Gannet: a Scheme for Task-level Reconfiguration of Service-based Systems-on-Chip

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 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 potential to completely separate data flows from control flows. In this paper we present the Gannet architecture and explain how Scheme can be used to describe task-level configuration of a Gannet SoC. The paper introduces the background for the work, presents the Gannet machine language and the compile process and explains how the Gannet SoC executes task description programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; C.1.4 [*Processor Architectures*]: Parallel Architectures

General Terms Distributed System-on-Chip architecture, Task-level reconfiguration

Keywords Service-based System-on-Chip, Network-on-Chip

1. Aim

The aim of this paper is to explain how Scheme can be used as a language for task-level reconfiguration of service-based Systems-on-Chip (SoCs).

Because the field of reconfigurable, heterogeneous multi-core Systems-on-Chip is very specialised and does not overlap much with the field of functional programming languages, the paper provides the necessary background about SoC architectures. The notions of task-level reconfiguration and service-based Systems-on-Chip are introduced and applied to the Gannet service-based SoC architecture. After setting the scene, the paper explains the design flow for the Gannet SoC and discusses why Scheme is a suitable language for the purpose of task-level configuration.

2. Background

2.1 The need for high abstraction level SoC design

As integration density of integrated circuits increases following Moore's law, it becomes increasingly clear that the current tools and methodologies for the design and verification of very large scale (VLSI) integrated circuits are not suited to exploit the full potential offered by the technology (productivity gap). Today's technology allows entire systems (e.g. a microprocessor with memory and peripherals, but more importantly systems with large number of data processing cores) to be integrated on a single chip, where only a few years ago such systems would have consisted of individual, packaged chips assembled onto a printed circuit board. For that reason, there is a growing demand for solutions allowing to design complex Systems-on-Chip (SoC) at high abstraction levels (Kulkarni et al. 2004; Lavagno et al. 2002).

This is even more the case for run-time reconfigurable systems. There are already a number of relatively high abstraction level tools for FPGA-based, statically reconfigurable systems (where the configuration does not change at run time). Most of these are based on C dialects with some support for concurrency (Handel-C, ImpulseC, SystemC) or on domain-specific languages (Kulkarni et al. 2004). Run-time reconfigurable hardware currently uses fine-grained reconfiguration mechanisms, using concepts such as context switching, differential bit file updates and reconfigurable instruction set processors (Hauck et al. 2004; Lam et al. 2006; Scalera and Vazquez 1998). However, none of these approaches qualifies as high abstraction level. The low granularity at which configuration takes place is not well suited for very large systems consisting of large numbers of heterogeneous processing cores.

2.2 Networks on Chip

The main issues with very large SoCs are connectivity (because the number of logic blocks on a SoC and the number of connections per block is very large) and design complexity (because the complexity of every individual logic block in a SoC is similar to the complexity of a single chip in a conventional system) (Sgroi et al. 2001). Traditional bus-style interconnects are no longer a viable option: synchronisation of hundreds of processing cores over large physical distances (on a SoC, the order of magnitude is 10^{-3}m or even 10^{-2}m where on a conventional IC it would be rather 10^{-4}m) is impossible; fixed point-to-point connections result in huge wire overheads. There is a consensus in the field that packet-switched Networks-on-Chip (NoCs) (Dally and Towles 2001) provide the best solution to this interconnect bottleneck because they offer flexible connectivity and an efficient mechanism for managing wires (Benini and De Micheli 2002; Grecu and Jones 2005).

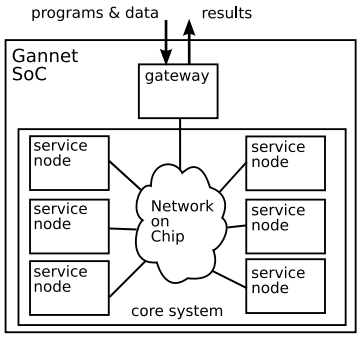

Figure 1. Gannet architecture

2.3 Design reuse

For very large SoCs, design reuse is essential (Stolberg et al. 2005). Design reuse is facilitated by the concept of what is generally called "Intellectual Property" (IP) cores. These 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.

2.4 Task-level reconfiguration and the service paradigm

Consider for example a handheld application, e.g. a mobile phone: the functionality of this type of device has increased so dramatically in recent times that a handset can perform a large number of different tasks (camera, video, SMS, web browsing,...). To make optimal use of the hardware resources, and in particular to reduce power consumption, the system should be reconfigurable at run time. We have proposed (Vanderbauwhede 2006a,b) a high abstraction level run-time reconfiguration mechanism called task-level reconfiguration. This mechanism organises the hardware blocks in the system as services, to be called on demand. Because of their self-contained nature, treating IP blocks as services is a natural abstraction. The interaction between the services is governed by a task description program. The framework of this so-called Service-based System-on-Chip architecture effectively turns the SoC into a distributed processor which executes task description programs.

It is important to note that the system does not require a von Neumann-style microprocessor (with a program counter, registers and a shared memory) to execute the program. Instead, the program is executed through the collective action of the Gannet framework and the IP cores contained therein.

3. The Gannet service-based architecture

The Gannet service-based architecture (Vanderbauwhede 2006a,b) is a *task-level reconfigurable system* that consists of a – potentially large – number of IP blocks connected via a Network-on-Chip (Fig. 1) . Every IP block is incorporated in a *service node*. All service nodes are connected over the NoC. Data and task configuration programs enter the system via a dedicated *gateway* circuit.

