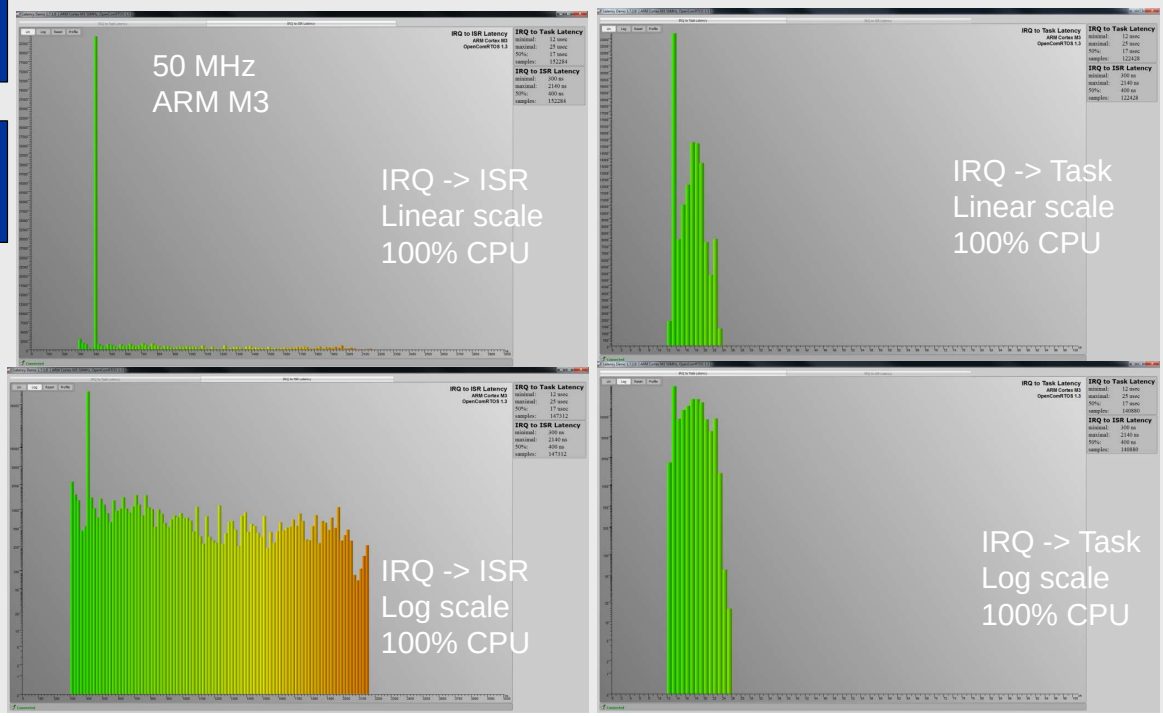
Can be reduced to 1200 bytes code and 200 bytes RAM

Standard processors (32bit)

OpenComRTOS L1 code size figures in bytes - Os				
Full support	MicroBlaze (100 MHz)		LEON3 (40 MHz)	ARM Cortex M3
	SP	MP	SP	SP
Hub shared	4756	5096	4904	2018
L1 Port	8	8	8	4
L1 Event	88	88	72	36
L1 Semaphore	92	92	96	40
L1 Resource	96	96	76	40
L1 FIFO	356	356	332	140
L1 PacketPool	296	368	268	120
Total L1 services	5692	6104	5756	2398
Driver (IntC -Timer)	812 - 284	852 – 284	1224 - 536	-
Uart driver		932		-
Semaphore loop	**33.64 usec**		**136.1 usec**	52,66 usec

Interrupt latencies

- From IRQ to first useful instruction in ISR or Task
- Not a single figure, an application dependent histogram



What influences Interrupt latency?

