Run-Time Integration of Reconfigurable Video Processing Systems

Pete Sedcole, *Member, IEEE,* Peter Y. K. Cheung, *Senior Member, IEEE,*
George A. Constantinides, *Member, IEEE,* and Wayne Luk, *Member, IEEE,*

***Abstract*— Embedded systems in Field-Programmable Gate Arrays can be customised and adaptive if assembled from modular components at run-time. This paper examines realising run-time system assembly by extension of platform-based design. Two major challenges are addressed in this paper. Firstly, the design of a reconfigurable platform architecture suitable for run-time system assembly is described. Different systems are constructed by integrating the platform architecture with different modular components, which employ the communication infrastructure supplied by the platform in order to interact. Secondly, where on-chip communications channels use shared media, we propose techniques for modelling the inter-module communication behaviour based on statistical time division multiplexing. The proposed techniques enable system designers to guarantee that logical communication requirements between the adjunct modules can be satisfied by the infrastructure. An in-depth analysis is presented, and then verified with cycle-accurate simulations for the Sonic-on-a-Chip reconfigurable platform for real-time video applications.**

I. Introduction

Through continuous effort and innovation, the semiconductor industry has maintained an unrelenting increase in VLSI transistor density over the last several decades. The pace of the transistor density increase has not been matched by corresponding advances in designer productivity, causing design costs to spiral upwards and threatening the continuation of the semiconductor roadmap [1]. To ameliorate this situation, new design methodologies have emerged to exploit a greater degree of design reuse, primarily through extensive, planned reuse of design focused around a standardised bus architecture, an approach known as platform-based design [2], [3]. Derivative systems are built by integrating a basic platform architecture kernel with a specific set of modules, which interact using the communication infrastructure defined by the kernel. Derivatives have a lower integration design effort than ad hoc block-based reuse, mainly due to the reduced design complexity of intermodular communication. While the initial development effort of the platform kernel may be high, this can be amortised over a number of derivatives resulting in overall lower design cost.

The user-exposed transistor density of Field-Programmable Gate Arrays (FPGAs) inevitably lags behind that of Application Specific Integrated Circuits (ASICs). Nevertheless, modern FPGAs are now reaching gate counts where design productivity is becoming a bottleneck, leading to the application of platform-based design techniques to reconfigurable systems [4]. However, a significant difference exists between FPGA and ASIC platform-based design; whereas ASIC derivative designs are necessarily fixed at design-time, the reconfigurability of FPGAs engenders the possibility of derivative designs generated and integrated at run-time. This we have termed *late integration* [5]. One of the most significant challenges in achieving late integration is the resolution of logical intermodular communication channels onto finite on-chip interconnect resources. The advantage of late integration is that it enables an instance of a system to be customised to the environment in which it is deployed, and adaptable to changes in the environment. Increasing the customisation of reconfigurable derivatives partly mitigates the reduced performance of FPGA-based designs compared to ASIC implementations.

For example, consider a video processing system for intelligent tracking surveillance cameras deployed in two situations: one monitoring an underground car parking garage and the other a busy street. The type and quantity of scene activity in the two situations are quite different; moreover, the lighting and conditions in the street scene are time-variant. Depending on the instantaneous operating conditions, different algorithms are required for optimal results; an ASIC derivative must be generic enough to support all possible algorithms (whether or not a particular algorithm is ever invoked), whereas an automated reconfigurable platform can, by monitoring the environment, selectively instantiate the momentarily optimal algorithms for the conditions.

One can view platform-based design as the application of constraints to the design space in order to simplify the integration effort of derivative designs. In reconfigurable platform-based design employing late integration, further restrictions are made, particularly with regard to on-chip communication, such that the integration phase is simple enough to be automated. Therefore, it is necessary to determine which (conventional derivate) design tasks may be reasonably performed automatically at run-time. The remaining design tasks must then be incorporated into the development of the reconfigurable platform architecture.

A. Contributions

This paper provides three main contributions. Firstly, we examine the tasks in platform-based design, in order to identify changes to the design flow necessary to achieve a late integration scheme, and the resulting impact on the system architecture. Derivative integration is primarily a function

P. Sedcole, P. Y. K. Cheung and G. A. Constantinides are with the Dept. of Electrical & Electronic Engineering, Imperial College London. W. Luk is with the Dept. of Computing, Imperial College London.

of resolving communication between the adjunct modules which form the system. Traditionally this task is performed at design-time, in which case simulations or statistical techniques can be used to set the communication schedules. These are too computationally demanding to be performed at run-time. Therefore, we propose circumscribing the communication behaviour of the modules forming the system.

This proposal leads to the second contribution, which is the first FPGA-based platform architecture that has been designed specifically to support automatic derivative generation and integration at run-time. The architecture comprises a bus-based network which provides connectivity between a number of customisable processing element modules. Constraints are imposed on the internal design of modules to limit the communication behaviour they may exhibit.

Finally, to demonstrate that the proposed approach enables rapid resolution of intermodular communication, we present a detailed analysis of the communication system. It is shown that the mapping of algorithms to the architecture results in predictable behaviour, enabling real-time requirements to be guaranteed. The analysis is verified through the use of cycle-accurate simulations for several example systems.

B. Organisation of the Paper

Section II describes research related to the work in this paper. The reconfigurable platform architecture requirements and the exemplar template of Sonic-on-a-Chip[1] is the subject of Section III. Following that, an analysis of the on-chip communication behaviour is presented in Section IV. Closed-form expressions are derived to determine arbitration parameters and required minimum buffer sizes for a given system. Finally, Section V presents the results of simulations, verifying the validity of the communication analysis.

II. Related Work

Several architectures for on-chip communication have been developed previously. Bus-based architectures are conceptually the simplest, and much work exists on high-level system design using buses (for example [7], [8], [9], [10], [11]). Many of these approach communication as a synthesis task; Gasteier and Glesner describe a method for determining a static bus scheduling for transfers between communicating low-level processes [8], which works for fully deterministic traffic. In [7], abstract communication channels (characterised by average and peak data rates) are assigned to shared media using an allocation algorithm. More complex models of communication traffic include trace-based communication analysis [9], which aids the exploration of architectural design space [10]. Communication synthesis also includes bus topology exploration; in [11] several custom bus architectures are automatically generated, using a library of bus components. Communication synthesis specifically for reconfigurable devices has also been examined [12]. Platform-based design benefits from standardisation, and a number of on-chip microprocessor bus standards have emerged over the last several years, such as AMBA [13], CoreConnect [14] and Wishbone [15].

The fundamental disadvantage of microprocessor buses is their lack of scalability, which can only be partially mitigated with bus hierarchies. Such buses typically employ a form of priority-based arbitration which is ineffective when the bus bandwidth utilisation is high. A more scalable alternative to a bus-based approach is the implementation of a packet network on-chip (NoC) [16], [17]. In a network, many transactions can occur in parallel over shorter, faster and less capacitive wires, leading to higher overall bandwidths and lower energy costs. However, these advantages come at a price; additional hardware (including buffer memory) for packet routing, and increased complexity in the communication system. Static routing schedules may be determined at design-time if the network traffic is sufficiently deterministic. Dynamic routing may be supported with complicated routing hardware [18]. As an alternative, in [19] a discrete number of static schedules are determined, which are dynamically switched as required. In complicated systems, on-chip network traffic exhibits time-variant behaviour, which may be modelled using statistical methods [20]. This can then be used to determine buffer sizes and reduce simulation times. This is related to techniques of queueing theory, which have long been applied to telecommunication networks. Statistical approaches produce probabilistic results. However, embedded systems often have hard real-time requirements; moreover, if sufficient information is known about the system *a priori* a deterministic solution can be found.

Our architecture work occupies an interesting space in-between pure microprocessor buses and on-chip networks. The communication infrastructure is a linear array of buses, but the bus arbitration protocol chosen is statistical time division multiplexing (STDM) which enables high bandwidth utilisation. Since data are packetized in STDM, the movement of data has similarities to networks, particularly for data that traverses two or more buses. Moreover, there is a high degree of separation between computation and system-level data movement within each processing node, which is typical of a network. For a system to be assembled at run-time, the communication architecture must be designed without exact knowledge of the traffic that the architecture must carry. In other words, the communication architecture cannot be customised to a specific set of communication channels. Therefore deterministic synthesis and scheduling approaches are not applicable, while simulation or trace-based methods are impractical.

Run-time reconfiguration in FPGAs has been proposed for bus-based systems [21] as well as on-chip networks [22], [23]. In all these cases, the focus is on the (undoubtedly important) practicalities of reconfiguration and connectivity. However, it is the thesis of this paper that for such reconfigurable systems to guarantee functionality, the construction of reconfigurable platform architectures must consider communication performance requirements explicitly and by design.

[1]Note that the use of *Sonic* in this paper is historical; the precursor to Sonic-on-a-Chip was named Sonic by its inventors S. D. Haynes *et al.* [6]. In this paper Sonic does not in any way refer to the on-chip interconnect technology of Sonics Inc.