In practice, the System-on-Chip will have a tile-based layout (Fig. 2). Every service node forms a tile (a rectangular area of the chip) which is connected to the Network-on-Chip via a local switch.

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. 3).

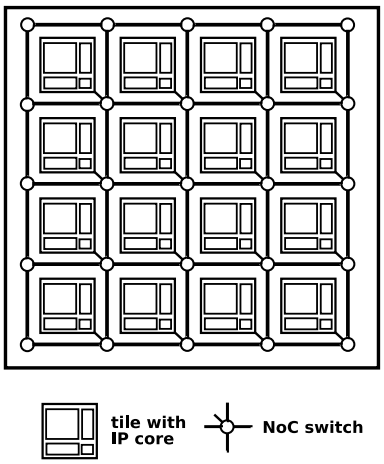

Figure 2. SoC with on-chip network

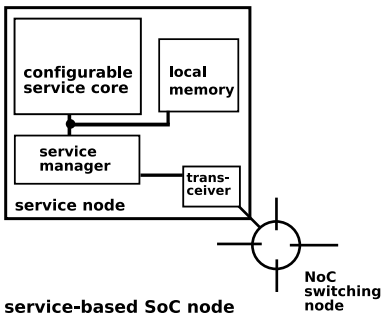

Figure 3. Gannet SoC tile with Service Manager

3.1 Functional task description language

Obviously, the functionality in such a system is not solely determined by the functionality of each core. The way the cores interact with each other is equally important. Thanks to the service manager circuit, the data flow between the services – and consequently the task performed by the system – can be described in a functional way, using a Scheme-like language.

A simple example of a task (which we will use further on) would be a system that takes images with 2 cameras and creates a 3-D image based from the data:

```
(create-3D (camera1) (camera2))
```

Assuming that the system has the required IP cores that map to these services, this example can actually be compiled and executed on a Gannet SoC, as detailed in Subsection 6.3.

The key role of the service manager is to marshall the data required by the IP core as well as the results produced by the core. In this way the service core is task-agnostic, i.e. it has no knowledge of the overall task the system is performing. A service core simply performs a computation on a set of input data and returns a result. Each service manager, governed by its corresponding part of the task description program, determines where to request the data and where to direct the result. The service manager does not modify the data in any way, it only directs the data flows.

3.2 Design Flow

Before we proceed to discuss the requirements imposed on the task description language by the Gannet system's architecture, we present the high-level design flow for a Gannet SoC (Fig. 4).

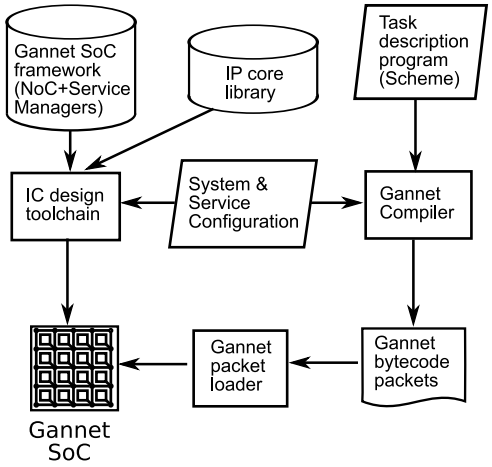

Figure 4. Gannet SoC Design Flow

The design flow has a hardware and a software component. Both are governed by a common description of the system and its services. This description contains a list of the services required by the system, the IP cores that will provide these services and the required configuration options for each core.

The hardware flow consists essentially of taking the IP cores from a library, configuring them if required and placing them in the Gannet fabric. For most of the cores the IP core library provides a configurable layout template and a dedicated program that will generate the final layout from this template.

The software flow essentially compiles a task description program to a set of packets containing the bytecode to be executed by the Gannet system.

3.3 Packet-based data transfer

Because the Gannet service-based architecture uses a Network-on-Chip for all communication between services, all data transfers are packet-based. In this section we discuss the Gannet packet structure.

3.3.1 Gannet packet

The unit of data transfer in the Gannet SBA is the packet, denoted as

$$\textit{Packet} ::= p(\textit{Packet-type,To,Return,Label;Payload})$$

The set of Gannet packet types is given by

$$\textit{Packet-type} ::= \textit{code} \mid \textit{reference} \mid \textit{data}$$

Depending on *Packet-type*, the *Payload* can be an *instruction* (code), a *code reference* (ref) or *data* (data). *To* and *Return* are the addresses of the destination and return services. *Label* is a Gannet symbol whose purpose will be discussed in Subsection 5.

3.3.2 Gannet symbol

An instruction is a flat list (further denoted with $\langle \ldots \rangle$) of *symbols*. A Gannet symbol is a structured byteword of a fixed number of bytes. The simplified[1] structure of a Gannet symbol is:

$$\textit{Symbol} ::= [\textit{Kind}:\textit{Name}:\textit{Subtask}]$$

Every symbol has a property called *Kind*. The service manager decides which action to take to process the task description based on a set of rules related to the symbol kind.

The set of kinds and the meaning of its elements will be explained in Subsection 5.

The content of the *Name* and *Subtask* fields depends on the kind. In general, *Name* is a symbolic name for a service, reference, variable or argument; *Subtask* provides information about the instruction to which the symbol belongs.