- Hardware:
  - Context switch
  - Bus congestion (multiple users)
  - DMAs
  - Peripherals disabling interrupts
  - “Special” support: HW loops, HW call stack
  - Interrupt controller
    - Jump table
    - Nesting capability, # int pins
- Software:
  - Critical sections, interrupts disabling sections
    - Data structures updating
    - Kernel loop (must be as small as possible)
      - In MP often multiple commands in kernel
    - Compiler conventions

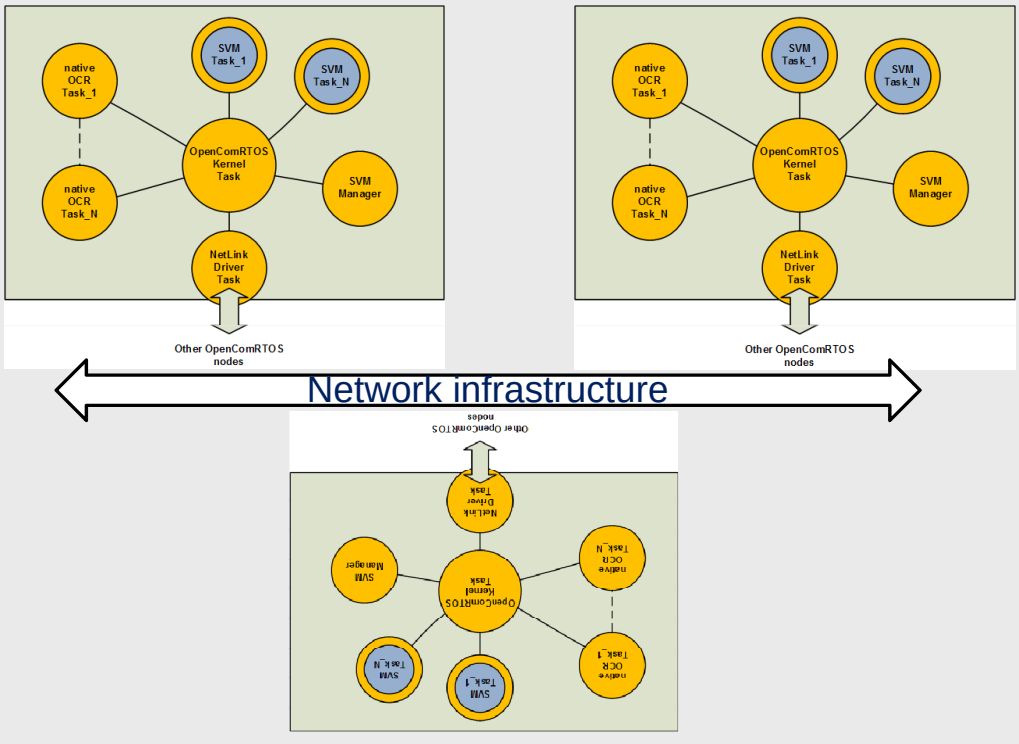
Program once, run anywhere

- Ultra low power:
  - CoolFlux DSP core (24bit, Harvard)
  - Code size full kernel: 2000w PM + 750w data
  - Interrupt latency:
    - IRQ to ISR: < 112 cycles
    - IRQ to task: < 877 cycles
  - Multicore capable
- Single chip multicore
  - Intel SCC 48core “super computer on chip + NoC switch” (in development)
- Heterogeneous targets:
  - Win32+Linux+ARM+MicroBlaze+XMOS+LEON3+ … demo programmed as single target

A Safe Virtual Machine for C

- Goal:
  - CPU independent programming
  - Low memory needs (embedded!)
  - Mobile, dynamic code
- Results:
  - Selected ARM Thumb2 instruction set of VM target
    - Compactness
    - Widely used CPU
    - **3.8 Kbytes of code for VM**
    - Executes binary compiled code
    - Capable of native execution on ARM targets
    - VM enhanced with safety support:
      - Memory violations
      - Stack violations
      - Numerical exceptions

Safe VM set-up



Universal packet switching

- Another new architectural concept in OpenComRTOS is the use of “packets”:
  - Used at all levels
  - Replace service calls, system wide
  - Easy to manipulate in datastructs
  - Packet Pools replace memory management
- Some benefits:
  - Safety and security
  - No buffer overflow possible
  - Self-throttling
  - Less code, less copying

