Gannet: a functional task description language for a service-based SoC architecture

Wim Vanderbauwhede

Department of Computing Science, University of Glasgow, UK

Abstract

There is a growing demand for solutions allowing to design complex 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 functionality of the system is defined by a functional task description language. The Gannet system performs tasks by executing task description programs. Thus Gannet not only provides a novel high abstraction-level SoC design concept, but effectively offers an alternative to the ubiquitous Von Neumann architecture. In this paper we present the Gannet architecture and language and introduce the concept of language services, which dramatically improve the performance.

1 THE GANNET SERVICE-BASED SOC ARCHITECTURE

There is a growing demand for solutions allowing to design complex systems-on-chip (SoC) at high abstraction levels [1, 2]. The Gannet project aims to address this need by proposing a novel system-on-chip architecture using a functional language paradigm. We propose a distributed service-based system-on-chip architecture which performs tasks through the interaction of services offered by IP cores. The tasks are described in a functional task description language. The physical Gannet fabric 1(a) consists of a matrix of processing cores (*tiles*) connected through an on-chip network. 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[3, 4]. Tomorrow's SoCs will be *very big* (billions of logic gates). The main issues with these very large SoCs are connectivity and design complexity [5]. 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 on-chip networks (NoCs)* [6] provide a solution because they offer flexible connectivity and an efficient mechanism for managing wires.

For very large SoCs, *design reuse* is essential [7]. Design reuse is facilitated by the concept of IP (Intellectual Property) cores. These are are highly complex, self-contained units offering a specific functionality, such as 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. Designing a Gannet SoC reduces to instantiating the IP cores in the Gannet fabric and creating a task description (a program) in the Gannet functional language.

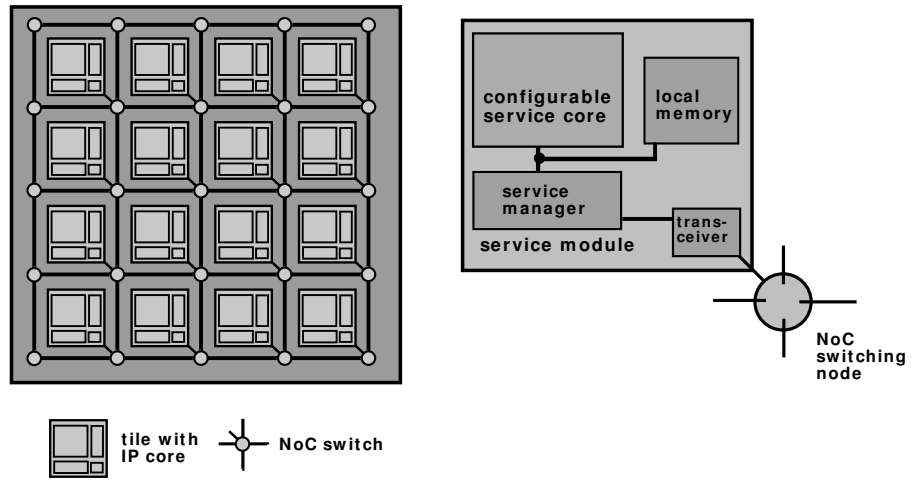

Figure 1. The Gannet Service-based System-on-Chip architecture

2 GANNET LANGUAGE CONCEPTS

The tasks performed by the Gannet system are defined in a functional task description language. A task description defines the interactions between the services by mapping every service to a function, and describing the flow of data between the functions in terms of function calls. Conceptually, the Gannet language (Gannet for short) is 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 [8]. Syntactically and semantically, it is a pure functional language, similar to LISP or Scheme [9, 10].

2.1 The Gannet machine

The Gannet SoC architecture can be viewed as a machine for running Gannet programs. However, Gannet is not a Von Neumann machine, but rather a dataflow-type distributed processing system [11, 12]. To manage the flow of *data* and *task descriptions* between the *heterogeneous IP cores* providing a *service*, every IP core interfaces with the system through a dedicated logic circuit called the *service manager*.