4. The Gannet language

The Gannet language is the equivalent of an assembly 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. The Gannet machine is defined as a set of service cores connected to a Network-on-Chip via a service manager. With this definition, the Gannet machine is a *custom instruction set coarse-grained distributed dataflow machine*. Every service core provides one or more "instructions". In the Gannet language this is represented as a function call.

Gannet syntax is an s-expression syntax completely free from syntactic sugar. In BNF, a Gannet expression must always obey

gannet-expression ::= (service-token argument-expression+)
argument-expression ::= (gannet-expression | literal)

where *service-token* represents a particular service. Every service in the system has a corresponding *service-token* in the program. There are no other keywords in the language, i.e. all flow control constructs are provided by services. The use of control services to provide "language" features is discussed in detail in Section 7; however, for a better understanding we will first discuss the compilation process and the execution model. An operational semantics for the Gannet language is presented in (Vanderbauwhede 2006b).

5. Compiling Scheme into Gannet packets

This section presents the flow for compiling Scheme into Gannet machine code, but first provides a justification for the choice of Scheme as high-level task description language.

5.1 Why Scheme?

As explained above, Gannet is a functional assembly language. It is not meant as a language to write the actual task descriptions in. Because of Gannet's functional nature, a functional higher-level language is a natural choice for this purpose. As we will see in Subsection 5.2, Scheme is not only syntactically but also semantically very similar to Gannet, which makes it an obvious choice. Moreover, Scheme is already popular with CAD vendors and tool users: two of the leading companies, Synopsys and Cadence, make extensive use of Scheme in their tools. Synopsys has embedded the GNU Guile interpreter while Cadence has its own Scheme dialect called SKILL (Petrus 1993).

It should be stressed that it is not the aim of Gannet to implement the full R^{5}R Scheme. The purpose of Gannet is to configure SoCs at service level, and the high-level language must have sufficient features for this task. For example, while support for numbers is required for purposes of flow control (see Section 7), support for

[1] The complete definition is *Symbol*::=[*Kind*:*Res*:*Quoted*:*Ext*:*Task*:*Name*:*Subtask*]. *Res* is reserved for future use; *Task* is used to distinguish between the symbols of different task running simultaneously on the system. *Quoted* and *Ext* will be introduced in Subsection 7.1.

strings is not required, nor is any type of OS interaction, including I/O.

5.2 Compilation flow

The Gannet "machine" processes *packets*. Consequently, the task description program will be compiled into packets.

Compilation of Scheme into Gannet packets is straightforward and consists of the following steps:

1. Transform the Scheme code into a core subset

   The subset supported by the Gannet system consists of `lambda`, `let`, `let*`, `set!`, `if`, `list`, `cons`, `car`, `cdr`, `length`, `eval`, and literals (including quoted expressions), variables and calls.
2. Transform the code into Gannet syntax
3. Decompose the Gannet code into an abstract syntax tree (AST)
4. Flatten the AST through reference substitution
5. Create a packet for each flattened node

Consider for example the following program:

```
(begin
(define (fact n acc)
  (if (< n 1)
    acc
    (fact (- n 1) (* acc n))))
(fact 5 1))
```

1. This will be transformed into

```
(let
((fact (lambda (n acc f) (if (< n 1) acc
    (f (- n 1) (* acc n) f)))))
(fact 5 1 fact)
)
```

2. Using the following rules:

   (a) $(let\ ((v\ \langle expr\rangle)+)\ \langle body\rangle)$
   $\Rightarrow (let\ (assign\ 'v\ \langle\ expr\rangle) +\ \langle\ body\rangle)$

   (b) $(lambda\ (arg+)\ (\langle body\rangle))$
   $\Rightarrow (lambda\ 'arg+'(\langle body\rangle))$

   (c) $(\langle\ lambda\text{-}def\rangle\ arg+)$
   $\Rightarrow (apply\ \langle\ lambda\text{-}def\rangle\ 'arg+)$

   (d) $(if\ \langle\ cond\rangle\ \langle\ expr1\rangle\ \langle\ expr2\rangle)$
   $\Rightarrow (if\langle\ cond\rangle\ '\langle\ expr1\rangle\ '\langle\ expr2\rangle)$

   the expression can be transformed into Gannet syntax:

```
(let
  (assign 'fact
    (lambda 'n 'acc 'f '(if (< n 1) 'acc
      '(apply f (- n 1) (* acc n) 'f))))
  (apply fact 5 1 'fact)
)
```

3. This expression is parsed into an AST which (slightly simplified) looks like this:

```
[E:([S:let]
  [E:([S:assign] [QV:fact]
    [E:([S:lambda] [QA:n] [QA:acc] [QA:f]
      [QE:([S:if] ([S:<] [A:n] [QL:1]) [QA:acc]
        [QE:([S:apply] [A:f] [E:([S:-] [A:n] [QL:1])
                              [E:([S:*] [A:acc] [A:n])] [QA:f])])])
    [E:([S:apply] [V:fact] [QL:5] [QL:1] [QV:fact])])]
```

   E denotes an expression; the other uppercase letterse denote the symbol kind of the token which they prefix. The set of kinds is:

   $Kind ::= S \mid R \mid L \mid V \mid A \mid QS \mid QR \mid QC \mid QV \mid QA$

   S for service, R for references, L for literals, V for let-variables, A for function arguments. Q denotes a quoted entity. As in Scheme, quoting turns expressions into literals. The Gannet quoting mechanism is explained in Subsection 7.1.