TABLE I
TASKS IN PLATFORM-BASED DESIGN

Development Stage	Conventional [2]	Reconfigurable
Platform	Hardware kernel	Hardware kernel I/O, clocks, test structures Floor-planning
Derivative	System design Functional verification I/O, clocks, test structures, power distribution Floor-planning Block implementation Assembly	Subsystem design Functional verification Block implementation Pre-assembly processing
Run-time		Environment analysis Assembly

III. A RECONFIGURABLE PLATFORM ARCHITECTURE

In this section the design of a reconfigurable platform architecture is examined. We first extend standard (ASIC) platform-based design with the requirements for reconfigurable platforms. This is followed by a description of our architectural template.

A. Architectural Requirements

For a platform architecture to support automatic, run-time derivative generation, the architecture must be developed further than in standard platform-based design. The creation of an integration platform comprises developing one or more hardware kernels which encapsulate the core common functionality of all the derivatives. A kernel includes buses, specialised component blocks, interface ports for attaching the 'virtual components' of derivative designs, central control and test functions. The kernel is a hard block of IP, although limited parameterisation is possible. A reconfigurable platform architecture includes kernel development; however, in order to make the run-time design effort low enough that it may be completed quickly and automatically requires limiting the degrees of freedom in the derivative designs. The platform development in the reconfigurable case therefore includes tasks that would normally be carried out in the development of derivatives, such as defining the clock tree and global floor-planning (see Table I).

Derivative design involves selecting virtual component modules required to complete the functionality of system, verification that the functionality meets specification, the implementation of all component blocks and final assembly. Conventionally, the derivative development phase is repeated several times, once for each specific derivative implementation. For a reconfigurable platform, a reduced set of design tasks can be achieved at design time. Rather than design and validation of the complete derivative system, a library of subsystems (each comprising several communicating virtual component blocks) is validated and implemented. Thus at run-time, the generation process is limited to extracting information about the environment, selecting and assembling together subsystems and setting programming parameters.

The most intensive integration task is validating that the system-level communication meets the requirements for the correct functioning of the system once assembly is complete. In ASIC development, this is usually achieved through the use of extensive simulations, trace-based methods, or statistical approaches such as those used in queueing theory. Computationally demanding approaches (such as simulations) are clearly impractical at run-time. Although statistical methods could be used, they have the disadvantage of producing probabilistic results: they cannot guarantee that hard real-time requirements will be met.

Instead, we propose imposing constraints on the communication infrastructure, protocols and virtual channels such that communication becomes predictable and analysable. During design time, the communication channels are characterised and parameterised, reducing the processing at run-time to simple calculations. This procedure is detailed in Section IV.

Note that the proposed approach precludes the use of standard microprocessor buses (for example, [13], [14]) which exhibit a level of flexibility that makes predicting communication behaviour problematic.

The physical design, performed at the platform development stage, involves the creation of a floor-plan in which the placement and routing of the hardware kernel, clock trees, I/O and the communication infrastructure are fixed. Note that the clock resources (such as wires and buffers) are already highly constrained in FPGA devices, however the clock trees must still be constructed by appropriately connecting and enabling the clock resources. The floor-plan must be flexible enough to allow for the instantiation of several modules, accounting for variation in number and (preferably) size. Fixed interface points are required to which modules are connected to the kernel structure; moreover, it is highly desirable that the communication infrastructure supports both inter-modular communication as well as transporting information between the modules and the kernel.

B. The Architectural Template

Having established in qualitative terms the requirements for a reconfigurable platform, we now briefly introduce a specific platform architectural template *Sonic-on-a-Chip*. The template is a generalised form of an architecture from which platform instances are distilled; its structure is illustrated in Figure 1. The template is an evolution of a board-level system developed by Haynes *et al.* which comprised multiple FPGAs. This original system was named Sonic [6] (later UltraSONIC [24]) hence the nomenclature 'Sonic-on-a-Chip' for the template. The targeted application domain is real-time video image processing.

The core of the template consists of a variable number of shared buses (named *SonicBuses*), connected by bridges. Customisable processing element virtual components (PEs) are attached to the SonicBuses at certain, fixed locations via socket interfaces. A series of *Chain buses* connect each PE to its adjacent neighbours, making use of physical locality to bypass the shared bus for fast local data transfers. Video data are processed as they stream through the processing elements; the Sonic processing subsystem is managed via a microprocessor-based control subsystem, which may additionally perform

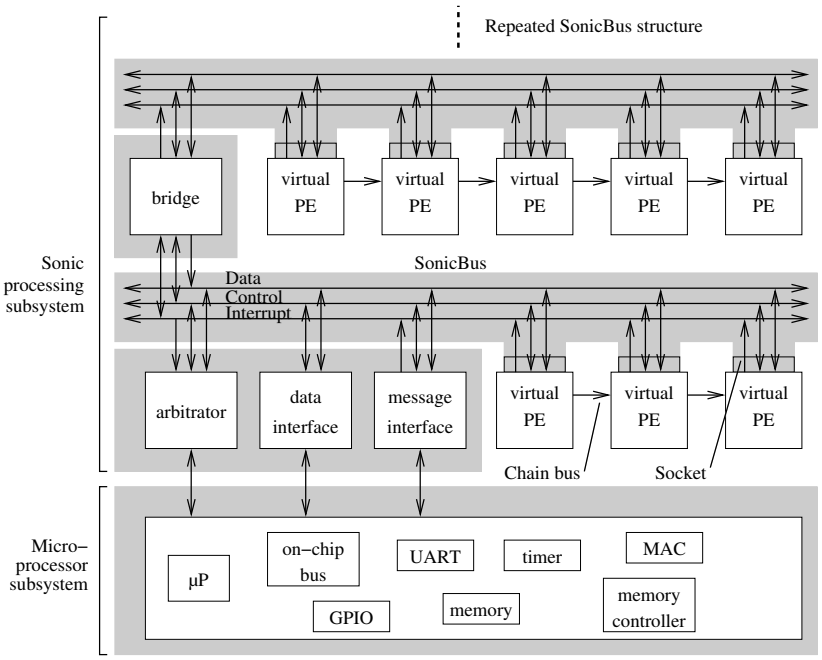


Fig. 1. The Sonic-on-a-Chip architectural template. The shaded areas form the static platform, derivatives are created by integration of customised virtual component processing elements.

other processing, control and I/O tasks, and resembles a conventional hardware kernel.

A specific platform is created from the template by designing the microprocessor subsystem and fixing the number of buses and the sockets on each bus; a process which is necessarily cognisant of the physical floor-plan of the whole system. Derivative systems are instantiated by selecting and attaching specific processing element modules to the SonicBuses via the socket interfaces. Internally, a PE comprises a router, an engine and memory / buffering as depicted in Figure 2. The engine processes data provided by the input stream buffer(s) and writes the resulting data to the output buffer(s). The engine design is fully user-defined; the remainder of the PE is fixed in design but has some limited scope for customisation, for example the number and size of the input and output buffers can be modified. Programmability of the engine is provided for through the use of engine registers, the purpose of which are, again, user-defined.

The router provides the interface between the input and output buffer stages and the inter-module interconnect and communication protocols. Communication in the Sonic subsystem is entirely source-driven: data are pushed from producer outputs to consumer inputs. Data-flow is managed through the use of the router control registers, which hold information on the destination module and port number for each of the output ports of the engine. The control registers also determine which (if any) of the input ports source data from the incoming Chain bus and which (if any) of the output ports write to the outgoing Chain bus.

The output buffers are simple FIFOs, however the input stream buffers are subtly modified so they can be used for data-reuse as well as normal buffering. This optimisation is particularly beneficial in FPGA designs where on-chip

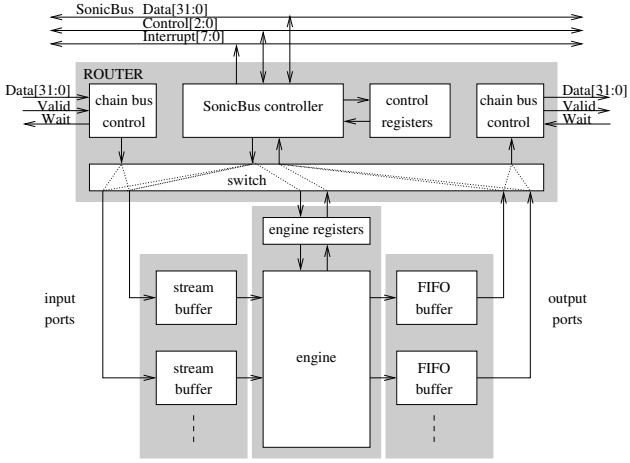


Fig. 2. The internal components and data-path of a processing element.

memory is significantly limited. As shown in Figure 3, data are streamed in serially to the input buffer as in a normal FIFO, while on the output side the engine supplies the address of a queued element relative to the front of the queue to read. A `Stall` signal is asserted if the address points to an empty location, and indicates that the engine operation should stall until the location is filled. As with normal FIFOs, data width conversion is possible. At each cycle, the engine-supplied `Advance[A:0]` signal will cause the queue to be shifted forward by the given number of positions.

