High-level Programming of Dynamically Reconfigurable NoC-based Heterogeneous Multicore SoCs

Wim Vanderbauwhede

School of Computing Science
University of Glasgow

Abstract

With the increase in System-on-Chip (SoC) complexity and CMOS technology capabilities, the SoC design community has recently observed a convergence of a number of critical trends, all of them aimed at addressing the design gap: the advent of heterogeneous multicore SoCs and Networks-on-Chip and the recognition of the need for design reuse through Intellectual Property (IP) cores, for dynamic reconfigurability and for high abstraction-level design.

In this chapter, we present a solution for High-level Programming of Dynamically Reconfigurable NoC-based Heterogeneous Multicore SoCs. Our solution, the Gannet framework, allows IP core-based Heterogeneous Multicore SoCs to be programmed using a high-level language whilst preserving the full potential for parallelism and dynamic reconfigurability inherent in such a system. The required hardware infrastructure is small and low-latency, thus adding full dynamic reconfiguration capabilities with a small overhead both in area and performance.

1 Introduction

Networks-on-Chip (NoCs) provide a scalable, efficient and performant communication medium to interconnect complex IP cores. To facilitate interoperation between IP cores, a number of standards have been proposed (e.g. VSIA, OCP/IP) [Kogel et al., 2005] regarding the interface between the IP cores and the communication medium. The purpose of such standards is to facilitate design reuse and as such they are aimed primarily at IP core developers. By their very nature, IP cores are agnostic of the system in which they are deployed. Consequently, the datapaths between the cores are not governed by the IP cores. Thus, a system consisting solely of IP cores and a communication medium is not dynamically reconfigurable. A Dynamically Reconfigurable SoC therefore requires an additional infrastructure to support reconfiguration of the cores and the datapaths. Moreover, to reduce design time, it is essential that a Dynamically Reconfigurable SoC can be programmed at high level.

In this chapter we analyse the requirements of such an infrastructure, in particular in view of allowing high-level programmability. We present our architecture of a dynamic reconfiguration infrastructure (DRI), the Gannet framework.

The chapter starts with the background to this work. We then analyse the requirements for NoC-based SoC infrastructures to support dynamic reconfiguration. The subsequent sections discuss the key components of the Gannet framework:

• The Gannet Dynamic Reconfiguration Infrastructure,

• The Gannet language used to program the system,

• The Gannet Machine Model, a formal model used to explain how the Gannet DRI executes Gannet programs.

Section “Examples of Dynamic Reconfiguration” presents examples of dynamic reconfiguration of communication (data path) and computation (IP core). In section “Implementation of the Dynamic Reconfiguration Infrastructure” we discuss the implementation of the Gannet DRI. The chapter concludes with an overview of the current status of the project and avenues for future research.

2 Background

Reconfigurable architecture platforms are gaining increasing popularity as flexible, low-cost solutions for a variety of media processing applications. They contribute to bridging the gap between general purpose processors and application specific circuits. As the application domain for reconfigurable platforms further expands, it becomes imperative that these systems provide a high degree of flexibility and adaptability, while at the same time being extremely cost-efficient. For many applications, fine-grained reconfigurable platforms such as FPGAs are not an acceptable choice because of their high area overhead and reduced performance compared to ASIC solutions. As a result, recent times have seen more focused research on coarse-grained reconfigurable architectures (CGRA), which constitute a middle ground between ASICs and FPGAs in terms of performance-versus-flexibility.

Using the terminology from the review paper on CGRAs by [ul Abdin and Svensson, 2008], the Gannet platform [Vanderbauwhede, 2008, Vanderbauwhede et al., 2008, Vanderbauwhede, 2007] is an Array of Functional Units – as opposed to Hybrid Architectures and Arrays of Processors. The former are typically using a combination of an ordinary processor with a reconfigurable fabric, e.g. [Singh et al., 2000, Mishra et al., 2006]) in the latter the reconfigurable fabric consists of fully-featured processors, e.g. [Butts, 2007, Taylor et al., 2004]).

Within its category, Gannet bears some similarity to MATRIX [Mirsky and DeHon, 1996], Silicon Hive [Cocco et al., 2004] and MORA [Lanuzza et al., 2007, Purohit et al., 2008], as it shares with these architectures the local-memory processing model, i.e. every functional unit has its own local memory. However, Gannet distinguishes itself from the other architectures by providing a generic interface layer to third-party processing elements where all other architectures discussed in [ul Abdin and Svensson, 2008] use either dedicated functional units or ordinary processors.

In contrast to most CGRAs, the Gannet platform was specifically designed as a *reconfiguration infrastructure* for SoCs using packet-switched NoCs. From that perspective, Gannet has similar goals as the work done at IMEC [Nollet et al., 2005, Nollet et al., 2004, Marescaux et al., 2004] on operating system control of NoC routing tables and support for run-time reconfiguration of functional units. The two approaches can be considered complementary as Gannet focuses on providing an interface layer which facilitates high-level programming of the interactions between the cores whereas the NoC OS focuses on operating system functionality such as process control and scheduling.

A further similarity is the recognition that it is essential to separate control flow from data flow to avoid the performance bottleneck caused by the processor bus in central-memory systems [Nollet et al., 2003]. There is however a fundamental architectural difference as Gannet does not require a centralised control system (e.g. a processor running an operating system).Furthermore, programming the routing tables is a low-level technique. Even when using a distributed programming model such as employed by the Æthereal researchers [Goossens et al., 2005], the programming model only addresses the NoC communication paths but not the actual functionality of the system, as it governs which slots are in use by which resource. In contrast, the Gannet platform can be considered as a distributed processor, which can be programmed using a high-level language. More precisely Gannet can be categorised as a coarse-grained demand-driven architecture.

Demand-driven (reduction) and data-driven (dataflow) parallel architectures have been extensively studied in the past [Veen, 1986, Treleaven et al., 1982], however, these were fine-grained architectures. The interest in these architectures decreased because they could not match the inexorable rise of the ever more performant von Neumann-style processors. It was generally recognised that communication presented the main bottleneck in fine-grained parallel architectures. However, with the advent of multicore SoCs and the adoption of Networks-on-Chip as an efficient communication paradigm for multicore systems, there is a renewed interest in coarse-grained architectures [Guerrier and Greiner, 2000, Lampinen et al., 2006].

In the specific case of the Gannet platform it is tempting to compare the proposed architecture with the numerous dataflow architectures for executing functional programs [Vegdahl, 1984, Amamiya and Taniguchi, 1990, Giraud-Carrier, 1994], as Gannet is a demand-driven parallel architecture and it adopts a functional paradigm for high abstraction-level programming. However, the crucial difference is that the aim of Gannet is not to provide an optimal architecture for executing general-purpose functional programs. On the contrary, the aim of our work is to facilitate high abstraction-level design of Dynamically Reconfigurable NoC-based Heterogeneous Multicore SoCs and for that purpose we have adopted concepts from dataflow architectures and functional programming. As a consequence, many concepts in Gannet will be familiar from this earlier work, in particular [Vegdahl, 1984] and [Ashcroft, 1986].

To contrast Gannet with other research on high level programming of dynamically reconfigurable systems, we can consider e.g. the work of [Najjar et al., 2003] using Single-assignment C (SAC) and the Lime project[Huang et al., 2008] using Java, in both cases to allow high-level programming of a host processor with attached FPGAs. While these approaches offer indeed high-level programming of dynamically reconfigurable systems, they are targeted at FPGA-based systems whereas Gannet targets NoC-based SoCs. To better illustrate the difference, imagine a NoC-based SoC where some of the IP cores are embedded FPGAs. To create the bit streams for those FPGAs one could use the SAC or Lime approach; however, the reconfiguration decision and management and the communication between the cores in the SoC would be handled by Gannet.

3 Requirements on SoC Infrastructures for Dynamic Reconfiguration

This section presents a number of key observations about Dynamically Reconfigurable NoC-based Heterogeneous Multicore SoCs and derives the requirements for a Dynamic Reconfiguration Infrastructure (DRI).

3.1 Characteristics of NoC-based Coarse-Grained Reconfigurable Architectures (CGRAs)

Assuming a NoC-based System-on-Chip consisting of a heterogeneous set of reconfigurable cores, we can make the following general observations:

• SoCs consisting of large numbers of Heterogeneous IP cores connected over a NoC are inherently parallel and can support large numbers of data flows between large numbers of cores. These data flows can be parallel (independent of each other) or concurrent (dependent on each other). Furthermore, it is in principle possible that a particular core will participate in multiple data paths. In general, data flows in a multicore SoC will not result in purely parallel operation but rather in concurrent operation, i.e. the flows can share resources.

• On the nature of the IP cores, we can observe that they are self-contained, by definition system-agnostic and are typically data processing units.

• Regarding communication, it is obvious that, because of the NoC, all data transfers are packet-based.

• In a dynamically reconfigurable system, it is essential that the data flows are controllable at run time. A system where the functionality of the cores can be reconfigured but the data path remains static would clearly be of limited use.

• In a NoC-based SoC with very large numbers of cores, centralised memory would present a huge performance bottleneck. Hence, IP cores will require local memory. Indeed, several CGRAs [ul Abdin and Svensson, 2008] and multicore processors [Pham and Aipperspach, 2006] have adopted local-memory architectures.

• Streaming data processing is essential in high-performance SoCs. This follows from the previous observation, as without central memory the system must operate in dataflow fashion. The nature of NoC-based SoCs as essentially a connected set of data processing units is optimally suited for this type of operation.

• High abstraction-level design of dynamically reconfigurable heterogeneous multicore SoCs is essential to reduce design time. This point is more generic and applies to any type of system design. Traditional low-level design approaches using HDLs are unable to cope with the complexity of today's SoCs. For Dynamically reconfigurable SoCs the reconfiguration of both data paths and core functionality must be expressed at a high level of abstraction and consequently the system architecture must enable high-level programmability.

3.2 Requirements on Dynamic Reconfiguration Infrastructures

Based on the above observations, we can derive the requirements for a Dynamic Reconfiguration Infrastructure (DRI) for NoC-based SoCs.

Since neither the cores nor the NoC provides control over the data flows between the cores in terms of deciding where to direct data generated by a core, the DRI must govern the data path. Indeed, the primary purpose of a DRI is to provide dynamic data path reconfiguration capabilities to the SoC. For that purpose, the DRI must provide run-time flow control operations. In doing so, the DRI must support parallel and concurrent operation of cores. As a central controller could present a bottleneck, the DRI should use distributed control as much as possible. Therefore, the DRI must support a local-memory model. For some types of control constructs, a centralised controller is the only option. In that case the DRI must ensure that the data flow is separated from the control flow.

Furthermore, considering the importance of streaming data processing for a very large class of applications combined with the observation that NoC-based SoCs are intrinsically ideally suited for streaming data processing, it is essential that the DRI should support this processing model. Finally, as the provider of dynamic reconfigurability, the DRI must support high-level programmability.

The DRI should also provide support for reconfiguration of the cores. However, as we will see further, this support follows naturally when all the above requirements have been taken into account.

4 The Gannet Dynamic Reconfiguration Infrastructure

In this section we propose a functional model for a Dynamic Reconfiguration Infrastructure (DRI) which we call Gannet. The Gannet DRI is a Service-based Architecture (SBA)[Vanderbauwhede, 2006b]. We introduce the Service abstraction and explain how the proposed model meets the DRI requirements set out above. We then present a high-level system overview of the Gannet DRI. Finally, we discuss the operation of a key component, the Gannet service manager.

4.1 The IP-Core-as-Service Abstraction

We start by observing that a NoC provides full connectivity between all its nodes. In principle, every node can exchange data with all other nodes in the system. We will call a node consisting of an IP core and a NoC interface a tile. From the above discussion it follows that the tile should also contain local memory. The aim of the Gannet DRI is to make the data exchanges between tiles programmable at high level. The choice of the programming paradigm is very important, as the paradigm will have to support all requirements of the DRI as identified in the previous section. The most popular programming paradigm, imperative programming (with C as the archetypical example) is a poor choice as it does not natively support parallelism or concurrency.

As IP cores are self-contained units with well-specified interfaces and functionality, we can regard them as "service providers", i.e. every core provides one or more services to the system. Typically, services are computational but flow control services including all types of storage are equally possible. We now introduce the service abstraction: a service is defined as a unit that consumes data, produces data and can modify its internal state. By "internal state" we mean all writeable memory elements on the service tile. With this abstraction, a service is very similar to a function. Consequently, we can adopt a functional approach to task graph composition.

Functional composition means that all data – apart from constant (hardwired) data (e.g. ROM content) – are the results of calls to other services. This idea is very similar to so-called functional languages such as Scheme, Haskell or ML [Sussman and Steele, 1975, Milner et al., 1990, Hudak et al., 1992b]: a program is a tree of nested functions. The key advantage of the functional paradigm is that it natively supports parallelism and concurrency and can easily support streaming data processing through pipelining.

