Gannet: a Service-based Architecture for Reconfigurable SoCs

Wim Vanderbauwhede
Department of Computing Science
University of Glasgow

- Why a new SoC architecture?
- Task control in NoC-based SoCs
- The Gannet system
- The Gannet language
- An operational semantics for Gannet
- Separation of data flow and control flow

Why a new SoC Architecture?

- tomorrow's SoC's will be **very big** (10^{10}logic gates)
  - traditional bus-style interconnect causes a bottleneck:
    - Synchronisation over large distances is impossible
    - Fixed point-to-point result in huge wire overhead
  - **on-chip networks** provide a solution
    - globally asynchronous/locally synchronous
    - flexible connectivity
- design reuse is essential => IP ("Intellectual Property") cores
- IP cores are highly complex, self-contained units
- treating such blocks as **services** is a logical abstraction

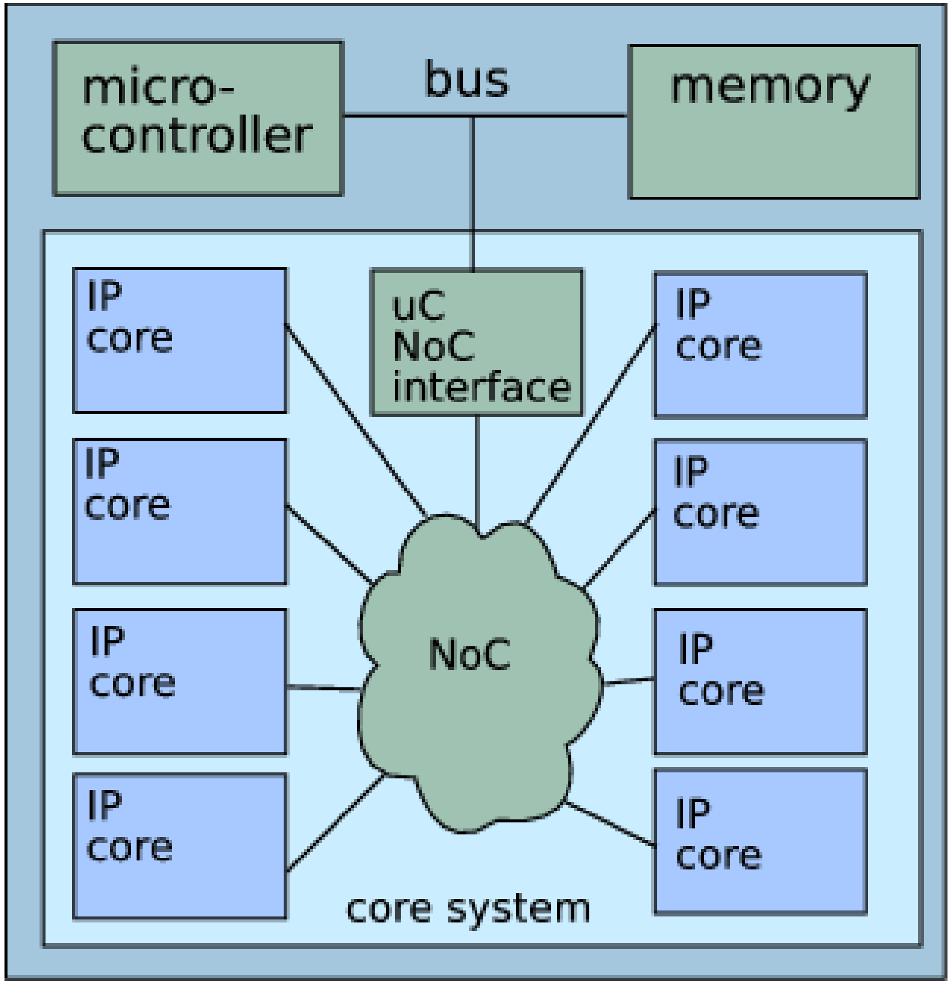
Overview

- We assume a generic SoC where
  - data is processed by IP cores interacting through a NoC
  - control structures are implemented on a microcontroller.
- We propose a service-based SoC architecture (the **Gannet** architecture) where
  - the control services are implemented using a Virtual Machine
  - IP cores acquire service behaviour through a generic data marshalling interface.

- Why a new SoC architecture?
- Task control in NoC-based SoCs
- Gannet system
- Gannet language
- An operational semantics for Gannet
- Separation of data flow and control flow

Task control in NoC-based SoCs

- NoC-based SoC with embedded microprocessor:

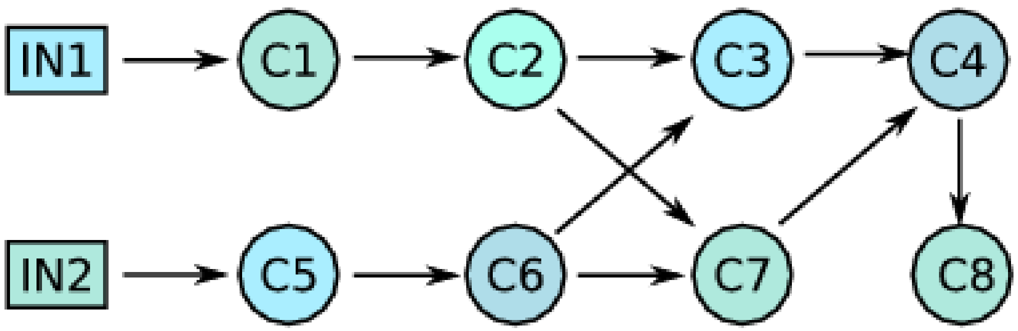


Task control in NoC-based SoCs

- SoCs in general use an embedded microprocessor for control.
- Conventional way of controlling hardware blocks using an embedded microprocessor: memory-mapped IO+ interrupts.
- In a NoC-based SoC, the microprocessor interacts with a NoC transceiver and transfers data as NoC packets $\Rightarrow$
  - efficient data transmission;
  - considerably reduction of required number of interrupts;
  - no significant operational difference with bus-based mechanism.

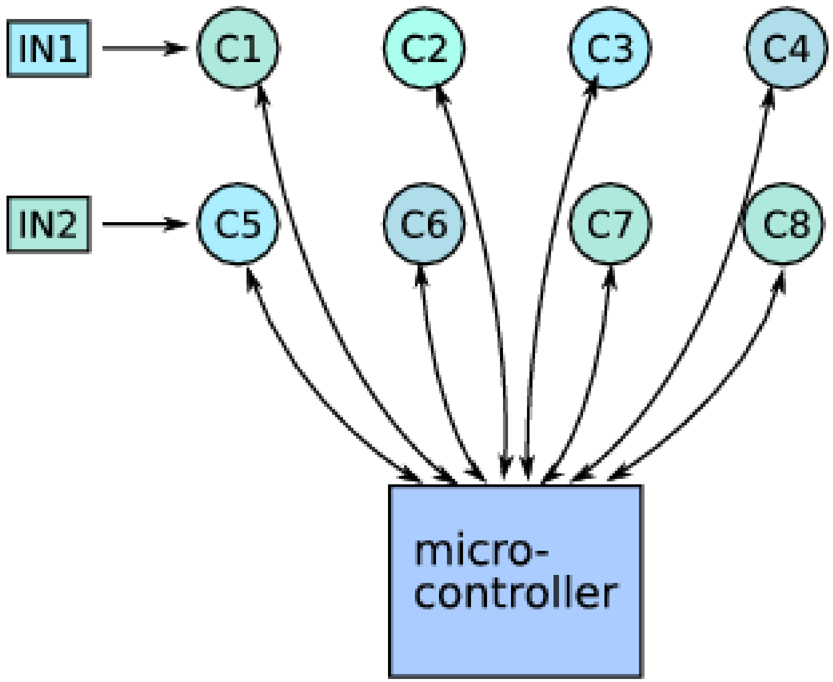
Task control in NoC-based SoC

- Non-task-level reconfigurable system:
  - microcontroller only sends control or configuration information to each core;
  - all data can flow between the cores.