The service manager is a processor for compiled Gannet task descriptions. Its design philosophy is based on simplicity and minimal action: the service manager must be small, fast and resource efficient. The compiled task description is a flat list of *symbols*, which are essentially words of a fixed number of bytes. For convenience, we will refer to compiled Gannet as bytecode. Every symbol has a property called *kind*; the most important kinds are *service* and *data*. The service manager uses rules linked to the symbol kind to process the task description; the two main rules are: service requests are delegated to the corresponding service; data are requested.

Memory is allocated for the results of the service requests and for the requested data. The Gannet architecture does not have a global shared memory, as using a global memory on a very large SoC would be detrimental to the system's performance. Instead, the service manager uses local memory for storing the task descriptions and the data required by the core 1(b). The memory is fully managed by the service manager. The service core only interacts with the service manager to exchange data via the memory. Consequently, the service core is task-agnostic, and the service manager is unaware of the nature of the service.

The bytecode of a Gannet task description program enters the system via a gateway circuit. This circuit allocates memory for external data and passes the program text on to the top-level service. This service requests all required data and delegates all subtasks to the corresponding services. In their turn, the requested services repeat the process until the lowest-level services (whose arguments are only data) are reached. The results are then propagated up the tree to top-level service.

2.2 Gannet task description language

The Gannet task description language is purely functional. The semantics are extremely simple: every expression must be a named function; every named function represents a Gannet service. Function arguments can be data or function expressions. Data symbols represent data that enter the system from the outside (as opposed to results from a function call). The overriding reason for Gannet's semantics is to fully exploit the parallelism offered by the distributed processing architecture. All branches in the computational tree can be executed concurrently, so that the execution time for a service is determined by the execution time of the slowest argument, rather than by the sum of the execution times for all arguments (as would

be the case with sequential execution). Therefore, the Gannet language must be functional rather than imperative.

2.3 Gannet symbols

In the rest of the section, we will use a syntax based on s-expressions, because they provide a the simple and familiar mapping of the Gannet machine code to a human-readable format. A Gannet program consists of a single s-expression. The syntax (using EBNF, [13]) is extremely simple, as Gannet has no keywords:

service_expression ::= '(' *service_symbol argument_expression*+ ')'
argument_expression ::= *service_expression* | *data_symbol*

As a trivial example, consider the calculation of the first root of a quadratic equation $\frac{-b+\sqrt{b^2-4ac}}{2a}$:

```
(/
    (+
        (- 0 b)
        (sqrt
            (-
                (* b b)
                (* 4 a c)
        )))
    (* 2 a)
)
```

A representation of the bytecode for the example in a human-readable fashion is given in Table 2(a).

The program is a list of symbols; a symbol word as used in the bytecode consists of 5 fields:

Kind: The symbol kind indicates the function of the symbol. We use the term "kind" to avoid confusion with data types. Kinds are used by the service manager to determine which action to take for a symbol. The basic kinds are:

Data (D): Any type of data
Service (S): An SBA service
Number (N): Any number constant (see 2.4)

Task: As multiple tasks can be running concurrently in the system, symbols must be labelled with the task number (not shown in Table 2). This facilitates task management as required e.g. for service reconfiguration.

Name: This field represents the name of the function or variable, or the value of constants and numbers, as taken from the S-expression. In practice, the names will be remapped to reduce the required namespace, and values of constants and numbers will be stored as payload with the name serving as a pointer. The representation in Table 2 shows the names for readability.

Subtask: Every service keeps a list of subtasks that are being processed. The subtask field is required to distinguish between the results for different tasks sent by a common service (e.g. the addition in the quadratic equation example has two different subtraction subtasks and three different multiplication subtasks).

Count: The purpose of the Count field depends on the symbol kind. A task description for a subtask can contain multiple calls to a certain service. The Count field is used to distinguish between the results for different tasks sent by a single subtask (e.g. the multiplications in the discriminant in the quadratic equation example). However, in the original task description forwarded by the gateway, the Count field of a Service kind symbol is used to indicate the length of the sub-expression. This allows to remove the brackets from the S-expressions, and facilitates parsing by the service manager.

