Separation of Data flow and Control flow in Reconfigurable Multi-core SoCs using the Gannet Service-based Architecture

Wim Vanderbauwhede
Department of Computing Science
University of Glasgow,
G12 8QQ Glasgow, UK
wim@dcs.gla.ac.uk

Abstract

This paper presents a mechanism for the separation of control and data flow in NoC-based SoCs consisting of multiple heterogeneous reconfigurable IP cores. This mechanism enables full data path control by an embedded microcontroller whilst avoiding the potential communication bottleneck and without requiring centralised control over the NoC. In this work, we assume a generic SoC where data processing is performed by reconfigurable IP cores interacting through a NoC and control structures are implemented on a microcontroller.

The proposed mechanism employs a service-based SoC architecture (the Gannet architecture) where the control services are implemented on a Virtual Machine and IP cores acquire service behaviour through the use of a generic data marshalling and interfacing circuit.

The paper aims at demonstrating how concepts from functional programming language and dataflow machine research can be applied to address the challenges faced in the design of reconfigurable heterogeneous multi-core SoCs. The presented work is theoretical in nature but is explained by example without formal analysis and notations.

1. Systems-on-Chip with multiple reconfigurable cores

With the growth in scale and complexity of integrated circuits, SoCs consisting of multiple heterogeneous reconfigurable IP cores are emerging[6, 4]. To address the communication bottleneck caused by bus-based architectures, Networks-on-Chip have been proposed [1] and successfully demonstrated [2, 3, 8]. One of the often quoted advantages of using a NoC, apart from power and area savings, is that it results in the decoupling of computation from communication [12]. However, in this statement a third aspect has been omitted: for reconfigurable systems, the control aspect is crucial. This paper discusses the issues related to data path control in NoC-based reconfigurable multi-core SoCs and proposes an architecture and mechanism to achieve a clear separation between the data flows and the control flows. By addressing this issue, the proposed SoC architecture facilitates system-level reconfigurability and system-level distributed processing.

1.1. Service-level and task-level reconfigurability

The reconfigurable system in this work (Fig 1) consists of a – potentially large – number of reconfigurable cores connected via a a Network on Chip (although in principle any flexible addressable communication medium could be used). A "reconfigurable core" is defined as a complex block of which the functionality is at least partly determined by some configuration information. For this work it is sufficient to assume generic reconfigurable IP cores. 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.

This paper presents a methodology for describing the interaction between cores. The proposed methodology allows fully concurrent operation and separation of data streams and control signals. The adopted nomenclature for the system is borrowed from service-based software architecture, e.g. [14]. Each independent core offers a service to the system. The collective interaction of the services in the system governed by a particular configuration is called a task. Hence the system in this work is reconfigurable at service level and at task level. In this paper we focus on the issues related to task-level reconfigurability.

1.2. Task control

Modern Systems-on-Chip contain in general at least one microcontroller. The conventional way of controlling a col-
lection of hardware blocks using an embedded microcontroller would be to interact with every block via a mechanism such as memory-mapped IO for data exchange and the use of interrupts to signal hardware events to the microcontroller.

Assuming a NoC, the microcontroller would interact with a NoC transceiver and transfer data via NoC packets. The NoC transceiver would generate interrupts to signal to the microcontroller. Using a NoC results in efficient data transmission and considerably reduces the required number of interrupts. Apart from the performance benefits, there is no significant operational difference with a bus-based mechanism. If the functionality of the system is defined at design time, this approach works well, as there is no need to define the data paths at run time (the routes by which data is exchanged between the blocks are fixed). However, if the system is reconfigurable, both at core level and at task level, the data paths are not hardwired but need to be defined at run time. In this scenario, the interaction with the microcontroller is likely to present a bottleneck. To illustrate this, consider the example of a generic data processing application (Fig 2):

Assuming a subroutine NoC_TRX which implements the interface between the microcontroller and the NoC, the naive C code for the example might be:

```c
Data* NoC_TRX(CoreAddress&,...);
/* variable declarations omitted. */
data1=NoC_TRX(C1,IN1);
data2=NoC_TRX(C2,data1);
data5=NoC_TRX(C5,IN2);
data6=NoC_TRX(C6,data5);
data3=NoC_TRX(C3,data2,data6);
data7=NoC_TRX(C7,data6,data2);
data4=NoC_TRX(C4,data3,data7);
result=NoC_TRX(C8,data4);
```

The variadic function NoC_TRX writes data to and reads data from a register interface for the NoC using MMIO.