By itself, an IP core will not act as a function as it has no control over the source of its data and the destination of the result of the computation. To achieve service-based (i.e. functional) behaviour, every tile of a Gannet SoC contains a special control unit (the *Gannet service manager*, SM), which provides a service-oriented interface between the IP core and the system.

Let us assume for the moment that a service can be modelled by a pure function, i.e. $y = S(x_1, x_2, ..., x_n)$. By "pure" we mean that the function has no side effects, i.e. it is a mathematical function. The service manager effectively performs evaluation of the function arguments. Consider following pseudo-code for a system with 3 services:

```
DCT( Block( Image(), blocksz ) )
```

The service `Block` takes a block of `blocksz*blocksz` pixels from `Image`. `blocksz` is a constant; `Image` is a service which obtains an image via IO, e.g. a camera. The service `DCT` performs Discrete Cosine Transform on the block of pixels provided by `Block`. All services consist of an IP core which performs the actual operation on the data and a Gannet service manager (SM). The operation starts at the `DCT` service: its SM requests data from `Block` and stores the constant `blocksz`. As soon as the SM receives a block of pixels from `Block`, it activates the DCT core. The core computes the transform. The SM will take care of sending the data to whichever service called the DCT. Similarly, the SM of the `Block` service requests data from the `Image` service; the SM of the `Image` service simply instructs the `Image` core to get an image and returns it to the caller (i.e. `Block`).

The HDL design and implementation of the SM are discussed in [Vanderbauwhede et al., 2008] and will be revisited in Section 8. In the following sections we present a high-level architecture of the DRI and the SM.

4.2 System Overview

Given a NoC-based System-on-Chip, the Gannet Dynamic Reconfiguration Infrastructure (DRI) consists of:

• The service managers acting as an interface between each IP core and the NoC;

• A gateway which acts as the interface through which the DRI is configured;

• A reconfiguration manager which acts as an interface to the service library, a reconfiguration database which contains configurations for the IP cores. The high-level view of a Gannet system is a collection of service tiles (IP core + SM + transceiver) connected via a NoC. This is illustrated in Figure 1; we call the combination of a service tile and a NoC switch a service node.

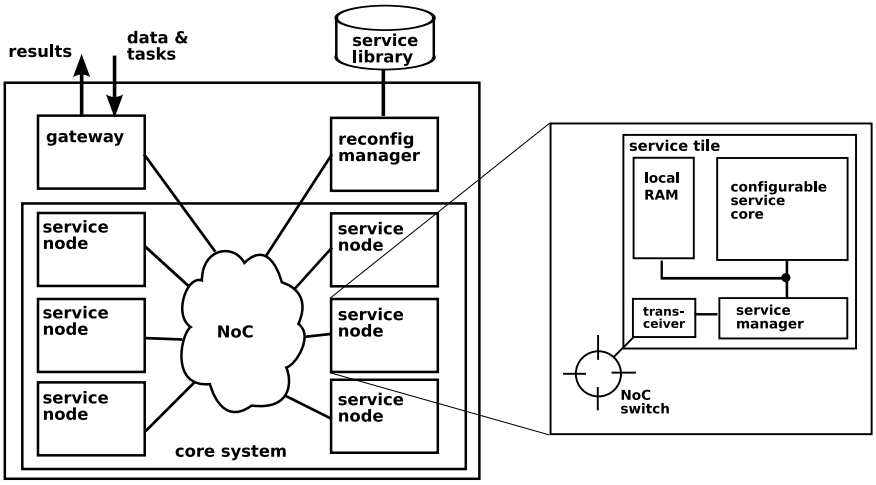


Figure 1: Gannet system architecture

The operation of the Gannet DRI can be described at high level as follows: thanks to the Gannet service manager, every service tile has the capability to know how it should communicate with other tiles in the system. Tiles with reconfigurable cores also have the capability to know how to reconfigure the core. This capability is provided by the SM; the actual information about the configuration of communication and computation is contained in a program that is executed in a distributed fashion by all tiles in the system. A Gannet system can execute several of these programs (called "task descriptions") concurrently. The task descriptions enter the system through the Gateway. The Gateway has an interface with the outside world, dependent on the overall system in which the Gannet SoC is deployed. For example, if the Gannet SoC would be part of a space probe, the interface would be a radio transceiver; if the SoC would act as an accelerator for a modern desktop PC, it could be PCI-express.

Gannet task descriptions are lists of packets, each of which contains a part of the complete program (called a "subtask"). The Gateway simply transmits the packets on the NoC, which transfers them to their destination Service. Once all services have received all required subtasks, the program is executed; in other words the Gannet SoC runs the specified task.

The dynamic reconfiguration mechanism in the Gannet DRI is based on a library of configurations for the Services. The Service reconfiguration manager is the interface through which the configuration data can be requested from off-chip storage. The actual reconfiguration of the IP core in a Service is handled by the SM. Due to the properties of the Gannet DRI, dynamic reconfiguration is very easily expressed in the task description.

4.3 Operation of the Gannet Service Manager

It will be clear from the above description that the Gannet service manager circuit plays a crucial role. As all communication between the cores is handled by it, it is important that the SM is a low-latency circuit, in other words marshalling the data for the core should take a minimal amount of clock cycles. Furthermore, the circuit must be small and low-power. As a rule, we could demand that the overhead of the SM should not be more than a few percent for speed, area and power.We have demonstrated in [Vanderbauwhede et al., 2008] that our novel design indeed achieves these goals.

The service manager is a queue-based system (Figure 2) where several queues are processed in parallel, depending on the type of packet in the queue. This pipelined parallel architecture results in very low latency;

as the queues are very shallow, the area consumption is very modest. A formal model for the operation of the SM is presented in the next sections.

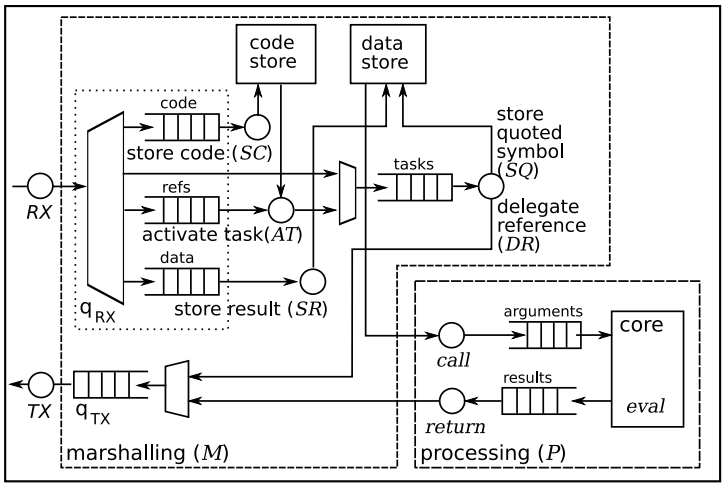


Figure 2: Gannet Service Manager (SM) architecture

We can consider the Gannet SoC as a machine for running Gannet programs. The Gannet SoC architecture is quite different from the familiar von Neumann-style processor-based SoC architecture: it is a distributed processing system without global memory. There is no program counter, and the program is not executed in a sequential fashion but in a demand-driven fashion [Wilkinson, 1996]. We introduce a more formalised description of the Gannet machine:

- The Gannet machine is a distributed computing system where every computational node *consumes packets* and *produces packets* and can store state information between transactions.
- A Gannet symbol is a multi-byte word which encodes operations, code references and constants. A symbol is identified by its *Kind* (*service*, *reference*, *constant*), the *Task* and *Subtask* to which it belongs, and its *Name*. Depending on the Kind, symbols can have additional fields to encode e.g. quoting, extensions, processing mode etc. We denote a symbol as a tuple *(Kind,Task,Subtask, Name, ...)*.
- A Gannet packet consist of a header and a payload. The payload can either be data or instruction code (Gannet symbols). The header consists of following fields: Packet *Type* (*code*, *reference*, *data*), destination address (*To*), return address (*Ret*), packet identifier (*Id*). We denote a Gannet packet as $p(Type,To,Ret,Id;Payload)$
  - A *code* packet contains the Gannet code for a subtask, compiled into a list Gannet symbols. We will use the notation $\langle s_1, \ldots, s_n \rangle$ to denote a list of Gannet symbols. A compiled subtask is called an instruction for the Gannet machine. The payload of a code packet is stored in the Code store.
  - A *reference* packet contains a *code reference*, a symbol which acts as an identifier for instruction code stored in the Code store

In the context of the formal model, computational cores such as IP cores will be called "service cores". The role of the NoC in the Gannet SoC is essential as a medium for transferring packet-based data between services. However, the actual implementation of the NoC will not impact on the functionality of the system – although it will of course influence its performance. Furthermore, the role of the IP cores in the Gannet SoC is purely computational, i.e. IP cores consume data and produce data without awareness of the source or destination of the data. All the intelligence in terms of the control of the communication between the cores is therefore encapsulated in the service manager. Consequently, the operation of the service manager completely determines the operation of the entire system.

With the above definitions, the operation of a Gannet service can be described in terms of the task code and the result packet produced by the task, using the following set of actions (see Figure 2):

- **Receive packet** (*RX*): when a service S_i receives a packet, the packet is stored in one of the queues (*code*, *refs*, *data*) dependent on the packet's *Type*. Presence of a packet in the queue triggers the next action:
  - For code packets: **Store code packet** (*SC*)**:** a service S_i receives a *code* packet $p(Code, S_i, S_j, R_{task}; subtask)$ where $subtask = \langle S_i\ a_1 ... a_n \rangle$. The instruction (compiled subtask code) is stored in the Code store for execution at a later time and referenced by the symbol R_{task}.
  - For reference packets: **Activate task** (*AT*)**:** the service S_i in $state_i$ receives a task *reference* packet $p(Ref, S_i, S_j, R_{id}; R_{task})$ [1]; the service activates the task referenced by R_{task}: $\langle S_i\ a_1 ... a_n \rangle$by transferring it to the tasks queue. This results in evaluation of the arguments $a_1, ..., a_n$ as follows:
    - If the argument is a reference symbol**:** *DR***: Delegate by reference packet:** the service manager requests activation of a subtask by sending a *reference* packet,containing the corresponding *reference* symbol, to the corresponding service
    - If the argument is a constant symbol: *SQ***: Store constant symbol:** all *constant* symbols (e.g. numbers) in the code ares stored in the local Data store.
  - For data packets: **Store returned result** (*SR*)**:** result data from subtasks are stored in the local Data store.

- *P***: Processing:** When all arguments of the subtask have been evaluated, the data are passed on to the service core by pushing them onto the *arguments* queue. The core performs processing on the data; the service produces a result packet $p_{res} = p(Type_i, S_j, S_i, R_{id}; Payload_i)$ where the $Payload_i$ is the result of processing the evaluated arguments $a_1, ..., a_n$by the core of S_i.

- **Transmit packet** (*TX*): when the packet resulting from the *P* or *DR* action is put in the TX queue, it will be transmitted over the NoC. The packet p_res is sent to S_o and there Payloadi is stored in a location referenced by R_{id}.

This operation sequence results in a fully parallel execution of all branches in the program tree in an unspecified order governed by the processing time of the packets. We call the set of actions $M = \{SC, AT, DR, SQ, SR\}$ the Marshalling set. Every action in the *M* set acts on a queue of packets (see Figure 2). Thus the Gannet service manager provides the marshalling of the data and the service core provides the processing.

To simplify the discussion we have omitted state changes in the above explanation. The *P* action can produce a change in the state of the service, this feature is discussed in detail in Section 6. The implementation of the Gannet service manager is discussed in detail in Section 8. In the next sections we discuss the Gannet language and we present a model for the Gannet machine. These two components are the foundation on which the Gannet system is built, therefore a detailed discussion is essential for the understanding of the final sections which present the actual implementation and programming of the Gannet DRI.

[1] 12

5 The Gannet Language

In this section we introduce the Gannet language [Vanderbauwhede, 2006a], and Intermediate Representation language intended as a target for High-Level Language compilers, and we discuss compilation of Gannet programs into bytecode for the Gannet machine.

Gannet programs define the interactions between the services by mapping every service to a *named function* and describing the flow of data between the services in terms of function calls. The Gannet language is an *intermediate representation language* (IR) for the Gannet Virtual Machine, comparable to .NET CIL [Gough, 2001] or PIR, the Parrot Intermediate Representation [Randal et al., 2004]. In other words, Gannet is not a high-level language but a compilation target language. A program written in Gannet syntax can be transformed in machine code in a trivial way.

