Extending an on-chip mesh network off the chip

Joseph Richard Auricchio
jauricchio@cs.ucsd.edu

submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science
University of California San Diego

1 Dec 2011

v1.0

Table of Contents

Introduction

The UCSD Computer Architecture research group is preparing to fabricate a prototype of the GreenDroid low-power processor architecture. GreenDroid is tiled architecture, with an on-chip mesh network carrying all communication between processor cores, requests to I/O devices, and loads and stores to main memory. In the prototype device, the on-chip network will not be connected to the device's I/O pins. All network messages heading to I/O devices must be multiplexed and carried over a single narrow bus to their destination on a northbridge FPGA or a host PC.

We present the design and implementation of a hardware module which bridges the on-chip mesh networks of two chips by transparently relaying messages over a genericized physical link. Viewed from inside the chip, it appears that all network nodes are attached to the same network fabric. The processor, main memory, and I/O devices communicate through on-chip network messages, unaware that they are on separate physical dies. We call this an **extended virtual on-chip network:** *extended* across several chips, *virtual* indicating the illusion of seamless connectivity[1].

The extended virtual on-chip network preserves the GreenDroid programming model, in which all devices are accessible solely through the network, while allowing devices to be implemented in different technologies than the processor itself. I/O devices may be implemented in inexpensive, reconfigurable FPGAs or emulated in software, rather than built into the expensive prototype chip.

Background: The GreenDroid processor architecture

For the past several process generations, processor architects have been challenged to improve performance within fixed power budgets. As CMOS devices scale smaller each year, architects face the *utilization wall:* the percentage of a chip which can actively switch drops exponentially with each successive process generation [Venkatesh]. Every 18 months, Moore's Law provides twice as many transistors, but the power required to switch a transistor does not decrease. To use twice as many transistors requires twice as much power. Most computer systems (including server, desktop, laptop, and

[1] "Virtual" by analogy with virtual private networks: not for their value in enhancing network security, but for the illusion they provide of remote devices sharing the same LAN.

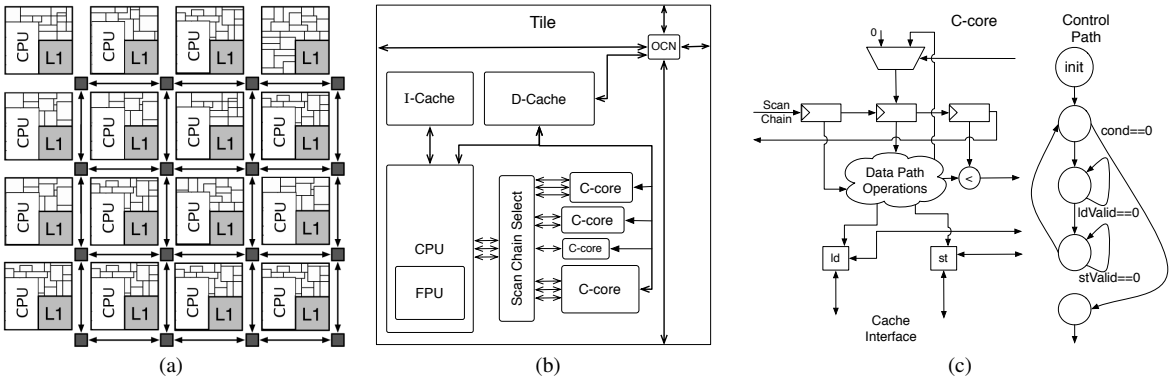


Figure 1. The high-level structure of a conservation-core-enabled system. A c-core-enabled system (a) is made up of multiple individual tiles (b), each of which contains multiple c-cores (c). Conservation cores communicate with the rest of the system through a coherent memory system and a simple scan-chain-based interface. Different tiles may contain different c-cores.

mobile systems) must operate within fixed power budgets. Next year's chip may bring twice as many transistors, but it can only use half of them at full speed.