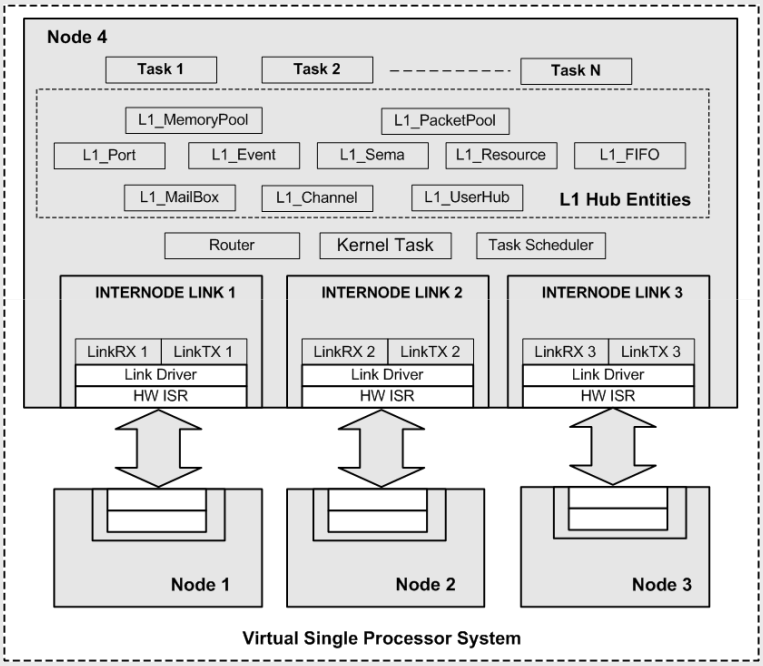
Transparent communication

- Tasks only “communicate” via Hubs
- Real network topology
  - Logical point-to-point links between nodes
  - Node Rx and Tx link driver task for each link end
  - Routing and gateway functionality
  - Works on any medium: shared buses, “links”, “tunneling” through legacy OS nodes using sockets, …
  - Heterogeneous and transparent parallel programming!
- Link driver tasks
  - OpenComRTOS application task with (TaskInput) Port
  - Driver task type per link type (UART, TCP/IP, …)
  - Not present/visible on the (logical) application level

