Interconnecting Heterogeneous Nodes in an Adaptive Computing Machine

Frederick Furtek, Eugene Hogenauer, and James Scheuermann

QuickSilver Technology, San Jose, CA 95119 USA
fred@calculus.com

Abstract. A distinguishing characteristic of field-programmable logic is the ability to route wires in the field, but previous authors have made compelling arguments for routing *packets*, not wires, between major system components. The present paper outlines the packet-switched network for interconnecting heterogeneous nodes in QuickSilver Technology's Adaptive Computing Machine (ACM). Special attention is paid to two truly innovative aspects of the ACM architecture: (1) the *Point-to-Point* (*PTP*) *protocol* for transferring real-time, streaming data and (2) the *node wrapper* which makes all nodes appear homogeneous regardless of their internal structure or functionality. The wrapper also provides a single, uniform and consistent mechanism for *task management*, *flow control* and *load balancing* across all node types. With the PTP protocol and the node wrapper, nodes as diverse as digital signal processors, reduced-instruction-set processors, domain-specific processors, reconfigurable fabrics, on-chip and off-chip bulk memories and input/output ports can communicate seamlessly. Moreover, once a node (including wrapper) has been configured, or reconfigured, by a supervisory node, it is able to operate autonomously without the need for global control.

1 Introduction

A distinguishing characteristic of field-programmable logic is the ability to route wires in the field, but previous authors have made compelling arguments for routing *packets*, not wires, between major system elements. Seitz[1] and Dally[2] argue that a dedicated packet-switched network offers several advantages in structure, performance and modularity:

- Electrical properties are optimized and well controlled
- Controlled electrical parameters enable aggressive signaling circuits
- Aggressive signaling circuits reduce power and increase propagation velocity
- Sharing wires among multiple communication flows makes more efficient use of wires
- A standard interface facilitates modularity

QuickSilver Technology's Adaptive Computing Machine (ACM) [3,4] extends the previous work by introducing network protocols and hardware mechanisms designed to make the transfer of real-time streaming data among heterogeneous *nodes* as seamless as possible, and does so without the need for global control. Two key as-

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 125–134, 2004.

pects of the ACM architecture – and the focus of this paper – are (1) the *Point-to-Point* (*PTP*) *protocol* for transferring real-time streaming data over the ACM network and (2) the *node wrapper* which makes all nodes appear homogeneous regardless of their internal structure or functionality. With the PTP protocol and the node wrapper, nodes as diverse as digital signal processors, reduced-instruction-set processors, domain-specific processors, reconfigurable fabrics, on-chip and off-chip bulk memories and input/output ports can communicate seamlessly.

The present paper introduces the ACM architecture including network topology and streaming protocols and describes how the node wrapper provides a single, uniform and consistent mechanism for task management, flow control and load balancing across all node types. The hardware task manager – which is inspired by Dennis's pioneering work in data-flow computing [5,6] – is a key component of the node wrapper and is discussed in detail.

2 The Adaptive Computing Machine

Adaptive Computing Machines are targeted at satisfying the signal and image processing needs of low-power, handheld, mobile, wireless devices and other forms of consumer electronics.

2.1 The ACM Network

The Adaptive Computing Machine consists of a collection of heterogeneous nodes interconnected by a scalable, fractal-based network (Figure 1). The network has a single *root* to which are connected:

- Network input and output ports
- System port (optional)
- Internal and external bulk memory (optional)
- K-Node (Supervisor Node)
- One or more descending *quadtrees* with heterogeneous leaf nodes

The quadtrees are implemented using 5-ported *switch elements*, each connected to a single parent and up to four children. The switch elements implement a fair, round-robin arbitration scheme and provide pipelining with multi-level look-ahead for enhanced performance. At present, the width of all paths is constant (51 bits), but the option is available to widen pathways as a tree is ascended, in the style of Leiserson's *fat trees* [7], in order to increase network bandwidth.