Task control in NoC-based SoC

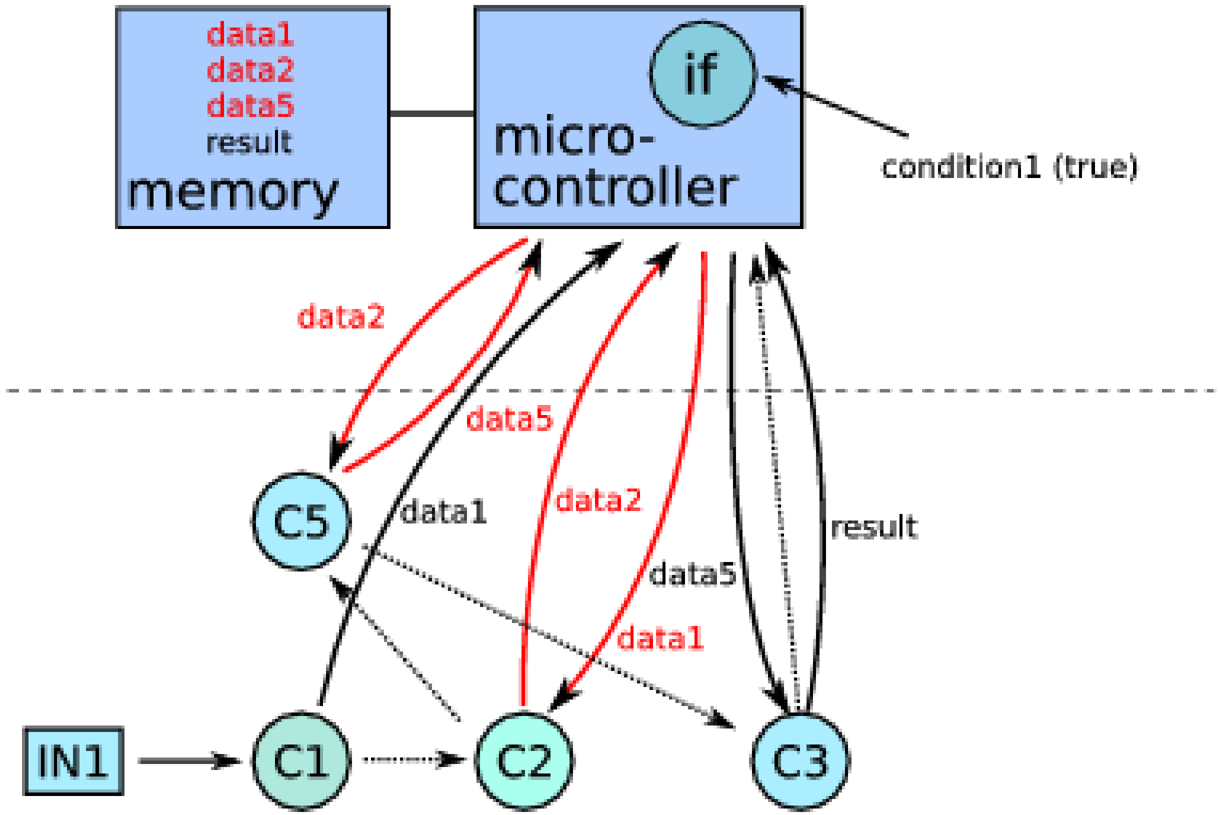
- Task-level reconfigurable system:
  - data paths are determined at run time by a program running on microcontroller;
  - all data pass via the microcontroller.



Example

```
// variadic function prototype
Data* NoC_TRX(CoreAddres&,...);
/* variable declarations omitted */
data1=NoC_TRX(C1,IN1);
if (condition1) {
    Data* data2=NoC_TRX(C2,data1);
    data5=NoC_TRX(C5,data2);
} else {
    Data* data2=NoC_TRX(C5,data1);
    data5=NoC_TRX(C2,data2);
}
result=NoC_TRX(C3,data5);
```

Example



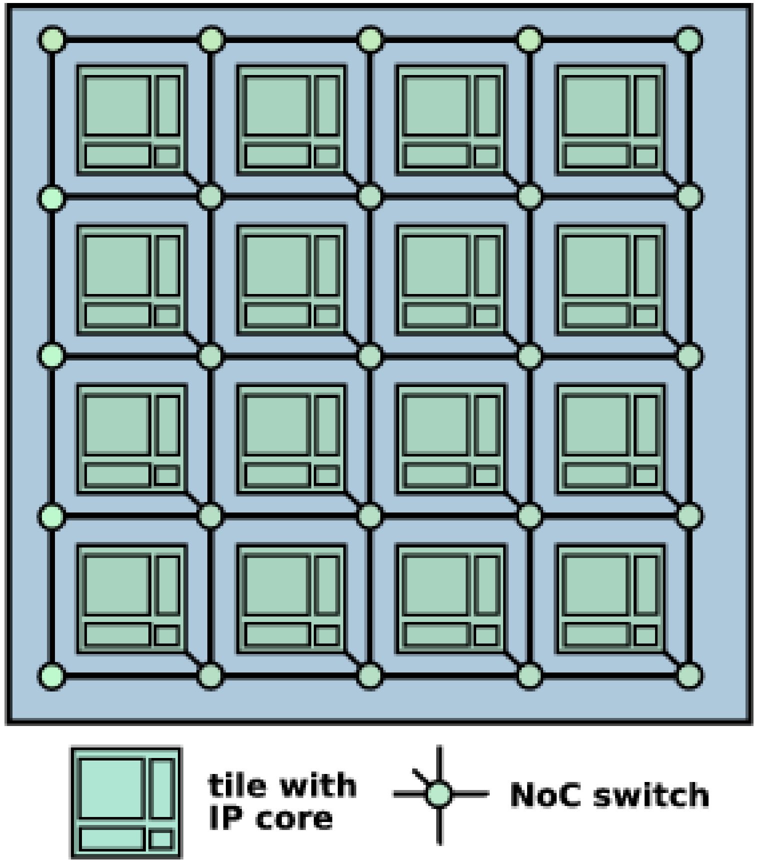
data1
data2
data5
result
memory
if
micro-
controller
condition1 (true)
data2
data5
data1
data2
data5
data1
result
C5
IN1
C1
C2
C3

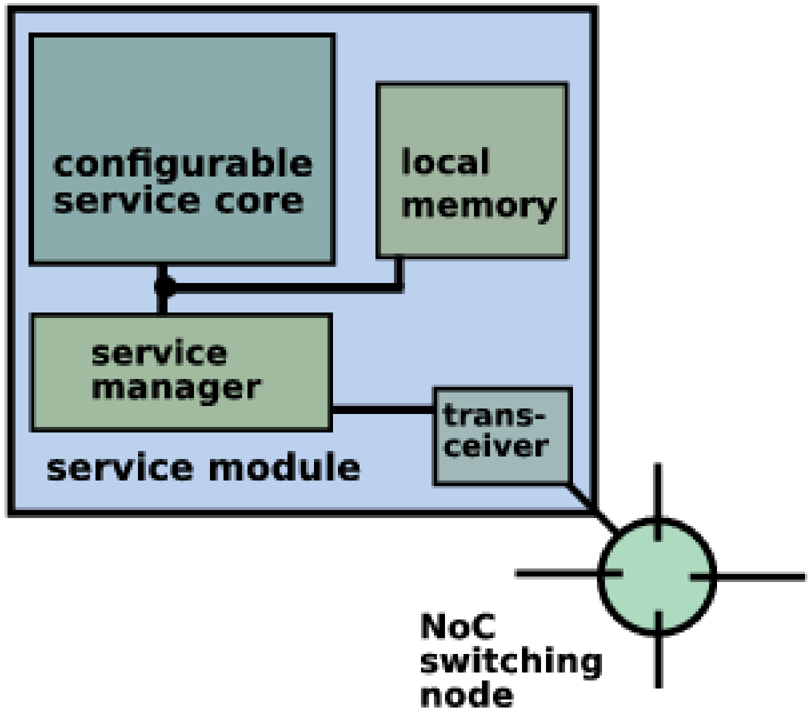
- Why a new SoC architecture?
- Task control in NoC-based SoCs
- Gannet system
- Gannet language
- An operational semantics for Gannet
- Separation of data flow and control flow

