A Formal Semantics for Control and Data flow in the Gannet Service-based System-on-Chip Architecture

Wim Vanderbauwhede
Department of Computing Science, University of Glasgow, UK
wim@dcs.gla.ac.uk

***Abstract*— There is a growing demand for solutions which allow the design of large and complex reconfigurable Systems-on-Chip (SoC) at high abstraction levels. The Gannet project proposes a functional programming approach for high-abstraction design of very large SoCs. Gannet is a distributed service-based SoC architecture, i.e. a network of services offered by hardware or software cores. The Gannet SoC is task-level reconfigurable: it performs tasks by executing functional task description programs using a demand-driven dataflow mechanism. The Gannet architecture combines the flexible connectivity offered by a Network-on-Chip with the functional language paradigm to create a fully concurrent distributed SoC with the option to completely separate data flows from control flows. This feature is essential to avoid a bottleneck at he controller for run-time control of multiple high-throughput data flows.**
In this paper we present the Gannet architecture and language and introduce an operational semantics to formally describe the mechanism to separate control and data flows.

Distributed System-on-Chip architecture, Operational Semantics, Service-based System-on-Chip, Network-on-Chip

I. The Gannet service-based SoC architecture

There is a growing demand for solutions allowing to design complex reconfigurable Systems-on-Chip (SoC) at high abstraction levels [1], [2]. The Gannet project aims to address this need by proposing a novel, task-level reconfigurable System-on-Chip architecture which uses a concurrent execution paradigm based on functional language processing. The Gannet architecture is a proposed distributed service-based System-on-Chip architecture which performs tasks through the interaction of services offered by heterogeneous data processing cores [3], [4].

The tasks are expressed using a functional *task description language*. The Gannet fabric consists of a set of *service nodes*, each offering one or more *services* to the system. All nodes are connected through a flexible interconnect medium.

In practice, the Gannet System-on-Chip (Fig. 1(a)) consists of a regular matrix of processing units (*tiles*) connected through a network-on-chip (NoC). The architecture is motivated by the growing complexity offered by the latest generation of IC manufacturing technologies. Following Moore's law, the complexity of integrated circuits has grown steadily in the past decades, from ICs with a few components via increasingly performant microprocessors to ever more complex Systems-on-Chip [5], [6]. Tomorrow's SoCs will be *very big* (billions of logic gates). The main issues with these very large SoCs are connectivity and design complexity [7]. Traditional bus-style interconnects are no longer a viable option: synchronisation of hundreds of processing cores over large distances is impossible; fixed point-to-point connections result in huge wire overheads. *Packet-switched Networks-on-Chip (NoCs)* [8] provide a solution because they offer flexible connectivity and an efficient mechanism for managing wires.

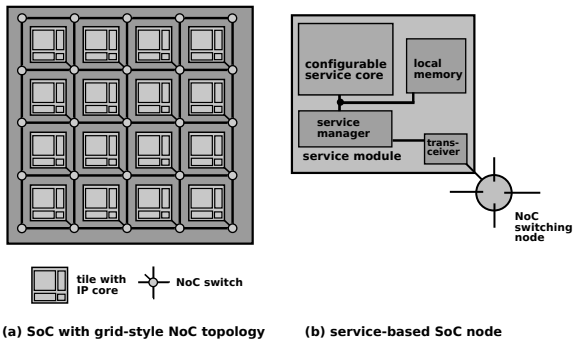

Fig. 1. Gannet service-based SoC with on-chip network

For very large SoCs, *design reuse* is essential [9]. Design reuse is facilitated by the concept of IP cores. These are are highly complex, self-contained processing units offering a specific functionality, such as data acquisition units, audio/video codecs, cryptography cores, TCP/IP packet filtering etc. They can be implemented as hardware logic circuits, as embedded microcontrollers running specific software, or combinations of both.

Because of their self-contained nature, treating IP blocks as *services* is a logical abstraction. To achieve service-based behaviour, every tile of a Gannet SoC contains a special control unit (the *service manager*), which provides a service-oriented interface between the IP core and the system (Fig. 1(b)). Designing a Gannet SoC reduces to instantiating the IP cores in the Gannet fabric and creating a task description program.

