



## ***POETS Orchestrator architecture***

### ***Design notes***

#### ***Volume I - Introduction***

*Ver 01*

*27 November 2018*

*Andrew Brown, Andrey Mokhov, Simon Moore, Jeff Reeve, Julian Shillcock, David Thomas*

#### **Documentation structure**

- Volume I :** Introduction  
The big picture.
- Volume II :** Application definition  
The interface between the domain-specific front ends and the Orchestrator XML file specification.
- Volume III :** The Orchestrator (internals)  
Internal structure and function of the domain-agnostic software layer.
- Volume IV :** The Orchestrator (user-facing)  
User documentation, and its interaction with the Orchestrator internals.
- Volume V :** Associated tools and capabilities  
Extra functionality overlaid upon the Orchestrator (simulation) plus stand alone graph and test circuit generators.
- Volume A :** Appendices  
Lists, tables enumerated constants, structure layouts and details, worker class documentation.



Revision history

Revision	Date	Description

This is an evolving set of documents, and the level of detail varies wildly throughout, from syntax diagrams to "...and then a miracle occurred....".

Throughout the lifetime of the project, the intention is that more and more details will be included, things will change, and the miracles will morph into hard quantitative details.

New stuff will appear, almost as if it had been intended to be there all along.

This is an Orchestrator design document. It is not an overall POETS design, or a hardware design, or a front-end design document. These require higher-order magic.

# ***POETS Orchestrator architecture***

## ***Design notes***

### ***Volume I - Introduction***

#### **Executive precis**

#### **1. Introduction**

- 1.1 The snake-oil pitch**
- 1.2 Application background**
- 1.3 The system**
- 1.4 The role of the Orchestrator**

#### **2. POETS - the system**

- 2.1 Background**
- 2.2 Mode of (abstract) operation**
- 2.3 A hardware model**
- 2.4 The (software) Orchestrator**
- 2.5 Hardware - Orchestrator interaction**



## 1 Introduction

### 1.1 The snake-oil pitch

Moore's Law has given us a doubling of logic density every eighteen months or so for over four decades. It has enabled microelectronics to move from a narrow professional niche into the hands and pockets of every consumer in the world. However, as process geometries continue to shrink towards the scale of the atom, we face the emergence of fundamental limits which the scaling of current methodology can no longer overcome; increasingly, far ranging architectural changes are required to utilise the potential of the technology. Three major challenges can be identified:

- Power dissipation: it is already not possible to power all parts of a chip at the same time (the *dark-silicon* problem). It has been demonstrated that multiple small CPUs are correspondingly more power efficient than fewer large ones, so the deployment of large cohorts of small CPUs is an obvious way forward.
- As process geometries continue to shrink, issues of reliability and robustness inevitably emerge. In a system of millions of cores, it is unrealistic to expect 100% functionality 'out of the box'; equally, cores will inevitably fail over the lifetime of the system. A POETS system is capable of inconspicuous self monitoring (and possible recovery) and graceful performance degradation in the face of core or communication fabric failure.
- A traditional argument against moving to large numbers of cores is the relative cost of computation and communication, currently around  $10^3$  in favour of computation - moving data (messages) traditionally is very (and non-linearly) expensive. POETS sidesteps this by 'embedding' the cores in a hardware communications fabric (which is truly parallel) and in which the messages are small and of fixed size (a few bytes). With POETS machines, the burden of high-level message choreography is completely removed: systems trade cores against complexity in both compute and communications. Much work is required to understand how to optimise the scheduling of workloads on such machines, but the nature of the task is changing: in the past, a large application was distributed 'evenly' over a few processors and much effort went into scheduling to keep all of the processor resources busy; today, the nature of the cost function is different: processing is effectively a free resource. The automatic (high-level) parallelisation of general-purpose codes remains a 'holy grail' of computer science, but fine-grain parallelisation is frequently signposted by the underlying mathematics. The problem has been in the past that solutions emerging naturally from the numerical solution technique do not map well (cheaply) onto existing architectures. POETS machines change this: the operational 'sweet spot' lies in the regime of low-level, high density message processing.

The problem is not *creating* large cohorts of processors, but how they might be productively *used* to perform or enable the sort of analyses that users of big compute demand. The project focuses on the potential of the hardware architectural point on the scale represented by the earlier SpiNNaker work. It will look at the areas of work where the hardware architecture would be well suited, how those needs can be supported on this architecture by methods and tools, and how the architecture may be further optimised, with the objective of providing the basis of knowledge to support valuable commercial exploitation opportunities anticipated to emerge for commodity HPC.

**How can meaningful computation be carried out in an environment such as POETS?**

POETS is an *event-driven* system. Cores carry out small calculations in response to the arrival of a message, based on a state subset held in local memories. These calculations may/may not result in the emission of further messages, which are immediately swept up by the communication infrastructure and delivered asynchronously, via hardware, to their target core. The target core is woken (by the hardware delivering the message), acts upon it - as above - and *returns to quiescence*, awaiting another stimulus. (This attribute is responsible for the energy frugality of POETS - you only power calculations when you perform calculations.) This is a programming model of immense power and enormous potential, and is completely orthogonal to conventional architectures.

**Why should we bother?**

Despite decades of attack, the general purpose parallelisation problem remains one of the most elusive "holy grails" of computer science. Inspired - possibly - by what we achieved by simply ignoring difficult general problems in our past neural simulation work, and focussing on the functionality we actually *needed*, our thesis here is that POETS is incredibly well-suited to an unexpectedly wide range of important engineering and physics problems, most of which are traditionally the domain of large, extremely resource hungry supercomputers. Event-driven programming, using thousands to millions of small, cheap, energy frugal cores is by far the best platform for some massive engineering problems that traditionally consume millions - tending to billions - of core-hours *and* watts, both of which translate directly into money.

**Proponents of exascale computing need event-driven machines if budgets are to remain sub-exascale.**

**What needs to be done?**

Our past work delivered the first large-scale existence proof of the power of this concept. If we are to exploit this hitherto underexplored and unconventional computing technology, there is still much research to be done. For nearly every step of the development trajectory to date, almost every tool and technique that a conventional software developer takes for granted has had to be re-engineered from scratch. Conventional support tools do not work with this system. We need standardised input formalisms, we need command, control, internal visibility and debug tools, we need to know where the limits are of an instance of the architecture, and how difficult and expensive it is to move these limits.

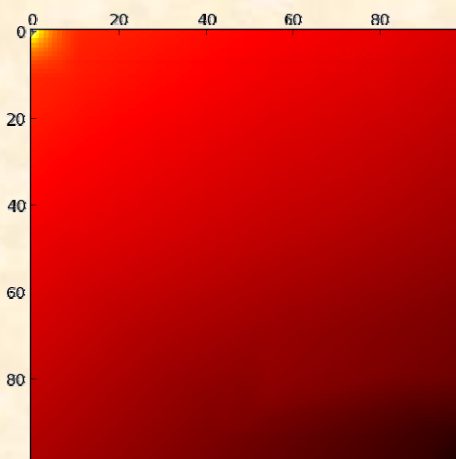
**Use case**

A large proportion of real engineering problems can be broken down to a discrete mesh, albeit one with sometimes millions of nodes. If we have millions of cores and a fast communications infrastructure, we can trade cores off against computational complexity, and exploit the near-perfect parallelism of the hardware interconnect.