4. By recursively substituting references for expressions:

   $$[E:([S:x]arg+)] \Rightarrow [R:x:i]$$

   the AST is flattened into a lookup table with the reference symbol as the key and the expression as the value :

[R:let:0]:	[E:([S:let] [R:assign:1] [R:apply:2])]
[R:assign:1]:	[E:([S:assign] [QV:fact] [R:lambda:3])]
[R:apply:2]:	[E:([S:apply] [V:fact] [QC:5] [QC:1] [QV:fact])]
[R:lambda:3]:	[E:([S:lambda] [QA:n] [QA:acc] [QA:f] [QR:if:4])]
[R:if:4]:	[E:([S:if] [R:<:5] [QA:acc] [QR:apply:6])]
[R:<:5]:	[E:([S:<] [A:n] [QC:1])]
[R:apply:6]:	[E:([S:apply] [A:f] [R:-:7] [R:*:8] [QA:f])]
[R:-:7]:	[E:([S:-] [A:n] [QC:1])]
[R:*:8]:	[E:([S:*] [A:acc] [A:n])]

5. This table is transformed into a list of packets using the simple rule:

   $$[R:x:n] : ([S:dest]...)$$
   $$\Rightarrow p(code, dest, GW, [R:x:n]; \langle [S:dest]...\rangle)$$

   Here *GW* indicates the address of the gateway node, i.e. a special NoC node which acts as the gateway to the outside world (see Section 3). The change from $(...)$ to $\langle ...\rangle$ is purely for readability.

6. Code execution on a Gannet SoC

As discussed in Subsection 5, a Gannet task description program consists of a set of *code* packets (packets which contain an instruction) which enter the system via a gateway circuit. This circuit allocates memory for the result and passes the packets on to the NoC, which delivers them to their destination services where they are stored until they are activated for execution. The gateway then activates the root task (top node of the computational tree) by sending a *reference* packet (a packet with a reference to the code for this task) to the corresponding service. The service manager of this service delegates all subtasks to the corresponding services (again by sending *reference* packets). In turn, the requested tasks repeat the process until the lowest-level tasks (whose arguments are not references) are reached. The results are propagated up the tree until the final result returns to the gateway. The mechanism is schematically depicted in Fig. 5, with the "cloud" denoting all services in the system. In practice, the system will run a large number of tasks in parallel. The actual number depends on the amount of local memory allocated at every service for storing task packets. In the next sections we will discuss this mechanism in detail.

6.1 The service manager architecture

The Gannet architecture is aimed at System-on-Chip designs with a large number of heterogeneous IP cores, few of which – if any – will be actual microprocessors. To manage the flow of data

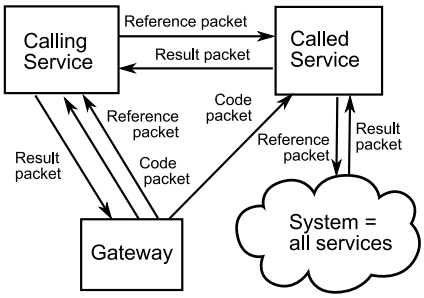

Figure 5. Gannet program execution

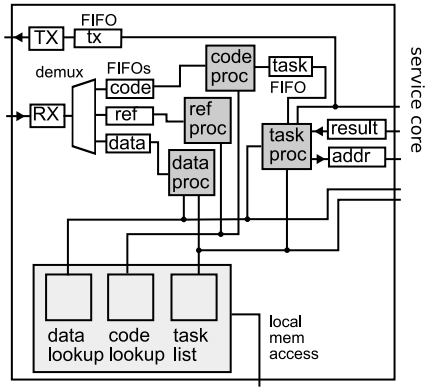

Figure 6. Service manager schematic

and instructions between the IP cores that provide the services, every IP core interfaces with the system through a dedicated logic circuit called the *service manager*. Its design philosophy is based on simplicity and minimal action: the service manager must be small, fast and resource efficient. A simplified design of the service manager is schematically represented in Fig. 6.

The circuit processes Gannet packets using a pipelined architecture. An input demultiplexer directs the incoming packets into FIFOs per packet type. Packet processing is event-driven; all FIFOs are processed in parallel. The figure shows the processing flow for the main packet types (*code*, *ref* and *data*). Code packets are stored in a code store. The payload of data packets is stored in the data store. Reference packets result in activation of corresponding tasks from the code store. Processing of the instructions that are the payload of the *task* packets (activated code packets) results in creation and dispatch of reference packets to other services. The next section presents a more detailed discussion of the service manager functionality.

6.2 Rule-based instruction processing

For the purpose of this paper, we can use a more abstract model of the service manager as a system that interfaces between the network and the service core:

- On the service core side, the service manager will notify the service core when all data required for processing are present and will receive a notification from the service core when the processing is finished and the result is available.
- On the network side, it can receive and transmit Gannet packets.

Based on the type of packet, a particular action will be taken. The smallest set of packet types and corresponding actions is:

- *code* packet ⇒ store packet
- *reference* packet⇒ activate corresponding code packet
- *data* packet⇒ store packet payload

Apart from the actions triggered by arrival of packets, there are 2 other actions which correspond to the interaction with the service core:

- all required data present ⇒ notify service core
- service core ready ⇒take result (data) from the service core and transmit over the NoC

The only non-trivial action is taken on receipt of a reference packet. This is a packet which contains the local memory address of a piece of code to be executed by the service manager. As described above, a Gannet program is compiled into packets the payload of which consists of bytecode representing a node in the AST. On activation of a code packet, the Service Manager parses the bytecode in a linear single-pass way, taking actions based on the kind of symbol.