II. Gannet machine operation

We can consider the Gannet SoC as a machine for running Gannet programs. The Gannet SoC architecture is quite different from the familiar von Neumann-style processor-based SoC architecture: it is a distributed processing system without global memory. There is no program counter, and the program is not executed in a sequential fashion but in a demand-driven dataflow fashion [10], [11]. We introduce a more formalised description of the Gannet machine:

- The Gannet machine is a distributed computing system where every computational node *consumes packets* and *produces packets* and can store state information between transactions.
- A Gannet packet consist of a header and a payload. The payload can either be data or instruction code (symbols). The header consists of following fields: Packet *Type* (*code*, *reference*, *data*), destination address (*To*), return address (*Ret*), packet identifier (*Id*). We denote a Gannet packet as $p(Type, To, Ret, Id; Payload)$

Thus in general terms, the semantics of a Gannet service can be described in terms of the task code and the result packet produced by the task as follows:

- SC**: Store code packet:** a 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 and referenced by R_{task}.
- AT**: Activate task :** the service S_i in *state*$_i$ receives a task *reference* packet $p(Ref, S_i, S_j, R_{id}; R_{task})$ [1]
  the service activates the task referenced by R_{task}: $(S_i\ a_1...a_n)$. This results in evaluation of the arguments $a_1..a_n$:
  - DR**: Delegate by reference packet:** the service manager requests activation of subtasks referenced by *reference sy*mbols by sending a *reference* packet to the corresponding service
  - SQ**: Store constant symbol:** all *constant* symbols (e.g. numbers) in the code ares stored in the local store.
  - SR**: Store returned result:** result data from subtasks are stored in the local store.
- P**: Processing:** When all arguments of the subtask have been evaluated, the data are passed on to the service core. The core performs processing on the date; the service, now in *state*$_i$*'*, produces a result packet
  $p_{res} = p(Type_i, S_j, S_i, R_{id}; Payload_i)$ where both *Payload*$_i$ and the state change to *state*$_i$*'* are the result of processing the evaluated arguments $a_1..a_n$by the core of S_i.
- p_{res} is sent to S_j where $Payload_i$ is stored in a location referenced by R_{id}.

This operation sequence results in a fully parallel execution of all branches in the program tree in an unspecified order governed by the processing time of the packets.

III. The Gannet language

The tasks performed by the Gannet system are expressed in a functional task description language. A task description defines the interactions between the services by mapping every service to a *named function*, and describing the flow of data between the functions in terms of function calls.

The Gannet language is the equivalent of an assembler language for the Gannet machine. By this we mean that a program written in Gannet syntax can be transformed in machine code in a trivial way. In this section we discuss the syntax, semantics an compilation of the language.

[1]The order of arrival is actually irrelevant: if the reference arrives earlier, activation will occur as soon as the code arrives.

A. Syntax

Gannet syntax is an s-expression syntax (similar to LISP or Scheme [12], [13]) completely free from syntactic sugar. In BNF, a Gannet expression must always obey

$$\begin{aligned} \textit{service-expr} &::= (\ \textit{service-symbol}\ \ '?\,arg-expr+\ ) \\ \textit{arg-expr} &::= \textit{service-expr} \,|\, literal-symbol \end{aligned}$$

where *service-symbol* represents a particular service. Every service in the system has a corresponding *service-symbol* in the program. There are no other keywords in the language, i.e. all flow control constructs are provided by services. The only additional syntactic construct is the quote.

Consider as a trivial example a SoC with 4 services: image capture (`img`) from several cameras, creating a composite image (`compose`), conversion to jpeg format or png format (`convert`), compression (`compress`) and encryption (`encrypt`). Then to obtain a compressed composite of raw image from cameras 1 and 3, the task description would be

```
(compress (compose (img cam1) (img cam3)))
```

To obtain a jpeg-converted, encrypted image from camera 2 it would be

```
(encrypt (convert jpeg (img cam2)))
```

Control services

To allow control over the flow of data, Gannet defines a number of *control* services. The core set consist of the lexical scoping constructs (**group**, **assign**, **read**), the branching construct (**if**) and the function definition and application constructs (**lambda**, **apply**).

a) Lexical scoping:

$$\begin{aligned} \textit{group-expr} &::= (\mathbf{group}\ '?\textit{assign-expr}+\ '?\textit{arg-expr}\ ) \\ \textit{assign-expr} &::= (\mathbf{assign}\ '\,\textit{var-symbol}\ \ arg-expr+\ ) \\ \textit{read-expr} &::= (\mathbf{read}\ '\,\textit{var-symbol}\ \ ) \end{aligned}$$

b) Conditional branching:

$$\textit{if-expr} ::= (\mathbf{if}\ \textit{service-expr}\ '?\textit{arg-expr}\ '?\textit{arg-expr}\ )$$

c) Function definition and application:

$$\begin{aligned} \textit{lambda-expr} &::= (\mathbf{lambda}\ '\,\textit{var-symbol}+\ '\,\textit{service-expr}\ ) \\ \textit{apply-expr} &::= (\mathbf{apply}\ \textit{lambda-expr}\ '\,\textit{arg-expr}+\ ) \end{aligned}$$

B. Gannet core language operational semantics

To provide understanding of how the Gannet *language* works (as opposed to the Gannet *machine*), we present a conventional small-step semantics for the Gannet core language.