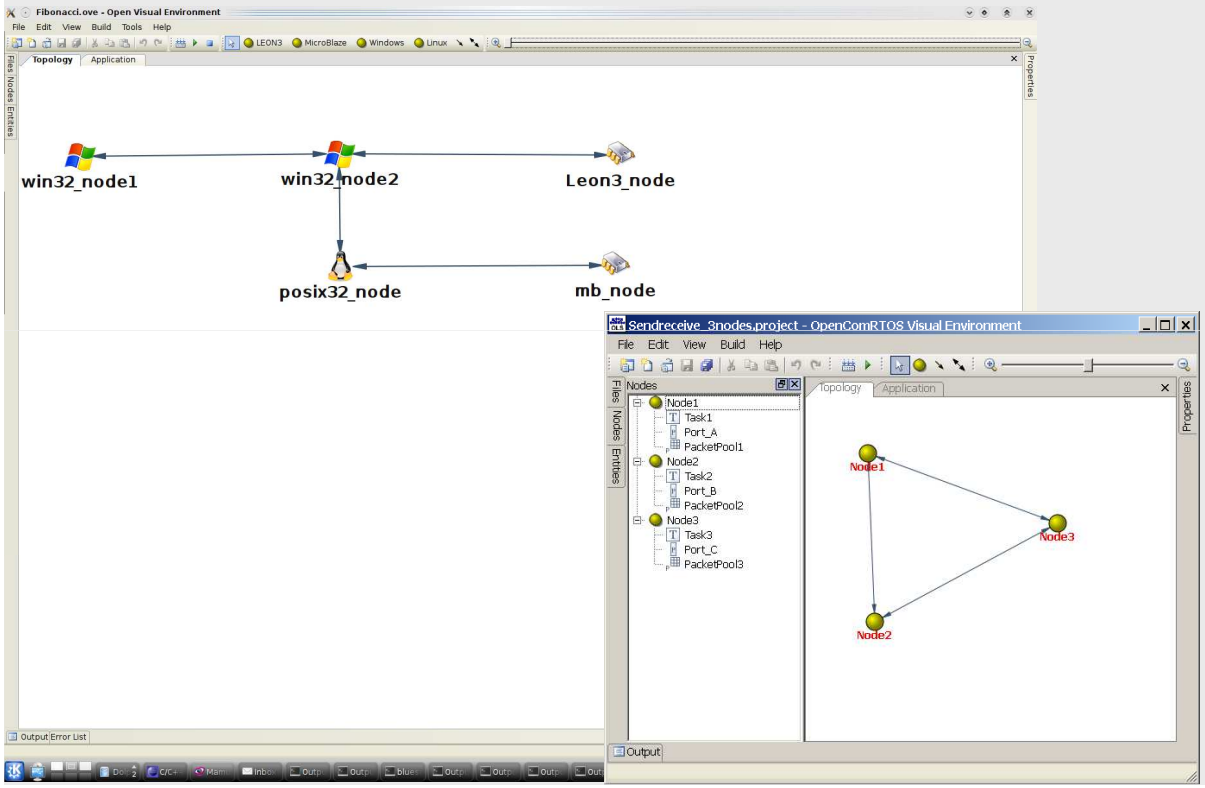
“Link” driver functionality

- Target/link specific communication implementation
  - L0_RxDriverFunction – retrieve packet from “wire”
    - E.g. socket read for Win32, Linux, ..
    - E.g. buffered UART communication for embedded target
  - L0_TxDriverFunction – put packet on “wire”
    - E.g. socket write for Win32, Linux, ..
    - E.g. buffered UART communication for embedded target
  - network <-> host byte ordering functions
  - Normal ISR framework can be used as applicable
- But fully transparent for the application software

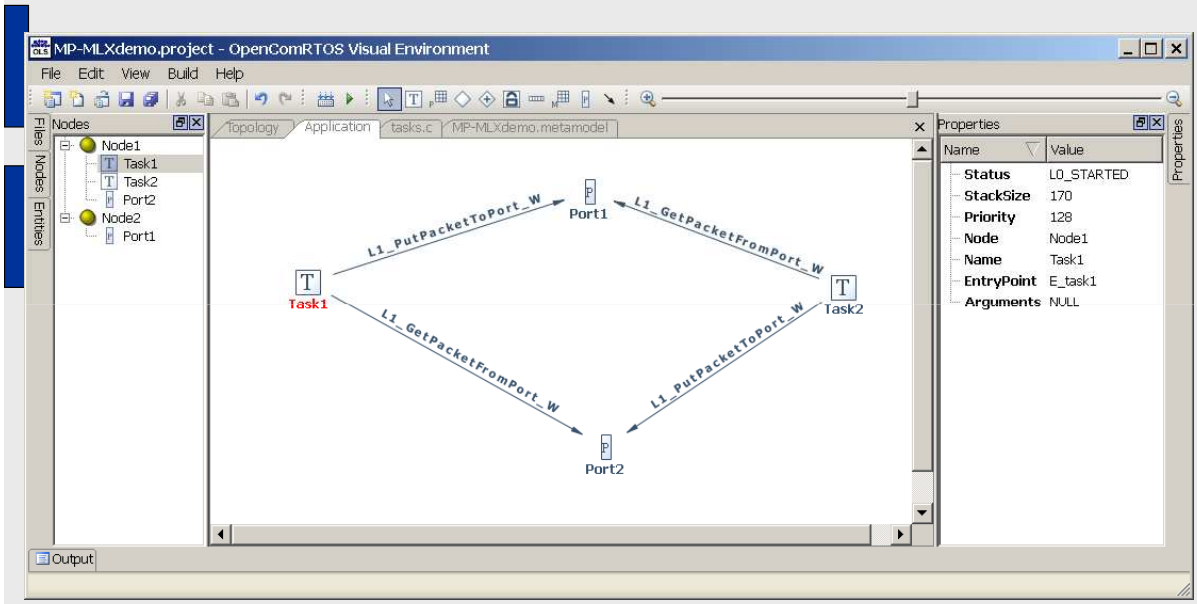
Virtual Single Processor programming model