5.1 Language Syntax

Gannet syntax is an S-expression syntax (similar to LISP [McCarthy, 1960] or Scheme [Sussman and Steele, 1975]) completely free from syntactic sugar. In EBNF [Bray et al., 2008], a Gannet expression must always obey (for simplicity we assume that all expressions include a trailing white space):

service-expr ::= '(' *service-symbol* "'"? *arg-expr*+ ')'

arg-expr ::= *service-expr* | *literal-symbol*

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

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

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

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

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

5.2 Language Semantics

An operational semantics for the Gannet language has been presented in [Vanderbauwhede, 2008]. In this section we focus on the key features that make Gannet suitable as an intermediate representation language for a a Dynamic Reconfiguration Infrastructure.

Strict Parallel Evaluation: A Gannet service can be considered as a function, though not necessarily a pure function. The *service manager* effectively performs evaluation of the function arguments; the evaluated arguments are passed on to the service core which computes a result based on the argument values and returns it. A key feature of the Gannet language is that the evaluation order is unspecified; the Gannet "machine" evaluates all arguments in parallel. As a result, a Gannet program will automatically exploit the capacity for parallelism present in the system.

Quoting to Defer Evaluation: Quoted expressions are passed unevaluated to the service core. In this way, service cores that implement control structures can evaluate expressions as required.

5.3 Custom Control Constructs

Because of Gannet's minimal design philosophy, there is not a unique fixed set of control structures. In principle, every developer can design a custom set of controls. However, for convenience we have defined a small core set, very similar to Scheme's [Kelsey et al., 1998, Vanderbauwhede, 2007].In Gannet parlance, these are called *control* services as they provide control over the flow of data. To allow control over the flow of data, Gannet defines a number of *control* services. The core set consist of the lexical scoping constructs (`group`, `assign`, `read`), the conditional branching construct (`if`),the function definition and application constructs (`lambda`, `apply`) and list operations (`list`, `head`, `tail`, `length`, `cons`).

Lexical scoping The `group` construct acts as a block that provides lexical scope to the variables declared inside it using `assign`, similar to *let* in Scheme, Haskell or ML. Accessing a variable requires an explicit `read`. Lexical variables are immutable.

group-expr ::= '`(group` ' "'"? *assign-expr*+ "'"? *arg-expr* '`)`'
assign-expr ::= '`(assign` ' "'" *var-symbol arg-expr*+ '`)`'
read-expr ::= '`(read` ' "'" *var-symbol* '`)`'

Conditional branching The ubiquitous *if-then-else* construct:

if-expr ::= '`(if` ' *service-expr* "'"? *arg-expr* "'"? *arg-expr* '`)`'

Function definition and application The `lambda` construct creates a λ-function (which can be bound to a lexical variable). Function application requires an explicit `apply`.

lambda-expr ::= '`(lambda` ' "'" *var-symbol*+ "'" *service-expr* '`)`'
apply-expr ::= '`(apply` ' *lambda-exp*r "'" *arg-expr*+ '`)`'

List operations List operations are similar again to those in in Scheme, Haskell or ML. Lists require an explicit `list` constructor.

list-expr ::= '`(list` ' "'" *arg-expr*+ '`)`'
single-list-op ::= '`head`' | '`tail`' | '`length`'
single-list-expr ::= '`(`' *single-list-op list-expr* '`)`'
cons-expr ::= '`(cons` ' *list-expr arg-expr* '`)`'

5.4 Streaming data Processing

Gannet has provisions for pipelining of operations on streaming data. This feature has no equivalent in languages intended for sequential processors, as pipelining only makes sense if operations can be performed in parallel. In multicore SoCs, pipelining is a key feature.

buf-expr ::= '(buf ' "'" *buf-var* "'" *service-expr* '`)`'
stream-expr ::= '(stream ' "'" *buf-var*'`)`'
eos-expr ::= '(eos ' "'" *vbuf-var*'`)`'
peek-expr ::= '(peek ' "'" *buf-var*'`)`'
get-expr ::= '(get ' "'" *buf-var* '`)`'

The `buf` expression stores the return value of *expr* in the buffer *buf-var*. The `stream` expression returns the buffered value, calls the *expr* bound to *buf-var* and stores the return value in the buffer. The `eos` expression returns `true` if the end of the stream is reached.

As an example, a 2-stage pipeline S1→[b1]→S2→[b2]→S3 can be written as:

```
(let
 '(buf 'b1 '(S1 ...)
 '(buf 'b2 '(S2 (stream 'b1)))
 '(S3 (stream 'b2))
)
```

5.5 Compilation of Gannet Code into Packets

This section explains how a Gannet program is compiled into packets for running on the Gannet machine. As Gannet-C is simply syntactic sugar for Gannet S-expressions, the transformation from Gannet-C into Gannet S-expressions is omitted.

The compilation process is very straightforward:

1. Decompose the nested S-expression into a list of flat S-expressions by replacing the nested expressions by references

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

2. Every token in the expression is replaced by a tuple (a structured byteword) containing the symbol's *kind* and a unique byteword representing the Gannet symbol corresponding to the token:

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

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

3. Create code packets: using the notation introduced above, a code packet is represented as

   $p_i = packet(\text{code}, n_{S_i}, GW, r_i; \langle s_i\ r_{i,1} ... r_{i,n} \rangle)$

   with n_{S_i} the name of the service S_i, *GW* is the "gateway", the interface between the Gannet SoC and the outside world.

4. Create a reference packet to the root task:

   $p_{root} = packet(\text{reference}, n_{S_{root}}, GW, r_{root}; r_{root})$

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

5.6 Compiling High-Level Languages into Gannet

The Gannet language is intended as a compilation target for high-level languages. Because of the functional nature of the Gannet language and system, functional programming languages such as SML or Scheme are ideal candidates for compilation into Gannet. In particular, we have reported on compilation of Scheme into Gannet [Vanderbauwhede, 2007]. Imperative languages such as C and Java are less suitable because they have limited notion of parallelism and concurrency, having been designed for sequential von Neumann-style processors. However, C dialects such as Single-assignment C [Scholz, 2003] and C-syntax dataflow-style languages such as Mitrion-C and similar languages [Koo et al., 2007] constitute suitable candidates.

6 The Gannet Machine Model

In this section we introduce the Gannet Machine Model, a semantic model to explain the operation of the Gannet DRI [Vanderbauwhede, 2008]. The semantic model is used to demonstrate how a Gannet SoC runs Gannet programs and how the Gannet DRI implements concurrency, global and local flow control and streaming data processing.

6.1 Notation and definitions

Notation

- The notation $\bullet$ is used to separate a packet from the other packets in the queue: $(p \bullet ps)$ denotes a packet at the head, $(ps \bullet p)$ a packet at the tail.
- The notation $*$ ("don't care") indicates that the value of a field does not influence the operation.
- The notation ... indicates the presence some non-specified entities. In general, unspecified entities are left out unless omitting them would cause ambiguity.
- The notation _ indicates allocated available storage space
- The notation $e \Downarrow w$ indicates large-step evaluation of e to w.
- We use following shorthand conventions:
  - *expression: e*
  - *service-symbol: s*
  - *reference-symbol: r*
  - *variable-symbol: v*
  - *argument-symbol: x*
  - *quoted symbol: prefix '*
  - *value: w*

Definitions

- The Gannet system consists of N service nodes $S_i(...)$, $i \in 1..N$, a packet-switched communication medium ("Network on Chip") and a gateway to the outside world, $G(...)$.
- The unit of data transfer in the Gannet SBA is the packet. Depending on the packet's *Type*, the *Payload* can be *data* or an *expression*.
- The packet receive and transmit FIFO queues of the services are represented by q_{RX} and q_{TX}. A received a packet is pushed onto the RX queue; a transmitted packet is shifted off the TX queue.

  The RX queue actually consists of four queues multiplexed by the packet's *Type*:

  q_{RX}*(tasks(...),data(...),refs(...),code(...)).*

  Thus $q_{RX}(ps \bullet p)$ is actually $q_{RX}(...pt(ps \bullet p)...)$ with$pt \in \{$*tasks,data,refs,code*$\}$. (In the actual design *tasks()* is not part of the RX queue, but placing it there simplifies the analysis.)
- Packet receive and transmit FIFO queues: q_{RX} and q_{TX}
- Apart from the RX/TX queues, a service node S_i consists of following entities:
  - The data store:$store_d(...(Label\ data)...)$. *Label* is a Gannet symbol, *data* is the stored content. Space allocated for data to be stored is denoted by _: $store_d(...(Label\ \_)...)$
  - The task packet store: $store_{tp}(...(Label\ p)...)$. p is the stored packet,*Label* is the Label field from the packet's header.

– The processing core $core(...)$ which performs the actual processing of the data.

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

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

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

Small-step semantics

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

Lines above the transition expression define items (e.g. packets) appearing on the left-hand side, lines below the transition expression define items appearing on the right-hand side.

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

6.2 Packet processing by the services

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

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

Packet transfer between services

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

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

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

Marshalling action set M

On receipt or activation of a *task* packet, a number of actions can be performed by the service manager, as explained in 4.3. As mentioned there, these actions are grouped in the M ("Marshalling") set. Application of the M set results in evaluation of all arguments of a service call. The actions of the complete set $M = \{SC, AT, DR, SR, SQ\}$ can be expressed as:

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

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

Processing action set P

The actions of the service core determine the functionality of the service. This functionality can be defined as the type, destination and payload content of the packet the service produces based on the values marshalled by the service manager.

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

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

The processing can be expressed as:

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

Hardware Memory Management

An important point to note is that all the memory is fully managed by the SM: it allocates memory for data required by a task and deallocates it when the task has finished. As a result, the language does not need any memory management constructs and it is not possible for different tasks to impinge on each other's memory. Static memory allocation would offer the same benefits but it would require memory to be allocated for all possible tasks at all times and would thus require a large amount of memory, most of which never to be used. Dynamic memory management introduces a small overhead for managing the address stack but the required area is negligible compared to the memory area for fully static allocation; the run-time overhead is only a single cycle for allocating an address; de-allocation takes a few more cycles but does not add to the run-time overhead as it does not block the processing.

6.3 Control and computational service semantics

In this section and the next, we consider actions at the level of the M and P sets without detailing the individual actions in each set. The combined M and P action sets result in the service transmitting a result packet in response to receiving a task packet and potentially modifying the local store. The semantics of the Gannet services can be described completely in terms of the task and result packets and the state of the store. The aim of the next two sections is to illustrate this mechanism.

Computational service semantics

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

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

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

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

Control Service Semantics

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

Lexically scoped variables Lexical scoping is implemented through the `group` and `assign` services. Variables are bound to an expression by the `assign` service:

$$
\begin{array}{rl}
 & p_{rx} = packet(\text{task}, \textbf{assign}, \textbf{group}, r_a; e_{assign}); \\
 & e_{assign} = \langle \textbf{assign}'\, v_j\, r_j \rangle;\ r_j \Downarrow w_j \\
 & S_{assign}(store_d(...)) \\
\longrightarrow^M & S_{assign}(store_d(...('v_j\, v_j)\, (r_j\, w_j)...)) \\
\longrightarrow^P & S_{assign}(store_d(...(v_j\, w_j)...)) \\
 & p_{tx} = packet(\text{data}, \textbf{group}, \textbf{assign}, r_a; v_j)
\end{array}
$$

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

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

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

$$
\begin{array}{l}
p_{group} = packet(\text{task}, \textbf{group}, i, r_l; e_{group}); \\
e_{group} = \langle \textbf{group} \ldots r_{assign,j} \ldots r_k \rangle; \\
r_k \Rightarrow \langle s_k ... r_j ... \rangle \Downarrow w_k; r_j \Rightarrow \langle \textbf{read}'\, v_j \rangle \Downarrow w_j
\end{array}
$$

$$
\begin{array}{rl}
 & S_{group}(store_{assign}(...)) \\
\longrightarrow^{M} & S_{group}(store_{assign}(...(v_j\, w_j)...(r_k\, w_k)...)) \\
\longrightarrow^{P} & S_{group}(store_{assign}(...)) \\
 & p_r = packet(\text{data}, i, \textbf{group}, r_l; w_k)
\end{array}
$$

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

- Functions are defined by the `lambda` service:

$$
\begin{array}{rl}
 & p_{lambda} = packet(\text{task}, \textbf{\textit{lambda}}, *, *; e_{lambda}); \\
 & e_{lambda} = \langle \textbf{lambda}\, {}'x_j ...\, {}'r_\lambda \rangle;\ r_\lambda \Rightarrow \langle s_j ... x_j ... \rangle \\
 & S_{lambda}(store_d(...)) \\
\longrightarrow^{M} & S_{lambda}(store_d(...(x_j\, x_j)...(r_j\, r_\lambda)...)) \\
\longrightarrow^{P} & S_{lambda}(store_d(...)) \\
 & p_r = packet(\text{data}, *, \textbf{lambda}, *; e_\lambda); \\
 & e_\lambda = \langle ... x_j ... r_\lambda \rangle
\end{array}
$$

- Function application by the `apply` service:

$$
\begin{array}{rl}
 & p_{apply} = packet(\text{task}, \textbf{apply}, j, *; e_{apply}); \\
 & e_{apply} = \langle \textbf{apply}\, r_\lambda ...\, {}'r_j ... \rangle;\ r_\lambda \Downarrow e_\lambda;\ {}'r_j \Downarrow r_j \\
 & S_{apply}(store_d(...)) \\
\longrightarrow^{M} & S_{apply}(store_d(...(r_\lambda\, e_\lambda)...({}'r_j\, r_j)...)) \\
\longrightarrow^{P} & S_{apply}(store_d(...)) \\
 & p_r = packet(\text{task}, *, j, r_w; e_w);\ e_w = e_\lambda[x_j/r_j]
\end{array}
$$

As can be seen from this semantics, the `apply` service does not bind values but rather substitutes code references. The reason for this behaviour is explained in the Section 6.6.