Kind	Name	Subtask	Count
S	/	1	16
S	+	1	12
S	-	1	3
N	0	0	1
D	b	0	1
S	sqrt	1	9
S	-	2	8
S	*	1	3
D	b	0	1
D	b	0	1
S	*	2	4
N	4	0	1
D	a	0	1
D	c	0	1
S	*	3	3
N	2	0	1
D	a	0	1

(a) Run-time task decomposition

Kind	Name	Subtask	Count
S	/	1	3
R	+	1	1
R	*	3	1
S	+	1	3
R	-	1	1
R	sqrt	1	1
S	-	1	3
N	0	0	1
D	b	0	1
S	sqrt	1	2
R	-	2	1
S	-	2	3
R	*	1	1
R	*	2	1
S	*	1	3
D	b	0	1
D	b	0	1
S	*	2	4
N	4	0	1
D	a	0	1
D	c	0	1
S	*	3	3
N	2	0	1
D	a	0	1

(b) Compile-time task decomposition

Figure 2. Representation of Gannet bytecode

In the current design, every symbol is 64 bits but for simpler systems it could be as little as 16 bits per symbol, depending on the number of allowed services (Name), service arguments (Count), tasks (Task) and subtasks (Subtask).

The next version of the Gannet compiler will decompose the task and transmit all subtasks separately. Consequently, the subtasks in the task description will be replaced by references (Kind R in Table 2(b)) which act as identifiers for the results. Although this increases the initial code size, it reduces the total transmitted code size, and increases the performance.

2.4 Extension of the Gannet language

The Gannet task description language as presented above allows in principle to describe arbitrary complex tasks, but has severe limitations:
- Recursive calls result in exponential memory utilisation.
- The code size is large due to the lack of loop constructs or subroutine calls.
- The potential for parallelism is limited as branches can not be joined.
- The lack of conditional branching limits the expressively of the language.

The above limitations will result in less than optimal system performance. However, these issues can be addressed by adding specialized services. These *language services* provide the Gannet system with familiar functional language constructs, although from the system's perspective they are ordinary services. To implement language services, a number of additional symbol kinds must be introduced and corresponding rules must be added to the service manager. It is the extension of the rule set which enhances the Gannet language's capabilities.

Aliasing

In general, many language services can be implemented as simple and small logic circuits. Because the overhead of the Gannet infrastructure (NoC switch, service manager) might be relatively large, it can be desirable to group a number of services together on a single service tile with a single service manager. For this purpose, the Gannet system supports an aliasing mechanism, comparable to the TCP/IP ports mechanism.

Quoting

First, we introduce the concept of quoting a Gannet expression. A quoted Gannet expression is represented by a new symbol kind. In EBNF:

data_symbol ::= *external_data_symbol* | *quoted_symbol*
quoted_symbol ::= *''expression*

The semantics of the quoted symbol are similar but not identical to Scheme's quoting mechanism. Essentially, quoting a function argument changes the calling convention from call-by-value to call-by-name. Quoted symbols are neither requested

nor delegated, but passed on to the service core as-is. To do so, the service manager requires a new rule, which is simply to store quoted symbols.

This simple addition to Gannet's semantics makes it possible for services to operate on Gannet expressions, thus paving the way for the introduction of language services.

Native datatypes

In general, Gannet services return untyped data. In principle, the service manager does not require knowledge about the nature of the data it handles. However, it is practical to introduce a limited set of native datatypes which are recognised as such by the service manager. We therefore introduce a new symbol kind *native* for numbers and booleans. In EBNF:

> *data_symbol* ::= *external_data_symbol* | *quoted_symbol* | *native_symbol*
> *native_symbol* ::= *number_symbol* | *boolean_symbol*
> *boolean_symbol* ::= '#t' | '#f'
> *number_symbol* ::= '**-**' ? *digit*+ ('**.**' *digit*+)?

The service manager must recognize the new symbol kind but requires no new rule. It simply stores native symbols, just like quoted symbols.

2.5 Language services

Thanks to the above extensions, the Gannet system can implement familiar functional language constructs as services. We introduce a minimal set of language services as an example and indicate how every language service helps to improve the system performance.