Importantly, data can only be discarded from the front of the stream buffer queue. The stream buffer is therefore less flexible than a cache or a scratch pad memory, and the way in which data are serialised is important if the features of the stream buffer are to be exploited. The advantage of the stream

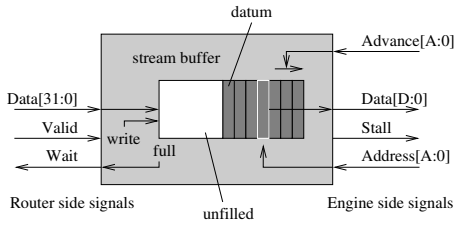


Fig. 3. The details of a stream buffer.

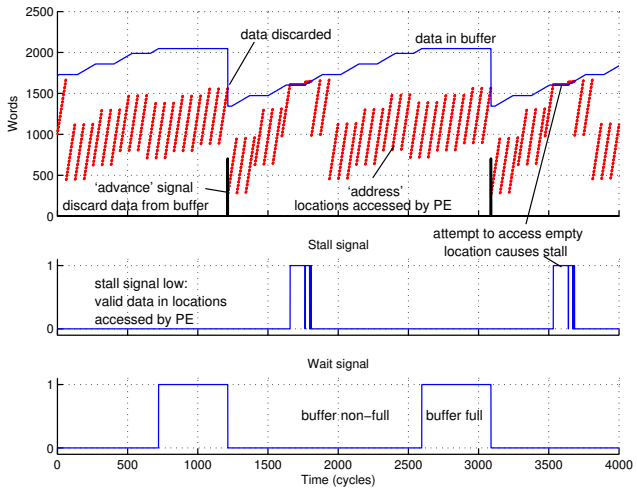


Fig. 4. The motion vector stream buffer behaviour. At the top, the graph shows the number of words in the buffer and the location accessed by the engine. If the location accessed does not hold valid data, the engine is stalled (middle graph). The wait signal is asserted when the buffer is full (lower graph).

buffer scheme is that, unlike more flexible methods, the scope of communication behaviour that can be exhibited is narrow and well-defined. This results in analysable and predictable communication patterns.

Example 1: Stream buffers in motion vector estimation. Consider a PE which performs block-based motion vector estimation, a computationally demanding operation which is used in MPEG video compression algorithms. The task is as follows: for a given square macro-block (typically 16×16 pixels) of a reference image, find the coordinates of the most similar block in a target image. To speed processing, the range of target locations to search is limited to a search window (say 44×44 pixels). Thus a motion vector estimation PE could be constructed with two input stream buffers (one for the reference macro-blocks, one for the search windows). Figure 4 is a graph of the state of the search window buffer in a simulation of a motion vector estimator PE which uses a three-step coarse-fine search algorithm. Note that since the search windows overlap, only some of the data are discarded from the buffer after performing each search. □

The trichotomy of the engine, router and buffering within each PE ensures that computation and inter-module communication are kept separate; this facilitates the design and reuse of the computational component (the engine). Moreover, this separation is fundamental to ensuring predictability in communication, which, as will be seen in Section IV, is a necessity in our analytical approach to the run-time integration of shared-medium communication channels.

C. SonicBus Communication Protocols

Two communication protocols are supported for transmission of information on the shared SonicBus. The first is in essence a statistical time division multiplexing (STDM) scheme: a series of consecutive bus cycles are allocated to a specific channel; if the channel becomes inactive during its allotted time (either by a lack of data to transmit or a lack of space to put the data at the consumer end) the bus is released early for re-arbitration. This is depicted in Figure 5. A channel is formed by transmission of data from an output buffer of one PE to an input stream buffer of another PE, thus the STDM scheme is used for the majority of data movement transactions in the Sonic subsystem. The second protocol is based on message passing, and used for reading from and writing to the router control registers and optional engine registers in each PE. This protocol is used infrequently, typically during initialisation or occasional monitoring of the status of a PE.

The routers within each PE are responsible for implementing the protocols, with the assistance of an arbitration unit (one for each bus). Within the unit, an arbitration table is programmed with an entry for each of the normal channels sharing the bus, as well as special entry for each PE, which is used when the PE sends a message. Each normal table entry includes the PE and port number of the producer side of the channel and the number of consecutive cycles allocated to the channel. The arbitrator begins by issuing a bus grant command to the producing PE and port for the programmed number of cycles. The appropriate PE router responds by issuing a stream data command which is detected by the receiving PE. Data are then streamed from the producer buffer to the consumer buffer until there is no more data, the receiving buffer is full, or the count is reached, at which point the producer issues a release command to pass control of the bus back to the arbitrator. The overhead of the STDM scheme is 3 cycles per channel transfer.

The SonicBus uses a 32-bit multiplexed bus with two extra control lines and an acknowledge signal. The control lines indicate the type of the information on the bus in the current cycle (address, data, or command), while the acknowledge signal is used by a receiving PE to indicate that the destination buffer either has space or is full. Interrupt lines are provided for message passing only.

The protocols are easily expanded over multiple buses by using buffering bridges. Each bridge includes queues for STDM channels and separate message queues. The arbitration unit for the secondary side bus (the bus further from the microprocessor control subsystem) is also built into the bridge. Each PE is assigned a module ID number which includes the bus on which it resides. The bridge behaves like any other PE, but picks up all traffic from either side of the bus which is destined for a PE on the other side of the bridge. The arbitration unit within the bridge is programmed using the standard message passing protocol.

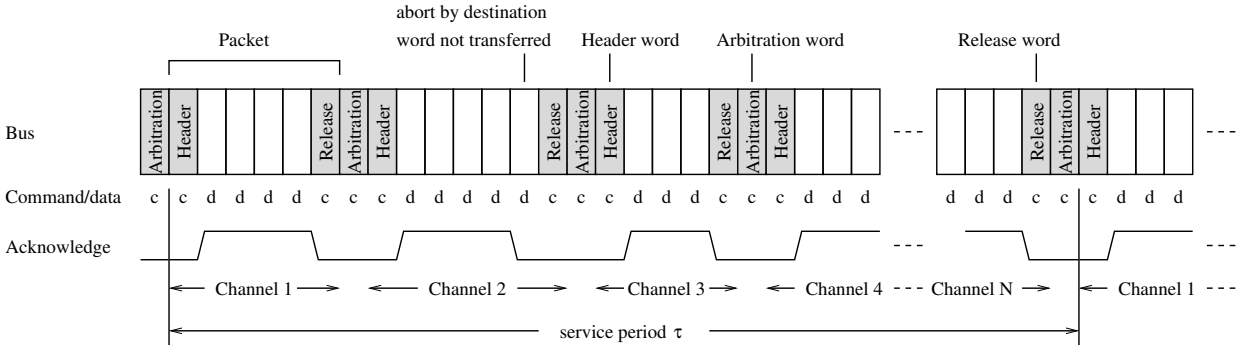


Fig. 5. Bus cycle-by-cycle view of statistical time division multiplexing.

IV. Communication Analysis

One of the most challenging issues in automatic run-time assembly of derivative systems is mapping communication channels to shared resources while ensuring performance targets are met. In this section we develop an analysis of the communication system based on the platform architectural template described above. The objectives of this analysis are, for each channel, to determine: (a) the arbitration parameters (the cycle count value), (b) the minimum required amount of buffering, and (c) an upper bound on the latency.

The analysis begins with a description of the necessary assumptions regarding the communication system, to which the Sonic-on-a-Chip architecture described above conforms. A first approximation is made by assuming that there is sufficient buffering throughout the system such that buffer saturation never occurs. The analysis is then modified to describe situations where buffer saturation does occur. The result is a method, summarised in Section IV-E, by which the system designer ensures that derivative designs constructed at run-time will achieve the desired performance at all times.

It should be noted that the analysis is not limited to Sonic-on-a-Chip, or even video processing, but is abstracted such that it may be applied to any communication system designed with similar constraints.

A. Scenario and Assumptions

We start with the assumption that the processing system is a process network comprising a number of processing nodes (PEs) connected by a series of communication channels, which is to be mapped to a system of buses connected by fully buffering bridges, as in the template. The features of a communication channel are depicted in Figure 6. A channel is defined by a continuous stream of data from the output of a producing node to the input of a consuming node across a shared communication medium. Assume that each node has been assigned to a bus. By using bridges which buffer data, the behaviour of each bus can be isolated and studied separately. We will ignore channels which are assigned to using the Chain bus connections, as they do not use shared media and thus are of no interest in this analysis. Therefore, for a particular bus we need to set the time-slot size for each channel to ensure throughput is met and determine the maximum latency and buffering required.

