Gannet: a service-based SoC architecture and task description language

Wim Vanderbauwhede

Overview

- What is the Service-Based Architecture (SBA)?
- How does it work?
- Gannet task descriptions
- The need for language constructs
- Conclusion

What is Gannet? (1)

A distributed System-on-Chip (SoC) architecture

- a collection of cores
- each core offers a a specific **service**
- all services are **fully connected** over an on-chip network (NoC)
- all information is transfered as **packets** over the NoC

What is Gannet? (2)

A language

- conceptually: an intermediate language, comparable to assembly language or the "intermediate representation" (IR) languages for virtual machines such as the JVM, the .NET CLR or Parrot.
- syntactically and semantically: a pure functional language, very similar to Scheme.
- the Gannet SoC architecture can be considered as a computing platform, a ”machine” to run the Gannet language.

Why Gannet?

- 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

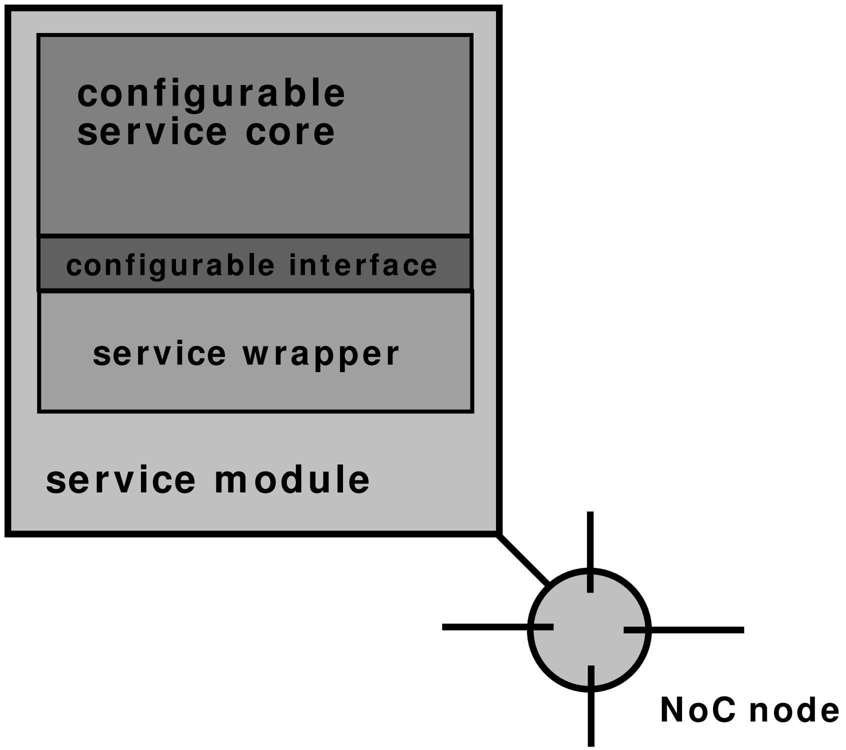
How does a Gannet SoC work?

The services collaborate in a data-driven manner:

- **data** enter the system
- to be processed by **services**
- the **results** of which are, like the data, processed by services
- this process evolves according to a predefined but configurable **task**
- the **description** of such a task is a Gannet **program**

Managing the services

- to manage the flow of **data** and **task descriptions** between the **heterogenous service cores**, every core interfaces with the system through a **service manager**