Variables and grouping

The service manager allocates memory for the result of every delegated subtask and all requested data. It is easy to show that for recursive calls this leads to exponential memory utilization: if a function representing a service $\mathcal{S}$ has n_a arguments, and D is the depth of the recursive call, then the worst-case required memory for storing data $\mathcal{M}_D$ is given by $\mathcal{M}_D = \sum_{i=1}^{D} n_a^i$. It is equally straightforward to show that the introduction of locally scoped variables reduces the worst-case memory utilisation to $\mathcal{M}_V = (D-1).n_a$.

We therefore introduce a variable binding service:

> *assign_expression* ::=' (' **assign** '' 'data_symbol *expression* ')'
> *grouping_expression* ::=' ('*grouping_service assign_expression*+ *expression* ')'
> *grouping_service* ::= **seq** | **conc**

The `assign` service binds the result of the service call to a variable. The grouping services give Gannet a block structure. Variables are local to the closest enclosing block and are visible to all nested blocks. The `seq` service imposes a sequential order on the evaluation; the `conc` service groups expressions without imposing any order of evaluation, effectively allowing parallel evaluation of the expressions. A consequence of allowing parallel evaluation is that Gannet's variables must be immutable.

It is important to understand the difference between the operation of the service manager and the service core. The service core implements the actual functional behaviour; the service manager's behaviour is independent of the actual function definition. Consider the assignment expression. The service manager of the `assign` service will store the quoted data symbol. Assuming the following expression is a function, it will delegate it to the corresponding service, allocate memory for the result and wait until the call returns. (Without the quote, the service manager would try to request the data represented by the data symbol.)

When the result of the service request (the RHS of the assignment) returns, the quoted data symbol and the return value are passed on to the `assign` service core. The core tries to bind the data symbol to the result and returns #t or #f to the service manager. The service manager then returns this result to the caller (i.e. the sequence service). Similarly, the service manager of the sequence service will delegate the `assign` calls, and wait for the results.

The current prototype of the Gannet system uses dynamic scoping, and therefore in some cases requires sequential evaluation of the assignment statement (similar to `let*` in Scheme). The next version will use lexical scoping similar to the `letrec*` construct in Scheme R6RS. The sequential grouping construct will then strictly speaking become redundant, as sequencing would only serve to allow side effects.

Variable symbols are represented by a new symbol kind, but no new rule is needed, as variables represent data and hence use the same "request" rule as external data.

Lists

In general, an IP core of a service could return a list of results, and each result might be required by a different service. Therefore Gannet needs services to create and manipulate lists. The current prototype has a minimal set of list services, with names borrowed from Scheme and Haskell:

list_expression ::= '('**list** *expression*+ ')'
'('**head** *list_expression* ')'
'('**tail** *list_expression* ')'
'('**length** *list_expression* ')'
'('**concat** *list_expression*+ ')'

Gannet' s lists, as its variables, are immutable. There is no need for special list symbol kind, and the service manager has no special rules for lists.

Conditional branching

Depending on its value, the result of a service request might have to be delegated to a different service. To support conditional branching, we introduce the `if` service:

if_expression ::='('**`if`** *cond_expression* `''`?*expression* `''`?*expression* ')'

In the current implementation of the `if` service, quoting of non-data expressions causes lazy evaluation, but is optional.

Functions

Without the potential for iteration and function calls, a Gannet task description could become very large. As every symbol requires at least one cycle of the NoC clock for transmission, a large code size results in long latency. This is the main motivation for introducing lambda functions. Because all operations in Gannet require a named service, an explicit function application service must be introduced as well.

lambda_expression ::= '('**`lambda`** ('`''`*data_symbol*)+ `''`*expression* ')'
application_expression ::= '('**`apply`** *lambda_expression expression*+ ')'

No new rule is required for the service manager, as function arguments are always quoted.

Arithmetic and Logic

To support conditional evaluation and operations on native datatypes, it is useful to introduce a minimal set of arithmetic and logic operations.

ALU_service ::= *arithmetic_service*|*logic_service*
arithmetic_service::= '+' | '−' | '*' | '/' | '<' | '=' | '>' | '!='
logic_service::= 'and' | 'or' | 'not' | 'xor'

Although every operation is conceptually a separate service, in practice these services will be aliased to a single arithmetic and logic core.