2.2 Network Words

All traffic on the ACM network is in the form of 51-bit *network words* (Figure 2) where the fields are defined as follows:

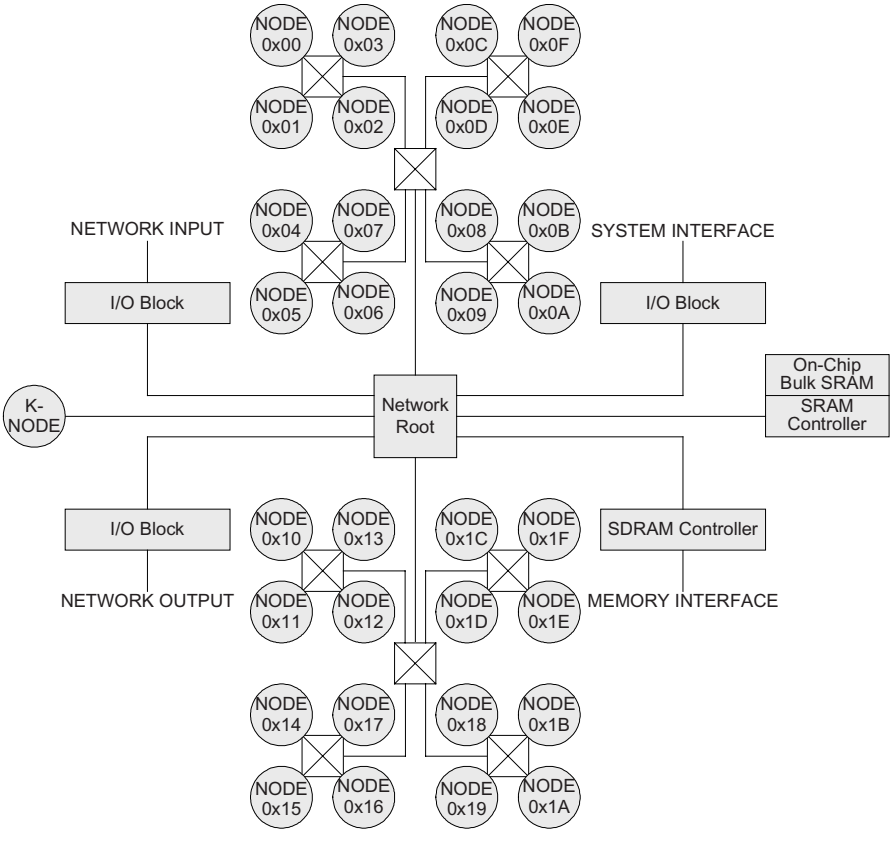

Fig. 1. ACM Network with 32 Leaf Nodes, K-Node (Supervisor Node), Off-Chip-SDRAM Controller and On-Chip SRAM with Controller

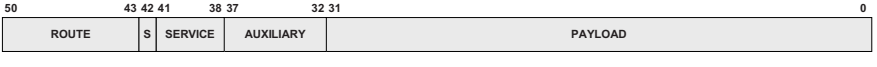

Fig. 2. A Network Word

Route – Destination address of the network word; The two high-order bits are the chip ID

S (Security Bit) – Bit allowing *peeks* (reads) and *pokes* (writes) to configuration memory; Set only for words sent by the K-Node

Service – Type of service

Auxiliary – Dependent on service type

Payload – Payload

The service field defines one of sixteen service types, two of which are of interest to us here:

Point-to-Point (PTP) – Streaming data

PTP Acknowledgement – Supports flow control for PTP data; Causes a *Consumer* or *Producer Count* at the destination node to be incremented or decremented

2.3 Nodes

Each node in the network has three elements as illustrated in Figure 3: a *node wrapper*, an *execution unit* (*EU*) and *memory* (*nodal memory*).

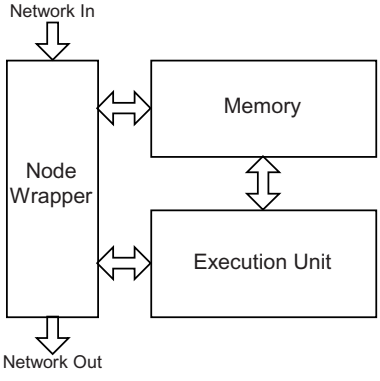

Fig. 3. A Cell

The wrapper makes the node identical in outward appearance to all other nodes regardless of its internal structure or functionality. The wrapper also relieves the execution unit from having to deal with myriad activities associated with task management and network interactions. Among other things, the wrapper is responsible for disposing of each incoming network word in an appropriate fashion – in one clock cycle.

The execution unit is responsible for executing *tasks*. It may take a wide variety of forms:

- Digital signal processor
- Reduced-instruction-set processor
- Domain-specific processor
- ASIC (application-specific integrated circuit)
- Reconfigurable (FPGA) fabric

But regardless of its form, the EU interacts with the node wrapper through a standard interface.