1) Notations and Definitions: We use a context-sensitive reduction semantics as introduced by Felleisen [14] and used in [15]. Because the Gannet machine is different from a von Neumann machine, some minor modifications to the notation are required. Full details can be found in [3].

a) Evaluation context: The context for evaluation must always be a service expression, unless the given expression is a service expression. Thus $C ::= [\,] \,|\, (s...C...)$. Evaluation of an expression is independent of neighbouring or enclosing expressions.

b) Store: The Gannet machine does not have a global memory. Rather, every service has its own local memory, with read-only access for the other services. This means that the $\mathbf{store}()$ concept must be contextualised. The context of the store is indicated with a subscript.

c) Shorthand notation: To keep the notation concise, we will use following shorthand for the terms introduced above:

$arg\text{-}expr : e$	$service\text{-}symbol : s$
$quoted\text{-}expr : qe$	$var\text{-}symbol : v$
$value : w$	$arg\text{-}symbol : x$

Furthermore, an expression e_i always evaluates to a value w_i: $e_i \rightarrow w_i$

d) Non-control-service semantics:: The operational semantics of a program running on the Gannet machine in the absence of control services amounts to simple δ-application:

$$C[(s\ e_1...e_n)] \quad \rightarrow \quad C[w];\ w = \delta(s, w_1, ..., w_n)$$

e) Control service semantics: As discussed above, the introduction of language services leads to a number of additional rules and symbols. We assume all expressions are well-formed.

f) Grouping and Variables: The grouping and assignment services effectively operate as a let-construct. As every service can only act on a local store, the **group**, **assign** and **read** services must be provided by a single service core.

The **assign** construct performs the binding:

$$\begin{aligned} & (\mathbf{store_{assign}}(...)\ C[(\mathbf{assign}\ qv\ e)) \\ \rightarrow\ & (\mathbf{store_{assign}}(...(v\ w)...)\ C[v]) \end{aligned}$$

Values bound to variables are requested using **read**:

$$\begin{aligned} & (\mathbf{store_{assign}}(...(v_1 w_1)...)\ C[(\mathbf{read}\ qv_1)]) \\ \rightarrow\ & (\mathbf{store_{assign}}(...(v_1 w)...)\ C[w]) \end{aligned}$$

The grouping construct **group** performs checks if all assigns were successful, and if so, it returns the value of the last argument, otherwise it returns an error symbol. It also deallocates the memory for the variables bound by its assign arguments:

$$\begin{aligned} & (\mathbf{store_{assign}}(...)\ C[(\mathbf{group}\ ...(\mathbf{assign}\ qv_i\ e_i)... \\ & (s...(\mathbf{read}\ qv_i)...)]) \\ \rightarrow\ & (\mathbf{store_{assign}}(...)\ C[w]) \end{aligned}$$

If the arguments to group are quoted, the semantics is different in that their evaluation is sequential rather than concurrent (similar to `let*` in Scheme).

g) Conditional branching: The syntax for the **if** service allows both quoted and unquoted arguments. Semantically, the behaviour is the same in both cases; however, as explained in Subsection V, quoting causes the result of the selected service to be redirected to caller of the **if** service.

$$\begin{aligned} & C[(\mathbf{if}\ e_p qe_t\ qe_f)] \\ \rightarrow\ & C[w_{tf}]; tf = w_p \neq 0?t : f \end{aligned}$$

$$\begin{aligned} & C[(\mathbf{if}\ e_p e_t\ e_f)] \\ \rightarrow\ & C[w_{tf}]; tf = w_p \neq 0?t : f \end{aligned}$$

h) Function definition and application: This is the operational semantics for lambda functions in Gannet.

Functions are defined using the **lambda** service:

$$\begin{aligned} & C[(\mathbf{lambda}\ qx_1...qx_n\ qe_a)] \\ \rightarrow\ & C[\langle\ x_1...x_n e_a\rangle] \end{aligned}$$

Function application is done by the **apply** service:

$$\begin{aligned} & (\mathbf{store_{apply}}(...) \\ & C[(\mathbf{apply}\ (\mathbf{lambda}\ qx_1...qx_n\ qe_a)\ \ qe_1...qe_n]) \\ \rightarrow\ & (\mathbf{store_{apply}}(...(x_1\ e_1)...(x_n\ \ e_n))\ \ C[e_a[\ x_i/\ e_i]) \\ \rightarrow\ & (\mathbf{store_{apply}}(...)\ \ C[w_a]) \end{aligned}$$

C. Compilation

This section explains how a Gannet program is compiled into packets for running on the Gannet machine.