In general terms, the semantics of a Gannet service (which consists of a service manager and a service core) can be described in terms of the *task code*, the internal *state* of the service and the *result packet* produced by the task as follows:

1. Initially, a service S_i receives a *code* packet $p(\textit{code}, S_i, S_j, r_{task}; \langle s_i\ a_1 ... a_n \rangle)$. The task is stored and referenced by r_{task}.
2. Later, the service S_i in *state*$_i$ receives a task *reference* packet $p(\textit{ref}, S_i, S_j, r_{id}; r_{task})$
   (Note that "later" is not essential: if the reference arrives earlier, activation will occur as soon as the code arrives.)
3. The service activates the task referenced by r_{task}: $\langle s_i\ a_1 ... a_n \rangle$. This results in evaluation of the arguments $a_1 ... a_n$.
4. The service, now in *state*$_i$ $'$, produces a result packet $p(\textit{Type}_i, S_j, S_i, r_{id}; \textit{Payload}_i)$ 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.
5. This packet is sent to S_j where *Payload*$_i$ is stored in a location referenced by r_{id}.

Note that the payload can be either data or an expression. While in general data processing services (typically provided by IP cores) will return data, control services can return data, bytecode or references to code.

6.3 Gannet code execution: an example

Returning to the example from Section 3, a system that takes images with 2 cameras and creates a 3-D image based from the data can be described as:

```
(create-3D (camera1 ) (camera2 ))
```

This will be compiled into

p_0: $p(\textit{ref,create-3D,GW}, r_1'; r_1)$

p_1: $p(\textit{code,create-3D,GW}, r_1; \langle s_1\ r_2\ r_3 \rangle)$

p_2: $p(\textit{code}, \textit{camera}_1, \textit{GW}, r_2; \langle s_2 \rangle)$

p_3: $p(\textit{code}, \textit{camera}_2, \textit{GW}, \textit{GW}, r_3; \langle s_3 \rangle)$

with

s_1: [S:create-3D]

s_2: [S:camera$_1$]

s_3: [S:camera$_2$]

r_1: [R:create-3D:memaddr$_1$]

r_2: [R:camera1:memaddr$_2$]

r_3: [R:camera2:memaddr$_3$]

The symbols r'_1,r'_2,r'_3 contain the memory location at which the result of the task should be stored.

The gateway circuit will take the packets and transmit them. The NoC will deliver them to the corresponding services.

In accordance with the above rules, the code packets will be stored and code packet $p1$ will be activated. Note that it doesn't matter if the reference packet p_0 would arrive before p_1 is stored: the action will simply be deferred by the service manager until p_1 is present.

Activation of p_1 results in parsing of the list of 3 bytewords $\langle s_1\, r_2\, r_3 \rangle$. The first word s_1 is a service symbol (S), used to select the service to be performed by the service core. This is because a single service core can provide multiple services. The next word is a code reference symbol: the value of the first field (R) indicates this. The second and third field indicate the service node address and the memory address where the code is stored. For every code reference symbol, the service manager dispatches a reference packet to the corresponding service. Thus for the given example service manager will dispatch

p_{r2}: $p(\textit{ref}, \textit{camera}_1, \textit{create-3D}, r'_2; r_2)$

p_{r3}: $p(\textit{ref}, \textit{camera}_2, \textit{create-3D}, r'_3; r_3)$

Arrival of these packets will trigger activation of resp p_2 and p_3

As both the camera tasks contained in p_2 and p_3 don't take any arguments, the service manager does not need to request or store any data. The service cores will simply acquire a picture and hand the resulting data over to the service manager. The service manager will transmit the data as a result packet to the caller, in this case create-3D:

p_{d2}: $p(\textit{data}, \textit{create-3D}, \textit{camera}_1; r'_2; \langle \textit{picture}_2 \rangle)$

p_{d3}: $p(\textit{data}, \textit{create-3D}, \textit{camera}_2; r'_3; \langle \textit{picture}_2 \rangle)$

Arrival of p_{d2} and p_{d3} will trigger the store data packet action.

When the service manager has marshalled all data required by the service core, it notifies the core. The service core then processes the data and produces a result. The resulting data are handed back to the service manager.

The service manager will transmit the data as a result packet to the caller, in this case, as this is the final computation, the gateway:

p_{d1}: $p(\textit{data}, \textit{GW}, \textit{create-3D}; r'_1; \langle \textit{3D-picture}_1 \rangle)$

The gateway stores the payload of p_{d1}at r'_1.