Gannet architecture

- A **service-based** architecture for **very large SoCs:**
  - a collection of processing cores (HW/SW).
  - each core offers a a specific **service.**
  - tasks are defined by the interaction pattern of the services.
- Task-level reconfigurability
  - task description programs, configurable at run time
- High abstraction-level design
  - single program governs behaviour of complete system

Gannet architecture





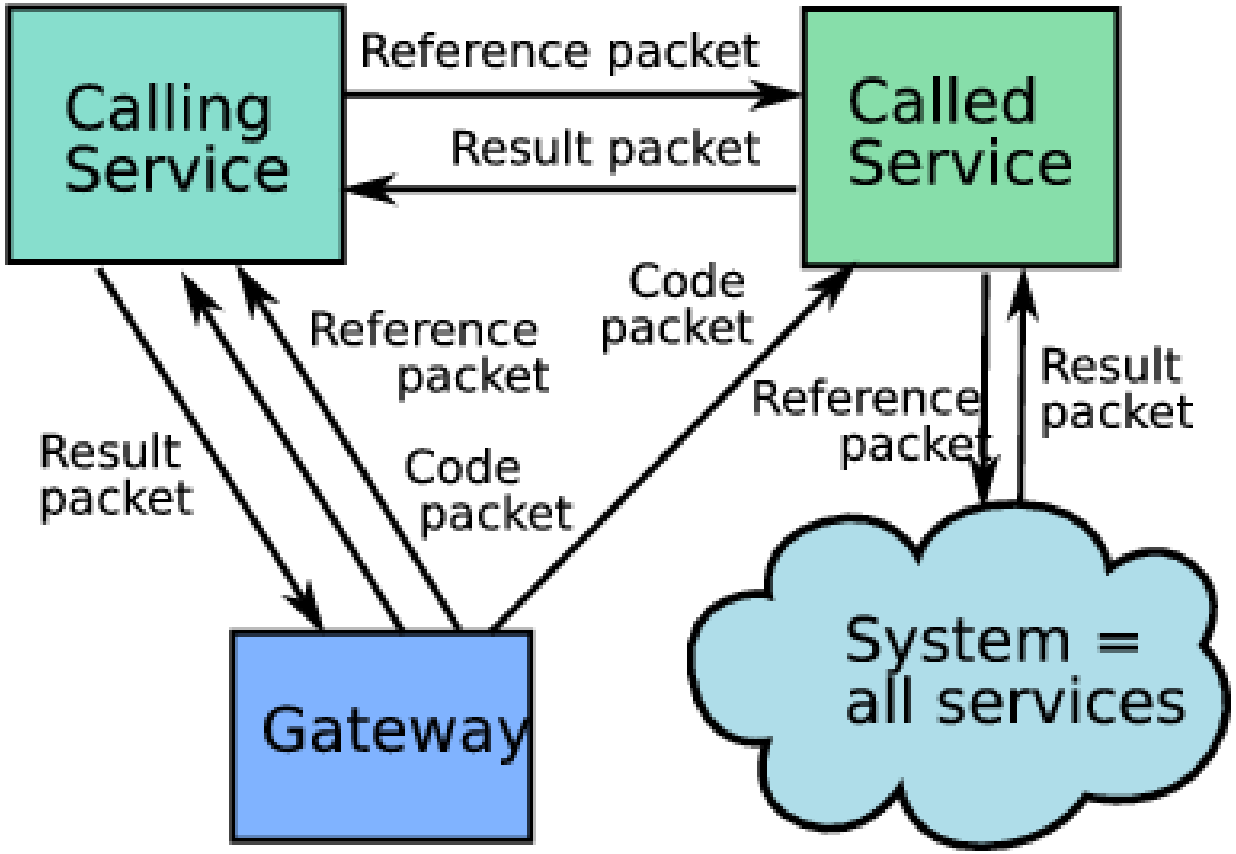
Gannet system operation

- The Gannet machine is a distributed computing system where every computational node **consumes packets** and **produces packets** and can store state information between transactions.
- We denote a Gannet packet as $p(Type, To, Ret, Id; Payload)$
- The semantics of a Gannet **service** (computational node) can be described in terms of
  - the **task code**
  - the internal state
  - the **result packet(s)** produced by the task

Gannet system operation

1. Service S_i receives a **code** packet $p(Code, S_i, S_j, R_{task}; task)$ where $task = (S_i\ a_1 ... a_n)$. The task is stored with reference R_{task}. Service S_i is in $state_i$.
2. Service S_i receives a task **reference** packet $p(Ref, S_i, S_j, R_{id}; R_{task})$
3. The service activates the task referenced by R_{task}: $(S_i\ a_1 ... a_n)$. This results in evaluation of the arguments $a_1..a_n$:
4. The service produces a result packet $p(Type_i, S_j, S_i, R_{id}; Result_i)$ and the state changes to $state_i{}'$.
5. This packet is sent to S_j where $Result_i$ is stored in a location referenced by R_{id}.

Gannet system operation



Control services in Gannet

- Any run-time reconfigurable system requires **control constructs** to be effective.
- In Gannet, these constructs (if/then, functions, blocks, variables, ...) are provided by **services**.
- Such **control services** can be efficiently implemented on an embedded microcontroller.
- Interleaving the services provided by the HW cores with control services can cause bottleneck due to memory bandwidth.