The interaction with the microcontroller effectively converts the parallel processing of the data streams into a time-multiplexed serialised process. Moreover, contention on the microcontroller bus can be very high as the data must be copied to memory. It is clear therefore that a mechanism is required which will allow the data to flow between the cores while being controlled by the microcontroller.
This issue has been recognised in the work done at IMEC [11, 8, 7]. In [11], the authors propose an operating system for heterogeneous reconfigurable SoCs (OS4RS); in [7, 8] they present details of the NoC architecture and design.

However, while the OS4RS supports direct communication between the cores, the problem of controlling the data flow for a given task has not been addressed. Instead, tasks have a static routing graph for their duration. For tasks such as in Fig 2, which do not interact with the microcontroller at run time, this is viable. However, suppose we modify the example to contain a control construct, e.g.

```c
Data* NoC_TRX(CoreAddress&,...);
/* variable declarations omitted */
data1=NoC_TRX(C1,IN1);
if (condition1) {
    Data* data2=NoC_TRX(C2,data1);
data5=NoC_TRX(C5,data2);
data6=NoC_TRX(C6,data2);
} else {
    Data* data2=NoC_TRX(C5,data1);
data5=NoC_TRX(C2,data2);
data6=NoC_TRX(C6,data2);
}
data3=NoC_TRX(C3,data5,data6);
data7=NoC_TRX(C7,data6,data5);
data4=NoC_TRX(C4,data3,data7);
result=NoC_TRX(C8,data4);
```

In this case it is not possible to create a static task graph which avoids the microcontroller, as the value for the condition is only known at run time. Nevertheless, it is clear that the `if...else` statement is a pure control construct, so there should be no need to route the data via the microcontroller.

The work presented in this paper can be seen as complementary to the work by IMEC: instead of focusing on the operating system which schedules the (static) tasks, our focus is on the control mechanisms within the task and we demonstrate how it is possible to separate the control information from the data.

2. The Gannet service-based architecture

We have proposed a general architecture for task-level reconfiguration of heterogeneous multi-core systems with large numbers of cores, called the Gannet architecture [15, 17, 16]. In this service-based SoC architecture (Fig 4) a generic circuit is added to interface between the core and the NoC. This circuit, the service manager, provides the core with data marshalling facilities and a simple rule-based mechanism for task processing and result dispatching.

The required information for task-level configuration (effectively an array of bytecodes) is loaded onto the service managers at boot time. The services then process and exchange data without requiring data transfers with a microcontroller. Indeed, for the example in Fig 2, the system would not even require a microcontroller as there is no runtime control.

2.1. Control services in Gannet

As presented in [17], it can be shown that the Gannet system (and in general any task-level reconfigurable system) will only function efficiently if a number of control services are added. These provide the system with familiar programming language constructs such as conditional branching, functions, blocks and variables. As argued in [17], the control services would be efficiently implemented on an embedded microcontroller.

However, as we have seen in Section 1.2, interleaving the services provided by the reconfigurable cores (the core services) with control services (e.g. conditional branching) effectively reintroduces the communication bottleneck due to data transfers to the microcontroller. Consider as another example a simple while loop:

```c
data2=NoC_TRX(C1,data1);
while(count>0) {
    data2=NoC_TRX(C2,data2);
count--;
}
data3=NoC_TRX(C3,data2);
```

Every iteration of the loop will result in data being passed back and forth between the microcontroller and the core service, which is clearly undesirable. It is imperative therefore that control services must be implemented in such a way that the data can flow directly between the core services.