The UCSD Computer Architecture group's GreenDroid processor architecture is an attempt to tackle the utilization wall using specialized energy-reducing logic cores called *conservation cores*. The GreenDroid processor targets the Android mobile operating system. The Android operating system is analyzed and profiled to find blocks of code that consume significant amounts of runtime and hence energy. These blocks are automatically synthesized into conservation cores. At runtime, when software execution enters a targeted block of code, the CPU pauses and the conservation core runs. The conservation core performs the needed computation very efficiently, without spending energy on instruction cache, instruction fetch & decode, pipeline registers, bypassing, reordering, or any of the other overheads in a general-purpose processor pipeline.

The GreenDroid chip is a collection of conservation cores and general-purpose CPU cores, tiled in a two-dimensional tiled array. **Figure 1** (borrowed from [Venkatesh]) shows the high-level structure of a GreenDroid system. Architecturally, the GreenDroid chip is an evolution of the Raw processor architecture [Taylor02]. It is a two-dimensional grid of computation tiles, connected by an on-chip mesh network. Each tile contains a general-purpose processor core (used for code not mapped to conservation cores), several conservation cores, an L1 data cache, and an on-chip network router. Memory and I/O traffic are routed through the network until they reach the edge of the tile array, where they are routed off-

chip to various I/O interfaces. Each link in the on-chip network is a full-duplex 32 bit wide channel. Messages are sent as one or more 32-bit data words.

All communication, including IPC, device I/O, and main memory access, travels as messages on the on-chip network. The main memory controller responds to read and write request messages. Peripheral I/O devices also communicate with messages on the on-chip network. The on-chip network serves roughly the same functions that PCI and HyperTransport/QuickPath do in modern PCs: they are the fabric linking together all devices in the system, including the CPU.

Project Overview

The GreenDroid research group plans to fabricate a prototype in silicon, sharing the die and the design effort with the UC Santa Cruz architecture group [Renau]. Research designs from both universities will share common power, clock, test, and debug resources. A common ring bus will connect all designs to each other and to I/O resources. The ring bus is called MURN ("Multi-University Research Network").

The GreenDroid design has its own on-chip network design and its own I/O assumptions (discussed above). In order for GreenDroid to fit into the common chip and interoperate with its neighbors from Santa Cruz, its on-chip network must be adapted to run on the MURN bus. Each network port around the edge of the GreenDroid tile array will be connected to a *virtual channel*. Traffic for all virtual channels will be multiplexed over the MURN bus, as many TCP streams are multiplexed over an Ethernet link. (See **Figure 2**).

Striving to be good computer scientists, when presented with a challenge, we attempt to generalize our solution to cover all possible future challenges of that type [Munroe]. We have not simply modified the GreenDroid on-chip network to run over the MURN bus. Instead, we have designed a modular adaptor that can transport the GreenDroid network over *any physical interface*. This is our contribution: A hardware module which connects to many GreenDroid on-chip network ports on one side, and on the other connects to a MURN bus; traffic on the network ports will be carried over the MURN bus to a matching remote module. We call this module the io_master[2].

---

[2] io_master: I/O = input/output; master as in bus master: a device which can both initiate and respond to bus transactions. Also, in another sense, it is "in charge of" many I/O links.

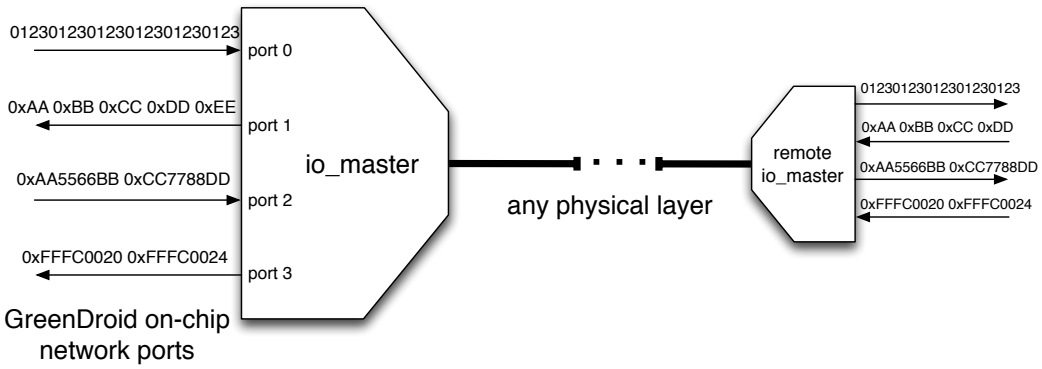


Figure 2: The io_master multiplexes many GreenDroid on-chip network ports onto any physical layer. At the other end of the physical link, a counterpart remote io_master demultiplexes data back to on-chip network ports. Each port is unidirectional, but the io_master carries both sending and receiving ports.