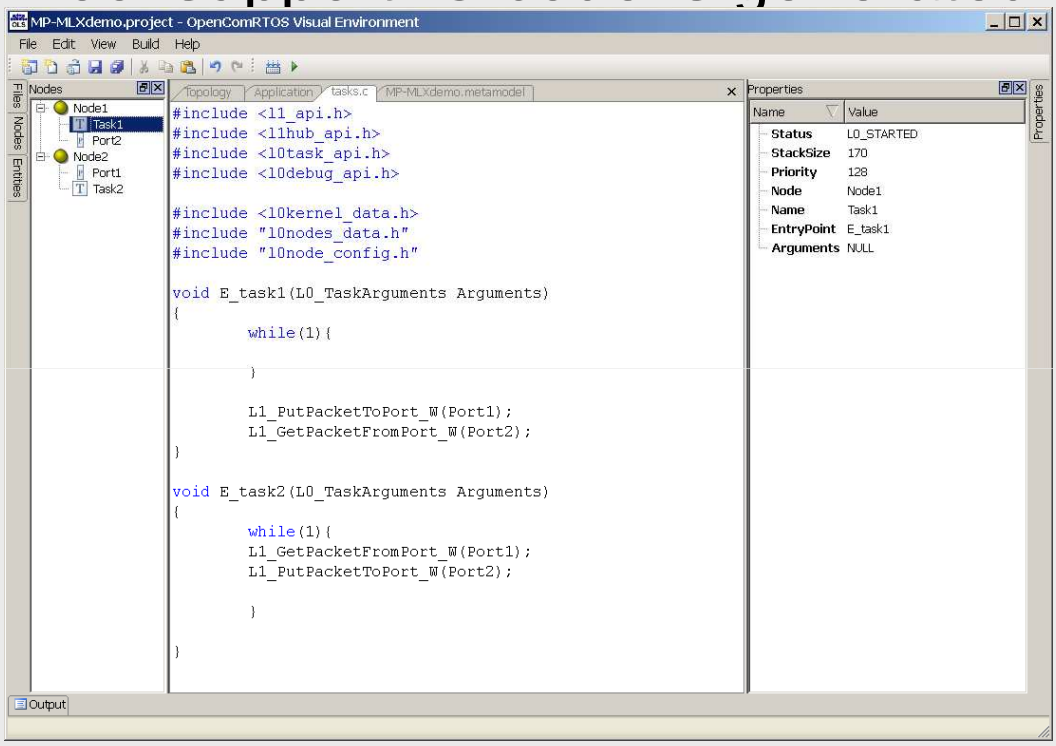
Tool support: Define Topology



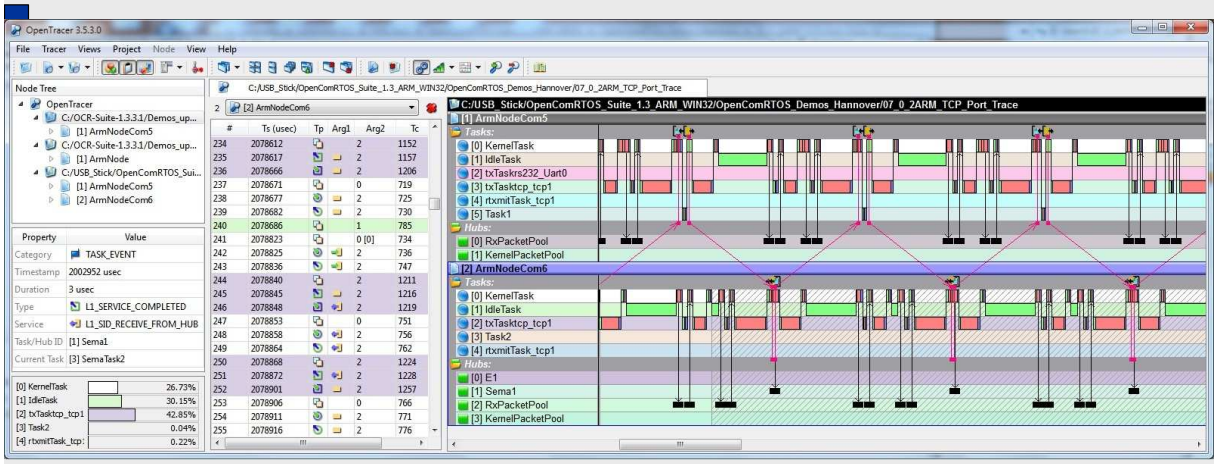
Tool support: Define Application



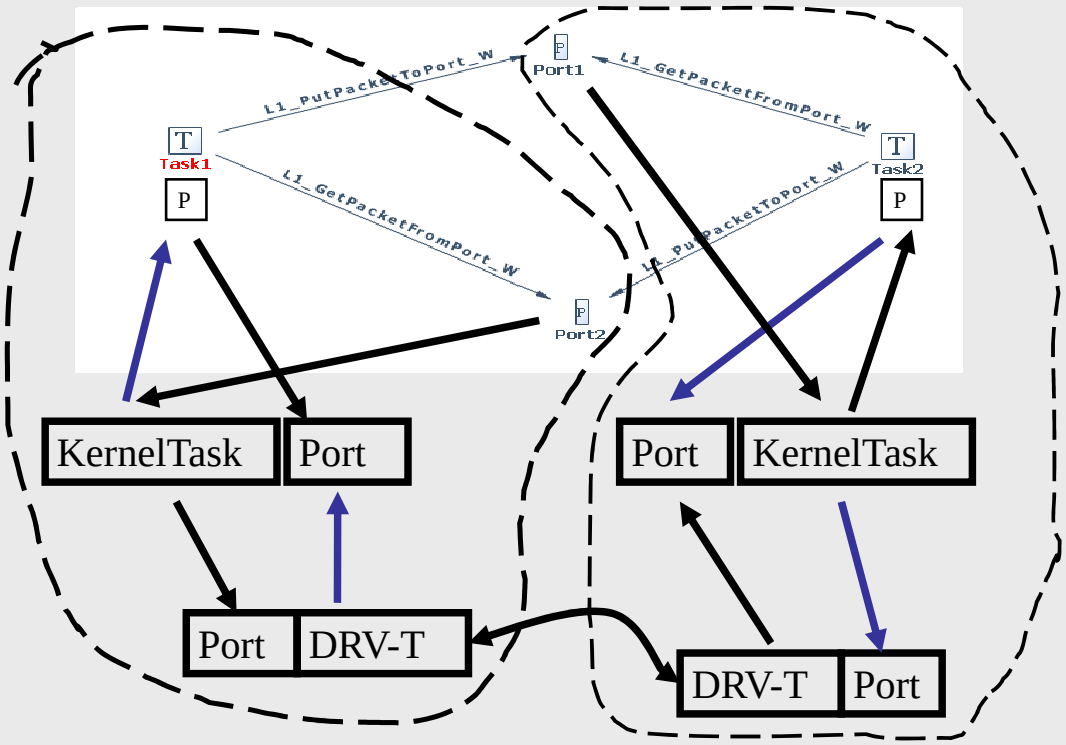
Tool support: C code is generated