2.2. The Gannet Virtual Machine

Ideally, the microcontroller would only exchange control information with the cores. Although it is technically not impossible to realise this objective using compiled code, it
would require on the one hand a language with functional characteristics (mainly the absence of side-effects but also undetermined execution order, laziness, concurrency) such as Haskell [5] or ML [10] – a language such as C is essentially sequential with fixed execution order and strict execution, which makes it a poor candidate for control of fully concurrent systems. But on the other hand for transparent interaction between the service manager and the microcontroller, the absolute memory addresses of the data structures must be accessible. Functional languages generally do not provide this low-level memory access as it could easily break the immutability of the variables and the requirement of no side effects. Furthermore, the program would effectively need to contain an interpreter which would create the bytecode for the service managers at run time.

For these reasons, our solution is to implement a Virtual Machine (VM) which interacts with the service managers. The VM is essentially a software implementation of the service managers, control service cores and a ‘virtual NoC’. It runs byte-compiled programs in the Gannet language as presented in [16]. Apart from addressing the issues discussed above, the VM has the additional advantage that it makes it very easy to initially implement functionality in software and the later move it to hardware. Moreover, the VM runtime is a small, portable C++ application and the Gannet bytecode is platform-independent.

2.3. Gannet system operation

The Gannet system consists of a set of services connected via a NoC, both either ‘virtual’ (in the VM) or physical. All communication is entirely packet-based. The simplified operation sequence is:

- initially, all service managers receive one or more code packets containing byte-compiled subtasks. Every subtask is essentially a list of symbols representing references to other subtasks.

- when a service managers receives a reference packet, it activates the corresponding subtask.

- execution of this subtask results in dispatch of reference packets to other subtasks (typically delivered by other services)

- leaf tasks (tasks without any references) are executed by the service core and the result packets are returned.

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

For a better understanding, we introduce a slightly more formalised description of the Gannet machine:

- The Gannet machine is a distributed computing system where every computational node consumes packets and produces packets and can store state information between transactions.

- A gannet packet consist of a header and a payload. The payload can either be data or instruction code (symbols). The header consists of following fields:
  - Packet type (code, reference, data)
  - Destination address
  - Return address
  - Packet identifier

We denote a Gannet packet as

\[ p(\text{Type},\text{To},\text{Ret},\text{Id};\text{Payload}) \]

Thus in the most general terms, the semantics of a Gannet service can be described in terms of the task code, the internal state and the result packet produced by the task as follows (see 2.4 for task notation):

1. initially, a service \( S_i \) receives a code packet

\[ p(\text{Code},S_i,S_j,R_{task};\text{task}) \]

where \( \text{task} = (S_i \ a_1...a_n) \) . The task is stored and referenced by \( R_{task} \).

2. later, the service \( S_i \) in \( \text{state}_{i} \) receives a task reference packet

\[ p(\text{Ref},S_i,S_j,R_{id};R_{task}) \]

3. the service activates the task referenced by \( R_{task} \): \( (S_i \ a_1...a_n) \). This results in evaluation of the arguments \( a_1...a_n \).

4. the service, now in \( \text{state}_{i}' \), produces a result packet

\[ p(\text{Type}_{i},S_j,S_j,R_{id};\text{Payload}) \]

where both \( \text{Payload} \), and the state change to \( \text{state}_{i}' \) 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 \( \text{Payload}_{i} \) 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 (typically provided by the VM) can return data, bytecode or references to code.

2.4. Gannet language fundamentals

The most important consequence of the Gannet system operation presented above is that the Gannet language, i.e. the language of the bytecode, must possess following properties:

- the evaluation order is unspecified
- lazy evaluation must be possible
- there should be no side effects across services
- updates of variables must be atomic (no race conditions)

The Gannet language is a machine code for the Gannet machine, i.e. it is a target for compilation rather than a language for designing applications. The listed language properties, which we will now cover in more detail, put it into the category of functional languages such as Scheme, Lisp, ML or Haskell [5, 9, 10, 13]. To be able to explain the language’s concepts, we introduce an intermediate language syntax (just like assembly language is an intermediate language syntax for machine code in an ordinary microprocessor). The syntax, known as S-expressions [9], is borrowed from Scheme [13] but the keywords and semantics are slightly different.