3 GANNET OPERATIONAL SEMANTICS

In section 2, we introduced the Gannet language and the concept of language services in an intuitive way, presenting it as a simplified version of a Scheme-like language. In this section we aim to present a formal grammar and operational

semantics for the language as it currently stands. Due to page limitations, the description is limited to the key language constructs (block structure, variable assignment, function application). A full grammar and operational semantics will be presented elsewhere.

We will describe the grammar and operational semantics of the Gannet language in the context of the Gannet machine. In this context, a Gannet program is a list of symbols, as introduced in 2.3.

We use following conventions:

> Angled brackets $\langle ... \rangle$ denote the boundaries of a symbol list. This is a notational convenience: the Count attribute of the first symbol in a symbol list is used to indicate the length of a given symbol list.
> Capitalised words in plural denote sets: $Items$
> Lowercase words in singular denote elements of a set: $item \in Items$

3.1 Grammar

To build a grammar for Gannet, we first introduce the sets of Symbols and Expressions. We use EBNF [13] to formally define the non-primitive elements of a set. All sets are disjoint unless otherwise indicated.

Symbols and Symbol lists

The different sets of symbols contains symbols with a particular value for the Kind field

$$\begin{aligned} Symbols &= Service\text{-}symbols \\ &\cup\ Number\text{-}symbols \\ &\cup\ Quote\text{-}symbols \\ &\cup\ Variable\text{-}symbols \\ &\cup\ Argument\text{-}symbols \\ &\cup\ Error\text{-}symbols \end{aligned}$$

$$symbol\text{-}list\ ::= \langle (symbol \,|\, symbol\text{-}list)+ \rangle$$

Expressions

Expressions are structured symbol lists, i.e. the kind of symbol in a particular position is not arbitrary.

$$\begin{aligned} Expressions &\subset Symbol\text{-}lists \\ Expressions &= Evaluable\text{-}expressions \\ &\cup\ Quoted\text{-}expressions \end{aligned}$$

$$
\begin{aligned}
\textit{Evaluable-expressions} &= \textit{Service-expressions} \\
&\cup \textit{Number-symbols} \\
&\cup \textit{Variable-symbols} \\
&\cup \textit{Error-symbols}
\end{aligned}
$$

$$\textit{service-expression} ::= \langle \textit{service-symbol}\ \ \textit{expression}+ \rangle$$

$$
\begin{aligned}
\textit{quoted-expression} ::= &\langle \textit{quote-symbol}\ \ \textit{expression}+ \rangle \mid \\
&\langle \textit{quote-symbol}\ \ \textit{argument-symbol} \rangle
\end{aligned}
$$

$$\textit{gannet-program} ::= \textit{service-expression}$$

The above definitions define a Gannet program as a Service-expression, i.e. a list of symbols starting with a Service-symbol. Roughly speaking, the service symbol is and identifier for a named function and the additional expressions in the list are identifiers for the arguments.

Values

The set of Values returned by Gannet services is defined as $\textit{Values} = \textit{Numbers} \cup \textit{Expressions}$.

Language service grammar

For the language services introduced above, the grammar needs to be more specific. Names in bold represent the service symbol for a particular language service. Any expression not conforming to the grammar will result in a compile-time error.

$$
\begin{aligned}
\textit{group-expression} &::= \langle \textit{group-service assign-expression}+\ \ \textit{expression}+ \rangle \\
\textit{group-service} &::= \textbf{conc} \mid \textbf{seq} \\
\textit{assign-expression} &::= \langle \textbf{assign}\ \textit{quote-symbol variable-symbol}\ \ \textit{expression} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\textit{lambda-expression} &::= \langle \textbf{lambda}\ \ \langle \textit{quote-symbol argument-symbol} \rangle * \ \ \textit{quoted-expression} \rangle \\
\textit{apply} - \textit{expression} &::= \langle \textbf{apply}\ \ (\textit{lambda-expression} | \textit{variable-symbol})\ \ \textit{expression}+ \rangle
\end{aligned}
$$

3.2 Operational semantics

To express the operational semantics of a programming running on the Gannet machine, we use a context-sensitive reduction semantics as introduced by Felleisen [14] and used in [15, 16]. However, the Gannet machine has some fundamental differences to a Von Neumann machine, which requires some minor modifications to the notation.