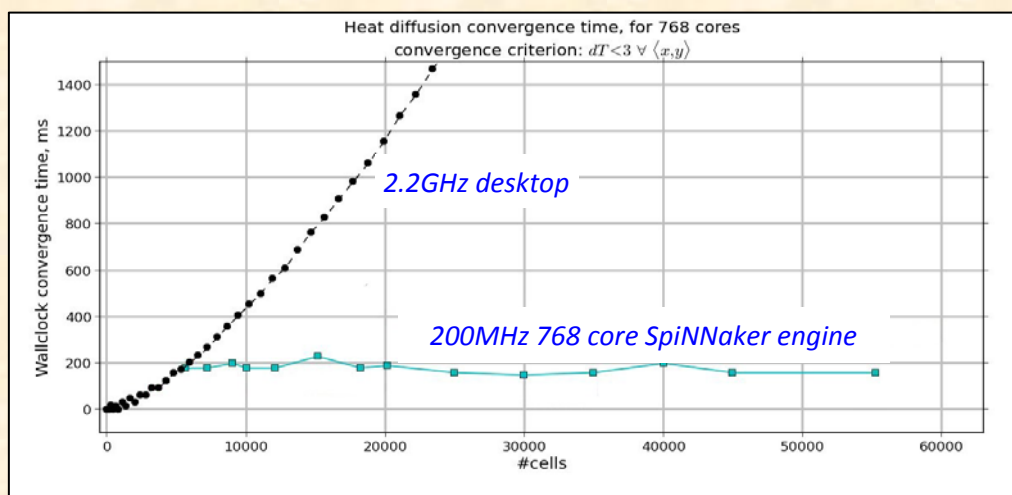


### Use case: Finite difference calculations

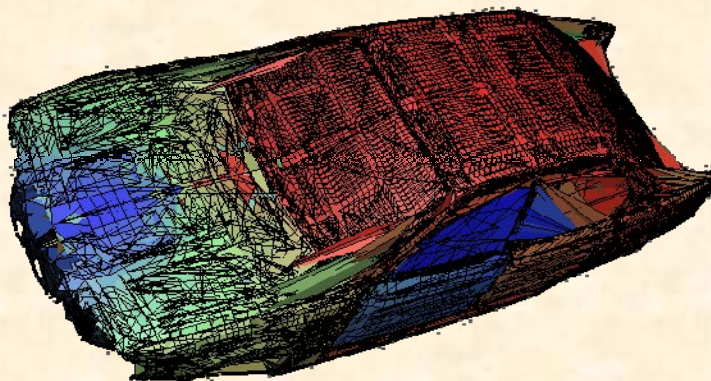
Consider the canonical finite difference heat equation on a 2D square grid: *each grid point is represented by an individual core*, which holds the grid point state (temperature) plus ghosts of the immediate neighbouring states, and communicates only with its direct neighbours by messages. On receipt of a message - any message - a grid point recomputes its state; if this has changed, it broadcasts the new state value to its neighbours. All the cores do this simultaneously (asynchronously), triggered by the arrival of messages. Pinning the opposite corner temperatures and letting heat flow freely produces the obvious result.



The interesting point is the wallclock solution time as a function of grid size: the algorithm, as cast, will continue to operate in constant time, using more and more cores as the user increases the grid size, until it runs out of physical resource.



There is, obviously, no reason why we need restrict our analyses to a uniform grid:







## 1.2 Application background

POETS is a hardware/software computational *accelerator*. It provides no capability that cannot be obtained from a conventional machine cluster. *For certain classes of problem*, however, it can provide speedups of several orders of magnitude. Further, *for certain classes of problem*, the throughput scales as a constant with problem size, up to some useful limit. This limit is set by the hardware, and can be easily moved by changing the hardware resource available - POETS is currently rooted in FPGA technology, but we leave open the possibility of non-FPGA implementations in the future.

The design intention is that POETS provides speed by exploiting low-level parallelism. For a given well-formed (from the perspective of POETS) problem, it will deliver a certain speedup. This speedup is intended to be constant, up to some (useful) application size. After this point, the application simply will not fit into the hardware, and no computation is possible at all. Adding hardware (the system is extensible) does not increase any speedups or throughput; it increases the size of the problem that can be handled.

POETS is not a general purpose machine. The characteristics of the computation are complex, and the expectation is that low-level POETS programs will be auto-generated by domain-specific specialised front ends (Applications), tailored to be sympathetic to the expectations, knowledge-base, and requirements of domain-expert user groups.

### *What class of problem solution technique can benefit from POETS technology?*

Problem domains for which POETS is useful are those in which the problem description can be represented as an (arbitrary) **application graph** (- this topology is *independent* of the POETS engine topology), and fall into two broad classes:

**Class A:** Systems capable of representation by discrete graphs where the gross behaviour is dominated by local interactions. Examples here include

- Wire-grid and spatial tiling of a volume for physical systems (diffusion- and transport-based problems, both continuous and discrete).
- Systems where the computational graph is divorced from the physical embodiment (electronic systems, communication networks).

Less obvious, perhaps, are

**Class B:** Systems that are to some extent embarrassingly parallel, but the set applicable here is somewhat broader than the kind of computation usually associated with the term, and has some overlap with class A problems above. A natural question is: *What problems fall into this category?*

Examples include any problem that can be progressively solved by the accumulation of many small computations, each of which adds something to the solution, an increment of accuracy, an increment of extra phase space volume searched, or an increment in alternative outcomes that have to be taken into account. Problems consisting of many average events with a few outliers that rely on huge numbers of trials or calculations can be solved much faster. Search algorithms over a space; summation of probabilities of independent events that aggregate to



give a lifetime or transition probability: POETS can overwhelm a problem with simulations and find the small number of desired solutions buried among a huge number of dull solutions.

This problem class may be further subdivided:

**B1:** The gradual accumulation of incremental results - for example, the slow increase in accuracy that typically currently resides in longer runs of, for example, a Monte Carlo simulation. Here the observables of the system are progressively better estimated by performing longer and longer averages over trajectories through the phase space of the system.

These problems can be considered solutions of deterministic differential equations, and are effectively another strategy to solving class A problems, where the solution gets more accurate the finer the grid and the longer a simulation is run, for example integrating the equations of motion of many point particles moving under Newton's laws as in a molecular dynamics simulation.

**B2:** The discovery of an unexpected outlier that is typically obscured behind a huge number of average results - the ability of a massively parallel system to explore paths that would almost never be found by a serial system because they are hidden within an ensemble of a vastly larger number of average paths in which a conventional computer spends most of its time. Effectively, it is *trading many cores for time* and bringing the discovery of rare events forward by starting the simulations from different points in the search space. (A difficulty here is that if the search space is so large that a single run cannot explore a large part of it, it may also be so large that even many runs do not sample all the interesting regions to start from.) However, if there are barriers in the space that limit the ability of any one run to explore the space, then many distinct starting points may remove or ameliorate the effect of the barriers.

Examples here include

- Simulation of N years of evolution in a set of interacting species on an island. As each stage in evolution is the combination of small random changes subjected to external pressure from the environment and other species, a huge number of independent evolutionary trajectories could be explored using POETS. As trajectories approach steady-states (extinction, dominance of single species) they can be stopped and resources freed for further simulations. The few simulations that lead to non-trivial trajectories can be pursued and used to predict likely outcomes. This is a tool in conservation ecology whereby human interventions can be simulated to explore how best to bring about desired changes in an ecosystem or, alternatively, how best to minimise such changes.
- Disease progression in a population, where the number of actors and transmission mechanisms are large, and some paths may be unknown: By assigning different behaviours to different actors, a variety of responses to interventions, transport disruption, personal behaviour and so on, can be explored and the set of trajectories pruned to explore how to maximise the likelihood of a desired outcome, for example minimising the number of infected individuals or minimising the spread of the disease.