The compilation process is very straightforward:

1) Decompose the nested s-expression into a list of flat s-expressions by replacing the nested expressions by references

$$\begin{aligned} & e_{root} = (S_{root}\ e_1...e_i...e_n) \\ & e_i = (S_i\ e_{i,1}...e_{i,j}...e_{i,n}) \\ & e_{i,j} = (S_{i,j}\ e_{i,j,1}...e_{i,j,k}...e_{i,j,m}) \\ & ... \end{aligned}$$

2) Every symbol in the expression is replaced by a tuple (a structured byteword) containing a unique number mapped to the symbol and its *kind*:

$$\begin{aligned} & e_i \Rightarrow r_i = (reference, n_{r_i}) \\ & S_i \Rightarrow s_i = (service, n_{S_i}) \end{aligned}$$

The resulting list of bytewords is called an *instruction*. Instructions are represented using pointy brackets: $\langle s_i\ r_{i,1}...r_{i,n}\rangle$is the instruction referenced by r_i.

3) Create code packets: using the notation introduced above, a code packet is represented as
$p_i = packet(code, n_{S_i}, GW, r_i; \langle s_i\ r_{i,1}...r_{i,n}\rangle)$
with $n_{S_i}$the name of the service S_i, GW is the "gateway", the interface between the Gannet SoC and the outside world.

4) Create a reference packet to the root task:
$p_{root} = packet(reference, n_{S_{root}}, GW, r_{root}; r_{root})$

The gateway transfers the packets onto the NoC in no particular order.

D. Services and instruction sets

The Gannet machine has been introduced as a *distributed processing system* built out of *tiles* connected via a Network-on-Chip. Every tile consists of an IP *core* providing services, a *service manage*r and a local store. The service manager processes packets based on their type and processes instructions (payloads of code packets) using a few simple rules. The instruction set of the Gannet machine is the set of services

offered by the cores. Consequently, the instruction set is entirely application-dependent and can be configured at design time or, if the cores are reconfigurable, at run time.

IV. Semantics of the Gannet machine

In this section we will present an small-step operational semantics for the Gannet *machine* based on operations on the packets which make up a Gannet program and on the content of the local store.

A. Notation and definitions

1) Notation:

- The notation $\bullet$ is used to separate a packet from the other packets in the queue: $(p \bullet ps)$ denotes a packet at the head, $(ps \bullet p)$ a packet at the tail.
- The notation $*$ ("don't care") indicates that the value of a field does not influence the operation.
- The notation ... indicates the presence some non-specified entities. In general, unspecified entities are left out unless omitting them would cause ambiguity.
- The notation _ indicates allocated available storage space

2) Definitions:

- The Gannet system consists of N service nodes $S_i(...)$, $i \in 1..N$, a packet-switched communication medium ("Network on Chip") and a gateway to the outside world, $G(...)$.
- The unit of data transfer in the Gannet SBA is the packet. Depending on the packet's $Type$, the $Payload$ can be *data* or an *expression*.
- The packet receive and transmit FIFO queues of the services are represented by q_{RX} and q_{TX}. A received a packet is pushed onto the RX queue; a transmitted packet is shifted off the TX queue.
  The RX queue actually consists of four queues multiplexed by the packet's $Type$:
  $q_{RX}(tasks(...), data(...), refs(...), code(...))$.
  Thus $q_{RX}(ps \bullet p)$ is actually $q_{RX}(...pt(ps \bullet p)...)$ with$pt \in \{tasks, data, refs, code\}$. (In the actual design *tasks()* is not part of the RX queue, but placing it there simplifies the analysis.)
- Packet receive and transmit FIFO queues: q_{RX} and q_{TX}
- Apart from the RX/TX queues, a service node S_i consists of following entities:
  - The data store:$store_d(...(Label\ data)...)$. $Label$ is a Gannet symbol, $data$ is the stored content. Space allocated for data to be stored is denoted by _: $store_d(...(Label\ \_)...)$
  - The task packet store: $store_{tp}(...(Label\ p)...)$. p is the stored packet,$Label$ is the Label field from the packet's header.
  - The processing core $core(...)$ which performs the actual processing of the data.

Thus an explicit notation for a service node $S_i$is:

$$S_i(q_{RX}(tasks(...), data(...), refs(...), code(...)), \\ q_{TX}(...), store_d(...), store_{tp}(...), core(...))$$

At any given moment, every service S_i can be performing any number of actions. Actions are data-driven. Furthermore, all services are operating concurrently in a completely asynchronous fashion.