Nodal memory is accessible to both the node wrapper and the execution unit. It is where the node wrapper deposits incoming streaming data and where the EU accesses that data. A node's own memory, however, is typically *not* where the EU sends *output data*. To minimize memory accesses, output data is usually sent directly to the node(s) requiring that data: the *consumer node*(*s*). Nodal memory is also used to store task parameters and is available to tasks for temporary (scratchpad) storage.

3 Transferring Streaming Data

In a multi-node system where nodes are both consumers and producers of streaming data, matching production and consumption rates is a fundamental problem. A producer task on one node may produce data at a rate that is either greater than or less than the rate at which a consuming task on another node can handle. If the producer is sending data at a *greater* rate than the consumer can handle, then data is eventually lost. If the producer is sending data at a *lesser* rate than the consumer can handle, then the consumer may be starved for data, thereby potentially causing the consumer to sit idle waiting for additional data.

To address these issues, the ACM provides – via the Point-to-Point protocol and the node wrapper – a single, uniform and consistent mechanism for *task management*, *flow control* and *load balancing*. Task management ensures that a task is placed in execution only when it has sufficient input data and when there is sufficient space in the consumer node(s) to accommodate the data produced by the task. Flow control guarantees that a producer task will never overwhelm a consumer task with too much data in too short a time. Load balancing permits a producer task to distribute data among several alternate consumer nodes, thus allowing the producer task to operate at a potentially higher rate.

3.1 Point-to-Point Channels

Streaming data is transferred between two nodes (*points*) via a **Point-to-Point channel** (Figure 4). Associated with each PTP channel are:

- A **Producer Node** (Node A in Figure 4)
- A **Producer Task** running on the Producer Node's execution unit that produces a finite-sized block of PTP data per task activation, that block of data being sent over the PTP channel as a sequence of PTP words (Task 1 in Figure 4)
- An **Output Port** on the Producer Node that is associated with the Producer Task (Output Port j in Figure 4)
- A **Consumer Node** (Node B in Figure 4)
- An **Input Port** on the Consumer Node via which the Consumer Task receives PTP data from the PTP channel (Input Port k in Figure 4)
- A circular **Input Buffer** in the Consumer Node's nodal memory into which the incoming PTP data is deposited (Input Buffer k in Figure 4)
- A **Consumer Task** running on the Consumer Node's execution unit that *consumes* a finite amount of the PTP data residing in the circular input buffer per task activation (Task 2 in Figure 4)

Data is conveyed over a PTP channel when the Producer Task transfers a 50-bit **Point-to-Point word** (Figure 5) to the node wrapper in the Producer Node. (The 51st bit, the Security Bit, is added later by the network.) The node wrapper, in turn, hands the PTP word over to the packet-switched network for transfer to the Consumer Node. The 8-bit Route Field of the PTP word provides the address of the Consumer Node, while the low-order 5 bits of the Auxiliary Field indicate to which of the Consumer

Node's input ports the data is directed. When the PTP word arrives at the Consumer Node, the node wrapper deposits the 32-bit payload into the circular input buffer associated with the indicated input port. The transfer is then complete.

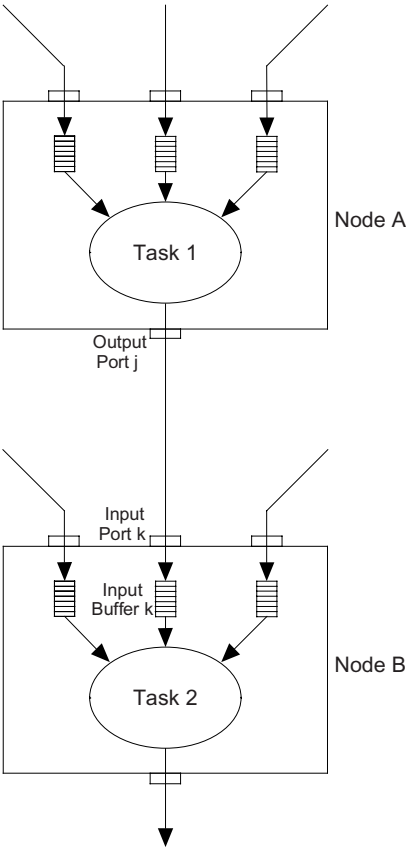

Fig. 4. A Point-to-Point Channel

Fig. 5. A Point-to-Point Word

4 Task Management, Flow Control, and Load Balancing

Having described a simple mechanism for moving streaming data between two points in the ACM, we now turn our attention to the mechanisms for task management, flow control and load balancing.

4.1 Consumer Counts and Producer Counts

As already noted, there is an input buffer associated with each input port. There is also a two's-complement signed count associated with each port, both input and output.