Tool support: Run and trace



Under the hood

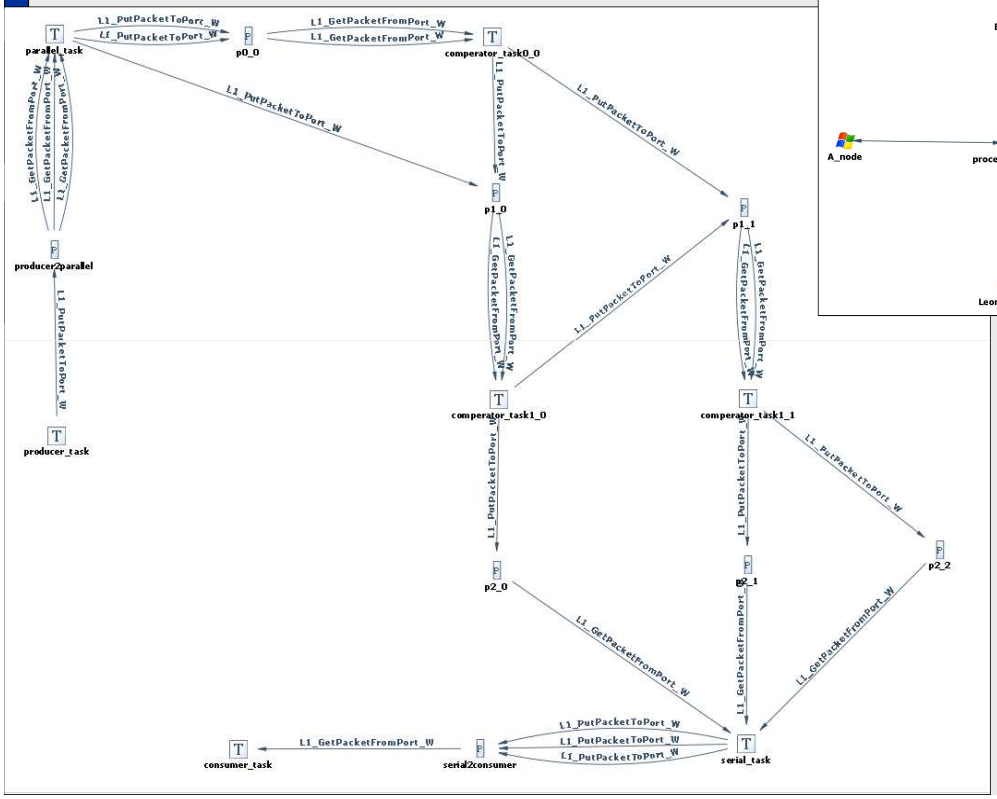


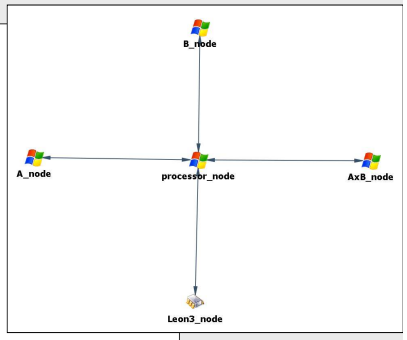
Heterogenous demo set-up

- Nodes: Leon3-WIN32-Linux-MicroBlaze