In this type of problem, the most important properties of the system being simulated are the correlations between its state transitions. If the system consists of non-interacting actors, such as the molecules of an ideal gas, there is no advantage to simulating huge numbers of trajectories - they all contain equivalent information. (An analogy with poker is useful here: if





one simulates the shuffling of a pack of cards a large number of times, and deals poker hands to a computer player, and then follows the results of each hand, nothing interesting will be learned unless the player can learn to change their playing strategy between hands. Simulating a large number of independent games will simply produce the probability of getting a particular hand, but the player does not benefit from this unless their future behaviour is modified as a result.)

On the other hand, if each hand is used to probe the response of the *other* players in the (simulated) game the player can use the huge number of simulations they can run to explore strategies against other players. If, simultaneously, the other players are also testing the player there is the means for feedback to drive the system into new states with long-term memory.

B2 problems are like solving the Fokker-Planck equation where the probability distribution gets more accurate as more simulations are run, revealing lower probability events in the solution space proportional to the number of simulations run in parallel. This can generate unexpected results whose probability is tiny because the large number of runs is sufficient to find them.

**B3:** A type of problem in which a large number of parallel simulations are run and the appearance of desired states in any of them propagates to other simulations and directs the parallel explorations through phase space by revealing and over-expressing correlations that appear in the problem. An example here is

- Optimising electronic synthesis involves the exploration of a behavioural design space, the limits and local geometry of which are defined by the circuit under synthesis. The most general purpose and robust approach to this is usually some form of simulated annealing, where the system traces a path through some multi dimensional design space in search of a "better" solution. Replacing the (single valued) trajectory through design space with a tree (each branch overseen by an independent POETS process) allows the tree to dynamically prune itself and grow extra branches in response to the local characteristics of the space being searched.

Potential application domains include:

- Discrete simulation (of just about anything: electronic logic, Petri nets...)
- Analogue simulation (electronic circuits, continuous systems)
- Hybrid simulation (neural systems - the SpiNNaker algorithm)
- Genome pattern matching
- Gyrokinetic plasma modelling
- Weather modelling (upper atmosphere and terrestrial)
- Tomographic image processing
- Image-guide surgery
- Drug screening
- Computational chemistry
- Neural synthesis
- Particle-particle and particle-field systems

In short, anything that can be transformed or decomposed into a large number of simple processes with a communication pattern characterised by asynchronous short-range messages.

**Key points:****Architecture**

- Extremely large numbers (1000000+) of *extremely* simple cores
- Short (a few bytes), uniform messages
- Hardware massively parallel communications network (on and off-chip)

**Advantages**

- Massive speedups for certain classes of problem:  $O(n^2) \rightarrow O(1)$
- Highly fault tolerant
- Low power: **25000 cores < 13A**

**Disadvantages**

- Not a general purpose architecture
- Cannot port existing codebases
- No existing support toolsets

**End users for POETS technology:**

<b>Who</b> are they?	Engineering, drug exploration, industrial tomography, image processing, simulation, big physics, weather modelling, genome scanning... industry, not domestic.
<b>What</b> do they want?	Commodity HPC: To be able to run complex, real-world simulations and explorations <i>without concern for the cost, either in terms of money or time.</i>
<b>Where</b> do they want it?	Desktop: The use should be as unthinking as a traditional compiler or simulator.
<b>When</b> do they want it?	Whenever they choose - and this is important: " <i>I'll just try this</i> " lets engineers stay focussed on the problem, not the implementation of the solution process. And, of course, " <i>this</i> " can be a raft of potential solutions in parallel.
<b>Why</b> do they want it?	If HPC access costs disappear from budget sheets, the specification and acquisition moves down the management/accounting chain to the realm of the people that actually use the tools. (Twenty years ago, computing hardware was a major capital item, and the maintenance infrastructure important continuous costs. Now, powerful PCs appear on budget sheets - <i>if at all</i> - as stationery.)



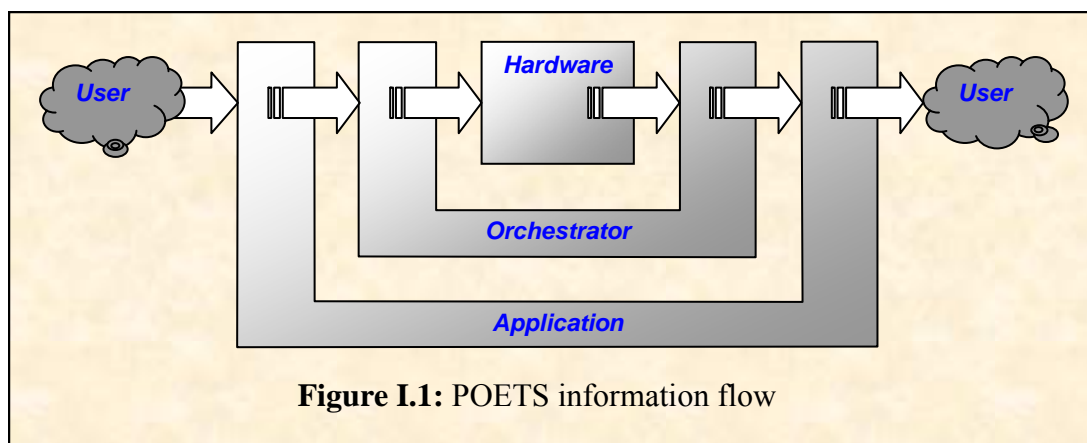
### 1.3 The system

POETS is a hardware/software computational *accelerator*. Broadly, it consists of three layers/components, outlined in figure I.1.

- The **Application**: This is the user-facing domain-specific high level software component that supports user interaction and performs user-level computation. Overall, the application consists of a set of processes, resident in different parts of the hardware ensemble that together create and control the graph-based representation of the user problem (in POETS known as a graph of **devices**) and its execution. Its main task is to translate the user problem into a form suitable for POETS processing, and also to *define* the lower members of the process hierarchy necessary to support this: the (set of) **Supervisor** (devices) and device realisations that oversee the computation at different levels of granularity.
- The **Orchestrator**: This is an intermediate level of *domain-agnostic* software that brokers and organises communication between the Application and the underlying hardware.
- The hardware **POETS engine**. This is the foundation hardware that delivers the computational results, to be exfiltrated by the Supervisor(s) (under the control of the Orchestrator) and handed out to the Application.

This set of documents are the principle design documents for the *Orchestrator*. It is an evolving set. It contains

- A brief and broad overview of the set of problem domains susceptible to the computational approach of POETS.
- An outline of the broad characteristics of the hardware and the Application, of sufficient detail to provide context, but focussed on their interfaces with the Orchestrator. *In these areas*, sufficient detail is given to provide a basis for the design decisions described relating to the Orchestrator.
- Detailed design description of the internals of the Orchestrator and its subsystems.



**Figure I.1:** POETS information flow



## 1.4 The role of the Orchestrator

The role of the Orchestrator is to oversee and control

- The mapping of a set of domain-specific, *arbitrary* application graphs onto the POETS hardware: This (the hardware) consists of a *fixed* compute hierarchy of small, conventional multi-threaded compute cores. Multiple users may run multiple application graphs simultaneously on the machine (known as **tasks**) - from the perspective of the hardware there is nothing to distinguish between a single disjoint application graph and a number of simply connected application graphs.
- The runtime management of the system: the Orchestrator provides a single, console-driven user interface so that an operator can oversee the loading, starting, stopping and unloading of tasks.
- Bulk data exfiltration: Such data as is generated by POETS is distributed throughout the hardware of the machine; when a task terminates, this must be collected, collated, triaged as necessary and presented to/by the overseeing MPI universe as conventional (set of) files.
- The dominant mode of usage is in batch mode: application graphs (tasks) may be loaded, set up, executed and unloaded. However, an extension to the original functionality allows interaction from individual task owners with their individual tasks; the Orchestrator brokers this mechanism.
- A job control subsystem allows console-free system control, so that users may submit jobs to the system with no human overseer.