6.4 Multi-operation Services

It is often desirable for a service to be able to perform more than one operation. In particular for service cores that take up little area, it is desirable to share one service manager amongst several service cores to limit the area overhead of the service manager. For example, for control services the granularity is quite fine and the implementation is relatively simple compared to the actual IP cores; on the other hand, control services need to be very flexible and in particular the sets of services related to lexical variables, lambda functions and list manipulations are closely interlinked and require a considerable amount of memory, although their data throughput is not large. It is therefore most area-efficient to combine control constructs into a single service. We call such a combined service a *multi-operation service*. The *Name* field of the service symbol indicates which operation must be performed for a given call (similar to the opcode in a microprocessor). The only difference with the single-operation services as presented above is the value of the To-field. For example, if we name the combined control service `control`, an `assign` task packet now becomes:

$$
\begin{array}{l}
p_{assign} = packet(\text{task}, \textbf{control}, j, r_j; e_{assign}) \\
e_{assign} = \langle \textbf{assign}\ {}'v\ r \rangle
\end{array}
$$

The advantages of combining services are less overhead thanks to the shared service manager and local store and the potential to factor out common functionality, thus further reducing the required area. Furthermore, in the particular case of control services, they can be implemented on a small embedded microcontroller for

maximum flexibility. The potential disadvantage is due to the lower degree of parallelism and the potentially large number of calls to be handled, i.e. the classic trade-off of speed for area. However, as we will discuss in section 6.6, this drawback can be mitigated by ensuring no control service, either combined or individual, is present in the data path.

6.5 Dynamic Service Configuration

Using the concept of a multi-operation service, to achieve dynamic service configuration in Gannet it suffices to add a *reconfiguration service* to every dynamically reconfigurable service core. This configuration service will request the configuration information from a central *reconfiguration manager*. A Gannet program that reconfigures a service S1 to performa discrete cosine transform (DCT) might look like this:

```
(let
    '(S1-reconf (config 'dct 's1))
    '(S1 (block (img '8))
)
```

The key point is that there is no change required to the Gannet DRI to support dynamic service configuration: it is simply a matter of adding the extra services. The global `config` service (i.e. the reconfiguration manager) is simply a database server which serves the precompiled configurations supported by each service core. The local `reconf` operation depends on the nature of the actual reconfigurable core. For example, if the core would be a microcontroller, `reconf` would simply load a different program into memory; if the core would be an embedded FPGA, `reconf` would load up a bitstream. To indicate to the reconfiguration manager what type of reconfiguration information is required for a given service, the config service takes two arguments: one describing the functionality to be configured and one describing the nature of the service core. The `reconf` call also takes two arguments, the actual reconfiguration data and a label for the configuration. This label is used to check if the current configuration is different from the new configuration. If the configurations are identical, the current configuration is kept.

In the remainder of this section we use the semantic model to explain two key advanced flow control features of the Gannet system: separation of control flow from data flow and streaming data processing.

6.6 Separation of Control Flow from Data Flow

An issue of crucial importance for the performance of SoCs with multiple high-throughput data flows is the separation of control flows and data flows. Although Gannet does not require centralised control, in many cases services will perform flow control tasks. This section introduces a technique used to avoid the potential bottleneck at such control services.

Bottleneck in Central-memory SoC Architectures

Consider a simple SoC which implements a video webcam. The system captures a video stream, compresses it using a lossy codec, encrypts it and transmits it over a network. The functionality for these four operations is provided by hardware IP cores. Following a conventional SoC design approach, this system will be implemented using a microcontroller which runs an embedded operating system (Figure 3).

The OS communicates with the hardware using device drivers. Data is transferred to and from the central memory either via the microcontroller or via a direct memory access (DMA) controller. Clearly this memory transfer is a bottleneck: if the number of datapaths or the number of operations per datapath would be very large, the memory access time will limit the data rate. The Gannet architecture allows separation of dataflow from control flow: the data will be transferred directly between the data processing cores (Figure 4), eliminating the bottleneck. Consequently the system will be able to support more and longer data paths. As the net number of data transfers is lower, the system will also consume less power than the conventional architecture.

Redirection of Data Flows

For optimal performance control flows must be separated from data flows, i.e. data should flow between non-control services while the actual data path is governed by the control service. If this would not be the case, data

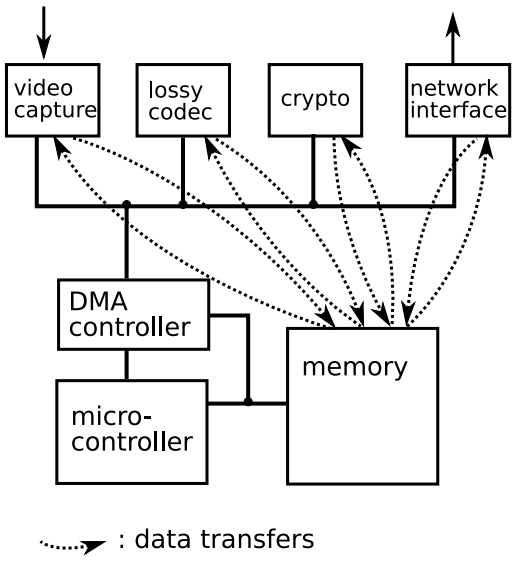


Figure 3: Simple videocam SoC with embedded OS

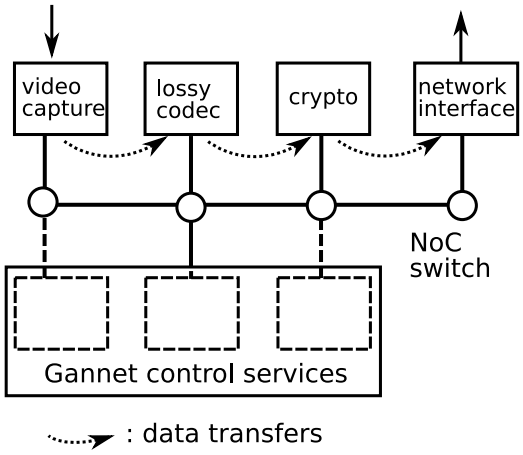


Figure 4: Gannet architecture for simple videocam SoC

would have to be copied to and from the control service's local memory, causing a performance bottleneck. The Gannet system as presented solves this issue via a combination of *deferred evaluation* and *result redirection*. To explain this mechanism, we use the control services introduced above as examples.

Consider the expression:

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

With the semantics as presented in Section 6.3, the `group` service receives the result of the evaluation of the `S2` call. It then passes this result on to `S1`. However, the behaviour is different when the last argument is quoted:

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

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

$$
\begin{array}{rl}
 & p_{group} = packet(\text{task}, \textbf{control}, S1, r_l; e_{group}); \\
 & e_{group} = \langle \textbf{group} \ldots r_{assign} \ldots {}' r_{S2} \rangle; \\
 & r_{assign} = \langle assign \, {}' v_j \; r_j \rangle \\
 & r_{S2} \Rightarrow \langle S2 \ldots r_j \ldots \rangle \Downarrow w_k; r_j \Rightarrow \langle \textbf{read} \, {}' v_j \rangle \rightarrow w_j \\
 & S_{group}(store_{assign}(\ldots)) \\
\longrightarrow^M & S_{group}(store_{assign}(\ldots(v_j \, w_j) \ldots({}' r_{S2} \, r_{S2}) \ldots)) \\
\longrightarrow^P & S_{group}(store_{assign}(\ldots)) \\
 & p_r = packet(\text{reference}, S2, S1, r_l; r_{S2})
\end{array}
$$

Another example is conditional branching as implemented by the `if` service.

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

The semantics without quoting (and thus without result redirection) are:

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

With the above semantics, both branches will always be evaluated. This is obviously undesirable in many cases. If the second and third arguments are quoted, evaluation is deferred to the core which will evaluate only one branch predicated on the value of the first argument.

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

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

$$
\begin{array}{rl}
 & p_{if} = packet(\text{task}, \textbf{control}, S1, r_1; e_{if}); \\
 & e_{if} = \langle \textbf{if} \, r_p \, {}' r_t \, {}' r_f \rangle; \; r_p \Rightarrow e_p = \langle s_p \ldots \rangle; e_p \Downarrow \textbf{true} \\
 & S_{if}(store(\ldots)) \\
\longrightarrow^M & S_{if}(store(\ldots(r_p \, \textbf{true}) \, ({}' r_t \, r_t) \, ({}' r_f \, r_f) \ldots)) \\
\longrightarrow^P & S_{if}(store(\ldots)) \\
 & p_r = packet(\text{reference}, S_{2t}, S1, S_1; r_t);
\end{array}
$$

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

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

If `apply` would bind the evaluated arguments to the λ-variables, all results would be copied to the `apply` service's local store and would have to be requested from there. This excessive back-and-forth copying of potentially large amounts of data is avoided by substituting the λ-variables for code references instead[3]:

34

$$
\begin{array}{ll}
 & p_{apply} = packet(\text{task}, \textbf{control}, S1, r_1; e_{apply}); \\
 & e_{apply} = \langle \textbf{apply}\, r_\lambda\, {}'r_{S3} \rangle;\ r_\lambda \Rightarrow e_\lambda;\ {}'r_{S3} \Downarrow r_{S3} \\
 & e_\lambda = \langle \textbf{lambda}\, {}'x\, {}'r_{S2x} \rangle \\
 & r_{S2x} \Rightarrow \langle S2\ x \rangle \\
 & S_{apply}(store(...)) \\
\longrightarrow^M & S_{apply}(store(...(r_\lambda\, e_\lambda)...\,({}'r_{S3}\, r_{S3})...)) \\
\longrightarrow^P & S_{apply}(store(...)) \\
 & p_c = packet(\text{code}, S2, j, r_w; e_w); \\
 & e_w = e_\lambda[x/r_{S3}] = \langle S2\ r_{S3} \rangle \\
 & p_r = packet(\text{reference}, S2, S1, r_1; r_w)
\end{array}
$$

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

6.7 Pipelined Streaming Data Processing

Pipelined streaming data processing is a key feature in SoCs, in particular for multimedia processing. In fact most SoCs have traditionally been constructed around a (non-reconfigurable) datapath for streaming data. It is therefore imperative that Gannet should support this processing model.

The semantics refer to the example in Section 5.4. It should be noted that `(buf 'b1 (S1...))` and `(stream 'b1)` are syntactic sugar, as the streaming and buffering capability are provided by the SM of service S, not by separate `buf` and `stream` services (as this would be extremely inefficient).

$$
\begin{array}{ll}
 & p_{buf} = packet(\text{task}, S_1, \textbf{let}, r_1; e_{buf});\ e_{buf} \Rightarrow \langle S_1^{buf} ... r_{stream} ... \rangle; \\
 & S_1^{buf} = (..., b, ...) \\
 & r\text{stream} \Rightarrow e\text{stream} \Downarrow w\text{stream} \\
 & S(q_{RX}(p_{buf} \bullet ps), q_{TX}(qs), store(...)) \\
\longrightarrow^M & S(store(...(r_{stream}\, w_{stream})\, (b\_)\, ...)) \\
\longrightarrow^P & S(store(...(b\, (w, p_{buf}))...));\ w = \delta(S_1, ..., w_{stream}, ...) \\
 & p_r = packet(\text{data}, S_1, \textbf{let}, r_1; b)
\end{array}
$$

The expression `(buf 'b1 (S1 ...))` is translated by the compiler into $\langle S_1^{buf}\, {}'b\, ... e_{stream} ... \rangle$; the superscript *buf* indicates that the service manager should buffer the result of the computation rather than return it. The result of the M actions is to store the buffer variable and to evaluate all arguments of S1 and store the values. The P action performs a δ-evaluation on the arguments; it then stores the result as a tuple (w, p_{buf}) and returns the buffer variable b.