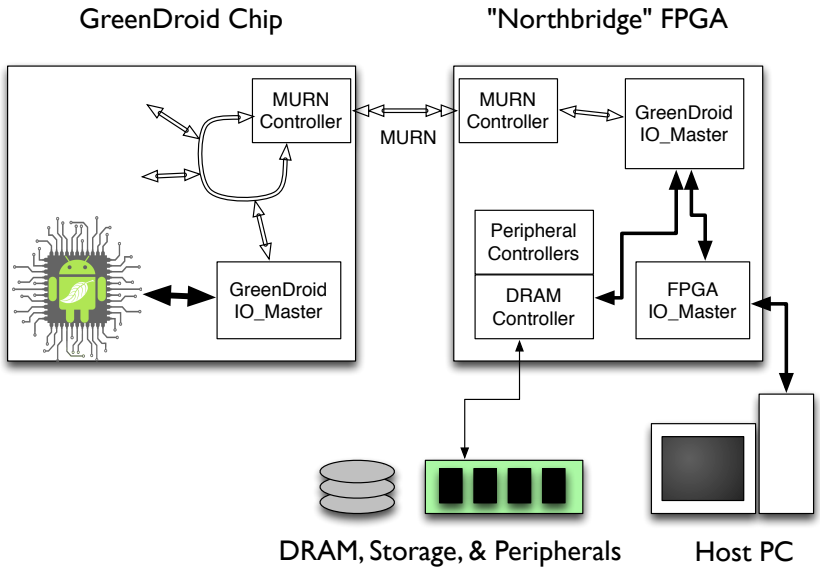
System Design

Outside the io_master

The full GreenDroid system comprises the GreenDroid prototype chip, a "northbridge" FPGA, and a host PC. **Figure 3** is a diagram of the system.

The prototype chip will be fabricated in collaboration with researchers at UC Santa Cruz. It contains several research designs from UCSC, the GreenDroid design from UCSD, and shared power, clock, and I/O resources. The "northbridge" FPGA is so called because it serves approximately the function of the northbridge chip in PC chipsets: it connects the CPU to main memory, some fast I/O devices, and to a southbridge for slower I/O devices. In the GreenDroid system the FPGA connects to the GreenDroid CPU, it contains a DRAM controller for main memory, it may contain a basic graphics

Figure 3: The complete GreenDroid prototype system comprises prototype chip including GreenDroid design, northbridge FPGA including memory controller, and host PC.



adapter, and it connects to the host PC. The host PC controls GreenDroid: it loads programs into the chip, starts them, and monitors their progress. The host PC also provides several emulated I/O devices, and performs filesystem requests for the GreenDroid programs. (That is, when a GreenDroid program reads and writes files, it's actually reading and writing the host PC's files. This allows us to run complex test programs on the GreenDroid prototype without the burden of first porting an entire operating system).

The GreenDroid design is only one module of several in the research prototype chip. It does not have direct access to the chip's I/O pins. It can communicate over the MURN ring bus to the other research designs and to some sort of off-chip interface.

The MURN bus will be responsible for communicating between chip and FPGA. We presume there will be an IP core or ready-to-use source code for the FPGA.

Inside the io_master

The io_master comprises many submodules. We will discuss them in the order an outbound message would pass through them.

Arbitrator: Many ports connect to the io_master. If several ports try to send a message at the same time, the arbitrator chooses a winner. The policy is round-robin. The arbitrator is also responsible for counting credits for flow control: it will not accept a message until the destination has buffer space available to accept the message. See the discussion of flow control below.

Encoder: The message is encoded into a "packet". Each packet contains the data message and the port number it was sent from. Some packets are used for flow control (see the discussion below).