7. Control services: flow control and performance

In practice, the system requires the ability to control the data flow. For that purpose, a number of additional services must be added to the system. They provide familiar functional programming language constructs such as if, lambda, let. It can be easily demonstrated (Vanderbauwhede 2006a) that without control services the performance of the system would be sub-optimal. As a simple example, consider a service that takes two arguments and that needs call itself recursively n times:

```
(S (S (S ...) (S ...)) (S (S ...) (S ...)))
```

Without any additional constructs, this would result in $2^n - 1$ code packets. Furthermore, on execution, memory has to be allocated for every call. As all calls are effectively dispatched in parallel, the service would need to allocate $2^n - 1$ memory locations. By introducing functions, lexically scoped variables and the possibility of sequential evaluation, we can reduce the code size and memory requirements. Obviously, the use of recursive functions requires a conditional branching control (if) to exit the recursion. (Another obvious consequence is that we need support for numbers in the language. Using Church numerals to count e.g. a number of iterations would be very inefficient.) The above example might then become:

```
(letrec ((recS (lambda (m_ n_)
  (if (< n_ 1)
    (S m_ m_)
    (recS (S m_ m_) (- n_ 1))))))
(recS m n))
```

Furthermore, it is likely that some services will produce multiple results and that these results may be required by more than one service. For example, consider the inverse of the create-3D service, which would take a 3D image an return two 2D images, destined for say the left and right display of a 3D headset. It is clear that this can't be expressed as a pure functional dependency. But we could easily write

```
(left-eye (car (2Dfrom3D
   (3D-image (camera1) (camera2)))))
(right-eye (cadr (2Dfrom3D
   (3D-image (camera1) (camera2))))))
```

To summarize, the service-based architecture requires control constructs to improve its performance. The set of control constructs for the Gannet system is as follows:

- Functions: `lambda`, function application (`apply`)
- Conditional branching: `if`
- Lexical scoping: providing scope (`let`, `let*`), variable binding, variable calling
- Lists: `list`, `cons`, `car`, `cdr`, `length`
- Quoting and `eval`

Because all functionality in the service-based architecture is provided by services, all control constructs must be provided by control services. The consequences for the Gannet language and for the Service Manager architecture are discussed in the next section.

7.1 The Gannet quoting mechanism: deferred evaluation

The ruleset of the service manager results essentially in evaluation of all arguments of a function before they are passed on to the function body. The reason for this is that the IP cores providing the services will in general not be able to handle Gannet symbols, which is what the unevaluated arguments are. Consequently, a call to e.g. the `lambda` service (`(lambda x (S x))` would result in `(S x)` being called, which is obviously not the intention.

For that reason, we introduce a mechanism to defer evaluation of arguments. This *quoting* mechanism simply marks a symbol as a literal to be stored as data[2]. Thus, on encountering a quoted symbol (as indicated by the symbol kind), the service manager action will be to store the symbol. Numerical literals are encoded by the compiler as quoted symbols, so the service manager does not need specific extensions to support numbers.[3]

[2] A symbol is marked as quoted by setting the *Quoted* bit.

[3] A Gannet symbol can consist of several bytewords (*extended* symbol). This is indicated by the *Ext* bit. If *Ext* is 1, the *Name* field will contain the number of additional bytewords.

The Gannet quoting mechanism is very similar to Scheme's. The difference is mainly that in Gannet, only individual symbols can be quoted: quoting expressions results in a quoted reference symbol. However, the compiler handles this difference transparently. For example, consider the Gannet `eval` service: `(eval '(+ 2 3))` is compiled into ⟨[S:eval:1] [QR:+:1]⟩ and [R:+:1] refers to ⟨[S:+:1] [QL:2] [QL:3]⟩.

7.2 Functions

In a Gannet system, function application is performed by the `apply` service. Thus

```
((lambda (n) (S n)) expr) ; Scheme
```

becomes

```
(apply (lambda 'n '(S n)) 'expr) ; Gannet
```

The reason why `expr` is quoted is explained in Subsection 8.2.

7.3 Conditional branching

In many (even most) cases, it is not desirable to evaluate both branches of an if-statement before choosing one. Consequently, the Gannet `if` becomes

```
(if (Scond ...) '(S1 ...) '(S2 ...)) ; Gannet
```

7.4 Variables

Gannet requires a service for providing scope, variable binding and variable calling:

```
(let ((a (S1 ...)) (b (S2 ...))) (S3 a b)) ; Scheme
```

becomes

```
(let (assign 'a (S1 ...)) (assign 'b (S2 ...))
  (S3 (read 'a) (read 'b))) ; Gannet
```

This is an example of a service core providing multiple services, as clearly the memory for storing the variables must be shared by `let`, `assign` and `read`: a service core can only access its local memory.

For variable updates, this core also provides the `set!` service.

Gannet does not provide `letrec`, so recursive functions must be passed explicitly as arguments:

```
(let ((fact (lambda (n a f)
   (if (< n 1) a (f (- n 1) (* a n) f)))))
(fact 5 1 fact))
```

7.5 Lists

Gannet lists are different from Scheme lists in that lists in Gannet are not built from pairs. The only list constructor is the `list` service. Furthermore, in Gannet, a quoted list of expressions is not the same as a list of quoted expressions. This is a consequence of the fact that only individual symbols are quoted, not the actual expressions. Thus for example

```
'(1 (+ 2 3) (+ 4 5) 6 7) ; Scheme
```

will be translated by the compiler to

```
(list 1 '(+ 2 3) '(+ 4 5) 6 7) ; Gannet
```

The translation consists of

1. Replacing quotes by the `list` operator (in Scheme)
2. Quote all list arguments (in Gannet)

The list operations `cons`, `car`, `cdr` and `length` behave like Scheme's.