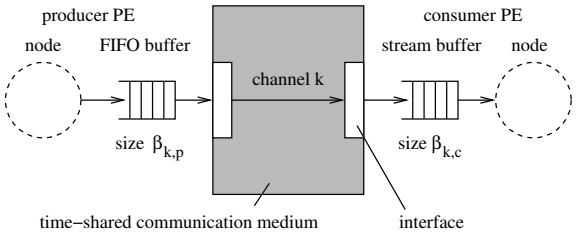


Fig. 6. An abstract communication channel k. Data are buffered on the producer side by a FIFO of depth $\beta_{k,\mathrm{p}}$ and on the consumer side by a stream buffer of depth $\beta_{k,\mathrm{c}}$.

In the analysis which follows, the processing nodes are assumed to have a common pattern of behaviour: One or more streams of data are stored in input buffers; the engine performs a number of accesses on the stored data and outputs results to the output buffer; input data which are no longer needed are discarded. This pattern is repeated indefinitely, such that the processing node has a base-line periodicity. Note that in some cases a node may exhibit different input and output periodicity; for example, a node which computes a histogram of the intensity values of an image may access and discard pixels one at a time (input periodicity of 1 pixel), whereas the results are only presented to the output buffer once per frame (output periodicity of 1 frame).

Table II lists some sample video processing algorithms and shows how they may be parameterised in pixels. All algorithms (with the exception of the motion vector estimator) process non-interlaced raster-scanned images of height r and width c.

B. First Approximation

A first approximation analysis is formulated by making several simplifying assumptions, including unlimited buffer sizes, and a constant rate of data production and consumption. The aim is to determine, for a given mapping of channels to a bus, what size time-slot to allocate to each channel, and the actual required amount of channel buffering. In the next subsection the assumptions are removed and the analysis extended to this more complex case.

Consider a bus with maximum bandwidth Γ supporting N channels. Each channel k has a required average throughput ϕ_k, and is allocated ω_k consecutive bus cycles for each data transfer (at one word per cycle) excluding the STDM overhead

TABLE II

CHARACTERISTICS OF VIDEO PROCESSING ALGORITHMS

(IMAGE SIZE $r \times c$)

Algorithm	Periodicity		Advance	Stored data
	Input	Output		
Window function (5×5 window) e.g. median filter	25	1	1	$4 \times c + 5$
2D convolution (3×3 kernel)	9	1	1	$2 \times c + 3$
Histogram (3 colour channels, 256 points)	1	768	1	1
1D DFT (parallel)	c	c	c	c
Motion vector estimation (16×16 macroblock)	6912	1	704 / 256	1936 / 256

TABLE III

NOTATION USED IN THE ANALYSIS

Symbol	Units	Description
N		Number of channels
M		Number of (saturating) V-channels
Γ	words/s	Maximum bus bandwidth
ϕ_k	words/s	Required average bandwidth for channel k
ϕ'_k	words/s	Required peak bandwidth for V-channel k
$\hat{\phi}_k$	words/s	Available bandwidth for I-channel k during peak bus usage
$\check{\phi}_k$	words/s	Available bandwidth for I-channel k during off-peak bus usage
Φ	words/s	Aggregate average bandwidth for all channels
Φ_{peak}	words/s	Aggregate peak bandwidth for all channels
Φ_{V}	words/s	Aggregate peak bandwidth for V-channels
Φ_{I}	words/s	Aggregate average bandwidth for I-channels
$\hat{\Phi}_{\text{I}}$	words/s	Aggregate available bandwidth for I-channels during peak bus usage
ω_k	cycles	Time-slot allocation for channel k
h	cycles	Arbitration overhead
τ	s	Service period
$\beta_{k,\text{p}}$	words	Producer-side buffer size for channel k
$\beta_{k,\text{c}}$	words	Consumer-side buffer size for channel k

of h cycles. Clearly the average bandwidth required must be less than that available:

$$\sum_{k=1}^{N} \phi_k = \Phi < \Gamma \tag{1}$$

If data are produced and consumed at a constant rate (ϕ_k for each channel k) and there are no buffer overflows, then the service period (the time taken for all channels to have completed one transfer each, see Figure 5) is:

$$\tau = \frac{1}{\Gamma} \sum_{k=1}^{N} (\omega_k + h) = \frac{1}{\Gamma} \left(\sum_{k=1}^{N} \omega_k + Nh \right) \tag{2}$$

During this time, $\phi_k \tau$ data are produced for channel k. In steady state $\phi_k \tau = \omega_k$. So we can solve for τ:

$$\tau = \frac{1}{\Gamma} (\tau\Phi + Nh) = \frac{Nh}{\Gamma - \Phi} \tag{3}$$

When allocating the time-slot ω_k for channel k, the lower bound is:

$$\omega_k \geq \frac{\phi_k Nh}{\Gamma - \Phi} \tag{4}$$

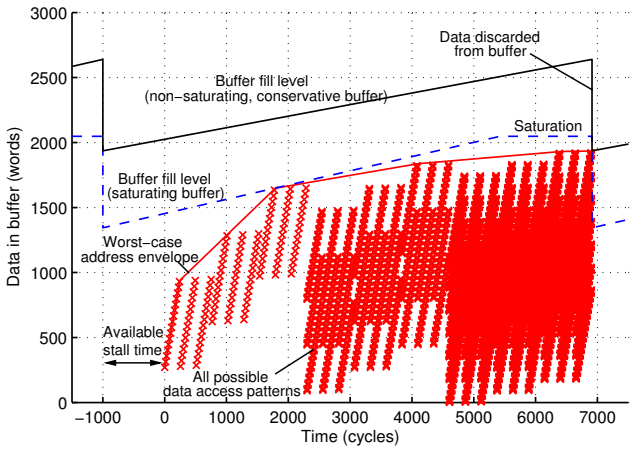


Fig. 7. Motion vector estimation search window buffering. The address pattern shows all possible address accesses over one fundamental period. Buffer fill levels for non-saturating and saturating buffer conditions are shown.

This value is used as the cycle count limit in the programming of the arbitration unit. The minimum source buffer size $\beta_{k,\text{p}}$ for each channel is the number of words which need to be stored during the time the channel does not have control of the bus:

$$\beta_{k,\text{p}} \geq \omega_k \left(1 - \frac{\phi_k}{\Gamma} \right) \tag{5}$$

However, avoiding buffer saturation comes at a cost of greater than necessary consumer side buffering; this is highly non-desirable as on-chip memory is limited, and particularly so in FPGAs. The following example illustrates this point.

Example 2: Simple input buffer sizing. Consider the motion vector estimation buffer case of Example 1. Each search window comprises $44 \times 44 = 1936$ pixels, with an overlap of $44 \times 28 = 1232$ pixels between adjacent search windows. To ensure that valid data are always available to the engine, a simple method for determining the buffer size is to store 1936 pixels (for the current search window) and an additional $1936 - 1232 = 704$ pixels (the non-overlapping part of the subsequent search window), totalling 2640 pixels. The actual memory consumed is a power of 2, 4096 pixels, and therefore 112% larger than required for storing just the current search window. □

C. Size-limited Buffers

In order to account for limited buffer sizes, we will modify the assumptions and allow buffers to saturate. Channel throughput is no longer constant, but has inactive periods (when the destination buffer is full), which must be balanced by periods where the throughput is higher than average. This is shown in Figure 7 for the case of Example 2. The two 'fill level' lines represent the amount of valid data in the buffer at any particular moment. In the case where the buffer saturates (lower line) the rate at which data is fed into the buffer is slightly higher than the rate for the simple example (upper line). This is necessary to compensate for brief period when the buffer is full. There are several important features to be noted:

1) In the analysis of the saturating buffer case, we also take into account the pattern of locations accessed in the buffer. This must be determined at design time. In Figure 7 all possible accesses are plotted for the MVE example. The buffer must be filled sufficiently quickly such that data accesses are all within the available buffered data.
2) To simplify the calculation of the required fill rate for a given access pattern, not all addressed locations need to be considered. The required fill rate can be quickly calculated from the envelope of possible accessed locations.
3) It is assumed that the engine processing rate will be at least as fast as the overall required throughput rate, and potentially faster. This is accounted for by introducing an allowable 'stall time' per fundamental period when determining the required fill rate. This can be seen to be 1000 cycles in the example of Figure 7.

Each of the N channels may now exhibit one of two modes of behaviour. If the destination buffer of a channel saturates, the channel bandwidth demand is time-variant, since during saturation no data transfer can be made. We will denote these channels *V-channels*. If the destination buffer never saturates, the bandwidth demand is constant. These are denoted time-invariant *I-channels*. Without loss of generality the channels are ordered so that the first M are V-channels. The V-channels have a required peak throughput $\phi'_k,\ 1 \leq k \leq M$. The bus must be able to support concurrent peak demands of the V-channels:

$$\Phi_{\mathrm{V}} = \sum_{k=1}^{M} \phi'_k < \Gamma \tag{6}$$

If the peak demand on the bus, including the time-invariant channels, is less than the bus capacity:

$$\Phi_{\text{peak}} = \sum_{k=1}^{M} \phi'_k + \sum_{k=M+1}^{N} \phi_k < \Gamma \tag{7}$$

then (4) can be used to calculate the STDM time-slot parameters ω_k by substituting ϕ'_k for ϕ_k and Φ_{peak} for Φ. If the inequality of (7) does not hold, let us term the bus usage *critical*. In a bus with a critical level of usage, bandwidth demands vary over time. During periods of peak activity by the V-channels the remaining I-channels are starved of bandwidth. This is compensated for during off-peak times. As a result the I-channels have increased buffering requirements.