3) Small-step semantics: The semantics expresses an action taken by a service. Actions (indicated with the arrow $\longrightarrow^A$) are triggered either by arrival of a packet or by completion of a computation by the service core. Every expression in the semantics describes the effect of the action in terms of the state of the service, i.e. of its stores and queues.

lines above the transition expression define items (e.g. packets) appearing on the LHS, lines below the transition expression define items appearing on the RHS.

$$\begin{array}{rrl} p_i = packet(...); ... & & \\ \qquad\qquad S_i(...p_i...) & \longrightarrow^A & S_i(...p_j...) \\ p_j = packet(...); ... & & \end{array}$$

B. Packet processing by the services

A service performs a set of actions which result in packets being received from and transmitted to other services.

A subset of actions (the *marshalling* set) is performed by the service manager, which is the generic data marshalling unit through which every service core interfaces with the system. It is important to note that the service manager is generic, i.e. its design and functionality is independent of the design and functionality of the service core. The complementary set of actions (the *processing* set) is performed by the service core.

1) Packet transfer between services: The set of actions to transfer packets between services consists of TX (transmit) and RX (receive). The semantics are straightforward:

$$\begin{array}{rrl} p = packet(*, i, j, *; *) & & \\ \qquad S_i(q_{TX}(p \bullet ps)) & \longrightarrow^{TX} & S_i(q_{TX}(ps)) \\ \qquad S_j(q_{RX}(qs)) & \longrightarrow^{RX} & S_j(q_{RX}(qs \bullet p)) \end{array}$$

Both actions carry the implicit assumption that the system's NoC will transfer the packet correctly between nodes S_i and S_j. Note that the actions don't happen synchronously: the NoC is asynchronous and the delay for transmission of the packet is unknown.

2) Marshalling action set M: On receipt or activation of a *task* packet, a number of actions can be performed by the service manager, as explained in II.These are grouped in the M ("Marshalling") set. Application of the M set results in evaluation of all arguments of a service call. The actions of the complete M set $\{SC, AT, DR, SR, SQ\}$ can be expressed as:

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

This expression describes the evaluation of function arguments by the Gannet service. For the sake of brevity, the actions leading on to the activation of the task packet have been omitted. Instead, it is simply assumed that the service manager receives a task packet. It is easy to show that this is equivalent to applying the $\{SC, AT\}$action set on arrival of a reference packet.

3) 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 the service produces based on the values marshalled by the service manager.

The service core implements a function cs_iwhich takes n arguments with values $w_1 ... w_n$ and produces a result w, optionally modifying the state of the store in the process.

The P set consists of the actions $\{call, eval, return\}$: the values are called from the store; the core performs its computation (*eval*) and returns a result.

The processing can be expressed as:

$$\begin{aligned} & S_i(q_{RX}(qs), q_{TX}(ps), store((s_1\ w_1)...(s_n\ w_n)\ state)) \\ \longrightarrow^P\ & S_i(q_{RX}(qs), q_{TX}(ps \bullet p), store(state')) \\ & p = packet(*, *, i, *; w);\ (cs_i\ w_1\ ...\ w_n) \rightarrow w \end{aligned}$$

C. Control and non-control service semantics

In this section and the next, all actions are combined in the M and P sets. The combined M and P action sets result in the service transmitting a result packet in response to receiving a task packet and potentially modifying the local store. The semantics of the Gannet services can be described completely in terms of the task and result packets and the state of the store. The aim of the next two sections is to illustrate this mechanism.

1) Non-control service semantics: Non-control services are services of which the core behaviour can be modelled as δ-application. This type of service includes all third-party IP cores in the SoC, as these cores have no knowledge of the Gannet system.

The resulting packet will be of type *data* and the state of the *store* is not modified by the evaluation.

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

Note that this does not mean that the IP core is stateless, only that it does not affect the state of the service manager stores.

2) Control service semantics: Control services provide functional language constructs to the Gannet architecture. Evaluation of a task by a language service can result in *a change of the state of the store* or the creation of a *result packet of type task*. To illustrate these mechanisms, this section presents the semantics for the `group`, `assign`, `if`, `lambda` and `apply` services.

a) Lexically scoped variables: Lexical scoping is implemented through the `group` and `assign` services:

Variables are bound to an expression by the `assign` service:

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

The `read` service retrieves the value bound to a variable from the store and returns it:

$$\begin{aligned} & p_{rx} = packet(task, \mathbf{read}, i, r_r; e_{read}); \\ & e_{read} = \langle \mathbf{read}\ qv_j \rangle \\ & S_{read}(store_{assign}(...(v_j\ w_j)...)) \\ \longrightarrow^M\ & S_{read}(store_{assign}(...(qv_j\ v_j)\ (v_j\ w_j)...)) \\ \longrightarrow^P\ & S_{read}(store_{assign}(...(v_j\ w_j)...)) \\ & p_{tx} = packet(data, i, *, r_r; w_j) \end{aligned}$$