For an input port, the count is referred to as a **consumer count** since it reflects the amount of data in that port's input buffer that is available to be *consumed* by the associated task. A consumer count is **enabled** when its value is non-negative – that is, when its sign bit is 0. An enabled consumer count indicates that the associated input buffer has the minimum amount of data required by an activation of the associated task. At system initialization, or upon reconfiguration, a consumer count is typically reset to –C, where C is the minimum number of 32-bit words required per task activation.

For an output port, the count is referred to as a **producer count** since it reflects the amount of available space in the downstream input buffer to accept the data that is *produced* by the associated task. A producer count is **enabled** when its value is negative – that is, when its sign bit is 1. An enabled producer count indicates that the downstream input buffer has space available to accommodate the maximum amount of data produced per activation of the associated task. At system initialization, or upon reconfiguration, a producer count is typically reset to P – S – 1, where P is the maximum number of 32-bit words produced per task activation and S is the size of the downstream input buffer in 32-bit words.

Notice that both consumer counts and producer counts are typically initialized to negative values, which means that consumer counts start out *disabled* while producer counts start out *enabled*. This initial state reflects the fact that input buffers are usually empty at system initialization/reconfiguration.

4.2 PTP Acknowledgements

Consumer and Producer Counts are updated by a system of *credits* and *debits* in the form of **forward acknowledgements** and **backward acknowledgements**. Both types of acknowledgements are network words (Figure 6) sent by a task as the last steps in a task activation. In both cases, the payload contains four fields: (1) a bit indicating the type of acknowledgement, (2) a port, (3) a task and (4) an Ack Value.

Fig. 6. A PTP Acknowledgement

The sequence of acknowledgements that a task performs at the end of each activation is as follows:

A. For each output port of the task:

1. Send a forward acknowledgement to the consumer node specifying the consumer input port and the consumer task; Ack Value is the number of PTP words the task just sent to the consumer input port

2. Send a backward acknowledgement (a *self ack*) to the node on which the task resides specifying the output port and the task; Ack Value is the number of PTP words the task just sent via the output port

B. For each input port of the task:

1. Send a backward acknowledgement to the producer node specifying the producer output port and producer task; Ack Value is *minus* the number of 32-bit words the task just consumed from the input port's buffer

2. Send a forward acknowledgement (a *self ack*) to the node on which the task resides indicating the input port and the task; Ack Value is *minus* the number of 32-bit words the task just consumed from the input port's buffer

4.3 The Hardware Task Manager

The hardware task manager is the part of the node wrapper responsible for updating consumer and producer counts in response to incoming acknowledgements. It also monitors the sign bits of those counts and launches a task when an appropriate set of counts is enabled. This last responsibility is met using two signed counts that are associated not with a port but with a task: a **task input count** and a **task output count**. A task's input (output) count reflects the number of task consumer (producer) counts that are enabled. A task count is said to be **enabled** when its value is non-negative. A task is **enabled** – and available for execution – when both its input count and its output count are enabled.

Incoming acknowledgements update various counts and cause tasks to be launched as follows:

A. If a *forward* acknowledgement is received:

1. Interpret the specified port as an *input* port, and add Ack Value to the corresponding consumer count

2. If the consumer count makes a transition from disabled to enabled (enabled to disabled), then increment (decrement) the input count of the specified task by 1

B. Else if a *backward* acknowledgement is received:

1. Interpret the specified port as an *output* port, and add Ack Value to the corresponding producer count

2. If the producer count makes a transition from disabled to enabled (enabled to disabled), then increment (decrement) the output count of the specified task by 1

C. If after Step A or B the specified task's input and output counts are both enabled, then place the task on the *ready-to-run queue* if it is not already on the queue; Launch the task when it reaches the head of the queue

These actions, in effect, embody the *firing rule* for tasks. They cause a task to be placed on the ready-to-run queue and ultimately executed when a sufficient number of consumer counts and a sufficient number of producer counts are enabled. What those

sufficient numbers are is determined by the initial values of a task's input count and output count. If I (O) is the number of input (output) ports associated with a task and $IC_{Initial}$ ($OC_{Initial}$) is the initial value of the task's input (output) count, and if we assume that all consumer counts are initially disabled and all producer counts are initially enabled as discussed above, then a task *fires* when

$-IC_{Initial}$ out of I consumer counts are enabled
AND
$(O - OC_{Initial})$ out of O producer counts are enabled