The `stream` call returns the buffered result and reschedules the buffering task:

$$
\begin{array}{ll}
 & p_{stream} = packet(\text{task}, S_1^{buf}, S_2, r_2; e_{stream});\ e_{stream} \Rightarrow \langle S_1^{stream}\, {}'b \rangle; \\
 & S(q_{RX}(p_{stream} \bullet ps), q_{TX}(qs), store(...(b\, (w, p_{buf}))...)) \\
\longrightarrow^M & S(store(...({}'b\, b)\, ...(b\, (w, p_{buf}))...)) \\
\longrightarrow^P & S(q_{RX}(ps \bullet p_{buf}), q_{TX}(p_w), store(...)); \\
 & p_w = packet(\text{data}, S_1, S_2, r_2; w)
\end{array}
$$

The expression `(stream 'b1)` is translated by the compiler into $\langle S_1^{stream}\, {}'b \rangle$; the superscript *stream* indicates that the service manager should return the value from the buffer and reschedule the task p_{buf}. Thus S2 will operate on the previous result returned by S1, i.e. the operations are pipelined.

The buffer/stream mechanism allows interleaving (multiplexing) of streams on every service in the pipeline, i.e. a single service will time-multiplex automatically over streams in a statistical fashion. Apart from the `buffer` and `stream` constructs, the buffer can be accessed without restarting the task via a `peek` instruction. As a consequence, the buffer mechanism also serves to implement a local store mechanism for caching (call-by-need) and accumulation.

6.8 Localised Control

Even with the above mechanism for separation of data flow from control flow, a control construct could still become a bottleneck simply because of the frequency of calls made to it. In particular the branching service (`if`) would, when inserted in a pipeline, have to support very large numbers of calls as it would be called on every cycle. This would also increase the pipeline's latency. Multiple pipelines with branches would exacerbate the problem. For performance reasons it would therefore be better to provide every service with flow control functionality, i.e. every service becomes a multi-operation service with a number of control operations. However, implementing the full set of control features introduced above would consume a lot of area. It is however possible to achieve a high degree of control with low overhead as follows:

Localised branch control

First of all we introduce a localised version of the `if` service (we will call the previously introduces `if` the "global `if`"). Syntactically, there is no change:

```
(S1 (if (Sp ...) '(St ... ) '(Sf ...)))
```

The compiler will however work out to which service the `if` belongs. Effectively, the expression becomes:

```
(S1 (S1-if (Sp ...) '(St ... ) '(Sf ...)))
```

A key difference with the global `if` is that there is no need for result redirection as the result from either `St` or `Sf` is destined for `S1`. This results in a considerable simplification of the implementation.

E.g. if the `Sp` call evaluates to true, the result of `St` will be sent directly to `S1`.

$$
\begin{array}{rl}
 & p_{if} = packet(\text{task}, S_1, S_1, r_1; e_{if}); \\
 & e_{if} = \langle S_{1,if}\, r_p\, {}'r_t\, {}'r_f \rangle;\ r_p \Rightarrow e_p = \langle s_p ... \rangle; e_p \Downarrow \textbf{true} \\
 & S_I(store(...)) \\
\longrightarrow^M & S_I(store(...(r_p\, \textbf{true})\, ({}'r_t\, r_t)\, ({}'r_f\, r_f)...)) \\
\longrightarrow^P & S_I(store(...)) \\
 & p_r = packet(\text{reference}, S_t, S_1, r_1; r_t);
\end{array}
$$

Unconditional return as block structure

The return value of the `if` service is predicated on its first argument. It is trivial to implement an unconditional return using the same infrastructure. The `return` service simply returns its first argument, e.g. `(S1 (return '(S2 ... )))`:

$$
\begin{array}{rl}
 & p_{ret} = packet(\text{task}, S_1, S_1, r_1; e_{ret}); \\
 & e_{ret} = \langle S_{1,\text{return}}\, {}'r_2 \rangle \\
 & S_I(store(...)) \\
\longrightarrow^M & S_I(store(...({}'r_2\, r_2)...)) \\
\longrightarrow^P & S_I(store(...)) \\
 & p_r = packet(\text{reference}, S_2, S_1, r_1; r_2);
\end{array}
$$

This may not seem particularly useful at first sight. However, remember that the SM will always evaluate all unquoted arguments of a call and that the number of arguments is unspecified. As a result, the `return` service can act similar to the `group` service. For example `(return 'e1 e2)` is equivalent to `(group e2 'e1)`; to emulate the behaviour of group with all arguments quoted, e.g. `(group 'e1 'e2 'e3)` we can use nested returns: `(return 'e3 (return 'e2 (return 'e1)))`.

Local variables

Using the buffer infrastructure, the local control can also support local variables. The main difference is that the buffer is bound to the enclosed service while the local variables are bound to the enclosing service, e.g. `(buf 'b1 (S1 ...))` binds at `S1`; `(S2 (return '(S3 v1) (var 'v1 (S1 ...))))` binds at `S2` and `return` is local to `S2`. The implication is that for buffers, the binding information is encoded in the service symbol but for local variables the binding must be encoded in the reference symbol:

$$
\begin{array}{rl}
 & p_{var} = packet(\text{task}, S_2, S_2, r_2; e_{\text{ret}});\ e_{\text{ret}} \Rightarrow \langle S_{2,\text{return}}\, {}'r_3\, r_1^{\text{var}} \rangle \\
 & {}'r_3 \Downarrow r_3 \Rightarrow \langle S_3 \ldots v_1 \ldots \rangle; r_1^{\text{var}} \Rightarrow \langle S_1 \ldots \rangle \Downarrow w_1; r_1^{\text{var}} = (\ldots, v_1, \ldots) \\
 & S(q_{RX}(p_{var} \bullet ps), q_{TX}(qs), store(\ldots)) \\
\longrightarrow^M & S(store(\ldots({}'r_3\, r_3)\,(v_1\, w_1)\ldots)) \\
\longrightarrow^P & S(store(\ldots(v_1\, w_1)\ldots)) \\
 & p_r = packet(\text{reference}, S_3, S_2, r_2; r_3);
\end{array}
$$

Localised Loops

It is possible to implement loops using the combination of return, if and local variables. To do so, we introduce a syntax for labelling expressions in Gannet: (label 'L1 (S1 ...)). The `label` construct is not a service as labelling is purely a compile-time feature. For example, (S2 (label 'L1 e1) L1) is equivalent to (S2 e1 e1). The compiler simply substitutes the label by the reference to the expression. Using labelling, we can now write e.g. a simple accumulation as follows:

```
(return
    '(label 'L (if (count)
        v
        '(return 'L (var 'v (S v))
    )
    (count 10)
    (var 'v (S ...))
)
```

In this example, the service S is called 10 times recursively and the final result is returned. The decrementing counter could be implemented as a stand-alone service but as counters are small it will be much more effective to implement it as a localised control.

7 Examples of Dynamic Reconfiguration

This section presents examples of Gannet task descriptions to illustrate dynamic (i.e. run-time) reconfiguration of the data path and the service core functionality in a Gannet SoC.

7.1 Example of Dynamic Data Path Reconfiguration

Consider a system for periodic image capture, for example for use in space exploration robots such as the Mars Rover. The system periodically captures images of its surroundings and transmits them to a satellite. Because of the orbit of the satellite, at certain times of the day the transmission bandwidth will be high and at other times low. Furthermore, at night images should be captured using an infrared camera rather than a visible light camera. Figure 5 illustrates the system. For clarity, only the paths have been shown.

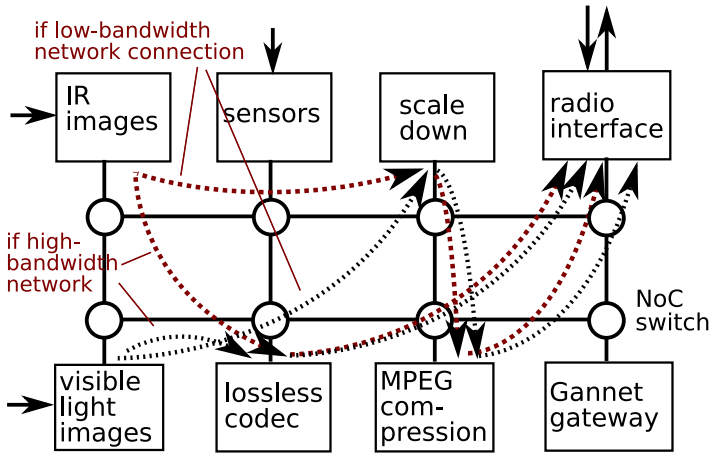


Figure 5: Example dynamic image capture system

The Gannet task description that governs the dynamic reconfiguration of the data path is shown below:

```
(let
'(assign 'hi-speed (label 'test_bw (> (bandwidth) TH_BW)))
'(assign 'infrared (label 'test_ir (> (tsr-sensor) TH_IR)))
'(if infrared
    '(buf 'b1i (img-capture-infrared))
'(buf 'b1v (img-capture-visible)))
'(if hi-speed
    '(buf 'b2hi (lossless-codec
(if infrared'(stream 'b1i)'(stream 'b1r))))
'(let
'(buf 'b1s (scale-down
(if infrared'(stream 'b1i)'(stream 'b1r))))
'(buf 'b2lo (compress (stream 'b1s)))
))
'(loop
'(radio-interface
'(if hi-speed'(stream 'b2hi)'(stream 'b2lo)))
'(wait SAMPLE_PERIOD)
'(update 'hi-speed test_bw)
'(update 'infrared test_ir))
)
```

The `assign` statements at the top test the bandwidth and light intensity and set the boolean variables `infrared` and `hi-speed`. Depending on the values of these variables, images are captured in infrared or visible light and transmitted uncompressed over a high-speed link to the satellite or scaled down, compressed and transmitted over a low-speed link. The lines with `buf` statements fill the initial pipeline; `stream` statements refresh the buffers (i.e. remove the old value and call the corresponding service which refills the buffer; see Section 6.7 for details) . The loop statement at the bottom is the equivalent of a `while(1)` in C or a `forever` in Verilog. The system periodically transmits the processed image over the link and updates the test variables. To understand the program it is important to realise that the description is actually a call tree, i.e. action is initiated by the final service: the `radio-interface` calls the `stream` command on e.g. the buffer `b2hi` (if bandwidth is high), this command results in the corresponding service (lossless-codec) calling the stream command on the buffer `b1i` (if infrared capture), this results in a call to `img-capture-infrared` which captures a fresh image. As explained in 6.6, the conditional statements simply redirect the requests, they are not part of the data path.

7.2 Example of Dynamic Service Reconfiguration

The example in this section is taken from [Noguera and Badia, 2006]. The paper reports on dynamic reconfiguration of systems with multiple FPGAs and uses a Sobel filter (used for edge detection in images) as a test case. As Gannet is a SoC infrastructure, we assume a dynamically reconfigurable SoC with three embedded FPGA cores (`S1`, `S2`, `S3`) rather than a discrete system. Furthermore, we assume a fixed core for actual capture of the image block on which the filter acts (`img-block`). The Gannet system is illustrated in Figure 6.

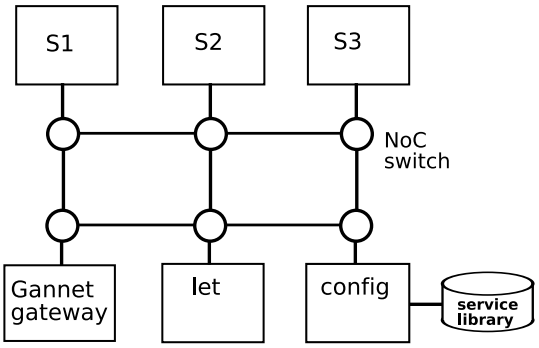


Figure 6: Example dynamically reconfigurable image processing system

The original paper uses an embedded processor for IO and to control the dynamic reconfiguration. Our aim is to illustrate how dynamic reconfiguration of service core functionality can be achieved using the Gannet DRI, without the need for an embedded microprocessor. Figure 7 shows the task graph of the Sobel edge enhancement application used in [Noguera and Badia, 2006].