The `group` service takes as arguments a number of assignment expressions and one or more expression that may call the assigned variables. The `group` and `assign` services have a shared store. The `group` service returns the result of the last expression and clears all variables resulting from the assignment expressions from the `assign` store. Consequently, `assign`-variables are lexically scoped.

$$\begin{aligned} & p_{group} = packet(task, \mathbf{group}, i, r_l; e_{group}); \\ & e_{group} = \langle \mathbf{group}\ ...r_{assign,j}... r_k \rangle; \\ & r_k \Rightarrow \langle s_k ... r_j ... \rangle \longrightarrow w_k; r_j \Rightarrow \langle \mathbf{read}\ qv_j \rangle \rightarrow w_j \\ & S_{group}(store_{assign}(...)) \\ \longrightarrow^M\ & S_{group}(store_{assign}(...(v_j\ w_j)\ ...(r_k\ w_k)...)) \\ \longrightarrow^P\ & S_{group}(store_{assign}(...)) \\ & p_r = packet(data, i, \mathbf{group}, r_l; w_k) \end{aligned}$$

3) Conditional branching: Conditional branching is implemented by the `if` service:

$$\begin{aligned} & p_{if} = packet(task, \mathbf{if}, j, r_j; e_{if});\ e_{if} = \langle \mathbf{if}\ r_p\ r_t\ r_f \rangle; \\ & r_{t|f} \Rightarrow \langle s_{t|f} ... \rangle; r_p \rightarrow w_p^B \\ & S_{if}(store(...)) \\ \longrightarrow^M\ & S_{if}(store(...(r_p\ w_p^B)\ (r_t\ w_t)\ (r_f\ w_f)...)) \\ \longrightarrow^P\ & S_{if}(store(...)) \\ & p_r = packet(data, j, \mathbf{if}, r_j; w_{tf});\ tf = w_p^B ? t : f \end{aligned}$$

With the above semantics, both branches will be evaluated. If the second and third argument are quoted, evaluation is deferred to the core which will evaluate only one branch predicated on the value of the first argument. This case is covered in Subsection V.

a) Lambda functions: Function definition and application is implemented through the `lambda` and `apply` services:

- Functions are defined by the `lambda` service:

$$
\begin{aligned}
& p_{lambda} = packet(task, \mathbf{lambda}, *, *; e_{lambda}); \\
& e_{lambda} = \langle \mathbf{lambda}\, q x_j ... q r_\lambda \rangle;\ r_\lambda \Rightarrow \langle s_j ... x_j ... \rangle \\
& S_{lambda}(store_d(...)) \\
\longrightarrow^M\ & S_{lambda}(store_d(...(x_j\ x_j)...\,(r_j\ r_\lambda)...)) \\
\longrightarrow^P\ & S_{lambda}(store_d(...)) \\
& p_r = packet(data, *, \mathbf{lambda}, *; e_\lambda); \\
& e_\lambda = \langle ... x_j ... r_\lambda \rangle
\end{aligned}
$$

- Function application by the `apply` service:

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

As can be seen from this semantics, the `apply` service does not bind values but rather substitutes code references. The reason for this behaviour is explained in the next section.

V. Separation of control flow from data flow

In this section we apply the presented small-step semantics to an issue of crucial importance for the performance of SoCs with multiple high-throughput data flows: the separation of control flows and data flows.

A. Background

Consider a simple SoC which implements a video webcam. The system captures a video stream, compresses it using a lossy codec, encrypts it and transmits it over a network. The functionality for these four operations is provided by hardware IP cores. Following a conventional SoC design approach, this system will be implemented using a microcontroller which runs an embedded operating system (Fig. 2).

The OS communicates which the hardware using device drivers. Data is transferred to and from the central memory either via the microcontroller or via a direct memory access (DMA) controller. Clearly this memory transfer is a bottleneck: if the number of datapaths or the number of operations per datapath would be very large, the memory access time will limit the datarate.

The Gannet architecture allows separation of dataflow from control flow: the data will be transferred directly between the data processing cores (Fig. 3), eliminating the bottleneck. Consequently the system will be able to support more and longer data paths. As the net number of data transfers is lower, the system will also consume less power than the conventional architecture.