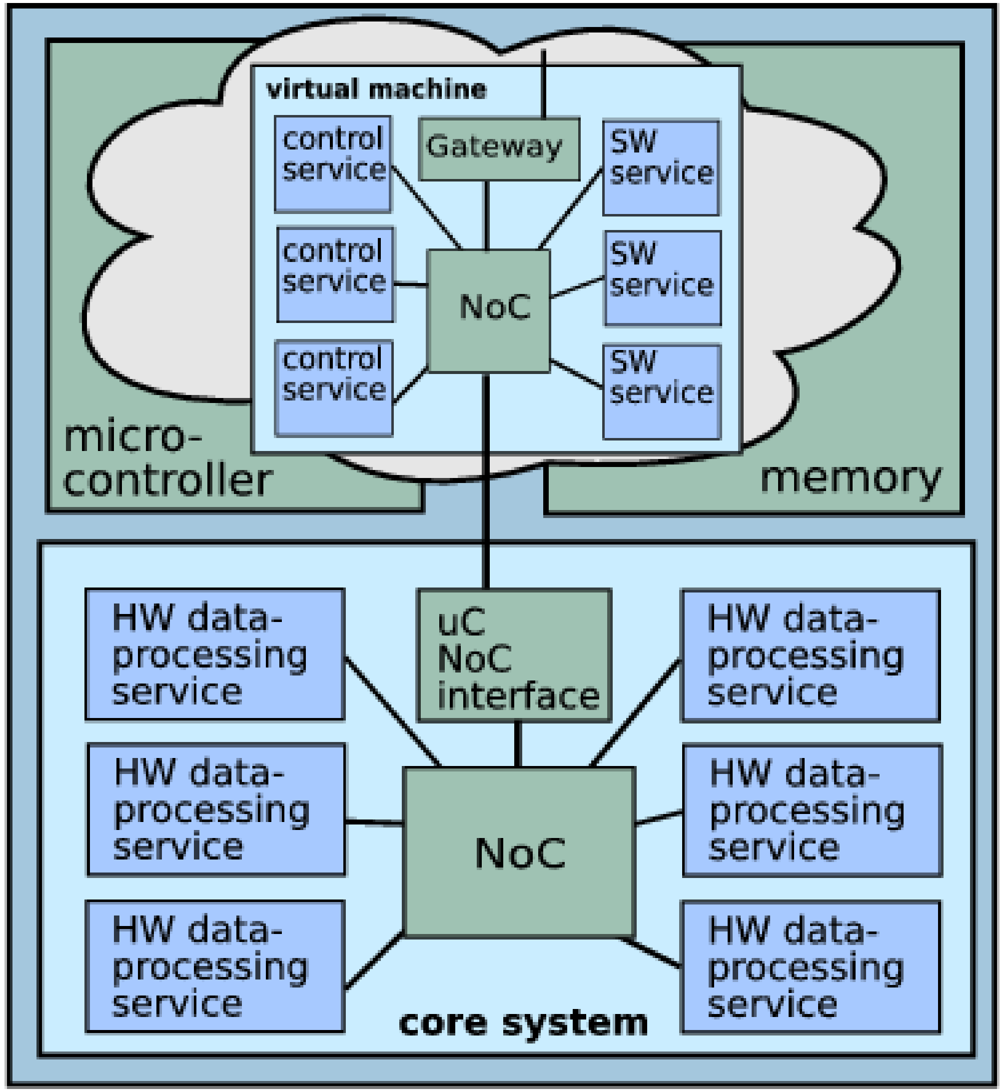
Control services in Gannet

- Ideally, the microcontroller would only exchange control information with the cores.
  - technically not impossible to realise using compiled code but would require
    - a language with functional characteristics (no side-effects, undetermined execution order, laziness, concurrency)
    - access to absolute memory addresses of the data structures
  - program would need to contain a JIT compiler to create bytecode for the service managers at run time.

The Gannet Virtual Machine

- A Virtual Machine (VM) which interacts with the hardware service managers:
  - software implementation of the service managers, control service cores and a ’virtual NoC’
  - small, portable C++ application
  - runs byte-compiled programs in the Gannet language
  - same bytecode used by VM and HW

The Gannet system



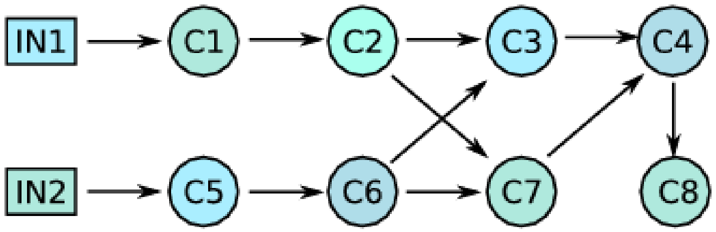
- Why a new SoC architecture?
- Task control in NoC-based SoCs
- Gannet system
- Gannet language
- An operational semantics for Gannet
- Separation of data flow and control flow

Gannet language

- The “assembly” language to program the Gannet system
- Intended as compilation target, not HLL
- A functional language, every service is mapped to an opaque function
- Gannet is a distributed machine for running this language
- Service = service manager + service core
- Service cores can be implemented in HW or SW

Gannet language syntax

- Previous example in S-expressions syntax:




```
(S8 (S4
        (S3
            (S2 (S1 IN1))
            (S6 (S5 IN2))
        )
        (S7
            (S6 (S5 IN2))
            (S2 (S1 IN1))
        )))
```

Gannet language syntax

- Example with control services (factorial):

```
(let
     (assign 'fact
          (lambda 'n 'a 'f
            '(if (< n '2)
               'n
               '(apply f (- n '1) (* a n) 'f)
           )))
     (apply fact '4 '1 'fact)
)
```

Gannet language properties

- Some key properties of the Gannet language:
  - the evaluation order is unspecified
  - eager by default but lazy evaluation is possible
  - no side effects across services
  - updates of variables are atomic (no race conditions)
- These properties
  - make the language fully concurrent (maximise parallelism)
  - and enable separation of control flow from data flow

Gannet language properties

- **Unspecified execution order:**
  - In a given function call it is not possible to predict the evaluation order of the arguments.
  - In practice, all arguments are evaluated in parallel; call blocks until all arguments are ready.

```
(let
    (assign 'a (S1 ...))
    (assign 'b (S2 ...))
    (S3 ... b ...
        (S4 ... a ...)... )
)
```

Gannet language properties

- **Lazy evaluation:**
  - By default, Gannet is **eager**, i.e. it always evaluates all arguments before passing them on to the service core.
  - It should be possible to evaluate arguments at need (“**lazy**”).
  - Laziness is expressed by prefixing an expression or symbol with a single quote:

    ```
    (assign 'a (S1 ...))
    ```

  - Quoting causes the evaluation of the symbol `a` to **deferred** to the service core.

Language properties

- **No side effects across services:**
  - A call to a given service should not result in a modification of the state of the rest of the system.
- **Updates of variables are atomic:**
  - No race conditions if several services simultaneously try to modify shared data.
  - The service manager processes all task requests in FIFO order.
  - Not possible to update an unassigned variable.

- Why a new SoC architecture?
- Task control in NoC-based SoCs
- Gannet system
- Gannet language
- An operational semantics for Gannet
- Separation of data flow and control flow

Notation

- $\bullet$: separates a packet from the other packets in the queue: $(p \bullet ps)$: packet at the head, $(ps \bullet p)$: packet at the tail.
- $*$ ("don't care"): the value of a field is not relevant.
- ... : presence some non-specified entities (esp. in a store).
- _ : allocated space in a store

Packet-level Semantics: Definitions

Definitions

- The Gannet system: N service nodes $S_i(...)$, $i \in 1..N$ and a gateway node $G(...)$.
- The unit of data transfer is the packet:

  $p = packet(Type, To, Ret, Id; Payload)$.

  $Payload$: **data** or an **expression (depending on** $Type$**)**

  Id : a Gannet symbol used as identifier for the payload

  To, Ret : addresses of destination and return services

Packet-level Semantics: Definitions

- Packet receive and transmit FIFO queues: q_{RX} and q_{TX}

  RX queue: 3 queues muxed by packet $Type$:

  $q_{RX}(ps \bullet p)$ ::= $q_{RX}(\ldots pt(ps \bullet p)\ldots)$, $pt \in \{tasks \mid code, refs, data, refs\}$
- Service node S_i also comprises:
  - The data store:$store_d(\ldots(Id\ data)\ldots)$
  - The task packet ($code$) store: $store_c(\ldots(Id\ p)\ldots)$

    p is the stored packet
  - The processing core $core(\ldots)$

Packet-level Semantics: Definitions

Thus an explicit notation for a service node S_i is:

$$S_i(q_{RX}(data(\ldots), code(\ldots), refs(\ldots)),$$
$$q_{TX}(\ldots), store_d(\ldots), store_c(\ldots), core(\ldots))$$

- All service nodes S_i operate **concurrently**
- The behaviour of a service S_i consists of a set of **actions**
- There is no central clock, all data transfers are asynchronous
- All actions are packet-driven

Packet transfer between services

The set of actions to transfer packets consists of:

- Transmit a packet (S_i): TX
- Receive a packet (S_j): RX

The semantics are straightforward:

$$
\begin{aligned}
p &= packet(*, i, j, *; *) \\
S_i(q_{TX}(p \bullet ps)) &\xrightarrow{TX} S_i(q_{TX}(ps)) \\
S_j(q_{RX}(qs)) &\xrightarrow{RX} S_j(q_{RX}(qs \bullet p))
\end{aligned}
$$

From now on, TX and RX are implicitly assumed.