Consider the case where $M = 1$: there is one time-variant, saturating channel b. The peak demand on bus bandwidth is:

$$\Phi_{\text{critical}} = \Phi_{\mathrm{V}} + \hat{\Phi}_{\mathrm{I}} \tag{8}$$

where $\hat{\Phi}_{\mathrm{I}}$ is the reduced total bandwidth available to the $N-1$ I-channels. Rearranging (4) and substituting variables:

$$\Phi_{\text{critical}} = \Gamma - \frac{Nh\phi'_b}{\omega_b} \tag{9}$$

(b is the time-variant channel) and also:

$$\omega_k \geq \frac{\hat{\phi}_k Nh}{\Gamma - \Phi_{\text{critical}}}, \quad M < k \leq N \tag{10}$$

for the I-channels. The variable $\hat{\phi}_k$ denotes the bandwidth available to channel k during the peak demand times, and is given by:

$$\hat{\phi}_k = \phi_k \frac{\hat{\Phi}_{\mathrm{I}}}{\Phi_{\mathrm{I}}}, \quad M < k \leq N \tag{11}$$

From these equations it will be possible to determine the time-slot size (ω_k) to allocate to each of the time-invariant channels, provided a value can be found for Φ_{critical} first.

Using (8), (9), (10) and (11) we can derive:

$$\omega_k \geq \frac{\phi_k \omega_b}{\Phi_{\mathrm{I}} \phi'_b} \left(\Gamma - \frac{\phi'_b Nh}{\omega_b} - \Phi_{\mathrm{V}} \right) \tag{12}$$

Now, the service period τ in the critical bus usage case also varies over time. During peak activity periods by the V-channels the service time will be longer than when these same channels are idle. For the $M = 1$ case, during the off-peak period (when channel b is idle) the service period is given by:

$$\tau_{\text{offpeak}} = \frac{1}{\Gamma} \left(\sum_{k=M+1}^{N} \omega_k + Nh \right) = \frac{Nh}{\Gamma - \sum_{k=M+1}^{N} \check{\phi}_k} \tag{13}$$

Now substitute (12) in (13), and simplify, noting that $\Phi_{\mathrm{V}} = \phi'_b$ in this case and $\Phi_{\mathrm{I}} = \sum_{k=M+1}^{N} \phi_k$. Solve for ω_b:

$$\omega_b \geq \frac{\phi'_b Nh\Gamma}{(\Gamma - \phi'_b)\left(\Gamma - \sum_{k=M+1}^{N} \check{\phi}_k\right)} \tag{14}$$

The channel b will be active for ϕ_b/ϕ'_b of the time, during which each time-invariant channel k has bandwidth $\hat{\phi}_k$. average bandwidth requirement for channel k to be met:

$$\frac{\phi_b}{\phi'_b}\hat{\phi}_k + \left(1 - \frac{\phi_b}{\phi'_b}\right)\check{\phi}_k = \phi_k, \quad M < k \leq N \tag{15}$$

Therefore:

$$\sum_{k=M+1}^{N} \check{\phi}_k = \frac{1}{1 - \frac{\phi_b}{\phi'_b}} \sum_{k=M+1}^{N} \left(\phi_k - \frac{\phi_b}{\phi'_b}\hat{\phi}_k \right) \tag{16}$$

$$= \frac{1}{1 - \frac{\phi_b}{\phi'_b}} \left(\Phi - \frac{\phi_b}{\phi'_b}\Gamma + \frac{\phi_b Nh}{\omega_b} \right) \tag{17}$$

So finally:

$$\omega_b \geq \left(\frac{\phi'_b Nh}{\Gamma - \Phi} \right) \left(\frac{\Gamma - \phi_b}{\Gamma - \phi'_b} \right) \tag{18}$$

This can be generalised for situations where $M > 1$:

$$\omega_b \geq \left(\frac{\phi'_b Nh}{\Gamma - \Phi} \right) \left(\frac{\Gamma - \sum_{b=1}^{M} \phi_b}{\Gamma - \sum_{b=1}^{M} \phi'_b} \right) \tag{19}$$

All the variables in this equation are known, so ω_b can be calculated for all V-channels $b \leq M$. One of these channels is then used to find Φ_{critical} using (9). This can be used to find the values for ω_k for the remaining I-channels $M < k \leq N$. This is illustrated in the following example.

Example 3: Calculation of arbitration parameters. A system comprises two of the motion vector estimation process nodes of Example 2, processing VGA sized images at different frame rates (22 and 18 frames-per-second). Each node has two input channels (the reference block and the search window) and one

TABLE IV
CHANNEL CHARACTERISTICS FOR EXAMPLE 3

Channel	Mean bandwidth ϕ_k (Mwords/s)	Peak bandwidth ϕ'_k (Mwords/s)	ω_k
1	18.59	24.84	235
2	15.21	15.30	145
3	6.76	6.76	40
4	5.53	5.53	33
5	0.03	0.03	1
6	0.03	0.03	1
Total	46.1	52.5	

output channel (the vectors), making six channels in total, with an overall mean bandwidth of 46.1Mw/s, mapped to a bus with capacity 50Mw/s. On inspection of the address patterns, buffer sizes and consumption behaviour of the channels it is determined that two of the destination channel buffers will saturate, increasing the peak bandwidth demand to 52.5Mw/s, as shown in Table IV. Using (19), and $N = 6$, $h = 3$, $\Gamma = 50$Mw/s, $\Phi = 46.1$Mw/s, $M = 2$, we find $\omega_1 = 210.6$ and $\omega_2 = 129.7$. The critical bandwidth demand is $\Phi_{\text{critical}} = 47.9$Mw/s from (9), and from (8) we find that $\hat{\Phi}_{\text{I}} = 7.74$Mw/s. Therefore, using (11) and (10) we find the values $\omega_k = \{35.9, 29.4, 0.1, 0.1\}$ for $k = \{3, 4, 5, 6\}$. These are rounded up to integer values, while ensuring that the ratio $\omega_k : (\sum_{i=3}^{6} \omega_i + Nh)$ for each k does not decrease in the process, giving $\omega_k = \{40, 33, 1, 1\}$ for $k = \{3, 4, 5, 6\}$. After similarly rounding and adjusting ω_1 and ω_2 we obtain the values for ω_k as shown in the right column of Table IV. □

D. Buffer Sizing and Latency

We have so far found a method for determining the time-slot sizes to use in the bus arbitration, including situations in which limited buffering causes time-variant behaviour on some channels. We now must determine the required buffer space on the producer-side of these channels and the effect on the size of the buffers for the remaining channels in the system. If the bus usage is critical, channels which do not exhibit time-variant behaviour require extra buffer space to compensate for periods where their bandwidth is temporarily restricted.

Consider a channel k which is a time-invariant channel: its bandwidth demand is constant. Due to the changes in bandwidth demands by time-variant channels, the *actual* throughput of channel k will be time-varying: $\phi_k(t)$. On the consumer side of the channel, there must be extra buffer space $\Delta\beta_k$ sufficient to prevent supply the processing node engine without causing stalls during deviances from the average throughput rate ϕ_k. Thus:

$$\Delta\beta_{k,\text{c}} \geq \int_{t_1}^{t_2} \phi_k - \phi_k(t)dt \quad \forall\{t_1, t_2\} : t_1 < t_2 \tag{20}$$

Determining the buffer sizes requires finding the worst cases for (20). This occurs when the throughput $\phi_k(t)$ reduces, due to all V-channels being active concurrently. Assuming the active channels are not source-limited and therefore (using the STDM protocol) consume the maximum amount of bandwidth available to them when active. The V-channels when idle due to buffer saturation consume a single bus cycle of their allocated time-slot before releasing the bus for arbitration. By inspection of Figure 5, one can observe that the time-varying throughput of channel k is therefore:

$$\phi_k(t) \approx \frac{\Gamma\omega_k}{\left[\sum_{i=1}^{M} \alpha_i(t)\right] + \left[\sum_{i=M+1}^{N} \omega_i\right] + Nh} \tag{21}$$

where for time-variant channel i, $\alpha_i(t) = \omega_i$ when the channel is active and $\alpha_i(t) = 1$ at other times. Therefore, the evaluation of the integral of (20) is computationally not difficult, since the worst-case (approximate) $\phi_k(t)$ is piecewise constant. However, it is necessary to determine the active and inactive times for each channel, and the interval (t_1, t_2). Assume that all M burst channels become active at time $t_1 = 0$, and each channel i has periodicity T_i, determined by the periodicity of the node it supplies. The procedure for determining the channel active times is relatively straightforward:

1) At time $t = 0$, each active channel $i \leq M$ starts with a number of words $r_i(0) = \phi_i T(i \to n)$ to be transfered before the channel will become inactive again.
2) For each channel calculate the transfer bandwidth $\phi_k(0^+)$ from (21).
3) Determine the time for the first channel to become inactive:
$$d_1 = \min\left(T_i \frac{\phi_i}{\phi_i(0^+)}\right), \quad \forall i \leq M \tag{22}$$
This channel is marked as inactive for $t > d_1$.
4) Record the number of words remaining to be transfered at time $t = d_1$ in the other channels:
$$r_i(d_1) = r_i(0) - \phi_i(0^+)d_1 \tag{23}$$
5) For each subsequent stage $n = \{1, 2, \dots\}$, the duration of the stage is given by:
$$d_{n+1} = \min \begin{cases} \dfrac{r_i(d_n)}{\phi_i(d_n^+)} & \text{active channels} \\ q_i T_i - d_n & \text{inactive channels} \end{cases} \tag{24}$$
where $\phi_i(d_n^+)$ can be calculated from (21) and
$$r_i(d_n) = r_i(d_{n-1}) - \phi_i(d_{n-1}^+)(d_n - d_{n-1}) \tag{25}$$
In the term $q_i T_i - d_n$, q_i is an integer value that is incremented each time channel i becomes inactive. At each stage, one channel becomes active or inactive, depending on which term in (24) is minimum.