## 2 POETS - the system

### 2.1 Background

From the project proposal:

*POETS - **Partial Ordered Event Triggered Systems** - technology is based on the idea of an extremely large number of small cores, embedded in a fast, hardware, parallel communications infrastructure - the **application network** communication is effected by small, fixed size, hardware data **packets** (a few bytes).*

*The POETS project describes research to investigate and prototype a software methodology and associated hardware platform to realise the potential of this architecture.*

*The physical implementation of such a system imposes a fixed and finite topology on the core graph, but a thin (software) layer on top of the cores allows the user to virtualise an arbitrary connectivity (application) graph on top of the physical one. Once this is done, the mapping of problem domain to processor mesh follows naturally.*

Problem domains for which POETS is useful are those in which the problem description can be represented as an (arbitrary, abstract) **application graph**. This topology is *independent* of the topology of the underlying hardware POETS engine.

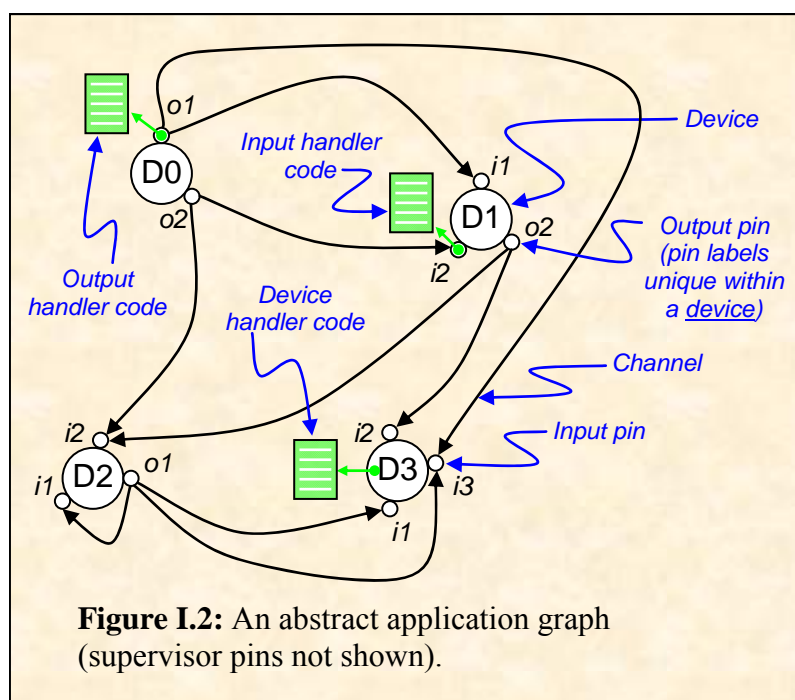
POETS leans heavily on graph theoretic concepts, and there are a lot of similar, but different concepts standing side-by-side here. A POETS terminology definition is supplied in Appendix A, intended to disambiguate the descriptive terms used throughout this document. By and large, specific terms appear in **bold** the first (few) times they appear throughout.

### 2.2 Mode of (abstract) operation

The **application graph** is a directed, tripartite graph: {**devices**, **channels**, **pins**} - see figure I.2. Associated with each abstract device is (1) some localised, persistent (partial) state vector, and (2) a small set of software **handler routines**. To a zeroth approximation, each device can be modelled as a small, abstract, state machine, which communicates via asynchronous **messages**. Access to the state machine is via one or more **receive handlers**, and communication from the state machine via one or more **send handlers**. Devices do not have independent threads of control; they *react* to incoming messages and send out consequent messages when the communications infrastructure allows.

When quiescent, they can execute an **OnIdle handler**. This can - in principle - block the device, but the design intention is that any OnIdle activity is brief, so that reaction to incoming messages is not delayed unduly. This is exactly analogous to the OnIdle handlers in the Windows message loop.

Associated with each device is a **supervisor** (each supervisor can - will usually - be connected to multiple devices). This is a process running on a conventional processor (with all the infrastructure that that provides), but it can be viewed as a special sort of device. Each device has at least one "supervisor input pin" (and associated handler) and at least one "supervisor output pin", but the actual connections between the device and supervisors are implicit, and



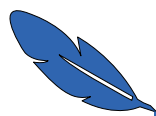
instantiated by the Orchestrator. The supervisor(s) provide a meta-graph sitting over the application graph. The supervisors are created by the Orchestrator, using partial definitions provided by the Application.

The supervisor pins - by necessity - also have a handler, nominally provided by the Application. *Default* supervisor functionality is provided by the Orchestrator, but as this (the Orchestrator) has no domain-specific knowledge, the default functionality is by necessity extremely limited. It is invoked principally during setup and data exfiltration. It provides an asynchronous mechanism whereby the overseeing (conventional) processes can inject (and force the extraction of) information to/from the application graph. They can also be used to supply real time (external application) excitation signals to devices.

Communication between devices is effected by the transmission of *packets* - small (~64 byte) quanta of data, carried along the application graph edges (**channels**) by the underlying hardware: computation is achieved by means of a "self-organising" packet storm. Leaving aside for the moment the issues of how the process starts/stops, the coarse sequence of operations is

- Initially, every device is quiescent. However, any device can indicate whether it currently wants to send packets from any of its output pins, so some devices may start-up ready and willing to send a packet.
- A packet arriving at a device is fielded by the appropriate receive handler; the handler executes, and in doing so may modify the device state and optionally change the set of output pins on which it would like to send.
- When there is spare (hardware) network capacity, a device which wishes to send a packet will be given the opportunity to send. When the send handler executes, the device can *further* update its state and either: (1) prepare a single packet to be sent, or (2) cancel a request to send.





- ▶ If a device needs to send multiple packets from a pin, it can leave the request to send on that pin enabled. Eventually spare network capacity will become available again and the send handler will execute again, allowing multiple packets to be sent as necessary.
- ▶ Packet *launch* may (temporarily) block if the underlying network or target device is congested. As with any asynchronous network, a situation can always be contrived whereby packets are injected into the communication fabric faster than they are drained from it, thereby overwhelming the physical network. An important design issue is what might be done about this state of affairs? Dropping packets at the physical point of local congestion is easy to do, but brings the inherent drawback that neither the sender nor the (intended) recipient can know there is a problem. Any attempt to detect this and transmit the occurrence data anywhere simply adds to the network congestion. In POETS, the problem is pushed back (by the hardware) to the point best qualified to react sensibly to it: the message injection point (sender). If a packet wants/needs (algorithmically) to be sent, and the physical network cannot support the transmission, the putative sending device is informed and still has control, and can decide what to do (abandon the attempt, delay the attempt, or delay and try to send a modified packet later on). If there is currently no local message capacity, then the send handler will not get called, so any packets remain implicitly encoded in the device state - whilst a device might *want* to send a packet, it is not *allowed* to prepare and inject that packet until network capacity is available and delivery can be guaranteed.
- ▶ Once launched, *unicast* packet delivery to devices in the application graph is *guaranteed*. However, the hardware does not allow the launch of multicast packets; if a packet needs to be delivered to multiple destinations, multiple copies must be sequentially launched (with the associated overhead), adding considerably to network congestion. If network congestion permits delivery of only some of these, the situation will obviously become confused; the *application* is expected to resolve this.
- ▶ Wallclock packet transit latency is non-deterministic and non-transitive.
- ▶ Any notion of simulated temporal fidelity must be carried explicitly by the packet payload. (Although wallclock time is available to the physical cores, this cannot possess the dynamic range of accuracy made possible by specifying it explicitly.)
- A packet arrives at a device and is delivered to the appropriate handler.
  - ▶ The receive handler executes, and in doing so may (1) modify the state of the receiving device and (2) change the set of output pins on which the device is currently requesting the opportunity to send a packet.
  - ▶ Devices cannot choose to block or delay the receipt of a message. However, only one handler (per device) will be active at any one time.
- When a send or receive handler terminates (the design intention is that the execution will be "brief"), the device returns to quiescence, awaiting the arrival of subsequent packets (or the opportunity to send more packets).
  - ▶ A device may send itself packets.