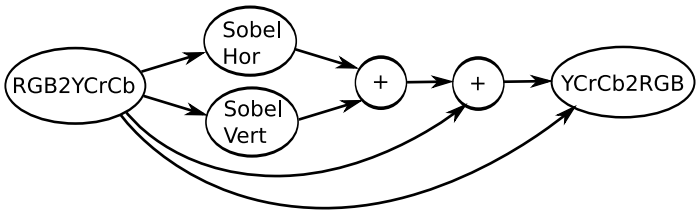


Figure 7: Sobel filter for edge quality enhancement

The Gannet task description for this Sobel filter without run-time reconfiguration is straightforward:

```
(let
'(buf 'br (RGB2YCrCb (img-block)))
'(let
```

```
(buf 'bsh (Sobel-Hor br))
(buf 'bsv (Sobel-Vert br))
)
'(buf 'ba1 (add (stream 'bsh) (stream 'bsv)))
'(buf 'ba2 '(add br (stream 'ba1)))
'(loop
(YCrCb2RGB (stream 'br) (stream 'ba2))
)
)
```

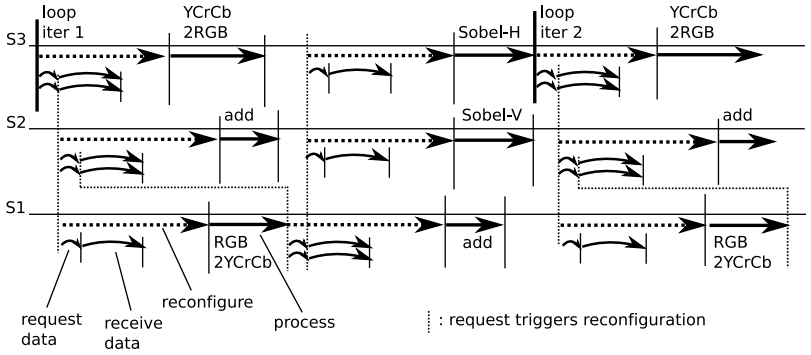
We now consider dynamic reconfiguration assuming a system with three reconfigurable cores consisting of embedded FPGA fabric (Figure 7). The horizontal and vertical Sobel convolutions are implemented on separate cores to allow them to be performed in parallel. A straightforward implementation of run-time reconfiguration, using the commands discussed in the section "Dynamic Service Configuration", is achieved as follows:

```
(let
'(buf 'br (RGB2YCrCb (img-block)
(S1-reconf 'RGB2YCrCb  '(config 'RGB2YCrCb 's1)) ))
'(let
(buf 'bsh (Sobel-Hor br
(S2-reconf 'Sobel-Hor '(config 'Sobel-Hor 's2)) ))
(buf 'bsv (Sobel-Vert br
(S3-reconf 'Sobel-Vert '(config 'Sobel-Vert 's3)) ))
)
'(buf 'ba1 (add
(stream 'bsh) (stream 'bsv)
(S1-reconf 'add '(config 'add 's1))
))
'(buf 'ba2 '(add br (stream 'ba1)
(S2-reconf 'add '(config 'add 's2))
))
'(loop
(YCrCb2RGB (stream 'br) (stream 'ba2)
(S3-reconf 'YCrCb2RGB  '(config 'YCrCb2RGB 's3))
)
)
)
```

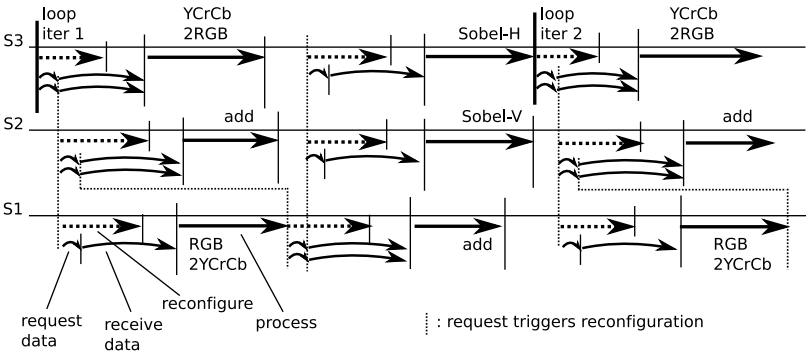
As can be seen from the code, it is sufficient to simply add the reconfiguration call as an additional argument to the original call, e.g.

```
'(buf 'br (RGB2YCrCb (img-block)
    (S1-reconf 'RGB2YCrCb  '(config 'RGB2YCrCb 's1)) ))
```

This works because the Gannet service manager will always evaluate all unquoted arguments of a service call. Consequently, the actual service core will not be called until the reconfiguration is finished. The return value of the reconfiguration command is ignored as the service core only reads the return values of its required arguments. The reconfiguration does not interfere with the streaming data processing because the buffers are managed by the Gannet service manager and are independent of the service core configuration. Because of the pipelined processing model, requesting and receiving data, computation and reconfiguration can occur concurrently, as illustrated in the following timing diagrams (Figure 8):



(a) Reconfiguration time is longer than data transfer/processing time



(b) Reconfiguration is shorter than data transfer/processing time

Figure 8: Timing diagrams for run-time reconfiguration of Sobel filter

The timing diagram shows reconfiguration as a straight, dotted arrow, processing as a straight, full arrow and data request and receipt as short and longer curved arrows. The times of the operations are only indicative. Case (a) shows the behaviour when reconfiguration times dominate; case (b) shows the behaviour when data transfer and processing times dominate, i.e. the block size of the image is much larger in case (b) than case (a). Both cases clearly illustrate the overlap between reconfiguration, data transfer and processing. This example is a nice illustration of the power of Gannet's programming model to support efficient scheduling for run-time reconfiguration.

8 Implementation of the Dynamic Reconfiguration Infrastructure

This section discusses the practical implementation of the Gannet DRI and programming tool chain. We discuss the assembler and compiler. We present results from a System-C cycle-approximate model of a Gannet SoC to illustrate the capabilities. We present a Verilog HDL implementation of the DRI [Vanderbauwhede et al., 2008] which demonstrates show that the Gannet DRI combines high performance and low overhead.

8.1 Assembler and Compiler

The compiler generates the code packets as explained in 5.5. Gannet programs are completely static from the point of view of code and variable addressing : all addresses for all instructions and variables are determined at compile time. This allows the SM to use direct lookups of code and variables. The main complexity is compiling the individual symbols: the symbol's Kind is determined from the context and the compiler needs to work out the scope and addresses for every symbol. The compiler is implemented in Haskell [Hudak et al., 1992a]using the Parsec parser combinator library [Leijen and Meijer, 2001]. The design is modular to make it easy to generate back-ends for a variety of platforms. Currently, the compiler exports Gannet bytecode, Gannet assembly language and Scheme.

The Gannet assembly language is the closest possible representation of compiled Gannet that is still human-readable. The Gannet assembler transforms the assembly into bytecode. The assembler (written in Perl) is used to explore new features of the system for which there is no compiler support.

8.2 Behavioural Model

We have implemented a behavioural model of the Gannet DRI using SystemC. The model allows cycle-approximate simulation of complete NoC-based SoCs with the Gannet DRI. The model is unclocked and uses the TLM library for transaction-level modelling, so internal pipelining is not simulated accurately, resulting in slightly pessimistic performance. The SM implementation (Figure 9) adheres closely to the Gannet machine model introduced in Section 4.3.

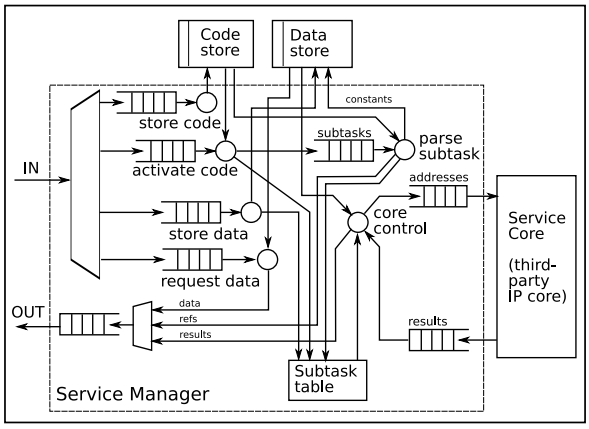


Figure 9: Gannet service manager implementation

The main differences are explicit arbitration and a separate path for returning requested data, which is required to make streaming more efficient. Other features not explicit in Figure 2 are a separate store for data pertaining to active subtasks (i.e instead of keeping the information as in the header of the request packet in the queue, the information is stored in a RAM for faster access and reduced contention. Furthermore, an FSM controls the interface between the SM and the core using a *core status* register: the core can be in one of four states *idle*, *ready*, *busy*, *done*. The ready state indicates that the SM has finished marshalling data; in the *busy* state the *core* is processing; in the *done* state the SM dispatches the result and cleans up the finished task. This process is illustrated in Figure 11 which shows the waveforms of the core status for a simulation of a random generated task running on a 16-core Gannet SoC providing matrix operations (Figure 10).

The task consists of the following matrix operations on 8x8 matrices:

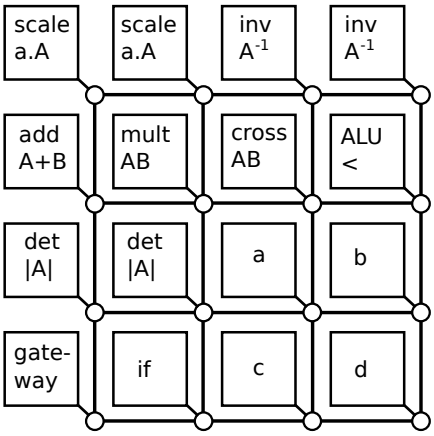


Figure 10: Gannet SoC providing matrix operations

```
(madd
    (cross (scale '0.5 (inv
        (if (< (det (a)) '0) '(mmult (a) (c)) '(mmult (a) (d)) ))) (tran (a)))
    (cross (scale '0.5 (inv
        (if (< (det (b)) '0) '(mmult (b) (d)) '(mmult (b) (c)) ))) (tran (b)))
) (madd
(cross (scale '0.5 (inv
                (if (< (det (a)) '0)
                    '(mmult (a) (c))
                    '(mmult (a) (d))
                )))
        (tran (a)))
    (cross (scale '0.5 (inv
                (if (< (det (b)) '0)
                    '(mmult (b) (d))
                    '(mmult (b) (c))
                )))
        (tran (b)))
)
```

We can observe that the if control service requires very few cycles thanks to the defered evaluation and redirection; we can also observe the parallel computation of the determinant and matrix inversion.

8.3 FPGA Implementation

The Gannet DRI is a complex system and has not yet been completely implemented in hardware. However, we have implemented an FPGA prototype of the Gannet Service Manager in Verilog 2001 targeting the Xilinx Virtex-II Pro XC2VP30 [Vanderbauwhede et al., 2008]. The results are summarised in Table 1 on page 28.

Optimisation Goal	Area	Speed
Slice count	822	917
BRAMs	14	8
Max. clock speed	170MHz	242MHz

Table 1: Gannet Service Manager synthesis results

We have also implemented our Quarc NoC switch [Moadeli et al., 2008b, Moadeli et al., 2008a] as well as a conventional mesh NoC switch as reference. The synthesis results for a flit size of one 32-bit word are presented in Table 2 on page 29. We can see that the SM slice counts are of the same order as the Quarc switch and considerably smaller than the mesh NoC switch. Note that the slice count for the switch does not include the transceiver. For a high-performance NoC, several virtual channels are required and the flit size will have

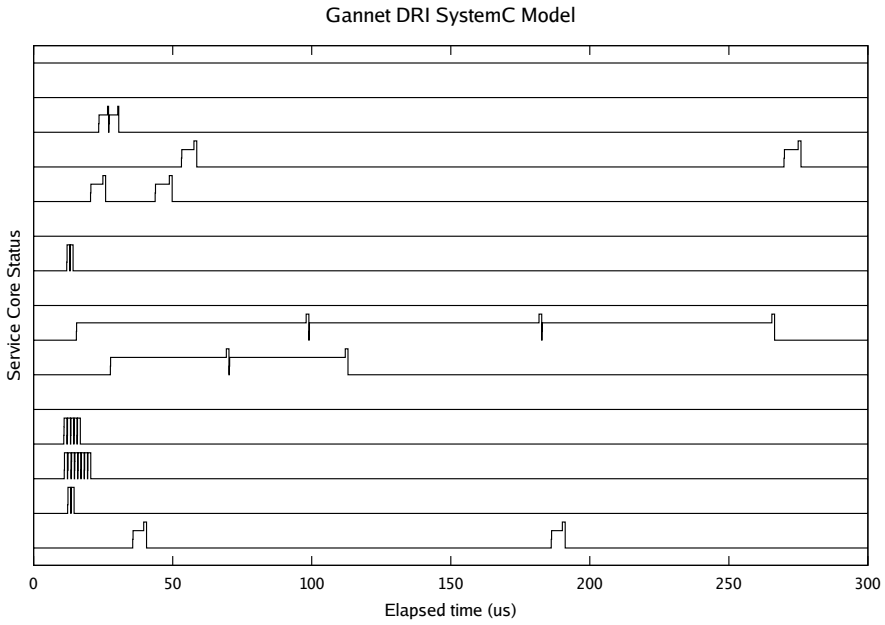


Figure 11: Core status transitions for a 16-core Gannet SoC running an arbitrary task

to be at least four words. Consequently, the area of the SM circuit will generally be a fraction of the NoC switch+transceiver area.