- Each “node” runs on instance of OpenComRTOS
- Only changes are the node-adresses
- Source code everywhere the same:

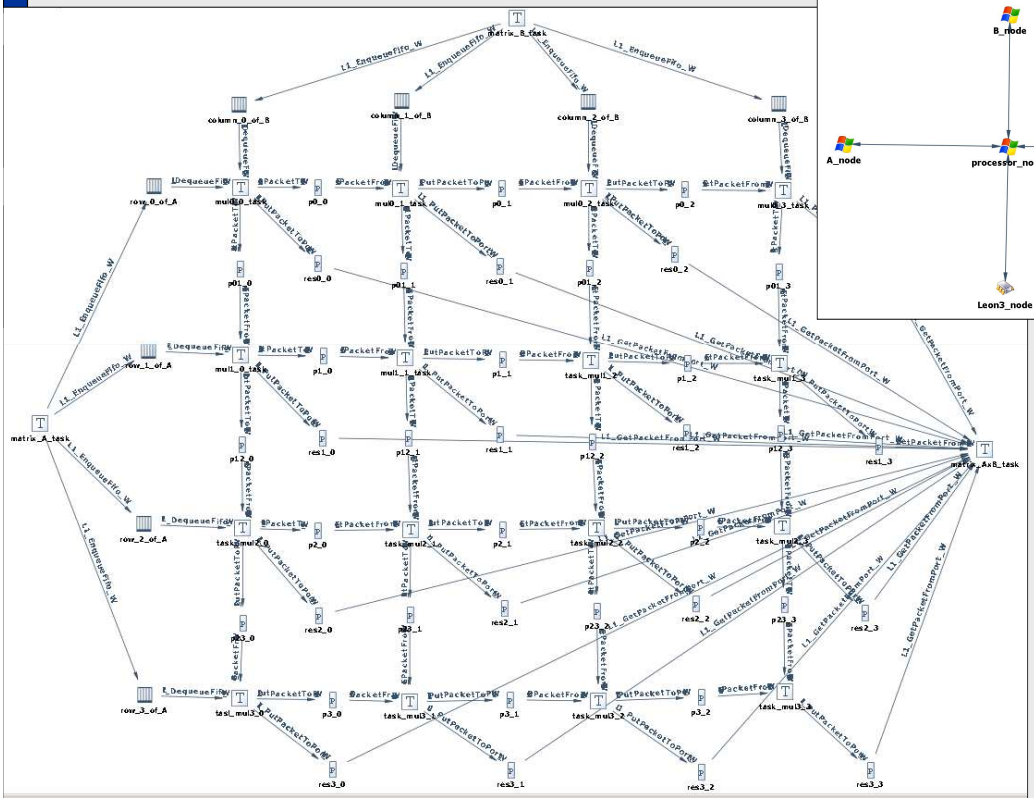
```
....
L1_PutPacketToPort_W (Port1)
...
L1_SignalSemaphore_W (Sema1)
...
```