Evaluation context

The context for evaluation must always be a service expression, unless the given expression is a service expression. Thus:

$$C \quad ::= \quad [\,] \,|\, \langle s ... C ... \rangle$$

Evaluation of an expression is independent of neighbouring or enclosing expressions. However, the evaluation of a service expression has two distinct, atomic stages:

- The *marshalling* stage: in this stage, the service manager's rule-based engine processes the symbol list as explained above. The result of this stage is that the symbol list is replaced by a value list.

- The *processing* stage: in this stage, the service core processes the value list and returns a value. However, the service core might also operate on the service's local memory, e.g. to achieve variable binding.

To make clear the distinction between the marshalling and processing stages, we will mark the arrows with M ($\underset{\rightarrow}{M}$) or P ($\underset{\rightarrow}{P}$).

Store

The Gannet machine doesn't have global memory. Rather, every service has its own local memory, with read-only access for the other services. This means that the **store**() concept must be contextualised. We will indicate the context of the store with a subscript.

Shorthand notation

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

$expression : e$	$service-symbol : s$
$service-expression : se$	$variable-symbol : v$
$quoted-expression : qe$	$argument-symbol : x$
$evaluable-expression : ee$	$number-symbol : n$
$value : w$	

error denotes the error-symbol (as expression-type value).

Non-language-service semantics – Gannet rules:

This section describes the operational semantics of a program running on the Gannet machine in the absence of language services, i.e. the semantics of the fundamental rules applied by the service manager.

Delegate Service expression This is the fundamental rule for the service-based architecture: service expressions are passed on to their corresponding service (as

identified by the service symbol's Name field). The called service returns a value to the caller service.

$$C[\langle s\ ...se_1...\rangle] \quad \underrightarrow{M} \quad C[s...w_1...]$$

Store Number Value Number symbols contain their numerical value in the Name field. In the marshalling stage, this value is extracted from the symbol.

$$C[\langle s\ ...n_1...\rangle] \quad \underrightarrow{M} \quad C[s...w...]\ \forall i : w_i \in Numbers$$

Language service semantics

As discussed above, the introduction of language services leads to a number of additional rules and symbols. We assume all expressions are correct according to the grammar (i.e no compile-time errors).

Store Quoted expression The value of a quoted expressions is the unevaluated expression without the quoting symbol.

$$C[\langle s\ ...qe_1...\rangle] \quad \underrightarrow{M} \quad C[s...e_1...]$$

Grouping and Variables The grouping and assignment services effectively operate as a let-construct. The **assign** construct performs the binding:

$$\begin{array}{lll} (\mathbf{store_{assign}}(...)\ C[\langle \mathbf{assign}\ qv\ e\rangle]) & \underrightarrow{M} & \\ (\mathbf{store_{assign}}(...)\ C[\langle \mathbf{assign}\ v\ w\rangle]) & \underrightarrow{P} & (\mathbf{store_{assign}}(...(v\ w)...)\ C[\#t]) \end{array}$$

If a variable was already bound, **assign** returns an error symbol:

$$\begin{array}{lll} (\mathbf{store_{assign}}(...(v\ w_1)...)\ C[\langle \mathbf{assign}\ qv\ e\rangle]) & \underrightarrow{M} & \\ (\mathbf{store_{assign}}(...(v\ w_1)...)\ C[\langle \mathbf{assign}\ v\ w\rangle]) & \underrightarrow{P} & (\mathbf{store_{assign}}(...(v\ w_1)...)\ C[\mathbf{error}]) \end{array}$$

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

$$\begin{array}{lll} (\mathbf{store_{assign}}(...)\ C[\langle \mathbf{group}\ ...\langle \mathbf{assign}\ v_i\ e_i\rangle ...\ e\rangle]) & \underrightarrow{M} & \\ (\mathbf{store_{assign}}(...(v_i\ w_i)...)\ C[\langle \mathbf{group}\ ...\#t...\ w\rangle]) & \underrightarrow{P} & (\mathbf{store_{assign}}(...)\ C[w]) \\ (\mathbf{store_{assign}}(...(v_i\ w_i)...)\ C[\langle \mathbf{group}\ ...\mathbf{error}..\ w\rangle]) & \underrightarrow{P} & (\mathbf{store_{assign}}(...)\ C[\mathbf{error}]) \end{array}$$