#Virtual Channels	Slice count	
	Quarc	Mesh
0	1,141	1,993
2	1,558	2,663
4	2,401	4,008

Table 2: Quarc NoC switch synthesis results

The functionally of the design has been extensively verified using the ISIM simulator provided with the ISE toolkit. The simulation results have been presented in [Vanderbauwhede et al., 2008]; the most important observation is that the SM requires a only 20 to 50 cycles (depending on the number of arguments) to activate a task.

8.4 Performance Evaluation

The performance of the Gannet DRI was evaluated with a focus on pipelined streaming data processing. We compare the performance with a reference implementation of a single-core system. Our main focus is to evaluate the overhead of the Gannet DRI in terms of reduction of throughput.

Choice of Application Domain

As the Gannet DRI is a generic framework, it is difficult to assess its performance without reducing the results to a particular application domain. We note however that obviously the overhead of the DRI will be determined by the number of cycles required for processing of data by the core and transferring of data by the Network-on-Chip. As our work is concerned with coarse-grained reconfigurable systems, it would be unfair to assess the performance of the DRI using fine-grained cores such as ALUs. On the other hand, for large data blocks and very computationally-intensive cores, the overhead will obviously be negligible. With these provisos, we decided to investigate the performance for a key class of applications, i.e. operations on 4×4 and 8×8 matrices of 32-bit floating-point numbers as used in many image processing applications. Image blocks of 8×8 pixels are commonly used in MPEG compressed video streams as the key transform, the 2D discrete cosine transform,

works on blocks of that size; the H.264 standard uses an integer transform on blocks of 4×4 pixels. However, we did not assume that a single core performs a complete frame compression or even a complete transform, as these would consume many cycles. Rather, we assumed that the cores perform fundamental matrix operations such as addition, multiplication and transposition. The cycle counts for these operations were obtained from software implementations in C++ compiled using gcc, optimising for speed. We used following rounded-down numbers (Table 3):

operation	**matrix size**	
	4×4	8×8
addition ($A+B$)	80	240
subtraction ($A-B$)	80	240
cross product ($A \otimes B$)	80	240
dot product ($A.B$)	320	1024
scaling ($a.B$)	80	240
transposition A^T	32	128

Table 3: Cycle counts for floating-point matrix operations

As Gannet supports variable numbers of argument, the first four operations can have from 2 to 4 arguments and the total cycle counts are adjusted assuming repeated binary operations. It should be noted that the nature of the operations and the exact number of cycles is of limited importance. The aim is to have a realistic set operations with realistic timings.

Choice of Network on Chip

The Gannet DRI makes no specific assumptions on the Network-on-Chip used in the SoC. For the simulations we have use cycle counts obtained from a Verilog implementation of our own Quarc Network-on-Chip architecture [Moadeli et al., 2008b, Moadeli et al., 2008a]. The Quarc is a NoC with deterministic routing and wormhole switching. The link width is equal to the word size used in the Gannet machine, i.e. 32 bits. The NoC can transfer a single flit with 4 cycles overhead; we have simulated the performance for flits of 4 words (8 cycles/flit) and 1 word (5 cycles/flit). The Quarc NoC has a maximum diameter of N/4, we have assumed N=64.

Design of Experiment

We used a Monte-Carlo approach to evaluate the performance of the system over a wide parameter space. Valid expressions consisting of streaming matrix operations (e.g. `(mult (add (A) (tran (cross (B) (C)))) (scale 0.5 (D)))`) are generated and simulated using the parameter values listed above and the throughput is established. The distances between the cores are chosen randomly. The length of the slowest path is extracted, as is the total number of operations in the expression. The experiment was performed 500 times for both block sizes (4x4 and 8x8) and both flit sizes (1 word and 4 words).

Results

Figure 12 shows the number of operations per expression versus the depth of the slowest path. The figure serves to visualise the distribution of the expressions generated by the Monte-Carlo.

The performance versus the reference implementation is presented in Figure 13. The cycle count for the reference implementation is assumed to be the sum of the cycle counts for all operations in the expression (i.e. no additional overhead for memory transfer etc.).

It is obvious that for small expressions with short paths the advantage of using a multi-core system is small; for more complex tasks however the system performs increasingly well. It should also be clear that the performance of an asynchronous pipeline is determined by the slowest node in the pipeline. To illustrate this point, Figure 14 shows performance results for expressions consisting of operations with a cycle count of 3072, i.e. the slowest operation in the set (a multiplication of four 8×8 matrices).

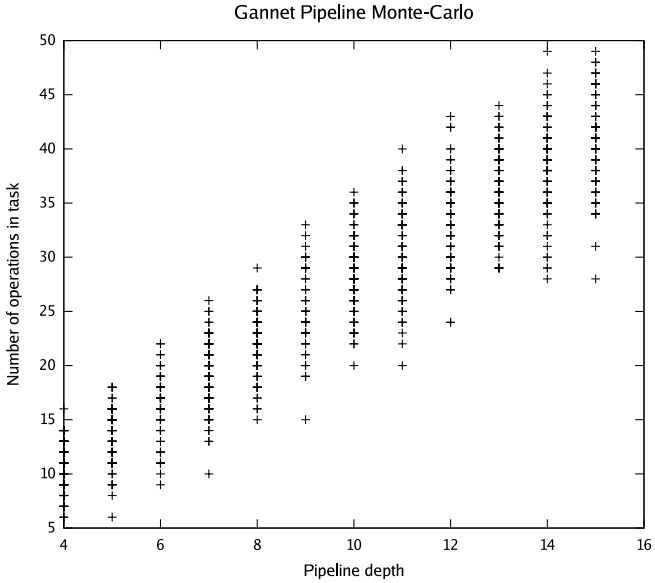


Figure 12: Number of operations per expression versus depth of the slowest path

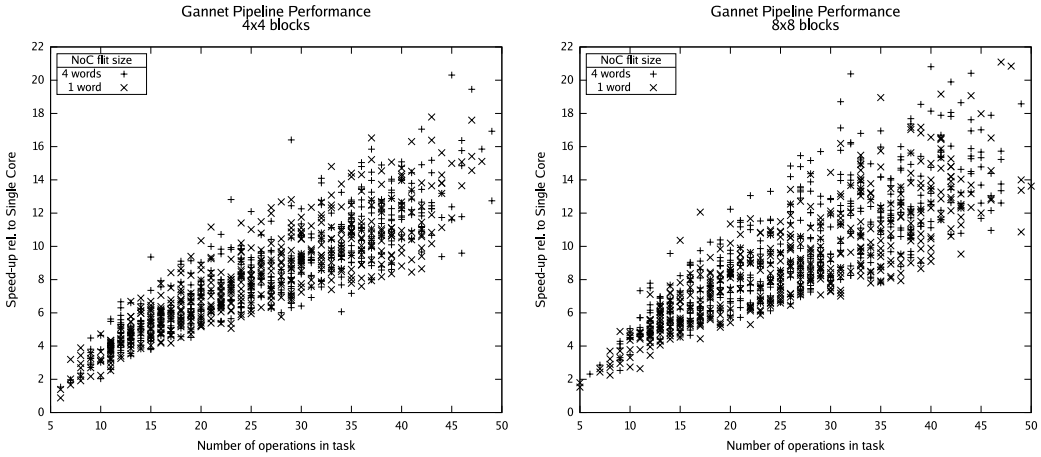


Figure 13: Performance of Gannet SoC vs serial reference implementation

Finally, the overhead of the Gannet DRI is presented in Figure 15. The overhead is determined by computing the throughput for an asynchronous pipeline without DRI, i.e. a static pipeline, and subtracting this figure from the simulated value. The throughput for an asynchronous pipeline is determined by the slowest process and defined as the size of the processed data block divided by the delay between blocks. An asynchronous pipeline needs a handshaking mechanism to indicate to the upstream process that it can send data. Consequently, the components of the delay are $\Delta t = \Delta t_{req} + \Delta t_{data} + \Delta t_{proc}$. The DRI adds an overhead for activating the task and dispatching the data: $\Delta t_{DRI} = \Delta t_{act} + \Delta t_{disp}$. Thus the relative overhead is given by $\Delta t_{DRI}/\Delta t$. We plot this overhead versus the throughput in millions of operations per second (Mop/s) assuming the system runs at 100 MHz.

It is clear from these results that the overhead of the Gannet DRI is generally very small. Aggregating the data points into a cumulative distribution (Figure 16) we see that for 4×4 blocks, the median overhead is 4.0% with the worst case overhead <10%, whereas for 8×8 blocks the median overhead is 1.5% with the worst case overhead <5% .

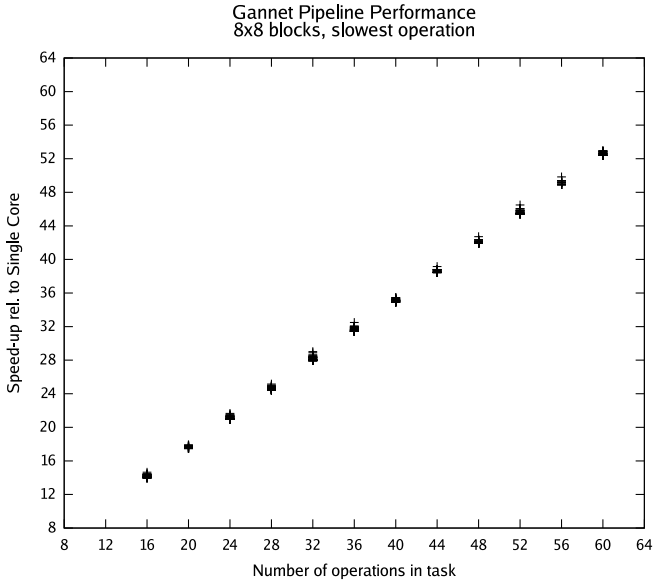


Figure 14: Performance of Gannet SoC vs serial reference implementation, for slowest operation

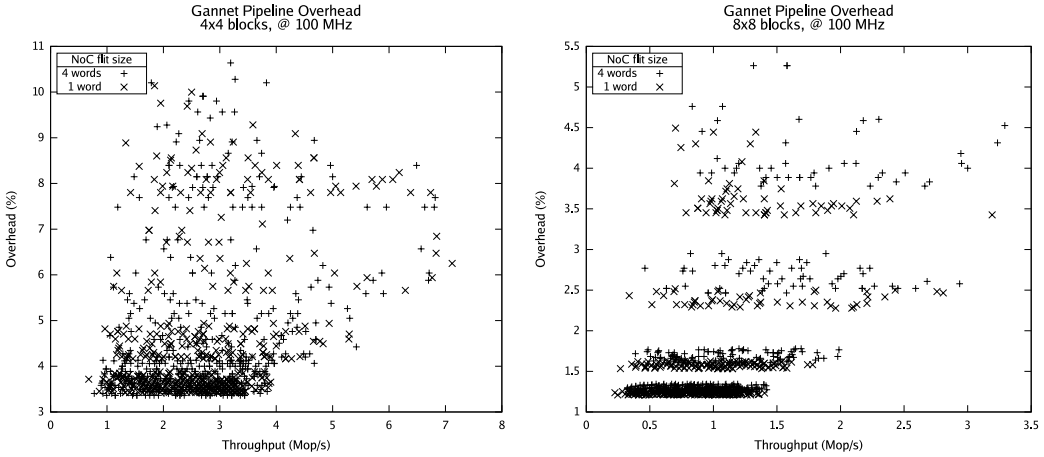


Figure 15: Overhead of the Gannet DRI vs throughput

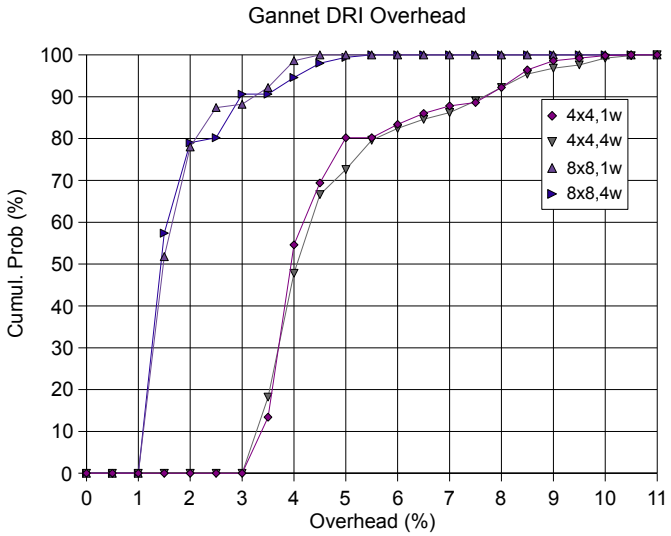


Figure 16: Cumulative distributions of Gannet DRI overhead

9 Conclusion and Future Work

In this chapter, we have presented the Gannet Dynamic Reconfiguration Infrastructure, a framework that provides high-level programming for Dynamically Reconfigurable SoCs. Our solution allows IP core-based Heterogeneous Multicore SoCs to be programmed using a high-level language whilst preserving the full potential for parallelism and dynamic reconfigurability inherent in such a system. We have demonstrated that the Gannet DRI provides full high-level dynamic reconfiguration capabilities with a small overhead both in area and performance.