Parallel sort





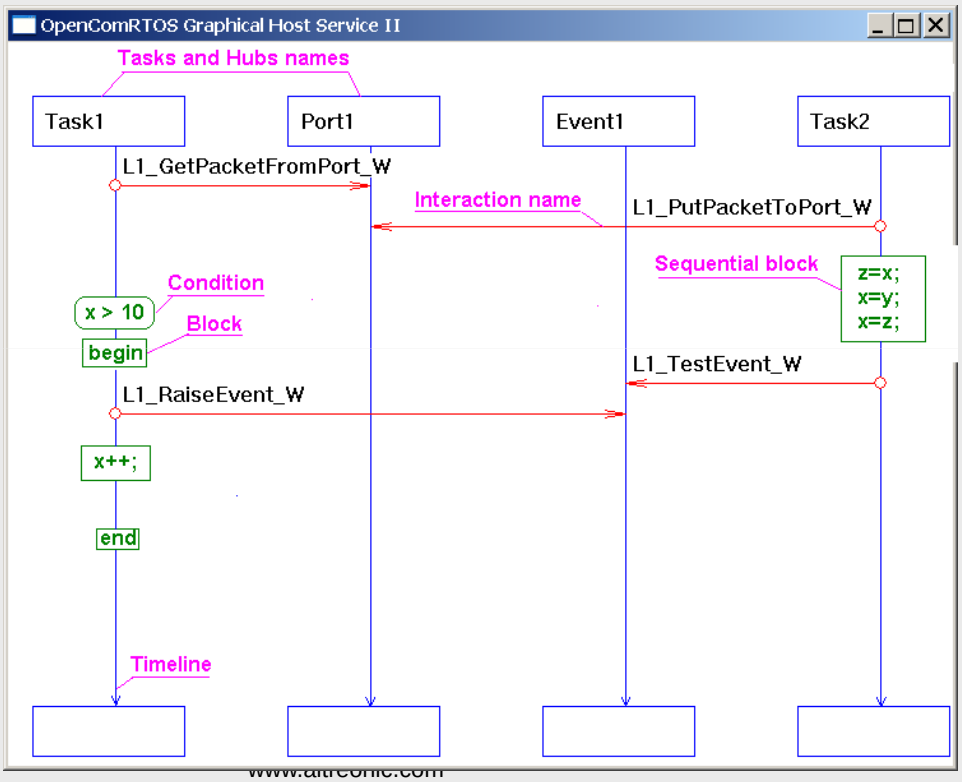
Parallel matrix multiplication



Roadmap

- Working on:
  - Interaction Sequence Diagram editor
  - Integration with MAST scheduleability analysis
  - Flowchart editor with C Code generation
  - Broadcast, barrier hubs
  - Protocol hubs:
    - Composing protocols from micro-protocols
  - Dynamic resource scheduling
  - Transparent fault tolerance
  - Back-end formal verification

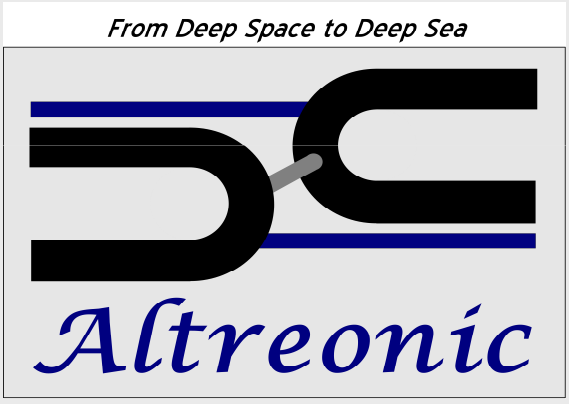
Prototype Interaction Sequence Chart



Conclusions

- OpenComRTOS is breakthrough “RTOS 2.0”
  - Network-centric => system communication layer
  - Priority or timer based scheduling => RTOS
  - Formally developed
  - Fully scalable, very safe, very small
  - Better performance
  - Portable & user-extensible
  - => Concurrent programming model
  - => works for any type of “multicore” target
  - ....
- Contact: Eric.Verhulst @ altreonic.com

From theoretical concept to products



“If it doesn't work, it must be art.
If it does, it was real engineering”