A call to a service $S$ can be written as $(S \, a_1 \, a_2 \ldots \, a_n)$. Thus the service is modelled as a function $S$ with arguments $a_1\ldots a_n$ (in C syntax this would be $S(a_1,\ldots,a_n)$). The arguments are symbolic representations (symbols) of expressions which will evaluate to the data to be passed on to the service core for execution. The number of arguments is variable and every argument can be an expression, so expressions can be nested arbitrarily. For example, the diagram in Fig 2 can be written as:

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

When the Gannet system executes this program, $S_8$ sends a task reference to $S_4$, which sends in its turn references to $S_3$ and $S_7$, etc. The leaf tasks ($S_1$ IN1) and ($S_5$ IN2) do not have any references, IN1 and IN2 represent data. The value for IN1 is passed to the $S_1$ core and the result of the processing is returned to $S_2$, etc. So in general a service will send task references to other services and receive the results of the task processing.

With this notation, we can clarify the requirements imposed on the Gannet language:

- unspecified execution order means that in a given function call it is not possible to predict in which order the arguments $a_1\ldots a_n$ will be evaluated. In practice, the result for the fastest chain of evaluations will return first. This might seem of little importance for an ordinary function call. However, every language construct in Gannet is a service and has the same structure and functional behaviour. For example, a grouping construct for a set of instructions (similar to a {...} block in C) is provided by the let service and expressed as:

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

The assign statements are local variable assignments (so corresponding C syntax for the first statement would be: `Data* a=S1(...);`).

Essentially all arguments will be evaluated in parallel. So e.g. $S_3$ and even $S_4$ could try to evaluate before $a$ has been assigned. For that reason the Gannet language must be single-assignment: multiple assignments to the same variable in a let block would lead to race conditions and undetermined results. Thus the following code is illegal:

```
{let
  (assign 'a 0)
  (assign 'a 1)
  a
}
```

A further consequence is that a service will not execute until all arguments have been evaluated (i.e. no partial execution).

- lazy evaluation means that it should be possible to evaluate arguments at need. By default, Gannet is eager, i.e. it always evaluates all arguments before passing them on to the service core. This is necessary as third-party cores cannot handle unevaluated symbols. However, the support for lazy evaluation is essential. In Gannet notation introduced above, laziness is expressed by prefixing an expression or symbol with a single quote. For example, the quote in front of the first argument of the assign statement (‘a) causes the evaluation of the symbol a to deferred, i.e. the symbol is passed as-is to the service core, which will evaluate it when required. If this would not be the case, the assign service would try to evaluate the symbol (e.g. a) and would fail simply because a has not yet been assigned a value. So (assign a 0) would never evaluate. The actual semantics of a quoted expression depends on the service core.
• **no side effects across services** means that a call to a given service should not result in a modification of the state of the rest of the system. If this were the case, concurrent evaluation would result in unpredictable behavior. The qualifier "across services" is essential as Gannet services are not purely functional since they can maintain state in between transactions.

e.g. the assign service expression

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

binds a value (resulting from evaluating (S1 ...)) to a symbol (a), thus changing the state of the service, and returns the symbol. In purely mathematical terms this is not a function, and the binding operation is a side effect. However, this side effect does not affect any other service in the system, thus preserving the potential for concurrency.

• **updates of variables must be atomic** means that there should be no race conditions if several services simultaneously try to modify shared data. If variables Gannet were truly immutable, concurrency would not be possible, as concurrency – in contrast to parallelism – by definition implies the sharing of common resources. To allow concurrency, updating variables must be possible. However, care must be taken to ensure that there are no race conditions possible during update. The Gannet service manager receives all task requests in a FIFO and always processes the current request completely before moving on to the next request in the queue, thus ensuring that update operations are always atomic. Furthermore, to have consistent single-assignment behaviour, updates are performed by a separate update service and it is not possible to update an unassigned variable.

### 3. Separation of control and data flows

As argued in Section 1.2, it is of crucial importance to be able to control the system in such a way that data are transferred directly between the hardware services rather than via the microcontroller and its associated memory. However, as the data paths are not fixed at design time (and for a reconfigurable system not even at compile time), it is equally important to send control signals from the microcontroller to the hardware blocks. In the Gannet architecture, the service manager enables any service to send both control and data packets to all other services. In practice, most (if not all) of the services that are used to control the system will be implemented on a single microcontroller, whereas the data processing services will typically be implemented in hardware. The reason for this division is mainly granularity and overhead: most control services operate at a very low granularity and consequently the overhead caused by implementing them hardware would be considerable. Typical data processing blocks will be large and complex, in which case the overhead caused by the service manager is small. Consequently, the problem reduces to ensuring that there is no unnecessary data transfer between control services and data processing services.