There are three main directions for our future work. These three avenues will be explored in parallel.

The first strand is to improve the efficiency and flexibility of the system. In the short term, this means adding support for multi-threaded cores and implementing the proposed dynamic reconfiguration mechanism. In the longer term this means exploring data-driven operation as an alternative to the current demand-driven control.

The second strand is to realise the integration of a full Gannet system on FPGA, using the Quarc NoC as the communication medium. A particular challenge is the tight integration of the SM with an embedded microprocessor for efficient implementation of global control services.

The third strand is to investigate the potential of the Gannet DRI as a distributed operating system for IP core-based Heterogeneous Multicore SoCs. This work will focus on task management features such as rescheduling, prioritised scheduling, interruption and cancellation as well as adding a policy-based run-time service re-allocation mechanism. These features will allow the Gannet DRI to be used as an infrastructure for adaptive hardware systems.

10 Key Terms and Definitions

Dynamic Reconfiguration Infrastructure (DRI): A System-on-Chip infrastructure to allow dynamic (i.e. run-time) reconfiguration of data paths and computational functionality in Heterogeneous Multicore Systems-on-Chip.

Demand-driven computational model: A model of computation based on reduction of the call tree of a functional program. This model supports parallel and concurrent computations.

Functional programming: A programming paradigm that treats programs as trees of functions, emphasising function application rather than sequential imperative statements. Examples of functional languages are Haskell, ML, Scheme.

Network-on-Chip: An on-chip communication medium using a network of switches to route data between components of the system.

Coarse Grained Reconfigurable Architecture (CGRA): A System-on-Chip (SoC) architecture which allows to reconfigure parts of the system at a coarse granularity, i.e. at the granularity of several instructions or gates.

Heterogeneous Multicore System-on-Chip: A System-on-Chip (SoC) with multiple non-identical processing cores.

Gannet: The Gannet (*Morus Bassanus*) is a seabird and is the largest member of the gannet family, *Sulidae*. It can be found around the coasts of Scotland, with the largest colonies on the Bass Rock.

Appendix: Gannet-C by Example: Stream Matching

As a possible high-level language we propose Gannet-C, a C-style-syntax language which essentially provides a layer of syntactic sugar and a static type system to Gannet. Apart from familiar C/C++ constructs, the most important syntactic constructs provided are `foreach`, which executes the body of the statement in parallel; `merge` which allows to merge multiple streams into one; and the `Stream` and `Buf` pseudo-objects to model data streams. A function of type `Stream<T>` returns a stream of type `T`; a buffer `Buf<T>` buffers a stream on instantiation and provides the accessor methods `stream()` and `eos()`. Gannet-C also provides two types of block structures: `par {}` and `seq {}`, for parallel resp. sequential evaluation. By default, all statements are evaluated in parallel.

The code below illustrates the key Gannet features (parallel evaluation, pipelining) using as example a program that takes a data stream and matches each block of data in the stream against one or more profiles (a "profile" is a set of patterns that should occur in the data block). We assume that the system has a service getData which provides the function get_data() (similar to a class method). Calling this function returns the next block of data from an input stream identified by a descriptor. The system also has a service scoreData which provides score_data(), the scoring function.

```
tuple ScoreTup { // like struct
  Data data;
  Score score;
};

// [more declarations]
service getData {
  Stream<Data> get_data(int iodesc);
  // [more declarations]
}
service scoreData {
  ScoreTup score_data(Data data, Prof prof);
}
// Gannet function, not a service
void report_score(ScoreTup) {
  // code here
}
foreach int i in (0..3) {
    Buf<Data> b1[i]=get_data(i);
}
Buf<Data> buf2 = merge {
  foreach int i in (0..3) {
      b1[i].stream();
  }
}
Buf<ScoreTup> b3=score_data(b2.stream(),prof1);
Buf<ScoreTup> b4=
    if (b3.score<thresh) {
      report_score(b3);
      score_data(b3.stream().data,prof2);
    } else {
      b3.stream();
    }
while (!b4.eos()) {
  report_score(b4.stream());
}
```

The functionality of the program is as follows: the first foreach loop fills the buffer array b1 with values from 4 input streams; the merge block merges the input streams into b2; b2 is scored against prof1 and the results, i.e. the score and the block scored, are transferred to b3. If the score in b3 is below a certain threshold, the stream from b3 is scored against prof2 and in parallel the score from b3 is reported, otherwise the data are transferred to b4. The complete process is controlled from the last while loop which reports the score.

References

[Amamiya and Taniguchi, 1990] Amamiya, M. and Taniguchi, R. (1990). Datarol: a massively parallel architecture for functional languages. In *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, pages 726–735.

[Ashcroft, 1986] Ashcroft, E. A. (1986). Dataflow and eduction: data-driven and demand-driven distributed computation. pages 1–50.

[Bray et al., 2008] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition).

[Butts, 2007] Butts, M. (2007). Synchronization through communication in a massively parallel processor array. *IEEE Micro*, 27(5):32–40.

[Cocco et al., 2004] Cocco, M., Dielissen, J., Heijligers, M., Hekstra, A., Huisken, J., Hive, S., and Eindhoven, N. (2004). A scalable architecture for LDPC decoding. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3.

[Giraud-Carrier, 1994] Giraud-Carrier, C. (1994). A reconfigurable dataflow machine for implementing functional programming languages. *ACM Sigplan Notices*, 29(9):22–28.

[Goossens et al., 2005] Goossens, K., Dielissen, J., and Radulescu, A. (2005). Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421.

[Gough, 2001] Gough, J. (2001). *Compiling for the .Net Common Language Runtime (CLR)*. Pearson Education, 1rst edition.

[Guerrier and Greiner, 2000] Guerrier, P. and Greiner, A. (2000). A generic architecture for on-chip packet-switched interconnections. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 250–256, New York, NY, USA. ACM.

[Huang et al., 2008] Huang, S., Hormati, A., Bacon, D., and Rabbah, R. (2008). Liquid Metal: object-oriented programming across the hardware/software boundary. *ECOOP 2008–Object-Oriented Programming*, pages 76–103.

[Hudak et al., 1992a] Hudak, P., Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M., Hammond, K., Hughes, J., Johnsson, T., et al. (1992a). Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM Sigplan Notices*, 27(5):1–164.

[Hudak et al., 1992b] Hudak, P., Jones, S. P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. (1992b). Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164.

[Kelsey et al., 1998] Kelsey, R., Clinger, W., and (Editors), J. R. (1998). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76.

[Kogel et al., 2005] Kogel, T., Haverinen, A., and Aldis, J. (2005). OCP TLM for Architectural Modeling (white paper).

[Koo et al., 2007] Koo, J., Fernández, D., Haddad, A., and Gross, W. (2007). Evaluation of a high-level-language methodology for high-performance reconfigurable computers. In *Application-specific Systems, Architectures and Processors*, pages 30–35.

[Lampinen et al., 2006] Lampinen, H., Perala, P., and Vainio, O. (2006). Design of a scalable asynchronous dataflow processor. pages 85–86.

[Lanuzza et al., 2007] Lanuzza, M., Perri, S., and Corsonello, P. (2007). MORA - A New Coarse Grain Reconfigurable Array for High Throughput Multimedia Processing. In *Proc. International Symposium on Systems, Architecture, Modeling and Simulation,(SAMOS07)*.

[Leijen and Meijer, 2001] Leijen, D. and Meijer, E. (2001). Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht.

[Marescaux et al., 2004] Marescaux, T., Nollet, V., Mignolet, J., Bartic, A., Moffat, W., Avasare, P., Coene, P., Verkest, D., Vernalde, S., and Lauwereins, R. (2004). Run-time support for heterogeneous multitasking on reconfigurable SoCs. *Integration, the VLSI journal*, 38(1):107–130.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195.

[Milner et al., 1990] Milner, R., Tofte, M., and Harper, R. (1990). *The definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[Mirsky and DeHon, 1996] Mirsky, E. and DeHon, A. (1996). MATRIX: a reconfigurable computing architecture with configurableinstruction distribution and deployable resources. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 157–166.

[Mishra et al., 2006] Mishra, M., Callahan, T., Chelcea, T., Venkataramani, G., Goldstein, S., and Budiu, M. (2006). Tartan: evaluating spatial computation for whole program execution. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 163–174. ACM New York, NY, USA.

[Moadeli et al., 2008a] Moadeli, M., Vanderbauwhede, W., and Shahrabi, A. (2008a). A Performance Model of Communication in the Quarc NoC. In *14th IEEE International Conference on Parallel and Distributed Systems, 2008. ICPADS'08*, pages 908–913.

[Moadeli et al., 2008b] Moadeli, M., Vanderbauwhede, W., and Shahrabi, A. (2008b). Quarc: A novel network-on-chip architecture. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 705–712, Washington, DC, USA. IEEE Computer Society.

[Najjar et al., 2003] Najjar, W., Bohm, W., Draper, B., Hammes, J., Rinker, R., Beveridge, J., Chawathe, M., and Ross, C. (2003). High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69.

[Noguera and Badia, 2006] Noguera, J. and Badia, R. (2006). System-level power-performance tradeoffs for reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(7):730–739.

[Nollet et al., 2003] Nollet, V., Coene, P., Verkest, D., Vernalde, S., and Lauwereins, R. (2003). Designing an operating system for a heterogeneous reconfigurable soc. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 174.1, Washington, DC, USA. IEEE Computer Society.

[Nollet et al., 2005] Nollet, V., Marescaux, T., Avasare, P., and Mignolet, J. (2005). Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 234–239. IEEE Computer Society Washington, DC, USA.

[Nollet et al., 2004] Nollet, V., Marescaux, T., Verkest, D., Mignolet, J., and Vernalde, S. (2004). Operating-system controlled network on chip. In *Proceedings of the 41st annual conference on Design automation*, pages 256–259. ACM New York, NY, USA.

[Pham and Aipperspach, 2006] Pham, D. C. and Aipperspach, T. (2006). Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196.

[Purohit et al., 2008] Purohit, S., Chalamalasetti, S. R., Margala, M., and Corsonello, P. (2008). Power-Efficient High Throughput Reconfigurable Datapath Design for Portable Multimedia Devices. In *International Conference on Reconfigurable Computing and FPGAs (Reconfig08)*, pages 217–222.

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

[Scholz, 2003] Scholz, S. (2003). Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(06):1005–1059.

[Singh et al., 2000] Singh, H., Lee, M., Lu, G., Kurdahi, F., Bagherzadeh, N., and Chaves Filho, E. (2000). MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE TRANSACTIONS ON COMPUTERS*, pages 465–481.

[Sussman and Steele, 1975] Sussman, G. J. and Steele, G. L. (1975). An interpreter for extended lambda calculus. Technical report, Cambridge, MA, USA.

[Taylor et al., 2004] Taylor, M., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A., Lee, W., Miller, J., et al. (2004). Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 2–13.

[Treleaven et al., 1982] Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. (1982). Data-driven and demand-driven computer architecture. *ACM Comput. Surv.*, 14(1):93–143.

[ul Abdin and Svensson, 2008] ul Abdin, Z. and Svensson, B. (2008). Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocessors and Microsystems*, In Press, Uncorrected Proof:–.

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

[Vanderbauwhede, 2006b] Vanderbauwhede, W. (2006b). 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.

[Vanderbauwhede, 2007] Vanderbauwhede, W. (2007). Gannet: a Scheme for Task-level Reconfiguration of Service-based Systems-on-Chip. In *Proceedings 2007 Workshop on Scheme and Functional Programming, September 30th, 2007, Freiburg, Germany*, pages 129–137. N/A.

[Vanderbauwhede, 2008] Vanderbauwhede, W. (2008). A Formal Semantics for Control and Data flow in the Gannet Service-based System-on-Chip Architecture. In *The International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2008*. N/A.

[Vanderbauwhede et al., 2008] Vanderbauwhede, W., Mckechnie, P., and Thirunavukkarasu, C. (2008). The Gannet Service Manager: A Distributed Dataflow Controller for Heterogeneous Multi-core SoCs. In *Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on*, pages 301–308.

[Veen, 1986] Veen, A. (1986). Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 18(4):365–396.

[Vegdahl, 1984] Vegdahl, S. (1984). A survey of proposed architectures for the execution of functional languages. *IEEE transactions on computers*, 100(33):1050–1071.

[Wilkinson, 1996] Wilkinson, B. (1996). *Computer architecture: design and performance*, chapter 10, pages 434–437. Prentice-Hall, 2nd edition.