Gannet's service manager

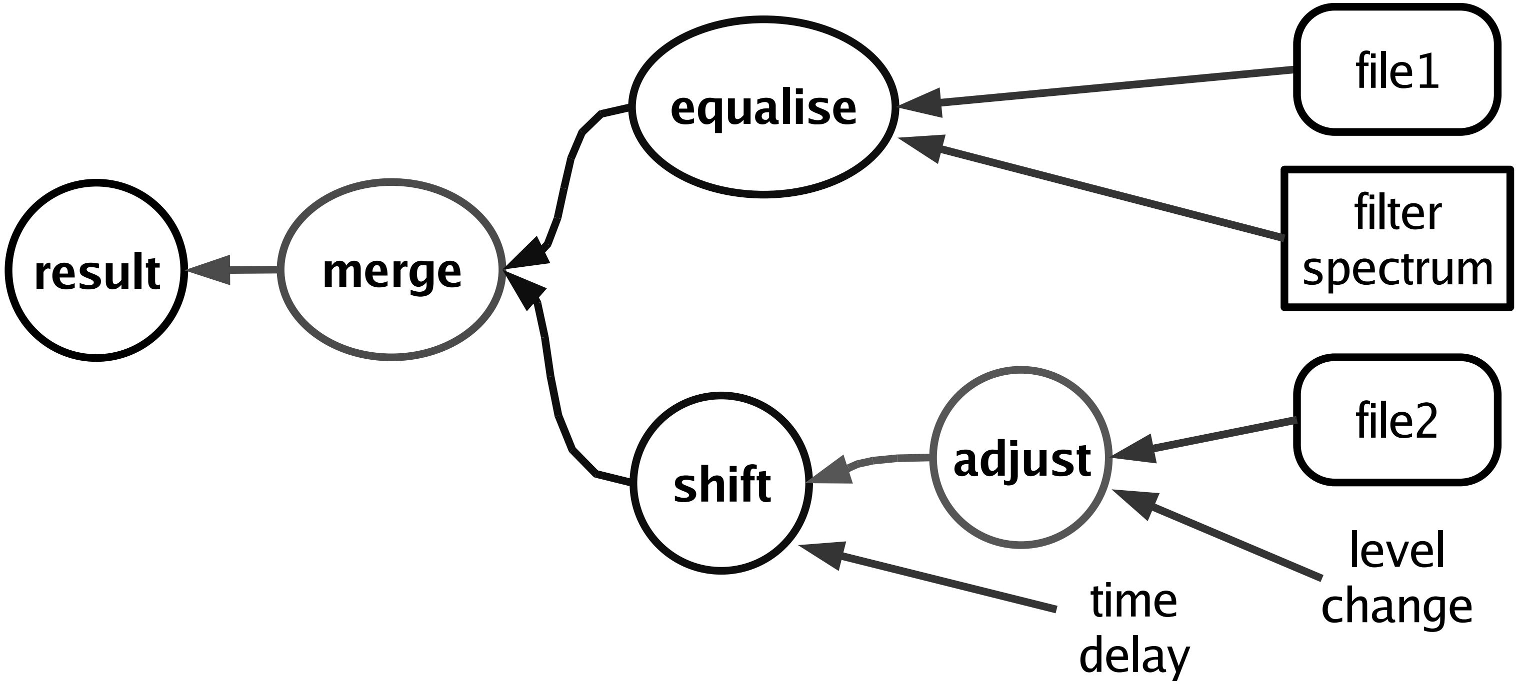
- the task description is a list of **symbols** representing either **data** or **services**
- essentially, the service manager uses two rules to evaluate the task description:
  - Data => request
  - Service => delegate
- it does the bookkeeping of all pending subtasks and the status of each of their arguments
- the service cores are task-agnostic

Gannet Task Description (1)

Example task: a system to process audio files.

- Suppose the system merges a number of audio files after having applied some processing to each file. E.g.:
  - Apply filtering with a given spectrum to the first file
  - Change the level of the second file and shift the wave a few seconds to synchronise with the first file
  - Merge both files

Gannet Task Description (2)



Gannet - Syntax

Task description format: S-expressions

- EBNF:
  - *expression* ::= (*service_symbol* *argument_symbol+*)
  - *argument_symbol* ::= *expression* | *data_symbol*

Example:

```
(merge
    (shift time_delay
        (adjust level_change file2))
    (equalise filter_spectrum file1))
```

Gannet - Symbols

Example:

```
S 9 merge
S 5 shift
D 1 time_delay
S 3 adjust
D 1 level_change
D 1 file2
S 3 equalise
D 1 filter_spectrum
D 1 file1
```

The need for a proper language

The Gannet task description approach

- allows in principle to describe arbitrary complex tasks
- but has severe limitations:
  - memory requirements
  - limited parallelism
  - no conditional branching
  - no loop constructs – program size

Solution: Adding language services

A short recap

What is it?

- Gannet is a system-level architecture for very large SoC's
- the Gannet language is used to define the functionality of the system
- so it is a task description language for a service-based SoC architecture

What is it not?