For each channel k there will be a time d_p at which $\phi_k(d_p^+) > \phi_k$. The integral of (20) is therefore calculated between $(0, d_p)$.

On the source side, the equation is slightly different. The producer FIFOs must be large enough to contain data generated by the node without causing a stall, even when the data generation rate is not constant. If the producer for channel k is node n and generates data at a rate $p_n(t)$, then the equation for the buffer space required is:

$$\Delta\beta_{k,\text{p}} \geq \int_{t_1}^{t_2} p_n(t) - \phi_k(t)dt \quad \forall\{t_1, t_2\} : t_1 < t_2 \tag{26}$$

To simplify this, we will compute a conservative estimate for the upper bound, by setting $p_n(t)$ to a periodic function:

$$p_n(t) = \begin{cases} p'_n & 0 < t < \frac{\phi_k T_n}{p'_n} \\ 0 & \frac{\phi_k T_n}{p'_n} < t < T_n \end{cases} \quad (27)$$

Here p'_n is the peak rate at which node n can produce data, and T_n is the periodicity of the node. (26) is now a piecewise constant function and can be computed in a similar way to (20).

For a time-variant channel b, the source side buffer must be sufficiently large to hold the data produced while the consumer-side buffer has saturated. Again, (26) must be evaluated, however in this case we find the worst case conditions by assuming that the destination buffer saturates at time $t = 0$ and $\phi_k(t)$ is the periodic function:

$$\phi_k(t) = \begin{cases} 0 & 0 < t < T_k \left(1 - \frac{\phi_k}{\phi'_k}\right) \\ \phi'_k & T_k \left(1 - \frac{\phi_k}{\phi'_k}\right) < t < T_k \end{cases} \quad (28)$$

In addition to the buffer space required resulting from variations in throughput, the buffer levels also ripple up and down over the duration of each service period τ. The height of this ripple is given by:

$$\Delta\gamma_k \geq \frac{\phi_k}{\Gamma} \left(\sum_{i \neq k} \omega_i + Nh \right) \quad (29)$$

The total spare buffer space required is found by adding $\Delta\beta_k$ and $\Delta\gamma_k$. Finally, the maximum latency introduced by the channel can be approximated by the buffer size and the average throughput rate:

$$l_k \leq \frac{\Delta\beta_k + \Delta\gamma_k}{\phi_k} \quad (30)$$

Example 4: Buffer sizing and latency. We now calculate the required buffer space for each channel from Example 3. Assume that data are fed into channels 1 to 4 at a constant rate, such that for input node n and corresponding channel k, $p'_n = \phi_k$. Thus, $\Delta\beta_{k,\mathrm{c}} = \Delta\beta_{k,\mathrm{p}} = \Delta\beta_k$. Channel 1 has periodicity $T_1 = 37.9\mu$s, and for channel 2, $T_2 = 46.3\mu$s. We find $d_1 = 28.3\mu$s (channel 1 becomes inactive) and $d_2 = 37.3\mu$s (channel 2 becomes inactive). For channel 3, from (21),

$$\phi_3(t) = \begin{cases} 4.23\text{Mw/s} & 0 < t < d_1 \\ 8.37\text{Mw/s} & d_1 < t < d_2 \end{cases}$$

Therefore:

$$\Delta\beta_3 = \left\lceil \int_0^{d_1} \phi_3 - \phi_3(t)dt \right\rceil = \left\lceil d_1(\phi_3 - \phi_3(0^+)) \right\rceil = 72$$

The ripple for channel 3:

$$\Delta\gamma_3 = \left\lceil \frac{\phi_3}{\Gamma} (\omega_1 + \omega_2 + \omega_4 + \omega_5 + \omega_6 + 6 \times 3) \right\rceil = 59$$

Other values for the buffers are listed in Table V. Note that for channels 1 and 2 (where consumer-side buffers saturate) the buffering values are the minimum producer-side buffer sizes. For channels 3 to 6, the totals are the minimum producer-side buffer sizes, and the total spare capacity required before saturation of the consumer-side buffers. □

TABLE V
SPARE BUFFER SPACE AND LATENCY FOR EXAMPLE 4.

Channel	$\Delta\gamma_k$	$\Delta\beta_k$	Total (words)	Latency (μs)
1	178	89	267	14.4
2	100	5	105	6.9
3	59	72	131	19.4
4	49	58	107	19.4
5	1	0	1	37.9
6	1	0	1	46.3

E. Method Summary

The aim of the analysis is to show how the system designer can ensure that derivative designs constructed at run-time will achieve required performance when sharing communication media. The process is summarised as follows. At design time, the system designer collects the following information about each processing node:

1) The envelope of the address pattern, including the baseline repeat period.
2) The magnitude of the consume delta function.
3) The maximum theoretical processing throughput, assuming no stalling due to lack of data or output buffer space.
4) The input and output buffer sizes.

Algorithms are created from communicating clusters of nodes. The designer determines a set of possible mappings of nodes to platform buses for each application. At run-time, the necessary algorithms and the associated performance requirements are determined by supervisory application software. The run-time system software selects a mapping for each algorithm and then verifies the performance requirements can be fulfilled by executing the following steps:

1) Calculate the required throughput for each node and channel, based on algorithmic throughput requirements.
2) Verify mean demand on each bus does not exceed available bandwidth.
3) Determine for each node the stall time for the engine.
4) From the stall time, the required throughput, and the address pattern envelope, determine for each channel if the destination buffer will saturate, or if not, the spare capacity in the buffer.
5) Based on the buffer saturation, divide channels by their bandwidth demand into time-invariant and time-variant.
6) Calculate the peak bandwidth demand of the time-variant channels, and verify the aggregate peak bandwidth demand is less than the bandwidth available.
7) Calculate the time-slot size (arbitration count) for each channel (ω_k).
8) Calculate the required spare buffer capacity for all source-side buffers and all time-invariant channel destination buffers, and verify this is less than the available capacity.
9) Verify the latency of each channel is acceptable.

The verification process is linear in computational complexity, that is $O(N)$. If each verification step in the process is successful, the required performance will be achieved. If any step fails, the selected mapping is not acceptable; at this point a number of options are available to the system. A different bus

assignment may be selected, if the bandwidth requirements of different buses are mismatched. A less aggressive approach could be to instantiate an alternative processing algorithm that has lower performance requirements, such as one with a lower quality of service, and is therefore more likely to be implementable. The high-level decision made here is system-dependent, and not covered in the scope of this paper.

It is emphasised that the run-time evaluation of communication parameters involve calculations with low computational intensity, and moreover the evaluation is performed infrequently relative to the operation time of the algorithms (which process continuous streams of video data). Taking the example mentioned in the introduction to this paper, where a surveillance camera responds to changes in lighting conditions and activity, the system could reconfigure once every few hours or every few minutes. Computing the communication parameters and verifying the operating performance would be met takes of the order of microseconds, and moreover may be done as a background task while the system continues to operate. The reconfiguration time is of the order of milliseconds for recent high density FPGAs [25]. Thus the overhead incurred is slight.

V. Experimental Results

In order to verify the late integration design methodology and communication analysis presented above, several prototypical platforms have been created. These have informed the development of a cycle-accurate simulation model of the communication system. This section presents results obtained from the prototyping and simulations.

The prototype platforms are based on implementations of Sonic-on-a-Chip (e.g., [5]) and target Xilinx Virtex-II Pro and Virtex-4 FPGAs. The platforms were designed in Verilog and VHDL, synthesised using Synplicity Synplify 7.2 and implemented using Xilinx EDK and ISE 6.3 tools. The prototype systems were designed to be assembled using an advanced modular dynamic reconfiguration scheme [26]. As functionality was the primary goal, the target speed for the SonicBus was a relatively modest 50MHz. Although late integration requires careful control over the physical routing of signals, the constraints did not contribute significantly to the achieved bus speed. The maximum achieved propagation delay for the constrained bus signals was 19.08ns (as reported by the vendor tools). This only improved by 1.22% by removing the routing constraints.