The Gannet service manager

- On **receipt** of a packet, a service performs a set of **actions** which can result in
  - packets being **transmitted**
  - a change in the state of the store
- A **subset** of actions is performed by the **service manager**
  - The service manager is a **data marshalling** unit and **interface** between the service core and the system.
  - The service manager is **generic**: functionality is independent of the functionality of the service core.

Store code packet: SC

On receipt of a task, the gateway distributes code packets to the corresponding service nodes.

$$p_{c,ij} = packet(code, i, G, r_{ij}; e_{ij});\ e_{ij} = \langle s_{ij} \ldots \rangle$$

$$S_i(q_{RX}(code(p_{c,ij} \bullet ps_c)), store_{tp}(\ldots))$$

$$\xrightarrow{SC} S_i(q_{RX}(code(ps_c)), store_{tp}(\ldots(r_{ij}\, p_{c,ij})\ldots))$$

Service manager actions

Activate a task from a reference packet: AT

$$p_{r,i} = packet(reference, i, j, r_i'; r_i)$$

$$p_{c,i} = packet(code, i, *, r_i; se_i); se_i = \langle s_i \ldots r_j \ldots \rangle$$

$$S_i(q_{RX}(tasks(qs), refs(p_{r,i} \bullet ps)), store_{tp}(\ldots(r_i\, p_{c,i})\ldots))$$

$$\xrightarrow{AT} S_i(q_{RX}(tasks(qs \bullet p_i), refs(ps)), store_{tp}(\ldots))$$

$$p_i = packet(task, i, j, r_i'; se_i)$$

Marshalling action set M

Delegate reference: DR

$$
\begin{aligned}
& p_i = packet(task, i, *, *; se_i);\; se_i = \langle s_i ... r_j ... \rangle \\
& S_i(q_{RX}(tasks(p_i \bullet ps)), q_{TX}(qs), store(...)) \\
\xrightarrow{DR}\; & S_i(q_{RX}(tasks(ps)), q_{TX}(qs \bullet p_{r,j}), store(...(r_j\_)...)) \\
& p_{r,j} = packet(reference, j, i, r_j; r_j)
\end{aligned}
$$

Marshalling action set M

Store returned result: SR

$$p_i = packet(data, i, *, s_j; w_j)$$

$$S_i(q_{RX}(data(p_i \bullet ps)), store(\ldots(s_j \_)\ldots))$$

$$\xrightarrow{SR} S_i(q_{RX}(data(ps)), store(\ldots(s_j w_j)\ldots))$$

Marshalling action set M

Store quoted symbol: SQ

$$p_i = packet(task, i, *, *; se_i); se_i = \langle s_i ... qr_j ... \rangle$$

$$S_i(q_{RX}(tasks(p_i \bullet ps)), store(...))$$

$$\xrightarrow{}^{SQ} S_i(q_{RX}(tasks(ps)), store(...(qr_j r_j)...))$$

Marshalling action set M

The action of the complete M set $\{SC, AT, DR, SR, SQ\}$ can be abstracted as:

$$
\begin{aligned}
& p_i = packet(task, i, *, *; se_i);\ se_i = \langle s_i\ a_1 \ldots a_n \rangle \\
& a_i ::= qr_i \mid r_i \\
& S_i(q_{RX}(p_i \bullet ps), q_{TX}(qs), store(\ldots)) \\
\xrightarrow{}^{M}\ & S_i(q_{RX}(ps), q_{TX}(qs), store(\ldots(a_1\, wr_1)\ldots(a_n\, wr_n)\ldots)) \\
& wr_i ::= w_i \mid r_i
\end{aligned}
$$

Processing action set P

- The actions of the **service core** determine the **functionality** of the service.
- This functionality can be defined as the type, destination and payload content of the packet produced and the state change of the store based on the values marshalled by the service manager.
- For data-processing services, the service core implements a function cs_i which takes n arguments with values $w_1 ... w_n$ and produces a result w.

Processing action set P

The P set consists of the actions $\{call, eval, return\}$:

$$
\begin{aligned}
& S_i(q_{TX}(ps), store((s_1\, w_1)\ldots(s_n\, w_n)\ state), core()) \\
\longrightarrow^{call}\ & S_i(q_{TX}q(ps), store(state), core(\,(cs_i\, w_1\ldots w_n)\,)) \\
\longrightarrow^{eval}\ & S_i(q_{TX}(ps), store(state'), core(\,w\,)) \\
\longrightarrow^{return}\ & S_i(q_{TX}(ps \bullet p), store(state'), core()) \\
& p = packet(*,*,i,*;w)
\end{aligned}
$$

Processing action set P

The P set actions can be abstracted as:

$$S_i(q_{RX}(qs), q_{TX}(ps), store((s_1\,w_1)\ldots(s_n\,w_n)\ state))$$
$$\xrightarrow{P}\ S_i(q_{RX}(qs), q_{TX}(ps \bullet p), store(state'))$$
$$p = packet(*, *, i, *; w);\ (cs_i\,w_1 \ldots w_n) \rightarrow w$$

Service semantics

- All actions can be abstracted to the M and P sets
- The semantics of a service can be described completely in terms of
  - the task and results packets
  - the state of the store

Non-language service semantics

- **Non-language services** are services of which the core behaviour can be modelled as delta application
  - the resulting packet will be of type **data**
  - the state of the **store** is not modified by the evaluation

$$
\begin{aligned}
& p_{rx} = packet(task, i, j, r_j; e_i);\ e_i = \langle s_i \ldots r_j \ldots \rangle;\ r_j \rightarrow w_j \\
& S_i(store(\ldots)) \\
\longrightarrow^{M}\ & S_i(store(\ldots(r_j w_j)\ldots)) \\
\longrightarrow^{P}\ & S_i(store(\ldots)) \\
& p_{tx} = packet(data, j, i, r_j; w_i);\ w_i = \delta(s_i, \ldots, w_j, \ldots)
\end{aligned}
$$

Language service semantics

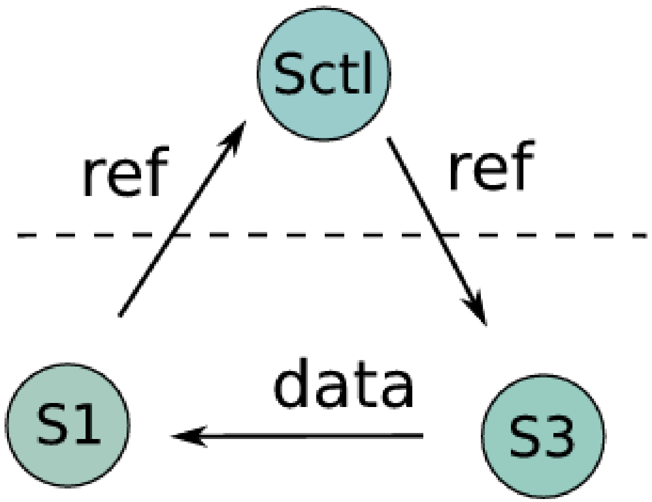
- **Language services** provide functional language constructs to the Gannet architecture
- Evaluation of a task by a language service can result in
  - the creation of a result packet of type **task, code** or **reference**
  - a change of the **state** of the **store**

- Why a new SoC architecture?
- Task control in NoC-based SoCs
- Gannet system
- Gannet language
- An operational semantics for Gannet
- Separation of data flow and control flow

Deferred evaluation and result redirection

The mechanism for separation of control and data flows: **deferred evaluation** and **redirection:**

```
(S1
   (Sctl
       '(S3 ...)
       ...
   )
   ...
)
```



S_1 sends $p(Ref, S_{ctl}, S_1, R_1'; R_{ctl})$;