## System configuration

**DRAFT**

- Computation ends when (1) the packet storm ends, or (2) some higher-order injected command causes termination.
  - ▶ The solution may be encoded within the set of final (modified) device states distributed over the application graph, or may be output dynamically during execution as devices send messages to supervisor nodes.
  - ▶ Solution exfiltration (output) from the application graph is carried out by cooperation between the devices and the optionally application defined epilogue code within the Supervisors. The Supervisors assemble the distributed solution into some coherent form and hand it out to the Application.
- Application input is provided statically through the topology and configuration of the graph instance, and can also be injected dynamically in real time via the supervisor pins during execution.

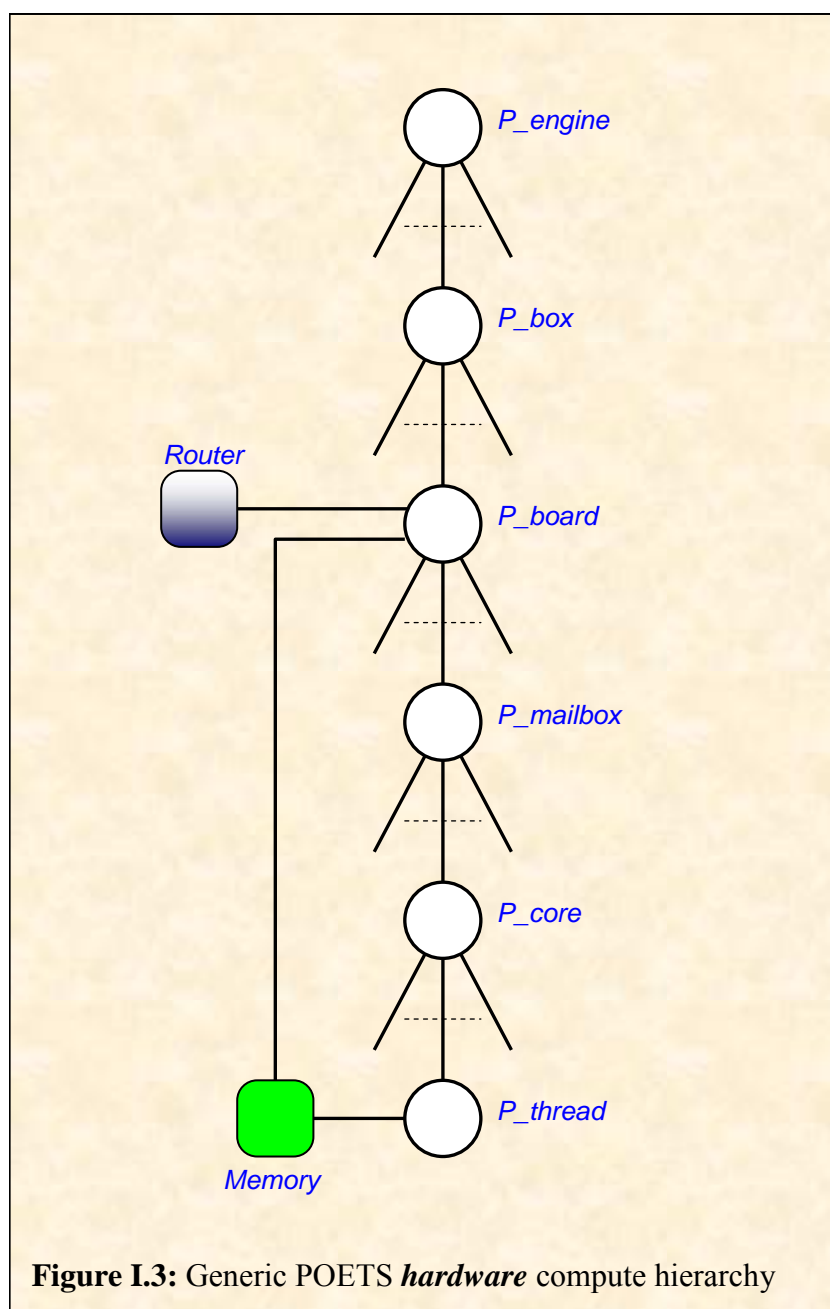
The design intent is that handlers only achieve computations through the sending and receiving of packets. Devices do not have an independent thread of control, and only have the ability to compute or send packets when explicitly given access to network and compute resources - they *react* to packets arriving or the opportunity to send packets. There is no deterministic dialogue (although a deterministic conversation can always be forcibly overlaid within the constraints outlined here) because the packet choreography is non-deterministic.

In principle, there are no constraints on the structure of the application graph: it may contain an arbitrary number of devices, interconnected by an arbitrary number of channels. (In practise there are hard limits to a few aspects of the graph - outlined later - but these are intended to be sufficiently remote that they will not impinge on the abstract principles of operation.)

The concept of "design intent" is used several times in this document. POETS is intended to be used in a certain way, with the components having certain (relative) properties. It is possible - easy - to take POETS out of the region of design intent, and the system behaviour will degrade significantly as a consequence. Currently, there is no intrinsic defence against this - the domain expert user must be aware of the limitations of the system to get any kind of performance from it.