Serializer: The packet is sliced into several "frames". The size of each "frame" is a property of the physical layer. Ethernet and RS-232 use 8-bit frames. MURN is expected to use 72-bit frames. A packet may span frames, and one frame may contain the end of one packet and the beginning of the next.

Physical layer transmitter: The frames are transmitted on the physical layer.

Physical layer receiver: The frames are received from the physical layer by a remote io_master.

Flow control FIFO: There is a large FIFO (first-in first-out queue) which buffers all incoming frames. If the deserializer and decoder are running slowly and cannot accept new data, the flow control FIFO will fill. Some physical layers do not perform hop-by-hop flow control, so if the deserializer cannot

accept new data, the physical layer may simply drop a frame. Ethernet, especially, works this way. In the GreenDroid on-chip network, drops are not permitted. The flow control FIFO is sized to ensure it can accept bursts up to the maximum number of credits (see discussion of flow control, below) across all ports, so that the io_master will always accept data from the physical layer, and hence avoid drops.

Deserializer: Data from several frames is concatenated. Enough frames are buffered to sum to a maximum-length packet. When requested by the decoder, a packet is dequeued from the deserializer.

Decoder: The decoder examines the frames buffered in the deserializer and attempts to recognize valid packets. When enough data has arrived to form a full packet, the decoder asks the deserializer to dequeue the entire packet. If packets vary in length (see discussion of encoding overhead below), the decoder is responsible for identifying the length of the packet. The decoder pulls the message data and port number out of the packet.

Receiver Buffers/Demultiplexer: The received packet data is buffered in a FIFO for its destination port. These buffers allow some ports to run slowly without delaying packets for other ports. Large buffers allow many messages to be in flight between sender and receiver, which can improve link utilization and throughput. (See the discussion of flow control below.)

Synchronizer: When the link between two io_masters becomes active, they perform a synchronization process. At the end of the synchronization process, the first bit of the first valid packet will be in the first bit position of the deserializer, ready for the decoder to decode it. During synchronization, each io_master transmits repeating patterns onto the physical layer, and detects and locks onto the patterns transmitted by the other.

Discussion

During the development and testing of the io_master, we met many challenges and made design decisions to overcome them. We will discuss three challenges: flow control across the link; maximizing performance by minimizing encoding overhead; and independence from any specific physical layer.

Flow control

GreenDroid's on-chip networks use *credit-based flow control*. The receiving side of a link has a buffer for incoming messages. The sending side has a counter of how many buffer slots (*credits*) are available. Each time the sender sends a message, it decrements its credit counter; when the counter reaches 0, the sender stops sending and waits. Each time the receiver accepts a message from the buffer, it replies to the sender, incrementing the credit counter. This method of flow control is simple to implement, makes efficient use of buffer space, and if round-trip time is known, permits full utilization of the link [Taylor05]. Credits and buffers allow many messages to be in flight from sender to receiver, which makes good use of the link. If the receiver stops processing messages, they will queue up in the buffer without being dropped. The sender stops sending data just as the buffer fills. When the receiver begins processing messages again, it drains the buffer at high speed. As the buffer empties, credits are returned to the sender, which begins sending again. Since the number of credits needed is based on the round-trip time, credit-based flow control is most useful for links with known, fixed round-trip times.

In the extended virtual network, both data messages and returned credits are sent as packets. Our initial io_master implementation uses a *virtual credit return channel*, using an unallocated port number. Credit return packets are a bit vector with three[3] bits per port, counting how many credits are now available to the sender on each port. Credit return packets are generated by the demultiplexer (which aggregates credits returned from the receiving ports) and processed by the arbitrator (which counts credits available to send).

A naive implementation would send a large number of credit return messages: in the worst case, one credit return message per data message, for overhead of 50%! We would like to decrease this overhead. Our first improvement is to return several credits in each message: each port returns up to 7 credits (expressed in 3 bits). This decreases credit return overhead to 12.5%. For very high-throughput links, we would like to decrease overhead even further. Our proposed second improvement is *credit decimation*: each bit in the credit return packet indicates the return of many credits. The *decimation*

[3] Adjustable at design time.

factor[4] is a design-time parameter describing how many credits are returned: the bit value is multiplied by 2^(decimation factor). For example, at a decimation factor of 4, each credit return message may indicate between 0 and 7*16 = 112 credits (3 bits encodes up to 7; credits are returned 2^4 = 16 at a time). Due to the loss of precision, credit decimation will waste some space in the receive buffer: up to 1 * 2^(decimation factor) entries may be underutilized. We consider it a good tradeoff to exchange precious link bandwidth for relatively cheap die area.