Request Variable Value rule The **assign** construct requires a new symbol kind, the variable-symbol. Values bound to variables are requested using following rule:

$$(\textbf{store}_{\textbf{assign}}(\ldots(v_1 w_1)\ldots)\ C[\langle s\ \ldots v_1 \ldots\rangle]) \quad \underrightarrow{M} \quad (\textbf{store}_{\textbf{assign}}(\ldots(v_1 w)\ldots)\ C[s\ldots w\ldots])$$

$$(\textbf{store}_{\textbf{assign}}((v_{i\neq 1} w)\ldots)\ C[\langle s\ \ldots v_1 \ldots\rangle]) \quad \underrightarrow{M} \quad (s\textbf{store}_{\textbf{assign}}(\ldots(v_{i\neq 1} w)\ldots)\ C[s\ldots\textbf{error}\ldots])$$

This rule effectively performs variable substitution ($e[v_i/w_i] \rightarrow w$).

At a higher level of abstraction, one could summarise the above as:

$$(\textbf{store}_{\textbf{assign}}(\ldots)\ C[\langle \textbf{group}\ \ldots\langle \textbf{assign}\ v_i\ e_i\rangle \ldots\ e\rangle]) \quad \rightarrow$$

$$(\textbf{store}_{\textbf{assign}}(\ldots(v_i\ w_i)\ldots)\ C[e[v_i/w_i]]) \quad \rightarrow \quad (\textbf{store}_{\textbf{assign}}(\ldots)\ C[w])$$

Function definition and application This is the operational semantics for lamba functions in Gannet.

$$C[\langle \textbf{lambda}\ qx_1\ldots qx_i\ldots qx_n\ qe_a\rangle] \quad \underrightarrow{M} \quad C[\langle \textbf{lambda}\ x_1\ldots x_i\ldots x_n\ e_a\rangle]$$

$$\underrightarrow{P} \quad C[\langle\ x_1\ldots x_i\ldots x_n e_a\rangle]$$

$$(\textbf{store}_{\textbf{apply}}(\ldots)\ C[\langle \textbf{apply}\ \langle \textbf{lambda}\ qx_1\ldots qx_i\ldots qx_n\ qe_a\rangle\ e_1\ldots e_i\ \ldots e_n\rangle]) \quad \underrightarrow{M}$$

$$(\textbf{store}_{\textbf{apply}}(\ldots)\ C[\langle \textbf{apply}\ \langle\ x_1\ldots x_i\ldots x_n\ e_a\rangle\ w_1\ldots w_i\ \ldots w_n\rangle]) \quad \underrightarrow{P}$$

$$(\textbf{store}_{\textbf{apply}}(\ldots(x_1\ w_1)\ldots(x_i\ w_i)\ldots(x_n\ \ w_n))\ C[e_a[\ x_i/\ w_i]) \quad \underrightarrow{P}$$

$$\begin{cases} (\textbf{store}_{\textbf{apply}}(\ldots)\ C[w_a]) & (if\ \#x = \#w) \\ (\textbf{store}_{\textbf{apply}}(\ldots)\ C[\textbf{error}]) & (if\ \#x \neq \#w) \end{cases}$$

Request Argument Value rule This rule is completely analogous to the rule for requesting values of Variable symbols, but the **lambda** arguments are bound on the **apply** store. This rule effectively performs argument substitution in the function body expression ($e_F[x_i/w_i] \rightarrow w_a$).