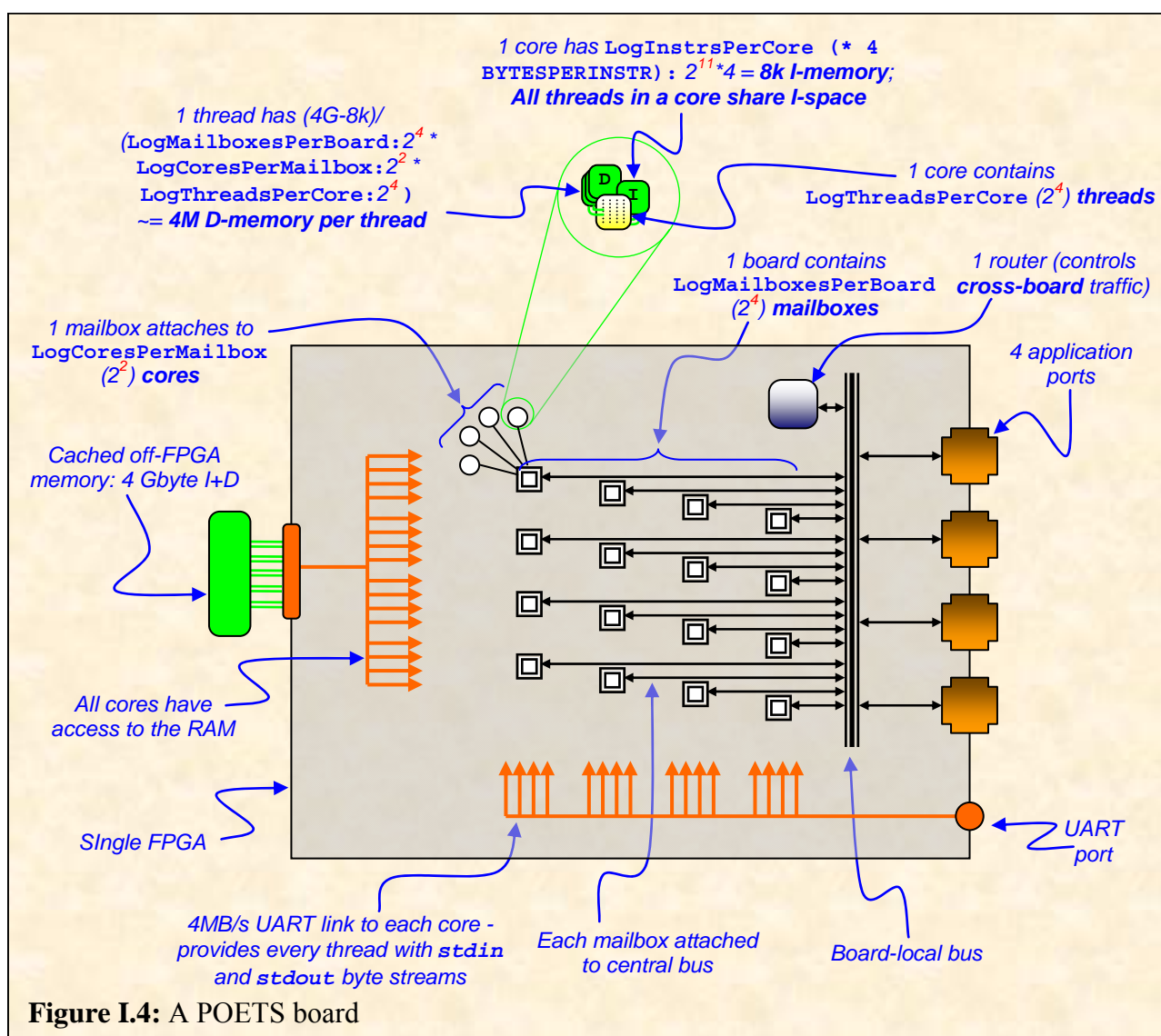
## 2.3 A hardware model

The physical hardware created by UoC takes the form of a (fixed) graph of FPGAs (**P\_nodes** - sometimes **P\_boards**), each P\_node containing a set of physical cores (**P\_cores**). Each P\_core is heavily hyperthreaded and executes some number of **P\_threads**. **There is no relationship between a P\_thread and a POSIX pthread - just no-one could think of any other rational name.** It is the P\_threads that execute the handler code of the devices. The Orchestrator (which incidentally uses POSIX pthreads itself) *constructs* the P\_thread binaries and maps the devices to the P\_threads. In order to accommodate application graphs with large device counts, the Orchestrator must map *tranches* of devices to each P\_thread (nominally around 1000). Within a tranche, the behaviour of the application devices is serialised, and any parallelisation benefits are lost. The current hardware also serialises most inter-core communication to a bus; it does not support broadcast (packet multicast), and everything must be serialised further and sent point-to-point.



**Figure I.3:** Generic POETS *hardware* compute hierarchy

POETS is research-in-progress; the hardware is realised on an FPGA farm and hence is subject to change. Broadly, the hardware model consists of a *compute hierarchy*: a **POETS engine** consists of a some number of **boxes**, each of which consists of some number of **boards**, each of which contains a **router** and some number of **mailboxes**; connected to each mailbox is some number of **cores**, which are all heavily multi-threaded. The hardware compute hierarchy is outlined in figure I.3, although the interconnection is more subtle than the figure suggests. Although each box contains a number of boards, the boards themselves are interconnected as a graph that bypasses the box level. Figure I.4 outlines one internal board structure, and figure I.5 illustrates how the boards are interconnected to form a network that is largely independent of the boxes hosting the boards.



The design intent is that the topology of the hardware network is arbitrary and is discovered dynamically; initial versions of the system rely upon a **hardware description file** to convey to the Orchestrator the structure of the hardware.

The graph of boards is referred to internally as the **machine topology graph**.

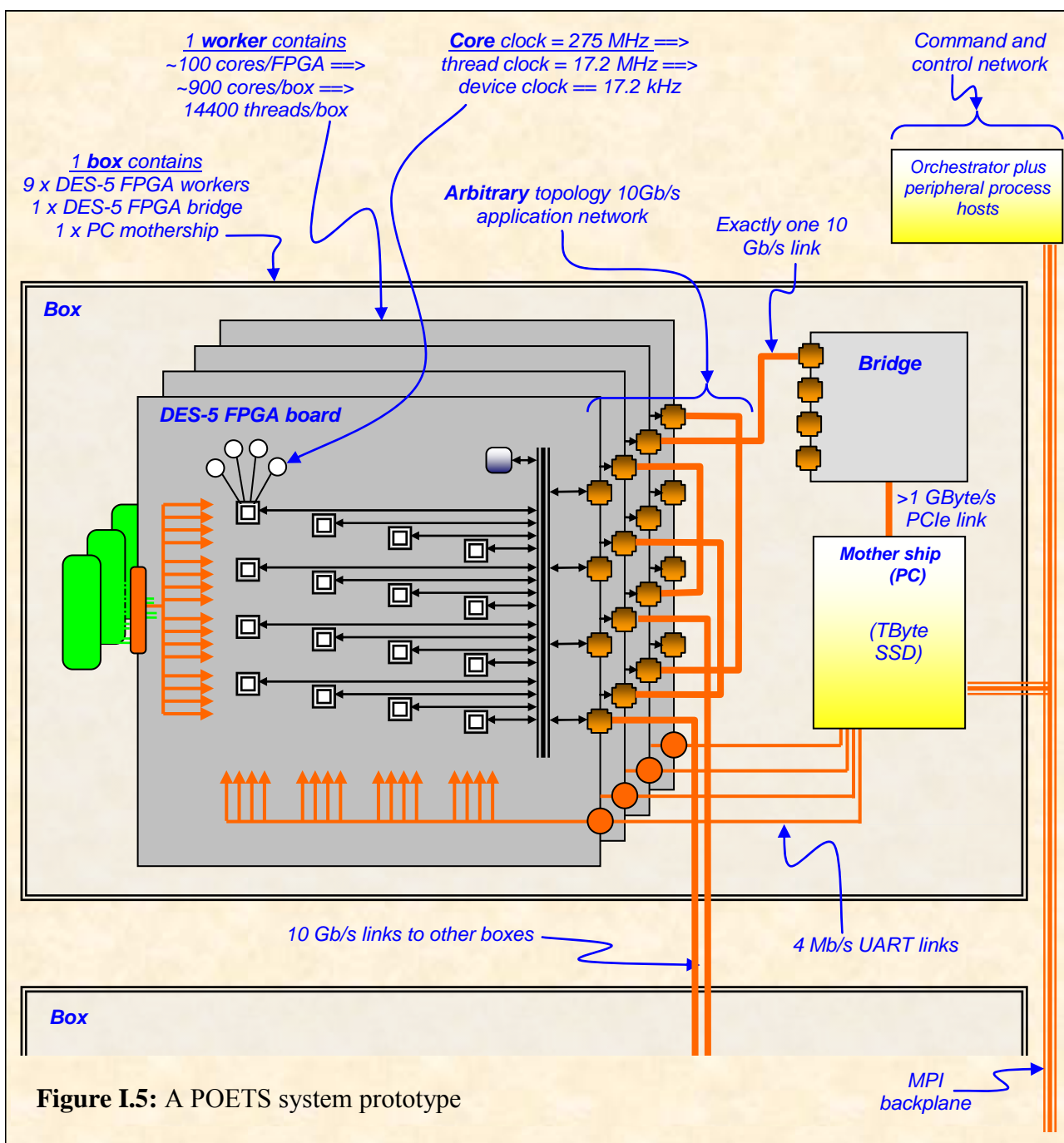


Figure I.5: A POETS system prototype



### 2.4 The (software) Orchestrator

The Orchestrator is a complex, heterogeneous parallel process software system, running under MPI, that controls the hardware. It consists of an MPI network, broadly outlined in figure I.6.