As well as the the motion vector estimators of Example 1, several other processing element nodes have been created for the prototypes. The performance and behaviour of the prototypes have been used in the creation of the simulation model. Several different systems were simulated. Detailed results are presented for the simple example system described in Example 3, comprising two motion vector estimation nodes. This type of node is interesting because it exhibits data-dependent behaviour and generates bus traffic which varies with time. The results of simulations of other systems are summarised at the end of this section.

The address patterns for the MVE nodes have been extracted from real data (a 'carphone' video sequence), and the

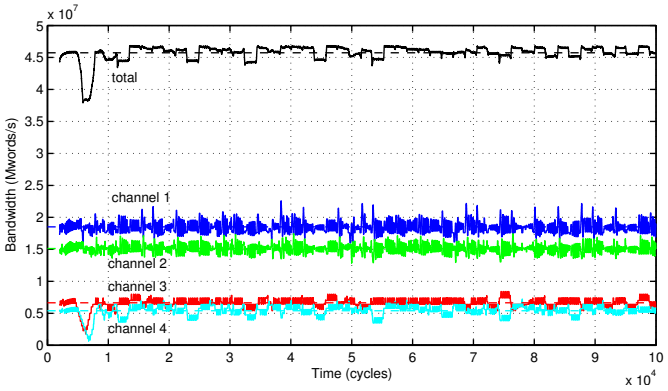


Fig. 8. Bandwidths of channels 1–4 in the simulated system, averaged over 2000 cycles. The arbitration scheme is effective at high bus utilisation (92% in this case), and copes with time-variant demand while maintaining the required average throughput for each channel.

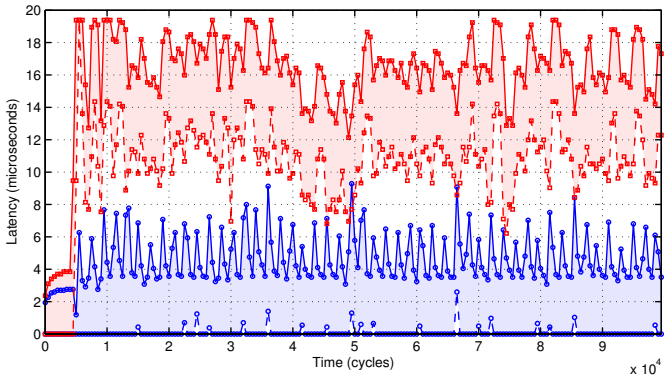


Fig. 9. The maximum and minimum latencies of channels 1 (circles) and 3 (squares). Measured latencies match the calculated expected values.

parameters calculated in Example 3 and Example 4 and listed in Table IV and Table V are used as nominal values. The simulated system has four input nodes; the rate at which these node supply data to the system is independently adjustable.

Figure 8 is a graph of the time-averaged bandwidth of the four main channels (1 to 4) and the overall bus bandwidth usage over a period of 2ms (10^5 bus cycles). For this simulation all input nodes were set to supply data at the fastest rate possible (one word per cycle). The graph shows that the STDM arbitration scheme is able to cope with high overall bandwidth utilisation and allocate bandwidth to each channel appropriately despite the time-variable demands of the channels. Indeed, the STDM scheme can achieve aggregate mean bandwidth usage arbitrarily close to the available bus bandwidth if sufficient buffering is available in each node. Note the mean bandwidth for each channel is as expected from Table IV.

The expected maximum latency for each channel (see Table V) was confirmed by the experimental data. Figure 9 shows the maximum and minimum latency for channels 1 and 3. The latency of each channel can vary significantly over time, which, as will be seen below, has an impact on the instananeous throughput of the nodes.

In a real system, data would not necessarily be supplied to the system faster than they can be processed. Instead, the

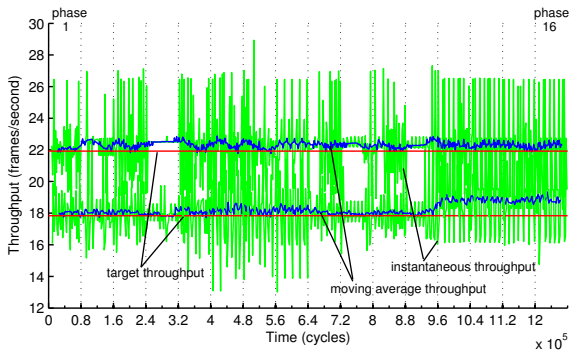


Fig. 10. The simulated block-by-block throughput of the motion vector estimators, as input data rates are varied. The instantaneous throughput is variable due to variable channel latency. The average throughputs meet or exceed the designed rates.

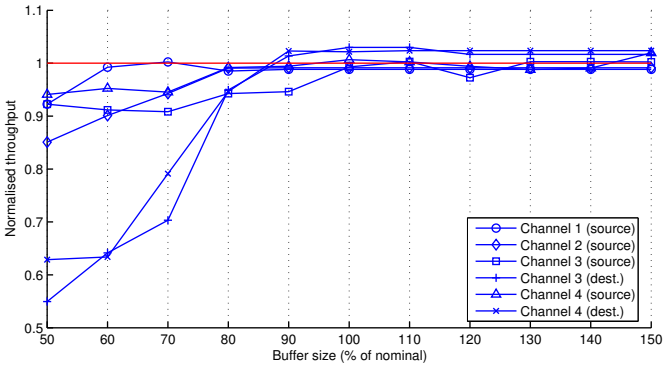


Fig. 11. Effect of varying buffer sizes on system throughput. Buffers smaller than the sizes calculated in Example 4 cause the system throughput to drop below the target rate.

rate at which data are available for processing may be limited, and the system is required to process the data at the supplied rate. The communication system must cope with scenarios in which some channels may be supplied by a rate-limited source, and other channels the data are available at an unmoderated rate. Regardless of the source data-rate, the communication parameters can be calculated based on the desired system throughput.

Figure 10 plots the block-by-block frame rates achieved by the motion vector estimators over a period of 26ms (1.28×10^6 cycles). In this case, the data rate of each input channel to the system is varied between fractionally below (by 0.1 to 0.2 frames per second) the designed-for rates (18 and 22 frames per second) and the maximum possible rate. Different channel combinations are tested for periods of 1.6ms (8×10^5 cycles). Note that in all cases the average throughputs meet or exceed the expected throughputs. The instanteous throughputs vary considerably; this is due to the variation in latency of the communication. The maximum latency for the channels supplying the higher-rate MVE processing node is 19.4μs (as per Table V). Compared with the mean processing time of a single macro-block (at 22 fps and 1200 macro-blocks per frame) of 37.9μs, the instantaneous throughput could be expected to vary between 15 and 45 frames per second.

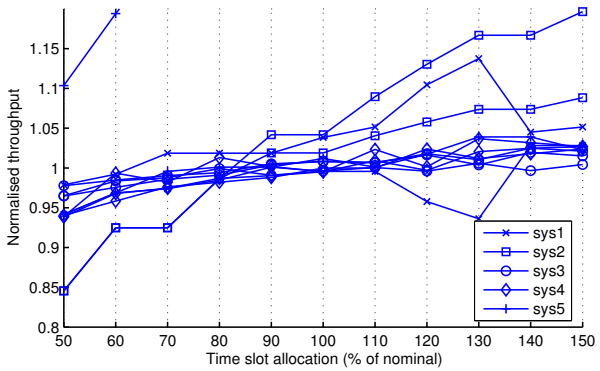
The buffer sizing calculations listed in Table V were verified by varying the size of the buffers by $\pm50\%$ of the nominal value and then measuring the corresponding throughput for the affected motion vector estimator. Rate-limiting of the source data channels was applied where it results in lower performance. Moreover, the address pattern used was set to the worst-case values (i.e., the addresses closest to the address envelope of Figure 7). The normalised outcomes are plotted in Figure 11. It can be seen that the calculated required buffer sizes are sufficient to avoid degrading the system throughput performance. In addition, the buffer sizes calculated are not significantly larger than necessary in this instance, with the exception of the source buffer for channel 1, which appears to be oversized by around 50%. In general, the calculated required buffer sizes are based on worst-case conditions, which may never occur in a given system, and therefore the calculations result in conservative estimates.