For example, for I = 4,

If $IC_{Initial} = -1$, then 1 out of 4 consumer counts must be enabled
If $IC_{Initial} = -2$, then 2 out of 4 consumer counts must be enabled
If $IC_{Initial} = -3$, then 3 out of 4 consumer counts must be enabled
If $IC_{Initial} = -4$, then 4 out of 4 consumer counts must be enabled

For O = 4,

If $OC_{Initial} = 3$, then 1 out of 4 producer counts must be enabled
If $OC_{Initial} = 2$, then 2 out of 4 producer counts must be enabled
If $OC_{Initial} = 1$, then 3 out of 4 producer counts must be enabled
If $OC_{Initial} = 0$, then 4 out of 4 producer counts must be enabled

4.4 Flow Control

Earlier, we said that flow control guarantees that a producer task will never overwhelm a consumer task with too much data in too short a time. In the context of the ACM, that means that a producer task will never overflow an input buffer of a consumer task. The mechanism that guarantees this property has been spelled out above:

1. The producer count associated with the output port of the producer task is initialized to P – S – 1 as described in Section 4.1.
2. The producer and consumer tasks perform the sequence of acknowledgments outlined in Section 4.2 upon the completion of each activation.

4.5 Load Balancing

The mechanism described in Section 4.3 – that launches a task when *just one* of its output ports is enabled – permits a producer task, usually a high-throughput task, to send the output of each activation to one of several alternate, usually lower-throughput, consumer tasks. The downstream processing load is thus distributed (balanced) among the several consumer tasks. To support this capability, the node wrapper makes available to the execution unit the identities of enabled input and output ports.

5 Conclusions

The node wrapper and the point-to-point protocol provide an asynchronous, distributed mechanism for handling streaming data in a system with heterogeneous nodes. This *distributed intelligence* supports task management, flow control and load balancing across a wide range of node types.

This work is connected to prior research in several related areas. Petri nets [8] were the first mathematical model to truly capture the notion of *concurrency*, and the firing rule described in Section 4.3 above can be seen as a generalization of Petri's original firing rule. Dennis's work on data-flow computing [5,6] – which inspired our work and was in turn inspired, at least in part, by Petri's work – is based on the principle that computing should be *data-driven*, that an operator (task) should fire (execute) when input data and output buffers are available. Seitz[1] and Dally[2] recognized the importance of making a transition from routing wires – the realm of field-programmable logic – to routing packets (tokens) – the realm of data-flow computing.

There is also a connection to Lysaght's work [9] on *logic caching* since the hardware task manager provides a mechanism for distributed, rather than centralized, control of logic caching. Finally, there is a connection to threshold logic which can be appreciated when one realizes that consumer and producer counts act as *threshold gates* with the output sign bit indicating whether a threshold has been reached.

References

1. Seitz, C. L.: Let's Route Packets Instead of Wires. In: Dally, W. J. (ed.): Proceedings of the 6th MIT Conference on Advanced Research in VLSI. MIT Press (1990) 133–37
2. Dally, W., Towles, B.: Route Packets, Not Wires: On-Chip Interconnection Networks. In: Proc. Design Automation Conf. (June 2001) 684–689
3. Master, P.: The Age of Adaptive Computing is Here. In: Glesner, M., Zipf, P., Renovell, M. (eds.): Proceedings of the 12th International Conference on Field Programmable Logic (FPL 2002). Lecture Notes in Computer Science, Vol. 2438. Springer-Verlag, Berlin Heidelberg New York (2002) 1-3
4. Guccione, S. A.: The Adaptive Computing Machine Combines DSP Programmability with the Power and Performance of Custom Hardware. In: Global Signal Processing Expo and Conference (GSPx). Dallas, TX (2003)
5. Dennis, J. B.: First Version of a Data Flow Procedure Language. In: Programming Symposium. Lecture Notes in Computer Science, Vol. 19. Springer-Verlag, Berlin, New York (1974) 362–376
6. Dennis, J. B.: The Evolution of 'Static' Data-Flow Architecture. In: Gaudiot, J.-L., Bic, L. (eds.): Advanced Topics in Data-Flow Computing. Prentice-Hall, Englewood Cliffs (1991) 35–91
7. Leiserson, C. E.: Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. In: IEEE Transactions on Computers, C-34(10) (October 1985) 892–901
8. Petri, C.A.: Kommunikation mit Automaten: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, Bonn (1962)
9. Lysaght, P., Dunlop, J.: Dynamic Reconfiguration of Field Programmable Gate Arrays. In: Moore, W., Luk, W. (eds.): Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications, Oxford (1993)