- Yet another programming language for a von Neumann architecture

A short recap

How does it work?

- A service consists of
  - a task-unaware service core
  - a generic service manager
- The **service manager decomposes task descriptions** using **2 rules**: request or delegate
- the service core performs **atomic operations**

A short recap

Why is it like that?

- IP cores are self-contained entities
- Service manager must be small and fast
- Task descriptions are decomposed by the distributed system, not in advance

Memory utilisation (1)

- The service manager allocates memory for the result of every delegated subtask and all requested data
- To calculate the memory requirements for a given task and service, we make the following assumptions:
  - the processing time for a task $\mathcal{T}$ is dominated by the service core
  - a service $\mathcal{S}$ has n_a arguments, of which n_S subtask and n_D data
  - a fraction α of all subtask requests calls $\mathcal{S}$ itself
  - the task $\mathcal{T}$ has a depth D, meaning that $\mathcal{T} = \mathcal{S}_D(\mathcal{S}_{D-1}(\ldots\mathcal{S}_i(\ldots(\mathcal{S}_1(d_1, \ldots, d_{n_a})\ldots)).$

Memory utilisation (2)

- Required memory for storing data ($\mathcal{M}_{D,d}$ for external data; $\mathcal{M}_{D,s}$ for subtask results):

$$\mathcal{M}_D = \mathcal{M}_{D,d} + \mathcal{M}_{D,s} =$$
$$\left[\sum_{i=1}^{D-1} n_D^i + (n_D + \alpha n_S)^D\right] + \left[\sum_{i=1}^{D-1} {n_S}^i + ((1-\alpha)n_S)^D\right]$$

- Worst case: $\alpha = 1$ and $n_a = n_S$

$$\mathcal{M}_D = \mathcal{M}_{D,d} + \mathcal{M}_{D,s} = n_a^D + \sum_{i=1}^{D-1} n_a^i = \sum_{i=1}^{D} n_a^i$$

Reducing memory utilisation

Introducing variables

```
(begin
    (var 'v1 (S d11 d12))
    (var 'v2 (S d21 d22))
    (S v1 v2))
```

- The `var` service binds the result of the service call to a variable
- The `begin` service is used for grouping expressions

The Gannet Quote

- the quote **'** indicates to the service manager that the following argument symbols must not be requested but **passed on as-is** to the service core
- is implemented as a separate symbol

  `(S1 'd1)`

  Symbols:

  `S 3 S1`

  `Q 2 '`

  `D 1 d1`

Immutable variables

```
(begin
    (var 'v1 (S d11 d12))
    (var 'v2 (S d21 d22))
        (var 'v12 (S v1 v2))
    (var 'v3 (S d31 d32))
    (var 'v4 (S d41 d42))
        (var 'v34 (S v3 v4))
    (S v12 v34))
```

This only halves memory utilisation: $\mathcal{M} = \sum_{i=1}^{D-1} n_a^i + n_a$

Reassignment (1)

```
(begin
    (var 'v1 (S d11 d12))
    (var 'v2 (S d21 d22))
        (var 'v12 (S v1 v2))
    (var 'v1 (S d31 d32))
    (var 'v2 (S d41 d42))
        (var 'v34 (S v1 v2))
    (S v12 v34))
```

This reduces the memory utilisation to: $\mathcal{M}_V = (D-1).n_a$

Reassignment (2)

- After an intermediate result has been calculated, the $n_a$variables used as arguments for the service can be reassigned, for all stages of the task:

$$v_{i.n_a+j} = \mathcal{S}(v_{(i-1).n_a}, \ldots, v_{(i-1).n_a+k}, \ldots, v_{i.n_a}), 1 \leq i < D, 1 \leq j,k \leq n_a.$$

- But:
  - variables are no longer immutable
  - Re-assignment requires the calls to `var` to be **blocking** to avoid race conditions**.**

Lexical scoping

```
(let
    (assign 'v12 (let
        (assign 'v1 (S d11 d12))
        (assign 'v2 (S d21 d22))
        (S v1 v2))
    (assign 'v34 (let
        (assign 'v1 (S d31 d32))
        (assign 'v2 (S d41 d42))
        (S v1 v2))
    (S v12 v34))
```

Lexical scoping

- **`let`/`assign`** requires **blocking** on **assign** call:

```
(let
 (assign 'v1 (Sfast d1))
 (assign 'v2 (let
     (assign 'v1 (Sslow d2))
     (assign 'v2 (S2 v1))
      (S3 v1 v2))
 (S3 v1 v2))
```

- but the variables are still immutable

Parallelism

- Often, services will produce more than one result, and each result can be required by a different service.