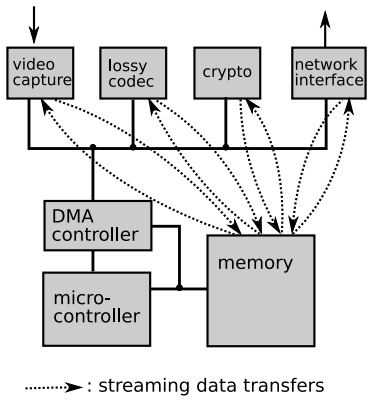

Fig. 2. Simple videocam SoC with embedded OS

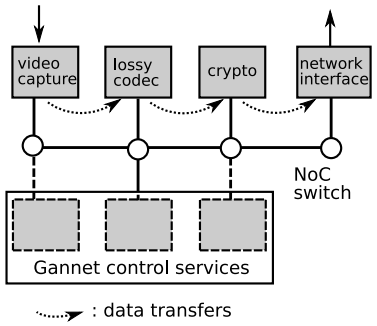

Fig. 3. Gannet architecture for simple videocam SoC

B. A combined control service

The granularity of the control services is quite fine and the implementation is relatively simple compared to the actual IP cores. For that reason it makes sense to combine all control constructs into a single service, as schematically indicated in Fig. 3. The *Name* field of the service symbol indicates which construct must be provided for a given call. The difference with the control services as presented above is limited to the To-field. For example, if we name the combined control service `control`, an `assign` task packet now becomes:

$$
\begin{aligned}
& p_{assign} = packet(task, \mathbf{control}, j, r_j; e_{assign}) \\
& e_{assign} = \langle \mathbf{assign}\ qv\ r \rangle
\end{aligned}
$$

The advantages of combining the control services are less overhead thanks to the shared service manager and local store and the potential to factor out common functionality, thus further reducing the required area. The potential disadvantage is due to the lower degree of parallelism and the potentially large number of calls to be handled, i.e. the classic trade-off of speed for area. However, the important issue is to make sure no control service, either combined or individual, is present in the data path.

C. Redirection of data flows

For optimal performance control flows must be separated from data flows, i.e. data should flow between non-control

services while the actual data path is governed by the control service. If this would not be the case, data would have to be copied to and from the control service's local memory, causing a performance bottleneck. The Gannet system as presented solves this issue via a combination of *deferred evaluation* and *result redirection*. To explain this mechanism, we use the control services introduced above as examples.

Consider the expression:

```
(S1 (group
        (assign 'v1 ...)
        (S2 ... (read 'v1) ...)
))
```

With the semantics as presented in Subsection IV-C.2.a, the `group` service receives the result of the evaluation of the `S2` call. It then passes this result on to `S1`. However, the behaviour is different when the last argument is quoted:

```
(S1 (group
        (assign 'v1 ...)
       '(S2 ... (read 'v1) ...)
))
```

In this case, evaluation the last expression is deferred to the `group` core. The core dispatches a reference packet to `S2` but sets the return address to `S1`:

$$
\begin{aligned}
& p_{group} = packet(task, \mathbf{control}, S1, r_l; e_{group}); \\
& e_{group} = \langle \mathbf{group} \, ... r_{assign} ... \, qr_{S2} \rangle; \\
& r_{assign} = \langle assign \; qv_j \; r_j \rangle \\
& r_{S2} \Rightarrow \langle S2 ... r_j ... \rangle \longrightarrow w_k; r_j \Rightarrow \langle \mathbf{read} \, qv_j \rangle \rightarrow w_j \\
& S_{group}(store_{assign}(...)) \\
\longrightarrow^M \quad & S_{group}(store_{assign}(...(v_j \; w_j) \; ...(qr_{S2} \; r_{S2})...)) \\
\longrightarrow^P \quad & S_{group}(store_{assign}(...)) \\
& p_r = packet(reference, S2, S1, r_l; r_{S2})
\end{aligned}
$$

Similarly, the `if` service performs redirection if the selected argument is quoted.

```
(S1 (if (Sp ...) '(S2t ...) '(S2f ...)))
```

So if the `Sp` call evaluates to true, the result of `S2t` will be sent directly to `S1`.

$$
\begin{aligned}
& p_{if} = packet(task, \mathbf{control}, S1, r_1; e_{if}); \\
& e_{if} = \langle \mathbf{if} \, e_p \, qr_t \, qr_f \rangle; \; e_p = \langle s_p ... \rangle; e_p \rightarrow \mathbf{true} \\
& S_{if}(store(...)) \\
\longrightarrow^M \quad & S_{if}(store(...(r_p \; \mathbf{true}) \, (qr_t \; r_t) \, (qr_f \; r_f)...)) \\
\longrightarrow^P \quad & S_{if}(store(...)) \\
& p_r = packet(reference, S_{2t}, S1, S_1; r_t);
\end{aligned}
$$

Finally, the `apply` service uses reference substitution for the same reasons. Consider the expression:

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

If `apply` would bind the evaluated arguments to the λ-variables, all results would be copied to the `apply` service's local store and would have to be requested from there. This excessive back-and-forth copying of potentially large amounts of data is avoided by substituting the λ-variables for code references instead:

$$
\begin{aligned}
& p_{apply} = packet(task, \mathbf{control}, S1, r_1; e_{apply}); \\
& e_{apply} = \langle \mathbf{apply} \, r_\lambda \, qr_{S3} \rangle; \; r_\lambda \rightarrow e_\lambda; \; qr_{S3} \rightarrow r_{S3} \\
& e_\lambda = \langle \mathbf{lambda} \, qx \, qr_{S2x} \rangle \\
& r_{S2x} \Rightarrow \langle S2 \; x \rangle \\
& S_{apply}(store(...)) \\
\longrightarrow^M \quad & S_{apply}(store(...(r_\lambda \; e_\lambda)... \, (qr_{S3} \; r_{S3})...)) \\
\longrightarrow^P \quad & S_{apply}(store(...)) \\
& p_c = packet(code, S2, j, r_w; e_w); \\
& e_w = e_\lambda[x/r_{S3}] = \langle S2 \; r_{S3} \rangle \\
& p_r = packet(reference, S2, S1, r_1; r_w)
\end{aligned}
$$

In a similar fashion any other control service can use the mechanism of deferred evaluation and result redirection to achieve separation of data flow from control flow. Consequently, the Gannet architecture provides fully task-level configurable data paths without incurring the performance bottleneck resulting from repeated transfers of large amounts data to and from a central memory.

VI. Conclusion

The Gannet project researches a novel *service-based* architecture for very large reconfigurable Systems-on-Chip. The proposed architecture results in a packet-based distributed processing system that is reconfigurable at task level.

In this paper we have presented a formal semantics for control and data flow in the Gannet service-based SoC architecture.This small-step reduction semantics formalises operation of the machine in terms of packet transfers between services, actions taken by service manager and actions taken by service core. We have applied the semantics to demonstrate how the Gannet system can achieve separation between data flows and control flows, thus avoiding the bottleneck presented by conventional in-path control mechanisms such as memory transfers. The properties of the Gannet system make it ideally for run-time reconfiguration of SoCs with multiple concurrent high-throughput datapaths.

Acknowledgements

The author acknowledges the support of the UK Engineering and Physical Science Research Council (EPSRC Advanced Research Fellowship).

References

[1] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a domain specific language to a platform fpga," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, (New York, NY, USA), pp. 924–927, ACM Press, 2004.

[2] L. Lavagno, S. Dey, and R. Gupta, "Specification, modeling and design tools for system-on-chip," in *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, (Washington, DC, USA), p. 21, IEEE Computer Society, 2002.

[3] W. Vanderbauwhede, "Gannet: a functional task description language for a service-based SoC architecture," in *Proc. 7th Symposium on Trends in Functional Programming (TFP06)*, Apr. 2006.

[4] W. Vanderbauwhede, "The Gannet Service-based SoC: A Service-level Reconfigurable Architecture," in *Proceedings of 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2006)*, (Istanbul,Turkey), pp. 255–261, June 2006.

[5] P. Hofstee and M. Day, "Hardware and software architectures for the cell processor," in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 1–1, ACM Press, 2005.

[6] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *IEEE Computer magazine*, vol. 35, pp. 70–78, Jan. 2002.

[7] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *DAC '01: Proceedings of the 38th conference on Design automation*, (New York, NY, USA), pp. 667–672, ACM Press, 2001.

[8] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proceedings of the Design Automation Conference*, (Las Vegas, NV, USA), pp. 684–689, June 2001.

[9] H.-J. Stolberg, M. Berekovic, S. Moch, L. Friebe, M. Kulaczewski, S. Flugel, H. Kluszmann, A. Dehnhardt, and P. Pirsch, "Hibrid-soc: A multi-core soc architecture for multimedia signal processing," *The Journal of VLSI Signal Processing*, vol. 41, pp. 9–20, August 2005.

[10] B. Wilkinson, *Computer architecture: design and performance*, ch. 10, pp. 434–437. Prentice-Hall, 2nd ed., 1996.

[11] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, no. 4, pp. 365–396, 1986.

[12] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[13] G. J. Sussman and J. Guy L. Steele, "An interpreter for extended lambda calculus," tech. rep., Cambridge, MA, USA, 1975.

[14] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theor. Comput. Sci.*, vol. 103, pp. 235–271, September 1992.

[15] J. Matthews and R. B. Findler, "An operational semantics for r5rs scheme," in *2005 Workshop on Scheme and Functional Programming*, Sept. 2005.