Again, at a higher level, one could summarise the above as:

$$(\textbf{store}_{\textbf{apply}}(\ldots)\ C[\langle \textbf{apply}\ \langle \textbf{lambda}\ qx_1\ldots qx_i\ldots qx_n\ qe_F\rangle\ e_1\ldots e_i\ \ldots e_F\rangle]) \quad \rightarrow$$

$$(\textbf{store}_{\textbf{apply}}(\ldots(x_1\ w_1)\ldots(x_i\ w_i)\ldots(x_n\ \ w_n))\ C[e_F[\ x_i/\ w_i]]) \quad \rightarrow$$

$$(\textbf{store}_{\textbf{apply}}(\ldots)\ C[\ w_F])$$

The aim of this section was to present an operational semantics as a mechanism to describe the execution of a program on the Gannet machine, not to present a complete operational semantics for the Gannet language. All language services introduced in Section 2.5 can be described in terms of the grammar and semantics introduced in this section.

4 DISCUSSION

The presented set of language services introduced in Section 2.5 is not exhaustive. As has already been stressed, Gannet is essentially a functional assembly language.

The aim is to compile a higher-level functional language to Gannet bytecode. An interesting candidate is the real-time design language Hume [17], because its design makes it well suited for concurrent hardware applications. Another obvious candidate is Scheme [18] because of its popularity and its structural similarities with Gannet. The set of language services will to some extent be dictated by the requirements of the target language. Ultimately, the addition of a given language service must result in better system performance. This translates to a smaller code size and/or a lower number of function calls, both of which lead to higher throughput. However, adding services results in a bigger SoC and a NoC with more nodes, which in turn lead to higher latency. Thus the choice of the set of language services is determined by a trade-off between throughput, latency and SoC area.

CONCLUSION

The Gannet project aims to facilitate high abstraction-level design of complex SoCs. Gannet combines a novel, *service-based* system-on-chip architecture with a functional *task description* language. From a computational point of view, Gannet is a distributed processing system. The processing units are *services* which consist of a service manager and a processing core. The *service manager* is a rule-based engine for processing compiled Gannet task description programs ('bytecode'). The paper explains the grammar and operational semantics of the language and introduces the concept of *language services*, auxiliary services which dramatically improve system performance. The proposed set of language services constitutes a specific but not unique way of expressing these capabilities. This flexibility is very attractive as the Gannet language is an "intermediate representation" language into which higher-level functional languages can be compiled. The composition of the set of language services can be optimized for the chosen higher-level language. In conclusion, Gannet introduces a novel high abstraction-level SoC design using a functional language paradigm.

Acknowledgement

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: ACM Press, 2004, pp. 924–927.

[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: IEEE Computer Society, 2002, p. 21.

[3] 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: ACM Press, 2005, pp. 1–1.

[4] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *IEEE Computer magazine*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[5] 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: ACM Press, 2001, pp. 667–672.

[6] 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, June 2001, pp. 684–689.

[7] 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, no. 1, pp. 9–20, August 2005.

[8] A. Randal, D. Sugalski, and L. Toetsch, *Perl 6 and Parrot Essentials, Second Edition*. O'Reilly Media, Inc., 2004.

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

[10] G. J. Sussman and J. Guy L. Steele, "An interpreter for extended lambda calculus," Cambridge, MA, USA, Tech. Rep., 1975.

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

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

[13] W3C. (2004, Feb.) Extensible markup language (xml) 1.0 (third edition)–w3c recommendation 04 february 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation

[14] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theor. Comput. Sci.*, vol. 103, no. 2, pp. 235–271, September 1992. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(92)90014-7

[15] J. Matthews and R. B. Findler, "An operational semantics for r5rs scheme," in *2005 Workshop on Scheme and Functional Programming*, Sept. 2005. [Online]. Available: http://repository.readscheme.org/ftp/papers/sw2005/matthews.pdf

[16] M. F. M. F. Jacob Matthews, Robert Bruce Findler, "A visual environment for developing context-sensitive term rewriting systems," in *International Conference on Rewriting Techniques and Applications (RTA2004)*, 2004. [Online]. Available: http://people.cs.uchicago.edu/ robby/pubs/papers/rta2004-mfff.pdf

[17] K. Hammond and G. Michaelson, "Hume: a domain-specific language for real-time embedded systems," in *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 37–56.

[18] R. Kelsey, W. Clinger, and J. R. (Editors), "Revised^5 report on the algorithmic language Scheme," *ACM SIGPLAN Notices*, vol. 33, no. 9, pp. 26–76, 1998.