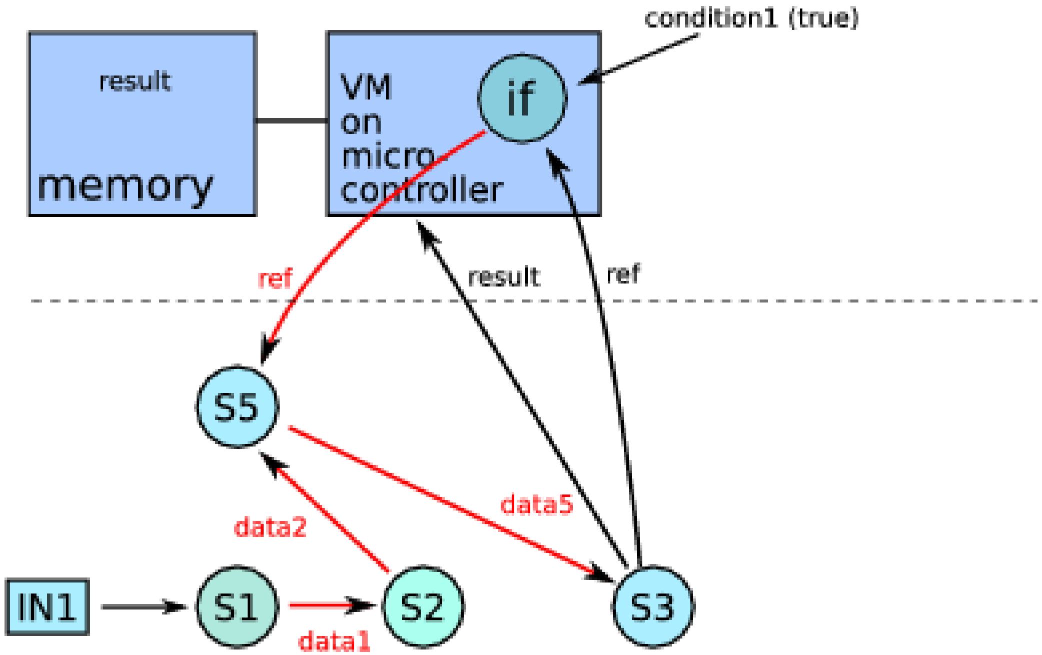
S_{ctl} sends $p(Ref, S_3, S_1, R_1'; R_3)$;

S_3sends $p(Data, S_1, S_3, R_1'; \langle data \rangle)$

Conditional branching

- Revisiting the earlier example:

```
(S3
   (if condition1
     '(S5 (S2 (S1 IN1)))
     '(S2 (S5 (s1 IN1)))
    )
)
```



S_{if} sends $p(Ref, S_5, S_3, R'_3; R_5)$;

S_5 sends $p(Data, S_3, S_5, R'_3; \langle data_5 \rangle)$

Conditional branching

(S3 ...)

$$p_{rx} = packet(task, \mathbf{S3}, \mathbf{gateway}, r_{res}; e_{S3});$$

$$e_{S3} \Rightarrow \langle \mathbf{S3}\, r_3 \rangle;\; r_3 \rightarrow w_3$$

$$S_{S3}(store(\ldots))$$

$$\longrightarrow^{M} S_{S3}(store(\ldots(r_3\, w_3)\ldots))$$

$$\longrightarrow^{P} S_{assign}(store(\ldots)); w_{res} = \delta(s_3, w_3)$$

$$p_{tx} = packet(data, \mathbf{gateway}, \mathbf{S3}, r_{res}; w_{res})$$

Conditional branching

(if ...)

$$p_{if} = packet(task, \mathbf{if}, S3, r_3; e_{if});\; e_{if} = \langle \mathbf{if}\, e_p\, qr_t\, qr_f \rangle;$$

$$qr \rightarrow r_{t|ft|f}; e_p \rightarrow w_p^B$$

$$S_{if}(store(...))$$

$$\xrightarrow{}^{M} S_{if}(store(...(r_p w_p^B)(qr_t r_t)(qr_f r_f)...))$$

$$\xrightarrow{}^{P} S_{if}(store(...));\; tf = w_p^B ?t : f$$

$$p_r = packet(reference, tf, S3, r_3; r_{tf})$$

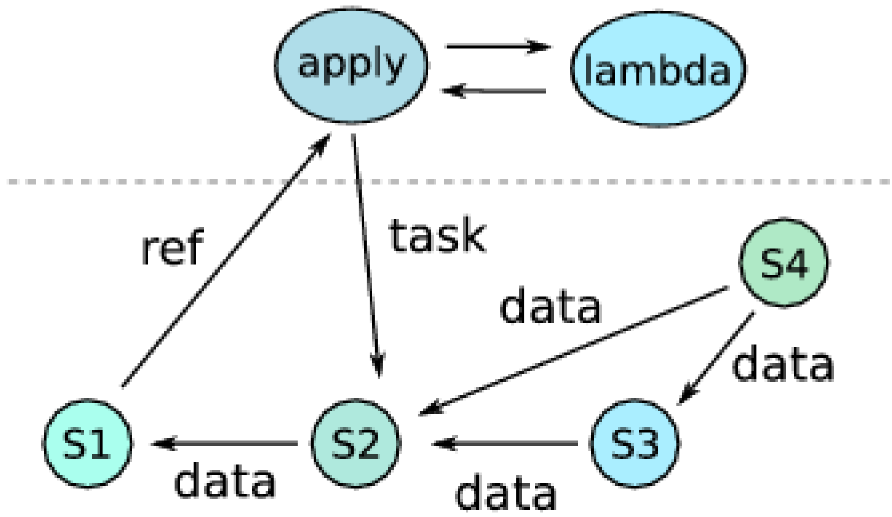
Conditional branching

(S2 ...)

$$
\begin{aligned}
& p_{rx} = packet(ref, \mathbf{S2}, \mathbf{S3}, r_3; r_{S2}); \\
& r_{S2} \Rightarrow e_{S2} = \langle \mathbf{S2}\, r_5 \rangle;\ r_5 \rightarrow w_5 \\
& S_{S2}(store(\ldots)) \\
\longrightarrow^{M}\ & S_{S2}(store(\ldots(r_5\, w_5)\ldots)) \\
\longrightarrow^{P}\ & S_{S2}(store(\ldots)); w_3 = \delta(s_2, w_5) \\
& p_{tx} = packet(data, \mathbf{S3}, \mathbf{S2}, r_3; w_3)
\end{aligned}
$$

Function definition and application

```
(S1
    (apply
        (lambda 'x
            '(S2 (S3 ... x ...) ... x ...)
        )
        '(S4 ...)
    )
    ...
)
```



$$S_1((\lambda x \rightarrow S_2(S_3(\ldots, x, \ldots), \ldots, x, \ldots))S_4(\ldots), \ldots)$$

Function definition and application

- Functions are defined by the `lambda` service:

$$p_{lambda} = packet(task, \mathbf{lambda}, *, *; e_{lambda});$$

$$e_{lambda} = \langle \mathbf{lambda}\, qx_j ... qr_\lambda \rangle;\ r_\lambda \Rightarrow \langle s_j ... x_j ... \rangle$$

$$S_{lambda}(store(...))$$

$$\longrightarrow^M S_{lambda}(store(...(qx_j x_j)...(qr_j r_\lambda)...))$$

$$\longrightarrow^P S_{lambda}(store(...))$$

$$p_r = packet(data, *, \mathbf{lambda}, *; e_\lambda);$$

$$e_\lambda = \langle ... x_j ... r_\lambda \rangle$$

Function definition and application

- Function application by the `apply` service:

$$
\begin{aligned}
& p_{apply} = packet(task, \mathbf{apply}, j, *; e_{apply}); \\
& e_{apply} = \langle \mathbf{apply}\, r_\lambda \ldots qr_j \ldots \rangle;\; r_\lambda \rightarrow e_\lambda;\; qr_j \rightarrow r_j \\
& S_{apply}(store(\ldots)) \\
\longrightarrow^M\; & S_{apply}(store(\ldots(r_\lambda\, e_\lambda)\ldots(qr_j\, r_j)\ldots)) \\
\longrightarrow^P\; & S_{apply}(store(\ldots)) \\
& p_c = packet(task, *, j, r_w; e_w);\; e_w = e_\lambda[x_j/r_j]
\end{aligned}
$$