8. Gannet is not quite Scheme

There are a number of aspects in which Gannet differs from Scheme. The reason for all differences lies in the nature of Gannet as a task description language for a SoC services provided by hardware blocks, as opposed to the general-purpose nature of Scheme as a language intended to run on a von Neumann-style microprocessor system. In particular the distributed nature of the system, with effectively no global memory, and the separation of argument evaluation by the service manager and computation by the service core result in a different code execution model. Furthermore, the service manager is meant to be a small circuit with very limited memory. This puts additional constraints on the language.

8.1 Restriction on lists

For practical reasons, Gannet lists can only consist of symbols. Consequently, any expressions not returning a symbol should be quoted. This means that in practice

`(list (camera1) (camera2)) ; Scheme and Gannet`

is not allowed as both services return non-symbol data, but

`'((camera1) (camera2)) ; Scheme`

and

`'((+ 1 2) (+ 3 4)); Scheme`

are fine, as the latter are translated to

`(list '(camera1) '(camera2)) ; Gannet`

and

`(list '(+ 1 2) '(+ 3 4)) ; Gannet`

respectively.

The reason for this restriction is that IP cores can potentially generate large amounts of data. As the service manager of the list service will evaluate all arguments, the resulting list of data could be very large. Storing and transferring lists of values could therefore be very inefficient, costly (in terms of power consumption) and slow and should therefore be avoided.

8.2 `let` versus `lambda`

In Scheme, `let` is syntactic sugar for `lambda`. In Gannet, `let` and `lambda` have similar semantics but a different implementation. A single service provides `let`, `assign`, `set!` and `read`. This service actually binds variables to local memory locations. Consequently, `set!` can be used to update the content of a memory location.

Functions are provided by two different services: `lambda`, which essentially constructs a list, and `apply`. The `apply` service does not bind the lambda arguments to memory locations. Instead, it simply substitutes the `lambda` argument symbols with the `apply` argument symbols (which therefor need to be quoted).

The reason for this behaviour is again that transfering large amounts of data over large distances (in SoC terms) is costly (in terms of power consumption) and slow. Consider a trivial example: `(S1 (apply (lambda 'x '(S2 x)) (S3))`.

Binding would result in the data being actually, physically transfered over the NoC from `S3` to `apply`, and then from `apply` to `S2`. This would be the case for every call to `apply`, as there is only a single `apply` service. Clearly, this would result in a serious bottleneck. With the substitution semantics,

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

results in the task `(S1 (S2 (S3)))`. The data are sent straight from `S3` to `S2`, so `apply` is not a bottleneck.

8.3 Closures

Because memory utilisation in SoCs must be minimised, Gannet does not support closures that capture `let`-variables. The reason is that a captured `let`-variable would escape from the `let` block and it would in general not be possible to reclaim the memory for this variable. This restriction is again motivated by resource constraints: storing a full environment would consume a lot of

the service manager's limited memory. However, because lambda application works by substitution, closures with `lambda` instead of `let` are supported. For example:

```
(let (assign 'x 5) (lambda 'y '(+ x y))
```

is not supported: the memory for x would be de-allocated on leaving the `let`; however,

```
(apply (lambda 'x '(lambda 'y '(+ x y))) 5)
```

returns `(lambda 'y '(+ 5 y))`. If the value is a quoted expression, the code reference is substituted.

8.4 Concurrent evaluation

A key feature of the Gannet system is that all arguments of a function are effectively evaluated in parallel. Consider e.g.

```
(S1 (S2 (S3 (S4 1 2) (S5 3 4))
        (S6 (S7 5 6) (S8 7 8)))
    (S9 (S10 (S11 9 10) (S12 11 12))
        (S13 (S14 13 14) (S15 15 16))))
```

This (admittedly contrived) example will result in parallel execution of all 8 leaf calls, then of all 4 dependent calls, etc. Concurrent evaluation makes optimal use of the inherent parallelism of the SoC (all hardware cores operate always in parallel), resulting automatically in the fastest execution. However, it requires that the program is not sensitive to the evaluation order. This is actually a more stringent requirement than Scheme's serial-but-unspecified evaluation order, but parallel execution is the overriding rationale for the Gannet SoC architecture.

8.5 Forcing sequential evaluation

Although parallel evaluation is very efficient, it is sometimes undesirable. Parallel execution also means that memory will be allocated for all parallel branches of computation, leading potentially to very high memory consumption as discussed under Section 7. Another important consequence of parallel evaluation is that e.g. the following code produces an unpredictable result, as there is a race condition between the `read` and `set!` calls:

```
(let
  (assign 'a (S1 ...))
  (set! 'a (S2 ...))
  (read 'a)
)
```

Consequently, the capability to force sequential evaluation is essential. Therefore the `let` service allows serialisation of evaluation through quoting. All quoted arguments will be evaluated sequentially (and obviously after the unquoted arguments). Thus the above example becomes

```
(let
  (assign 'a (S1 ...))
  (set! 'a (S2 ...))
  '(read 'a)
)
```

As the update will be deferred until the assignment was successful, only the last argument needs to be quoted.