TABLE VI
CHARACTERISTICS OF THE SIMULATED SYSTEMS

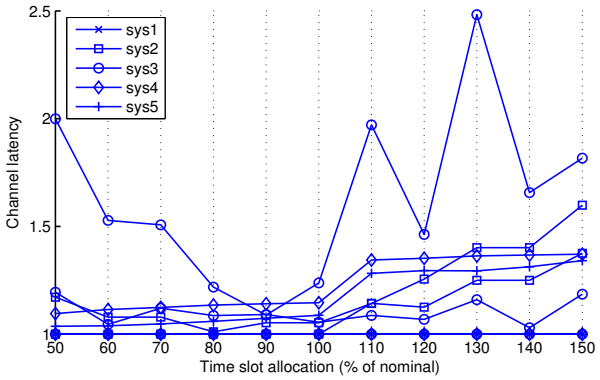
	sys1	sys2	sys3	sys4	sys5
Node	*Node throughput rate, frames/s*				
MVE 1	22.0			20.0	22.0
MVE 2	18.0				
Block 2D DCT		22.0	32.0	32.0	
Foreground separation		22.0	18.0		
Median filter			20.0		
Histogram				12.0	22.0
	Bus parameters				
Channels (N)	6	5	7	7	5
Time-variant channels (M)	2	0	1	2	1
Average total bandwidth Φ (Mw/s)	46.1	33.8	48.5	46.4	32.1
Peak total bandwidth Φ_{peak} (Mw/s)	52.5	33.8	50.9	50.9	38.4
Critical	yes	no	yes	yes	no

Thus far, for consistency all examples and experiments have been based on a single node type, namely motion vector estimators. This node type has been used because it exhibits interesting data-dependent behaviour. It is important to note that our approach is applicable to a wide variety of node types and system compositions. In addition to the two-MVE system described above (from now denoted `sys1`) four other sample systems (`sys2` to `sys5`) were designed and simulated. The parameters of the systems are given in Table VI, showing that they are constructed from different mixtures of various types of processing nodes operating at different rates. The designs of the systems are contrived to create a range of communication scenarios. The simplest case (`sys2`) is where no buffer saturation occurs. In `sys2`, buffer saturation does occur for one channel, but the peak bandwidth demand is less than the maximum bus bandwidth. The most complex cases are `sys3` and `sys4`, which each have seven channels, buffer saturation as well as unbalanced demand between the channels.

Time-slot sizes (ω_k) and minimum buffer sizes were calculated for all systems as per the method given in Section IV. To verify the correctness of the time-slot size calculations, the values for ω_k were collectively varied by $\pm50\%$ and the effect on system throughput and channel latency measured. Figure 12(a) plots the throughput for each node of each system, normalised to the desired throughput rate as listed



(a) Normalised throughput vs ω_k variation



(b) Relative channel latency vs ω_k variation

Fig. 12. Effect of changing the time-slot allocation from the nominal calculated values on (a) node throughput, (b) channel latency. Results from all five systems are overlayed in the plot.

in Table VI, as ω_k is varied. All buffers were sized $2\times$ the required minimum and data were supplied at an unlimited rate. With the correctly calculated time-slot allocation ω_k, measured throughput was at least the desired rate (to within measurement error of 0.5%). When the time-slot values were collectively reduced, the desired throughput rates were not met. Increasing the time-slot values for all channels generally resulted in higher throughput rates, although not in all cases. This is due to some channels dominating the available bandwidth, starving other channels.

Figure 12(b) shows the effect on channel latency, relative to the latency of systems with nominally sized buffers and supplied with data from rate-limited sources. It can be seen that latency generally increases as the time-slots for all channels are increased. This is because more the time between the channel having access to the bus increases. The minimum latency case actually occurs marginally below the nominal values calculated by the proposed method.

VI. Conclusion

Platform-based design in Field-Programmable Gate Arrays offers the unique prospect of derivative systems created automatically at run-time. Such systems can be customised and adapted to variations in the operating conditions, but require the judicious application of architectural constraints to reduce the integration complexity, particularly in satisfying communication requirements.

This paper presented the first platform architecture (Sonic-on-a-Chip) designed specifically for run-time assembly of FPGA-based derivative systems. The target domain of the architecture is real-time video image processing. To ensure that inter-modular communication in Sonic-on-a-Chip is analysable, it is firstly separated from computation by dividing each compute node into distinct parts: a router and an engine, connected by innovative buffers. Secondly, an appropriate arbitration scheme, statistical time-division multiplexing, is employed for the shared SonicBuses. Moreover, interactions between buses are isolated by using fully buffering bridges. Finally, by ensuring the compute nodes designs always exhibit periodic behaviour, the characteristics of the nodes can be encapsulated in parameters which describe the periodicity, as well as data consumption, storage and access patterns.

An analysis was presented of the constrained communication system, based on the nodal parameters and architectural constraints. The analysis accounts for limitations in buffer sizes, which is especially important in FPGAs where on-chip memory is a particularly limited resource. It was shown how to calculate the time-slot parameters of the STDM protocol in order to meet the real-time throughput requirements of all channels. In addition, a method for determining the minimum necessary sizes of all buffers in the communication system was detailed, along with estimations on the maximum channel latency. All calculations involve simple closed-form formulae and are suitable for execution at run-time.

The analysis has been verified in simulations of five different sample systems using a cycle-accurate model of the communication system. The systems can support a variety of processing node types and different throughput requirements.

Current and future work includes generalising the architectural template and extending our approach to applications other than video systems.

Acknowledgements

The authors thank Kostas Masselos for his comments and suggestions. The support of the Commonwealth Scholarship Commission, the New Zealand Vice Chancellors' Committee and the UK Engineering and Physical Sciences Research Council (Platform Grant number EP/C549481/1) is gratefully acknowledged. The support from Xilinx Inc., in particular Patrick Lysaght, Brandon Blodget, James Anderson and Adam Donlin, is also greatly appreciated.

References

[1] Semiconductor Industry Association, "International technology roadmap for semiconductors," 1999.

[2] H. Chang, L. Cooke, M. Hunt, G. Martin, A. J. McNelly, and L. Todd, *Surviving the SOC revolution: a guide to platform-based design*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[3] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design*, vol. 19, no. 12, pp. 1523–43, Dec. 2000.

[4] P. Lysaght, "FPGAs as meta-platforms for embedded systems," in *Proc. IEEE Int. Conf. Field-Programmable Technology*, 2002.

[5] P. Sedcole, P. Y. K. Cheung, G. A. Constantinides, and W. Luk, "A structured methodology for System-on-an-FPGA design," in *Proc. Int. Conf. Field-Programmable Logic and Applications*, 2004.
[6] S. D. Haynes, J. Stone, P. Y. K. Cheung, and W. Luk, "Video image processing with the Sonic architecture," *IEEE Computer*, vol. 33, no. 4, pp. 50–57, April 2000.
[7] J.-M. Daveau, T. B. Ismail, and A. A. Jerraya, "Synthesis of system-level communication by an allocation-based approach," in *Proc. Int. Symp. System Level Synthesis*, 1995.
[8] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," *ACM Trans. Design Automation of Electronic Systems*, vol. 4, no. 1, pp. 1–11, Jan. 1999.
[9] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Trans. Computer-Aided Design*, vol. 20, no. 6, pp. 768–83, June 2001.
[10] ——, "Design space exploration for optimizing on-chip communication architectures," *IEEE Trans. Computer-Aided Design*, vol. 23, no. 6, pp. 952–61, June 2004.
[11] K. K. Ryu and V. J. Mooney, "Automated bus generation for multiprocessor SoC design," *IEEE Trans. Computer-Aided Design*, vol. 23, no. 11, pp. 1531–1549, Nov. 2004.
[12] M. Eisenring and M. Platzner, "Synthesis of interfaces and communication in reconfigurable embedded systems," *IEE Proceedings – Computers and Digital Techniques*, vol. 147, no. 3, pp. 159–165, May 2000.
[13] *AMBA*, ARM Ltd. Spec., Rev. 2.0, 1999.
[14] *The CoreConnect Bus Architecture*, IBM Inc. White paper, 1999.
[15] *Wishbone: System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores*, Silicore Inc. Spec., Rev. B.3, 2002.
[16] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. Design Automation Conf.*, 2001.
[17] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.
[18] E. W. et al., "Baring it all to software: RAW machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, Sept 1997.
[19] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Trans. VLSI Syst.*, vol. 12, no. 4, pp. 711–26, July 2004.
[20] G. V. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *IEEE Trans. VLSI Syst.*, vol. 12, no. 1, pp. 108–119, Jan. 2004.
[21] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, and U. Rückert, "Dynamically reconfigurable system-on-programmable-chip," in *Proc. Euromicro Workshop on Parallel, Distributed and Network-based Processing*, 2002.
[22] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection networks enable fine-grain dynamic multi-tasking in FPGAs," in *Proc. Int. Conf. Field-Programmable Logic and Applications*, 2002.
[23] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A dynamic NoC approach for communication in reconfigurable devices," in *Proc. Int. Conf. Field-Programmable Logic and Applications*, 2004.
[24] S. D. Haynes, H. G. Epsom, R. J. Cooper, and P. L. McAlpine, "UltraSONIC: A reconfigurable architecture for video image processing," in *Proc. Int. Conf. Field-Programmable Logic and Applications*, 2002.
[25] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proc. Int. Conf. Field-Programmable Logic and Applications*, 2006.
[26] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *IEE Proceedings Computers & Digital Techniques*, vol. 153, no. 3, pp. 157–164, May 2006.