Lexical scoping and Sequential evaluation

- The `group` service creates a block providing lexical scope
- Variables are bound to an expression by the `assign` service
- The `update` service updates the value bound to a variable

```
(group
      (assign 'v1 1)
      (assign 'v2 2)
     '(update 'v1 (+ (read 'v2) 1))
     '(update 'v2 (+ (read 'v1) 2))
     '(+ (read 'v1) (read 'v2))
 )
```

Lexical scoping and Sequential evaluation

- Semantics of the `assign` service:

$$
\begin{aligned}
& p_{rx} = packet(task, \mathbf{assign}, \mathbf{group}, r_a; e_{assign}); \\
& e_{assign} = \langle \mathbf{assign}\, qv_j\, r_j \rangle;\ r_j \rightarrow w_j \\
& S_{assign}(store_d(\ldots)) \\
\longrightarrow^M\ & S_{assign}(store_d(\ldots(qv_j\, v_j)\,(r_j\, w_j)\ldots)) \\
\longrightarrow^P\ & S_{assign}(store_d(\ldots(v_j\, w_j)\ldots)) \\
& p_{tx} = packet(data, \mathbf{group}, \mathbf{assign}, r_a; v_j)
\end{aligned}
$$

Lexical scoping and Sequential evaluation

- Semantics of the `update` service:

$$p_{rx} = packet(task, \mathbf{update}, i, r_r; e_{update});$$

$$e_{read} = \langle \mathbf{update}\, qv_j\, r_k \rangle$$

$$S_{read}(store_{assign}(\ldots(v_j\, w_j)\ldots))$$

$$\xrightarrow{}^{M} S_{read}(store_{assign}(\ldots(qv_j\, v_j)\,(r_k\, w_k)\,(v_j\, w_j)\ldots))$$

$$\xrightarrow{}^{P} S_{read}(store_{assign}(\ldots(v_j\, w_k)\ldots))$$

$$p_{tx} = packet(data, i, *, r_r; v_j)$$

Lexical scoping and Sequential evaluation

- Semantics of the `group` service:

$$p_{group} = packet(task, \mathbf{group}, i, r_l; e_{group});$$

$$e_{group} = \langle \mathbf{group} \ldots r_{assign,j} \ldots r_k \rangle;$$

$$r_k \Rightarrow \langle s_k \ldots r_j \ldots \rangle \longrightarrow w_k; r_j \Rightarrow \langle \mathbf{read}\, qv_j \rangle \rightarrow w_j$$

$$S_{group}(store_{assign}(\ldots))$$

$$\longrightarrow^{M} S_{group}(store_{assign}(\ldots(v_j w_j) \ldots (r_k w_k) \ldots))$$

$$\longrightarrow^{P} S_{group}(store_{assign}(\ldots))$$

$$p_r = packet(data, i, \mathbf{group}, r_l; w_k)$$

Lexical scoping and Sequential evaluation

- Quoting changes the semantics:

$$p_{group} = packet(task, \mathbf{group}, i, r_l; e_{group});$$

$$e_{group} = \langle \mathbf{group} \ldots r_{assign,j} \ldots qr_{k1} \ldots qr_{kN} \rangle;$$

$$S_{group}(store_{assign}(\ldots))$$

$$\longrightarrow^{M} S_{group}(store_{assign}(\ldots(v_j\, w_j)\ldots(qr_{k1}\; r_{k1})\ldots(qr_{kN}\; r_{kN})\ldots))$$

$$\longrightarrow^{P} S_{group}(store_{assign}(\ldots(v_j\, w_j)\ldots(r_{k1}\; \_)\;(qr_{k2}\; r_{k2})\ldots(qr_{kN}\; r_{kN})\ldots))$$

$$\longrightarrow^{M} S_{group}(store_{assign}(\ldots(v_j\, w_j)\ldots(r_{k1}\; w_{k1})\;(qr_{k2}\; r_{k2})\ldots(qr_{kN}\; r_{kN})\ldots))$$

$$\longrightarrow^{P} S_{group}(store_{assign}(\ldots(v_j\, w_j)\ldots(r_{k1}\; w_{k1})\;(r_{k21}\; \_)\ldots(qr_{kN}\; r_{kN})\ldots))$$

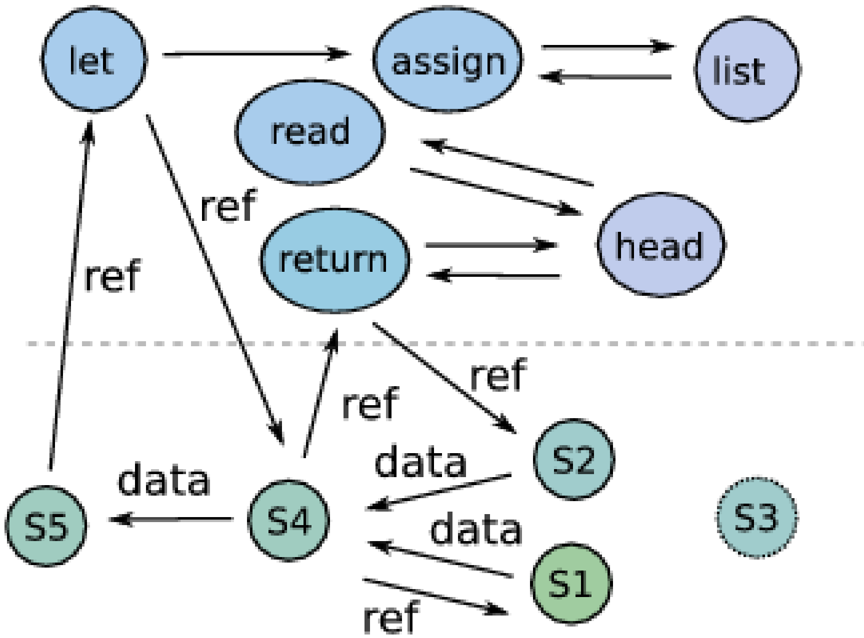
$$\ldots$$

$$p_r = packet(ref, i, \mathbf{group}, r_l; r_{kN})$$

Lists

- `list`: list constructor
- `head`: first element of the list
- `return`: unquotes its argument