The quoted variant of the Gannet let syntax is also used to support Scheme's `let*`:

```
; Scheme
(let ((x 0))
  (let ((x 5) (y x)) y) ; => 0
  (let* ((x 5) (y x)) y) ; => 5
)
; Gannet
(let (assign 'x 0)
  '(let (assign 'x 5)
        (assign 'y x) y) ; => 0
  '(let '(assign 'x 5)
        '(assign 'y x) 'y) ; => 5
)
```

For the example given in 8.4, we can save memory by forcing sequential evaluation:

```
(let
'(assign 'a
  (let
    '(assign 'a
      (let '(assign 'a (S4 1 2))
           '(assign 'b (S5 3 4)) '(S3 a b)))
    '(assign 'b
      (let '(assign 'a (S7 5 6))
           '(assign 'b (S8 7 8)) '(S6 a b)))
    '(S2 a b)))
'(assign 'b
  (let
    '(assign 'a
      (let '(assign 'a (S11 9 10))
           '(assign 'b (S12 11 12)) '(S10 a b)))
    '(assign 'b
      (let '(assign 'a (S14 13 14))
           '(assign 'b (S15 15 16)) '(S13 a b)))
    '(S9 a b)))
'(S1 a b))
```

By combining lexical scoping and sequential evaluation, the memory consumption now grows linearly with the depth of the expression rather than exponentially.

9. Status

The work on the compiler for the Gannet language has focused on the back-end, i.e. compilation of Gannet syntax into binary packets. This is the only stage where non-trivial development is required: the first stage of the compilation (transforming Scheme to a subset) has been well-studied and there are plenty of Open Source implementations available; transforming the Scheme subset into Gannet syntax is quite straightforward.

The back-end compiler is written in Haskell. It produces host platform-independent bytecode which can be executed on any Gannet implementation. The current runtime for Gannet is written in Ruby (Thomas et al. 2004) and automatically translated to C++ with the option of using SystemC libraries. The main reason for the choice of Ruby is that it is a very clean language with excellent object support and Ruby objects map easily to hardware modules. On the other hand, by writing generic rather than idiomatic Ruby, translation to STL-based C++ (Austern 1998) is very efficient.

The purpose of the C++ version is to run as a Virtual Machine on embedded systems. The Gannet VM makes it possible to implement some of the services in software and others in hardware, and have a single task description program that governs the behaviour of such a combined software/hardware system.

SystemC (Ruf et al. 2001) is a system-level design and modelling library for C++. It provides a framework to simulate SoCs at high levels of abstraction. The SystemC version of Gannet is the closest approximation of the actual hardware SoC.

Although the Gannet architecture is aimed at very large application-specific Systems-on-Chip (e.g. for handheld applications), for practical reasons, a proof-of-concept of the architecture is being developed on an FPGA board. Because FPGAs provide reprogrammable hardware logic, they are ideally suited for development and prototyping. The circuit design of the service manager is currently being implemented, as are a set of core services. Because of resource constraints (the target board is the Xilinx University Program Virtex-II

Pro board), the number of cores will be small (maximum 8). The main purpose of the FPGA implementation is to help estimate the resource utilisation of the system.

10. 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 proposed the use of Scheme as a high-level task description language for the Gannet service-based Systems-on-Chip architecture. We have explained how the service-base SoC architecture can work as a coarse-grained distributed dataflow machine. We have introduced the packet-based machine code for the Gannet architecture, the compilation process and the code execution process.

The paper has also explained the need for control constructs to improve the system's performance and the main differences in runtime behaviour for Scheme control constructys when executed on a von Neumann-style microprocessor and on the Gannet system.

In conclusion, we have demonstrated that Scheme is ideally suited as a high abstraction-level design language for task-level reconfigurable heterogeneous multi-core Systems-on-Chip.

Acknowledgments

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

References

Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer magazine*, 35(1):70–78, January 2002.

W. J. Dally and B Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Las Vegas, NV, USA, June 2001.

Cristian Grecu and Michael Jones. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Trans. Comput.*, 54(8):1025–1040, 2005.

Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera reconfigurable functional unit. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(2):206–217, 2004.

Chidamber Kulkarni, Gordon Brebner, and Graham Schelle. Mapping a domain specific language to a platform FPGA. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 924–927, New York, NY, USA, 2004.

Siew-Kei Lam, Bharathi N. Krishnan, and Thambipillai Srikanthan. Efficient management of custom instructions for run-time reconfigurable instruction set processors. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 261–264, 2006.

Luciano Lavagno, Sujit Dey, and Rajesh 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*, page 21, Washington, DC, USA, 2002.

Edwin S. Petrus. SKILL: a Lisp based extension language. In *LUV '93: Proceedings of the third international conference on Lisp users and vendors*, pages 71–79, New York, NY, USA, 1993.

J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller. The simulation semantics of systemc. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 64–70, Piscataway, NJ, USA, 2001.

S. M. Scalera and J. R. Vazquez. The design and implementation of a context switching FPGA. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 78, Washington, DC, USA, 1998.

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*, pages 667–672, New York, NY, USA, 2001.

Hans-Joachim Stolberg, Mladen Berekovic, Soren Moch, Lars Friebe, Mark Kulaczewski, Sebastian Flugel, Heiko Kluszmann, Andreas Dehnhardt, and Peter Pirsch. HiBRID-SoC: A Multi-Core SoC Architecture for Multimedia Signal Processing. *The Journal of VLSI Signal Processing*, 41(1):9–20, August 2005.

Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, October 2004.

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)*, pages 255–261, Istanbul,Turkey, June 2006a.

W. Vanderbauwhede. Gannet: a functional task description language for a service-based SoC architecture. In *Proc. 7th Symposium on Trends in Functional Programming (TFP06)*, April 2006b.