#### 3.1. Deferred evaluation and result redirection

The mechanism through which the separation of control and data flows is achieved is a combination of deferred evaluation and redirection. Deferred evaluation (quoting) has been introduced above; redirection means that the result of a call does not return to the caller but is redirected elsewhere, typically to the caller’s parent (Fig 5).

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

**Figure 5. Redirection mechanism**

Without redirection, the result of the evaluation of (S3 ...) would return to the control service Sctl, which would return it to S1. With redirection, the control service is bypassed. Redirection is implemented by setting the return address in the reference packet generated by the control service to the address of the caller (S1). However, without deferred evaluation this mechanism would not work because (S3 ...) would be evaluated during the task processing (due to the eager-by-default behaviour), causing the result to be returned automatically to the control service Sctl.

To illustrate how this mechanism allows effective separation between control and data flows we consider a number of examples. It should be noted that the mechanism is not limited to the presented constructs, indeed the choice of language constructs in Gannet is up to the system designer as long as they meet the requirements set out in Section 2.4. In the following diagrams, control services implemented by the VM (i.e. running on the microcontroller) are located above the dashed line, core services below it. For
the sake of clarity, only the signals crossing the boundary and the data paths have been labelled. All control services in the examples are called from a core service and in their turn call other core services.

3.2. Conditional branching construct

One of the essential programmatic control structures is the conditional branching construct, e.g. if or case.

```plaintext
(S1
  (if cond
    '(true ...)
    '(false ...))
)
```

Figure 6. Conditional branching construct

Fig 6 illustrates the operation of the if service. Based on the value of the first argument (false in the diagram), a task reference is sent to one of the services that make up the other arguments.

3.3. Function definition and application

A fundamental concept in any programming language, but in particular in functional languages, is the ability to create functions. Gannet uses the concept of lambda-functions, i.e. anonymous functions that can be bound to a variable or applied to an argument.

The lambda service creates the code for the function with variables for the arguments; the apply service substitutes the arguments (S4 ...) for the corresponding variable in the function definition, thus creating a task. Once the substitution is done, the code for the task is sent to the corresponding services and executed.

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

Figure 7. lambda function construct

3.4. Lists

Another fundamental concept in functional languages is that of lists, which are similar to linked lists in C. Although lists are very much a memory organisation concept, again it is possible to manipulate lists of services in such a way that the data don’t have to be stored in memory. The list in this example stores quoted expressions rather than values. This is somewhat similar to storing function pointers in C.

```plaintext
(let
  (assign 'l (list '(S2 ...) '(S3 ...)))
  (S4 (return (head l)) (S1 ...))
)
```

Figure 8. list constructs

The head service takes the first element of a list; the return service simply evaluates its argument and returns it. If the
argument is quoted, it uses redirection. In this example, $S_4$ does return data to the VM, as the call to $S_4$ is not quoted. Again, the diagram illustrates how quoting and redirection can be used to separate control and data flows.

The above examples demonstrate how it is possible to implement essential control structures in the Gannet VM in such a way that there is a clear separation between the control path and the data path.

4. Conclusion

In this paper, we have demonstrated how the Gannet service-based SoC architecture can address a number of crucial issues faced by task-level and service-level reconfigurable heterogeneous multi-core SoCs.

We have explained the issues relating to task-level reconfiguration, in particular the communication bottleneck created by the microcontroller responsible for configuring the data path at run time. We have introduced the Gannet architecture as a solution and explained the rationale for implementing control services as a Virtual Machine running on an embedded microcontroller rather than as native compiled code.

The most important contribution of this paper is the mechanism for the separation of control flow and data flow based on deferred evaluation and packet redirection. Using this mechanism, the Gannet system provides full control over the data paths in a fully concurrent heterogeneous multi-core SoC, ensuring that the data flows directly between the cores, thus alleviating the potential bottleneck caused by the microcontroller. We have demonstrated this mechanism using as example a number of core control constructs implemented in the Gannet language.

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

References