Encoding overhead

The io_master multiplexes many types of network traffic onto a variety of physical layers. Since the GreenDroid processor accesses its main DRAM memory over the on-chip network, cache line reads and writes will be carried over the io_master. Software performance is *extremely* sensitive to main memory latency. We wish to maximize memory performance, even over very slow physical layers. In order to maximize performance (maximize throughput and minimize latency), we will attempt to minimize the overhead of packet encoding.

All packets must be encoded to travel over the physical link. At minimum, the receiver must recover the port number and message data. The trivial encoding thus uses 36 bits: 32 bits for data identity encoded + 4 bits for port number[5], for 12.5% overhead.

Table 1: Trivial data packet encoding for 0xDEADBEEF sent by port 8

port number (4 bits)	data (32 bits)
1000	11011110101011011011110111011111

Credit return packets use a bit vector of 3 bits per port to return up to 7 credits per port. Credit return packets are differentiated from data packets by using a reserved channel number. Each credit return packet is nearly as big as a data packet. Overhead depends on how frequently return packets are sent and how many credits they return—see the previous section in flow control.

[4] It should rightly be called the “decimation exponent”, but we consider this to be a less euphonious term.

[5] This is calculated at design time based on how many ports are actually connected to the io_master. We anticipate the prototype chip will need 12 ports, therefore 4 bits for port number.

Table 2: Simple credit return packet encoding for (0, 1, 0, 3, 0, 0, 3, 0, 1, 0) credits on ports (0, ... 9). Port number 11, in this example, is allocated to the credit return virtual channel.

port number (4 bits)	credits (3*10=30 bits)
1011	000 001 000 011 000 000 011 000 001 000

If there are credits to return from many channels at the same time, the bit vector is an efficient representation. However, if only one or a few channels are in use, most of the bits will be wasted as zeroes. A proposed alternative encoding for credit return packets includes a port number and count of credits to return on that port. This can reduce the packet size to 11 bits, a reduction of 67%.

Table 3: More efficient credit return packet encoding for 3 credits on port 6

port number (4 bits)	credits (3 bits)	credit port number (4 bits)
1110	011	1010

Finally, we have designed a dense encoding that can even save the bits allocated to port number. This encoding uses the first several bits as a packet type code, with more common packet types encoded in fewer bits, in the spirit of Huffman coding. To avoid sending port numbers, this encoding uses *channel prediction*, which we believe to be a novel contribution[6]. Both sender and receiver implement a channel predictor, a small logic block akin to a standard processor's branch predictor. The channel predictors learn the stream of channels that are currently in use, and make a prediction for the channel of the next message. The two predictors are identical and run in lockstep with each packet, so they will always make the same prediction. If the sender's predictor correctly predicts the channel number of the next packet to be sent, the channel number will not be encoded into the packet. The receiver, seeing this, uses its (identical) prediction to route the packet. Channel numbers are only sent on mispredictions. Even a very low prediction rate will save some bits: at pure chance, 1/N correct predictions will save log(N)/N bits per packet. In practice we anticipate much higher prediction rates, as the predictor adapts to which channels are heavily used and which are inactive.

[6] Credit for channel prediction goes to Professor Taylor, not to this author.

Table 4: Length-optimized encoding.

prefix	suffix	credit?	data?	new c chan?	new d chan?	length
00	whitespace					2
01	32b data		Y			34
10	32b data		Y			34
1100	4b chan, 32b data	Y	Y		Y	40
1101	4b chan, 4b chan, 32b data	Y	Y	Y	Y	44
1100	4b chan	Y		Y		8
111100	none	Y				6

We have designed the io_master's encoder and decoder to be self-contained modules. Only the encoder and decoder module know the details of packet formats; the other modules either deal with a stream of encoded bits, or with the unencoded credit counts, port numbers, and message data. This decreases the design effort to try novel encodings and evaluate their performance. We anticipate the final implementation of the io_master will be able to switch at runtime between several encodings, as traffic flow varies.