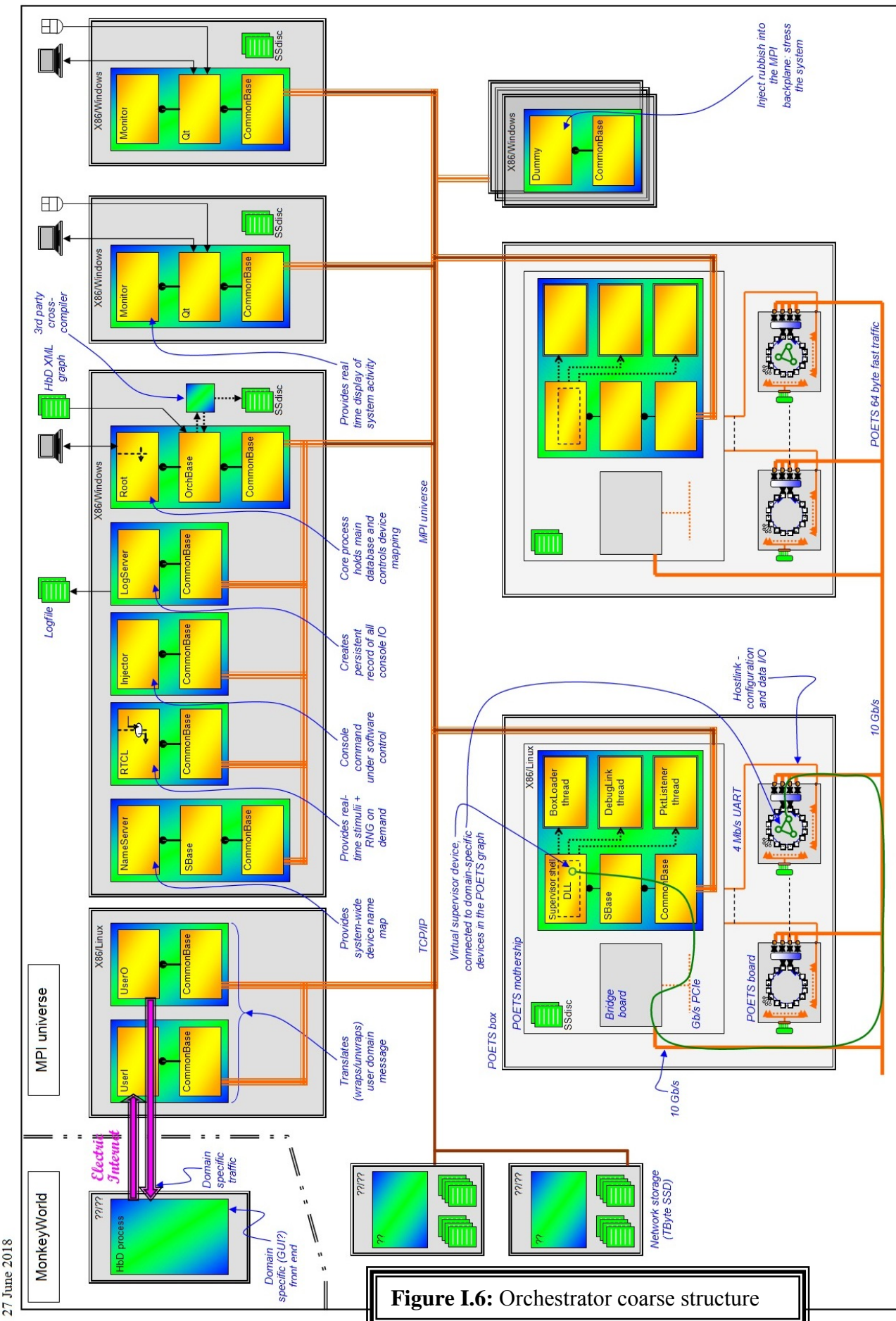
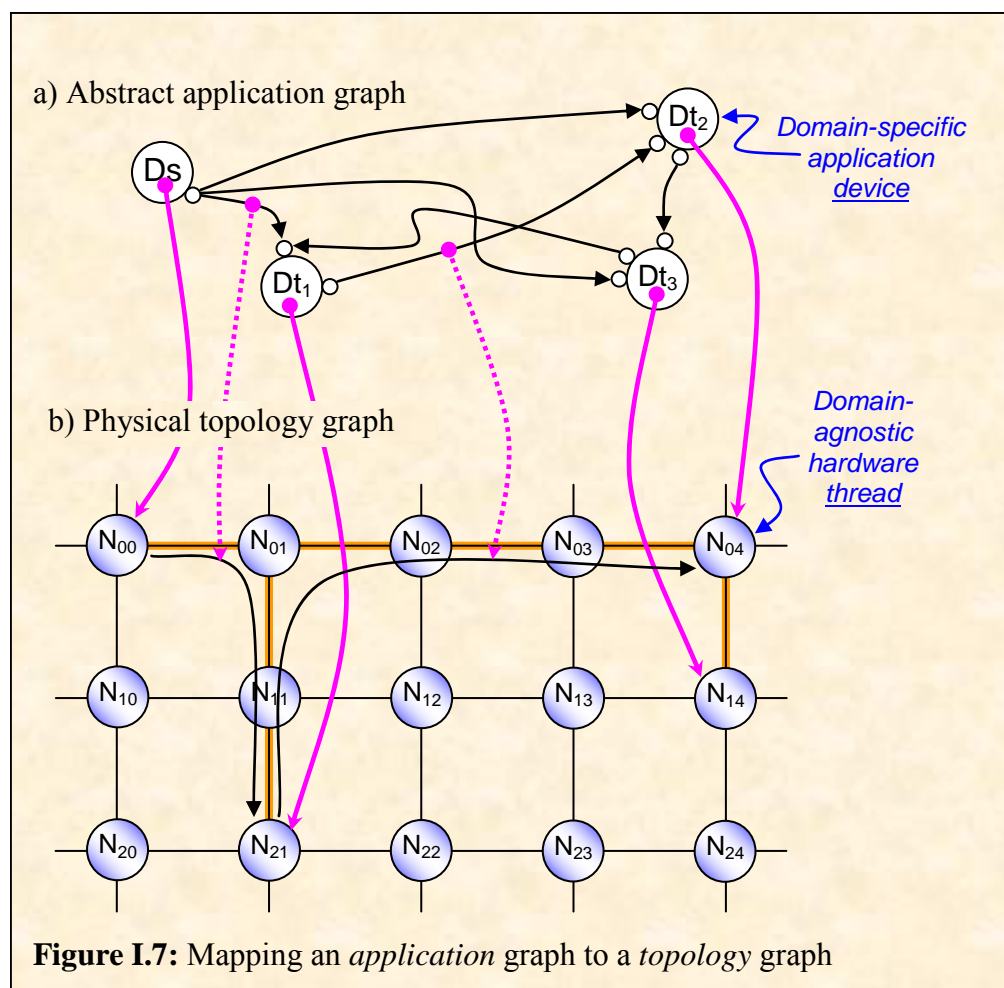


Figure I.6: Orchestrator coarse structure



The Orchestrator MPI universe is heterogeneous and hybrid (some of the processes are multithreaded) and is configured such that the MPI processes themselves communicate asynchronously with each other. This is outside the design intent of the MPI standard (although not in conflict with it) and brings with it an interesting set of implementation challenges.