Grouping: **`list`** service

```
(begin
    (var 'l1 (S1 d1 d2))
    (var 'l2 (list
        (S2 (car l1))
        (S3 (car (cdr l1)))
        (cons l1 d4))
    (S4 (car l1) (car l2)))
```

Parallelism

- Lists are not immutable
- Lists are passed by reference

Conditional branching

- Depending on the value of the result, it might be delegated to a different service

Conditional branching: **`if`** service

```
(if (S_test_condition d1) (S1 d1) (S2 d1))
(if (S_test_condition d1) '(S1 d1) '(S2 d1))
```

- Quoting causes **lazy** evaluation, but is optional
- Service testing for condition must return **true** or **false**
- Practically speaking, tests required the introduction of **numbers**

Program size - recursive functions

- Without iterative or recursive constructs, a Gannet task description could become very large

Recursive functions: **`lambda`** service

```
(begin
    (var 'f (lambda 'x 'y '(* x y))
    (apply f d1 d2)
)
```

Gannet Language Services

- Globals
  - (**`begin`** *expression+*)
  - (**`var`** **`'`** *data_symbol expression*)
- Lexicals
  - *assign_expression* ::= (**`assign`** **`'`** *data_symbol expression*)
  - (**`let`** *assign_expression+ expression*)
- Lists
  - *list_expression* ::= (**`list`** *expression+*)
  - (**`car|cdr`** *list_expression*)
  - (**`cons`** *list_expression expression*)

Gannet Language Services

- If
  - (**if** *cond_expression ' *expression ' *expression*)
  - *cond_expression ->* #t|#f
- Lambda
  - *lambda_expression* ::= (**lambda** *(' data_symbol)+ ' expression*)
  - (**apply** *lambda_expression expression+*)

Trade-offs

Memory/speed

- blocking causes **serialisation**

```
(begin
     (var 'v1 (S1 d1))
     (var 'v2 (S2 d2))
     (var 'v3 (S3 d3))
(S4 v1 v2 v3))
```

  - with blocking: $\tau_{S1} + \tau_{S2} + \tau_{S3}$
  - without blocking: $max(\tau_{S1}, \tau_{S2}, \tau_{S3})$
  - so **lower memory utilisation** results in **lower speed**

Trade-offs

- **lexicals** are **slower** than **globals**: determining scope takes time
- but globals are persistent for the time of the task, lexicals only within the scope
- especially relevant for **lists** and **functions**

Code size/space overhead

- more language services make Gannet more efficient
  - potentially smaller code size (total number of calls)
  - but takes up more SoC space
- smaller code size leads to higher throughput

Trade-offs

Programmer efficiency

- Gannet is an intermediate language
  - Must support enough language constructs to allow direct conversion from a higher-level language
  - Scheme R5RS is the obvious candidate, but maybe not the best one: system-level programmers might prefer C++, Java

Conclusion

- Gannet:
  - a **service-based** System-on-Chip architecture
  - a **task description** language
- The need for **language** services
  - Which ones do we need?
  - What do they look like?
- Trade-offs

More on Gannet

Not covered

- Types and Kinds, Built-ins
- Memory management
- Implementation
- Design flow

Symbol Kinds

- Symbols Kinds are use by the service manager to link a symbol to a specific language service:

Service	Kind	Used for
data	D	external data
var	V	globals
let/assign	L	lexicals
lambda	A	lambda arguments
built-in	B	built-ins
quote	Q	quoted expressions

Data Types

- for immutable variables and built-ins, introducting data types can result in important reduction of memory utilisation
- in that case, Gannet would need type support
- Gannet Symbols already have a datatype field, but it is currently only used for built-ins: int, bool, string, float
- How do we add type support to the Gannet language?