Physical layer independence

We have designed the io_master to function independently of the physical layer it runs over. There is a very simple, general interface between io_master and physical layer; any link that can present this interface can carry GreenDroid on-chip network traffic. The io_master sees the physical layer as a channel of "words" of fixed bit width that are conveyed to the remote io_master as they were sent, without reordering or loss. Words are fixed in size for a particular physical layer, but different physical layers may have different word sizes. RS-232 serial (UART) uses 8-bit words. Our initial gigabit Ethernet implementation uses 8-bit words written into 64-byte frames. We anticipate that MURN will transmit 72-bit words.

Implementation

We have implemented the io_master in SystemVerilog HDL and merged it into the GreenDroid source code. GreenDroid was synthesized for a Xilinx Virtex-5 FPGA. During development, we also tested the io_master in Synopsys VCS simulation.

On the host PC, we have written a software io_master module in C++ and merged it into the GreenDroid test infrastructure, which was largely inherited from the Raw processor.

We have tested the io_master with several test programs, including MCF and VPR from the SpecCPU benchmark suite, and Autocorrelation, Cjpeg, and Viterbi from EEMBC. These programs are copied from the host to the GreenDroid's instruction memory (over the gigabit Ethernet io_master), then they perform file and console I/O (again over the io_master), and load and store cache lines from a software-emulated main memory device (also over the io_master), then finally return their results to the host (over the io_master). The programs run to completion and produce correct results.

Conclusion

The utilization wall demands processor architects to design novel microarchitectures to continue improving performance and power consumption. However, practical concerns when fabricating prototype chips can impose design constraints. The GreenDroid processor was designed to send all I/O and memory traffic over an on-chip mesh network to I/O controllers placed around the perimeter of the tile array. When the GreenDroid prototype chip is fabricated, it will not have direct access to the chip's I/O pins, but must route all its I/O over a relatively narrow ring bus. To connect the GreenDroid processor to a northbridge FPGA and a host PC requires a hardware component that can transport on-chip network messages over a different physical layer. We present the design and implementation of such a hardware component: the GreenDroid io_master. The io_master tunnels many on-chip network ports over RS-232, Ethernet, MURN ring bus, or another physical layer, and presents the abstraction of a single seamless mesh. By extending the on-chip network across dies, we may implement I/O devices and test software in convenient FPGA and PC environments, while preserving the GreenDroid programming model.

Acknowledgements

I wish to thank the following people for their assistance with, accommodations made for, and/or contributions to this work: Professor Michael Taylor, Alex Amirnovin, D J Capelis, Nathan Goulding-Hotta, Keaton Mowery, Jack Sampson, Jon Schifman, Beth Scott, and Qiaoshi Zheng. I further extend my deepest thanks to all who have supported me during graduate school: my parents Rick and Amy and my sister Jessie, Teresa Mao, Scott Perry, Rushi Chakrabarti, Paul Knight, Ava Pierce, Ben Morris, Lauren Brown-Cornell, Michelle Olofson, and many more too numerous to name.

References

[Goulding-Hotta] N. Goulding-Hotta, et al. “GreenDroid: Exploring the next evolution in smartphone application processors”. Communications Magazine, IEEE 49(4):112-119, Apr 2011.

[Munroe] R. Munroe. “The General Problem”. XKCD, Nov 2011. [http://xkcd.com/974/]

[Renau]J. Renau. “Computer Architecture @ UCSC: The MASC Group”. Slides presented in CMPS/CMPE 200, Fall 2011. [http://classes.soe.ucsc.edu/cmps200/Fall11/facpresent/jose.pdf]

[Taylor02] M. Taylor et al. ‘‘The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs”. IEEE Micro, vol. 22, no. 2, 2002.

[Taylor05] M. Taylor. “The Raw Prototype Design Document, v5.02”. MIT Dept. EECS, Dec 2005.

[Venkatesh] G. Venkatesh, et al. “Conservation Cores: Reducing the Energy of Mature Computations”. ASPLOS (ACM), Mar 2010.