One of the tasks of the Orchestrator is to map the (domain-specific) application graph to the hardware topology graph (figure I.7).



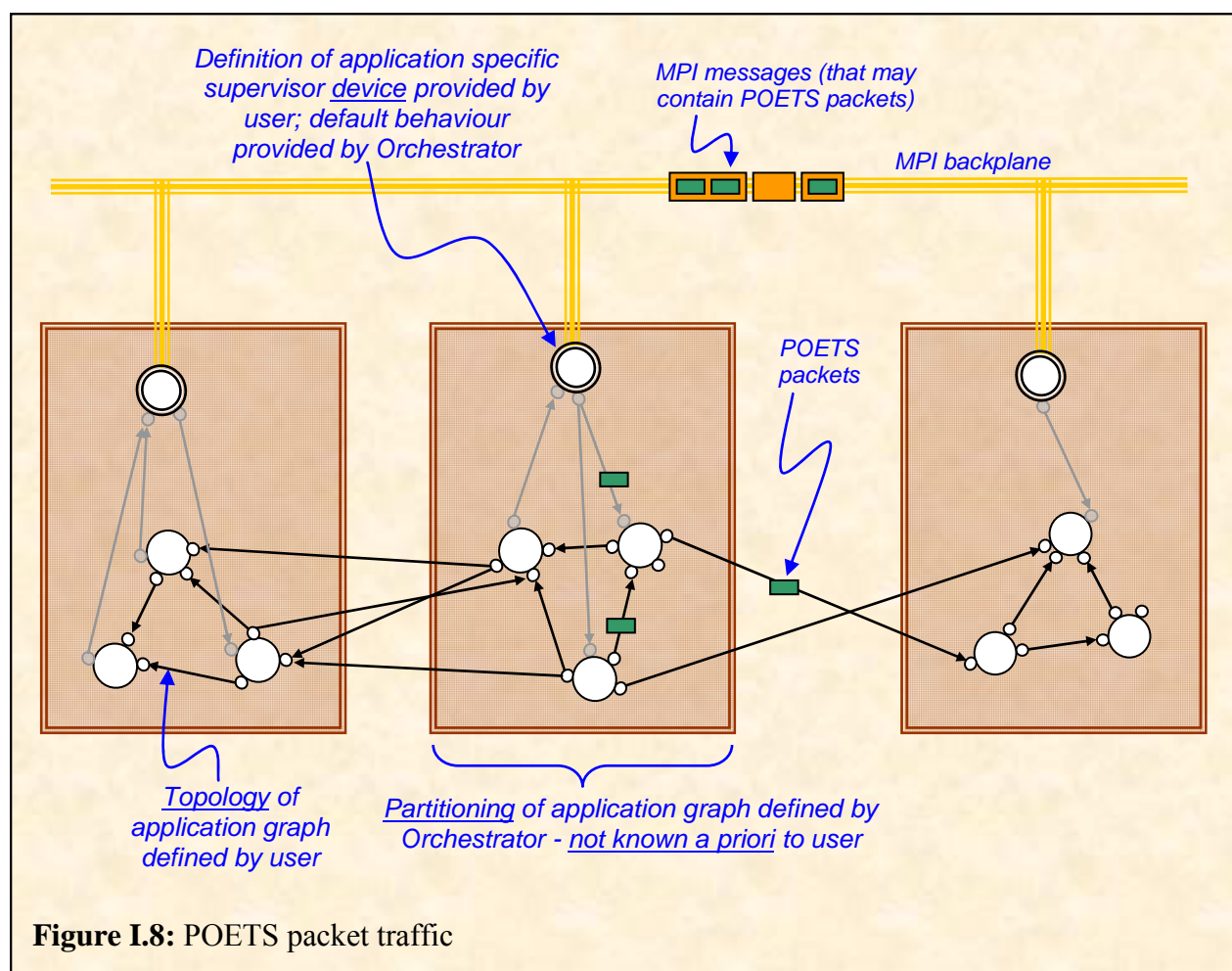
This allows the functionality of the application graph to be realised independently of and transparently to the underlying hardware compute fabric - the application graph is *virtualised*.

For many applications, the topology of the core graph and the device:P\_thread map have little or no effect on the outcome of any application computation; the total wallclock solution time may change (in the limit, dramatically), but any numerical solution will be invariant to the changes in the above. It is responsibility of the handler internals to ensure that the solution is invariant with the non-determinism and time-shear of the incoming packet flux (and that the solution exists at all). It follows that the handler code is (usually) non-trivial and (usually) machine generated.

It is also possible that the application will embrace large-scale non-determinism, tolerating randomness introduced by the underlying hardware. It is up to the application designer to ensure correct behaviour, for example using asynchronous algorithms for numerical solutions

which can tolerate time-shear. There may also be weak non-determinism even in synchronous systems: for example, a device might sum floating-point values from different devices, which will produce different results depending on the order in which the values arrive.

During the mapping process, the application device graph is distributed over the hardware thread graph, controlled by the Orchestrator, and (the intention is that) the behaviour of the application graph (modulo the comments about time-shear above) is unaffected by this mapping. However, in order to facilitate command/control/data input/exfiltration, the user may also define the behaviour of a **supervisor** device. The Orchestrator will incorporate the *functionality* of the supervisor into each box/board (see figure I.6). The supervisor functionality is defined by the user, but the mapping of devices to threads is not, in general, visible to the user, so the supervisor behaviour has to be defined without knowledge of the partition data - figure I.8.




**Figure I.8:** POETS packet traffic

In summary, then, the domain-specific devices (hosted on hardware threads) communicated amongst themselves via packets sent over the 10Gb/s POETS hardware links (figure I.5). Occasionally, (from the perspective of POETS compute traffic) it is necessary for this network to communicate with the outside world. This it does by sending packets to (what it sees as) *supervisor devices*, which are actually code fragments residing in a conventional process hosted on a conventional architecture **mothership**. (Supervisor shell/DLL in figure I.6). These mothership processes are interconnected via a MPI universe, and communicate with other processes in the universe via MPI messages (which may contain packets).


# Orchestrator Internals

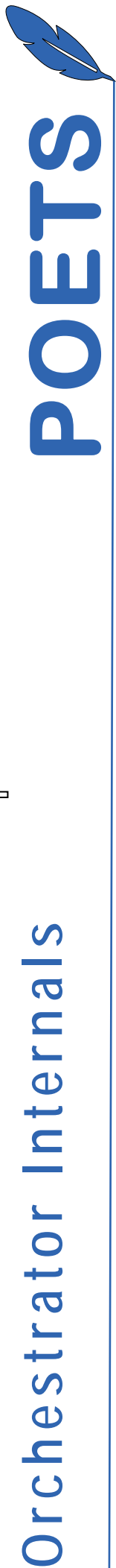
---

## POETS



# Orchestrator Internals

- # Orchestrator Internals
- 
- ## POETS
- 



# Orchestrator Internals

---

## POETS