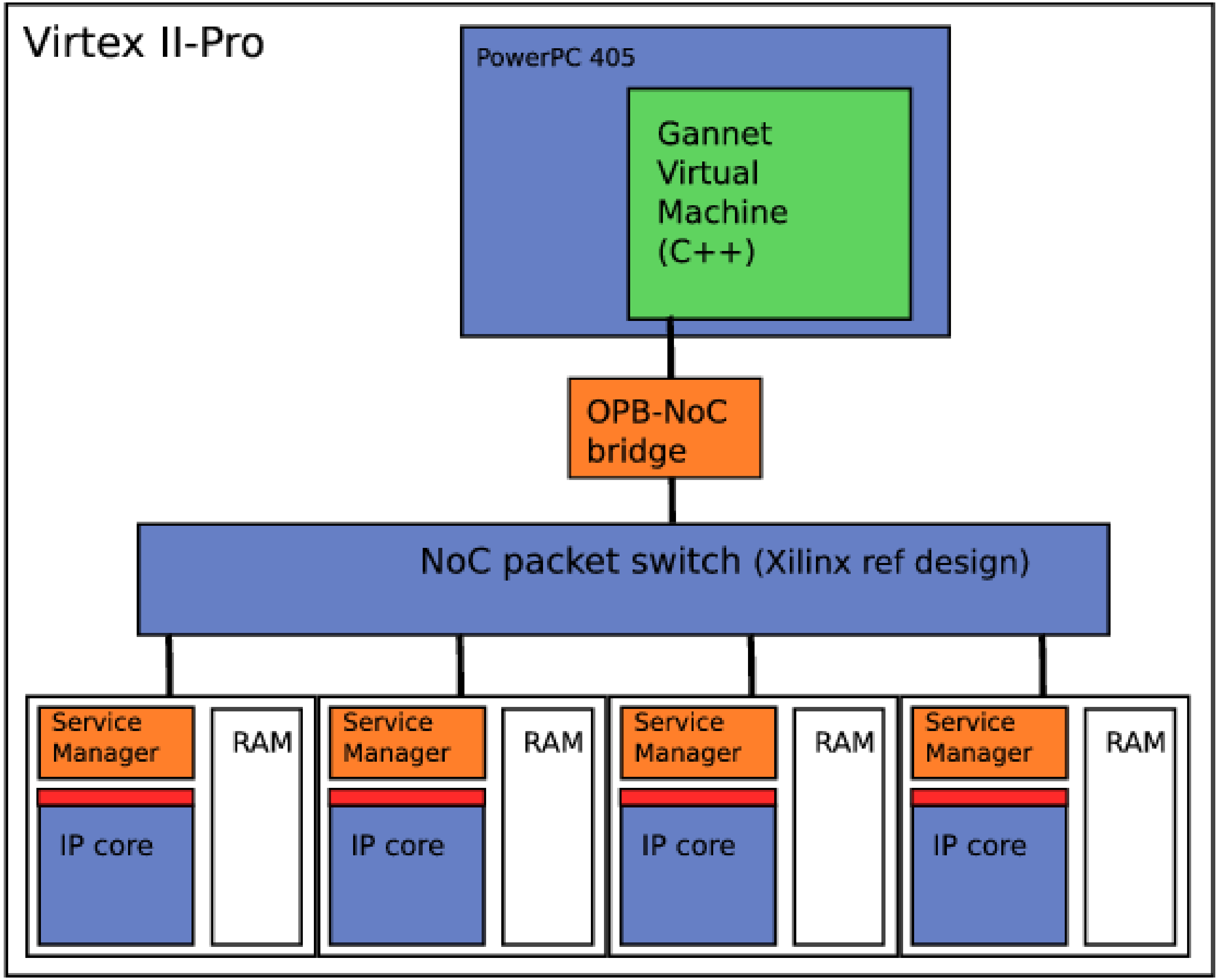
```
(S5 (let

    (assign 'l (list '(S2 ...) '(S3 ...)))

    '(S4 (return
              (head (read 'l))) (S1 ...))

))
```



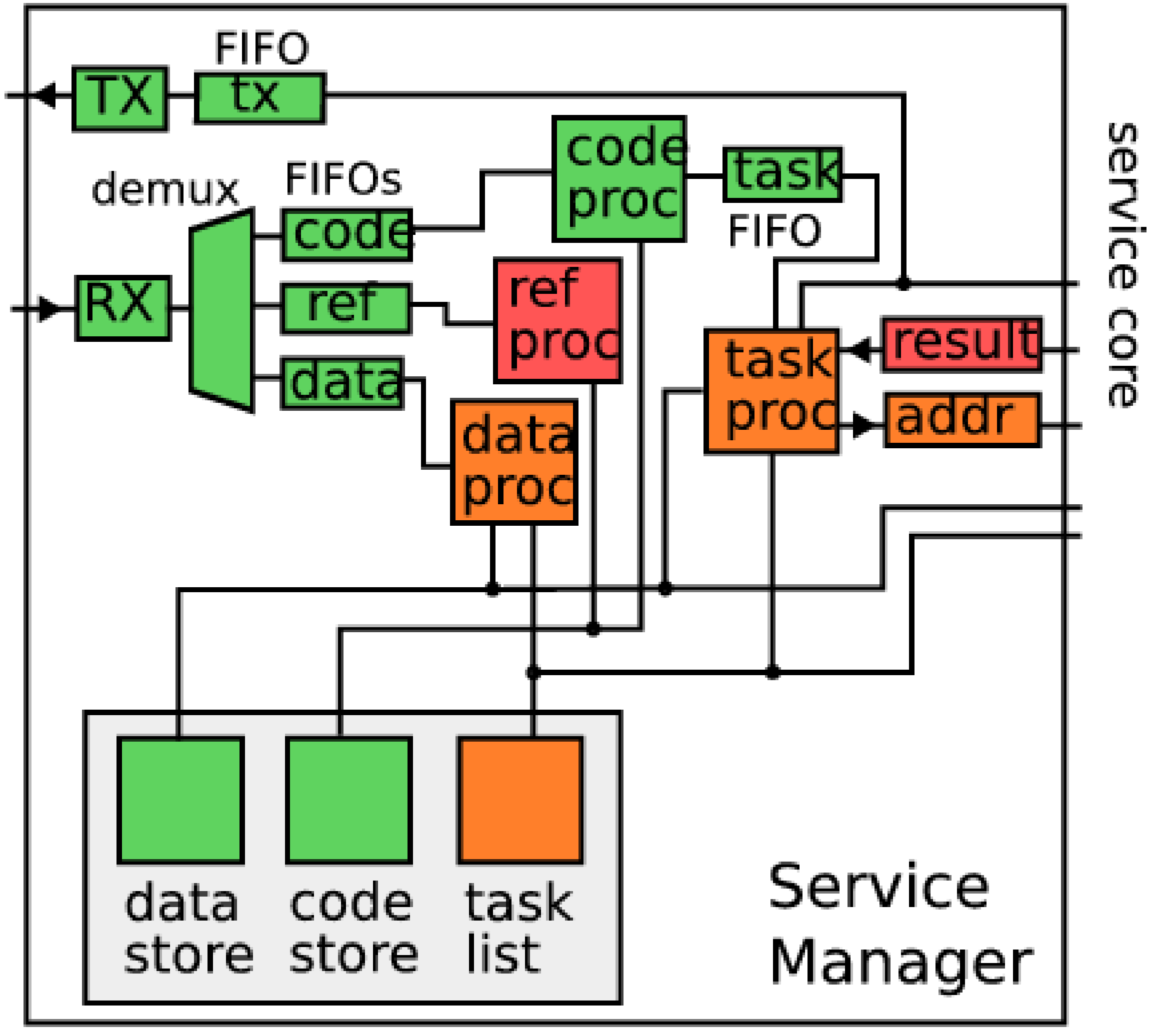
Conclusion

- Gannet: a service-based SoC architecture for high-level design of reconfigurable heterogeneous multi-core SoCs.
- Alleviate bottleneck resulting from memory bandwidth limitations: mechanism for the **separation of control flow and data flow** based on **deferred evaluation** and **packet redirection**.
- Gannet system and language
  - provides full control over data paths in multi-core SoC;
  - provides full concurrency;
  - ensures that data can flow directly between the cores.

Status


Virtex II-Pro
PowerPC 405
Gannet Virtual Machine (C++)
OPB-NoC bridge
NoC packet switch (Xilinx ref design)
Service Manager
RAM
IP core
Service Manager
RAM
IP core
Service Manager
RAM
IP core
Service Manager
RAM
IP core

Status


FIFO
TX
tx
demux
FIFOs
code
proc
task
FIFO
code
RX
ref
ref
proc
data
result
task
proc
addr
data
proc
service core
data
store
code
store
task
list
Service
Manager