SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering [§]

Bhavya K. Daya, Chia-Hsin Owen Chen, Suvinay Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P. Chandrakasan, Li-Shiuan Peh[†]

Massachusetts Institute of Technology

Abstract

In the many-core era, scalable coherence and on-chip interconnects are crucial for shared memory processors. While snoopy coherence is common in small multicore systems, directory-based coherence is the de facto choice for scalability to many cores, as snoopy relies on ordered interconnects which do not scale. However, directory-based coherence does not scale beyond tens of cores due to excessive directory area overhead or inaccurate sharer tracking. Prior techniques supporting ordering on arbitrary unordered networks are impractical for full multicore chip designs.

We present SCORPIO, an ordered mesh Network-on-Chip (NoC) architecture with a separate fixed-latency, bufferless network to achieve distributed global ordering. Message delivery is decoupled from the ordering, allowing messages to arrive in any order and at any time, and still be correctly ordered. The architecture is designed to plug-and-play with existing multicore IP and with practicality, timing, area, and power as top concerns. Full-system 36 and 64-core simulations on SPLASH-2 and PARSEC benchmarks show an average application runtime reduction of 24.1% and 12.9%, in comparison to distributed directory and AMD HyperTransport coherence protocols, respectively.

The SCORPIO architecture is incorporated in an 11 mm-by-13 mm chip prototype, fabricated in IBM 45nm SOI technology, comprising 36 Freescale e200 Power Architecture™ cores with private L1 and L2 caches interfacing with the NoC via ARM AMBA, along with two Cadence on-chip DDR2 controllers. The chip prototype achieves a post synthesis operating frequency of 1 GHz (833 MHz post-layout) with an estimated power of 28.8 W (768 mW per tile), while the network consumes only 10% of tile area and 19 % of tile power.

1. Introduction

Shared memory, a dominant communication paradigm in mainstream multicore processors today, achieves inter-core communication using simple loads and stores to a shared address space, but requires mechanisms for ensuring cache coherence. Over the past few decades, research in cache coherence has led to solutions in the form of either snoopy or directory-based variants. However, a critical concern is whether hardware-based coherence will scale with the increasing core counts of chip multiprocessors [18, 21]. Existing coherence schemes can provide accurate functionality for up to hundreds of cores, but area, power, and bandwidth overheads affect their practicality. Two main scalability concerns are (1) directory storage overhead, and (2) uncore (caches+interconnect) scaling.

For scalable directory-based coherence, the directory storage overhead has to be kept minimal while maintaining accurate sharer information. Full bit-vector directories encode the set of sharers of a specific address. For a few tens of cores it is very efficient, but requires storage that linearly grows with the number of cores; limiting its use for larger systems. Alternatives, such as coarse-grain sharer bit-vectors and limited pointer schemes contain inaccurate sharing information, essentially trading performance for scalability. Research in scalable directory coherence is attempting to tackle the storage overhead while maintaining accurate sharer information, but at the cost of increased directory evictions and corresponding network traffic as a result of the invalidations.

Snoopy coherence is not impacted by directory storage overhead, but intrinsically requires an ordered network to ensure all cores see requests in the same order to maintain memory consistency semantics. Snoopy compatible interconnects comprise buses or crossbars (with arbiters to order requests), or bufferless rings (which guarantee in-order delivery to all cores from an ordering point). However, existing on-chip ordered interconnects scale poorly. The Achilles heel of buses lie in limited bandwidth, while that of rings is delay, and for crossbars, it is area. Higher-dimension NoCs such as meshes provide scalable bandwidth and is the subject of a plethora of research on low-power and low-latency routers, including several chip prototypes [16, 17, 27, 30]. However, meshes are unordered and cannot natively support snoopy protocols.

Snoopy **CO**herent **R**esearch **P**rocessor with **I**nterconnect **O**rdering (SCORPIO) incorporates global ordering support within the mesh network by decoupling message delivery from the ordering. This allows flits to be injected into the NoC and reach destinations in any order, at any time, and still maintain a consistent global order. The SCORPIO architecture was included in an 11 mm-by-13 mm chip prototype in IBM 45 nm SOI, to interconnect 36 Freescale e200 cores, comprising private L1 and L2 caches, and two Cadence on-chip DDR

[§]This work was supported by DARPA UHPC grant at MIT (Angstrom) and by the Center for Future Architectures Research (C-FAR), one of six SRC STARnet Centers, sponsored by MARCO and DARPA.

[†]Tushar Krishna and Jim Holt performed the research while at MIT as PhD student and visiting research scientist from Freescale respectively.

978-1-4799-4394-4/14/$31.00

controllers. The SCORPIO NoC is designed to comply with the ARM AMBA interface [2] to be compatible with existing SoC IP originally designed for AMBA buses.

Section 2 covers prior work on snoopy coherence on unordered networks. Section 3 delves into the overview and microarchitecture of the globally ordered mesh network. Section 4 describes the 36-core chip with the SCORPIO NoC. Section 5 presents the architecture evaluations, design exploration, and area and power results. Section 6 discusses related multicore chips and NoC prototypes, and Section 7 concludes.

2. Background

Various proposals, such as Token Coherence (TokenB), Uncorq, Time-stamp snooping (TS), and INSO extend snoopy coherence to unordered interconnects. TokenB [23] performs the ordering at the protocol level, with tokens that can be requested by a core wanting access to a cacheline. TokenB assigns T tokens to each block of shared memory during system initialization (where T is at least equal to the number of processors). Each cacheline requires an additional $2 + \log T$ bits. Although each token is small, the total area overhead scales linearly with the number of cachelines.

Uncorq [29] broadcasts a snoop request to all cores followed by a response message on a logical ring network to collect the responses from all cores. This enforces a serialization of requests to the same cacheline, but does not enforce sequential consistency or global ordering of all requests. Although read requests do not wait for the response messages to return, the write requests have to wait, with the waiting delay scaling linearly with core count, like physical rings.

TS [22] assigns logical time-stamps to requests and performs the reordering at the destination. Each request is tagged with an ordering time (OT), and each node maintains a guaranteed time (GT). When a node has received all packets with a particular OT, it increments the GT. TS requires a large number of buffers at the destinations to store all packets with a particular OT, prior to processing time. The required buffer count linearly scales with the number of cores and maximum outstanding requests per core. For a 36-core system with 2 outstanding requests per core, there will be 72 buffers at each node, which is impractical and will grow significantly with core count and more aggressive cores.

INSO [11] tags all requests with distinct numbers (snoop orders) that are unique to the originating node which assigns them. All nodes process requests in ascending order of the snoop orders and expect to process a request from each node. If a node does not inject a request, it is has to periodically expire the snoop orders unique to itself. While a small expiration window is necessary for good performance, the increased number of expiry messages consume network power and bandwidth. Experiments with INSO show the ratio of expiry messages to regular messages is 25 for a time window of 20 cycles. At the destination, unused snoop orders still need to be processed leading to worsening of ordering latency.

3. Globally Ordered Mesh Network

Traditionally, global message ordering on interconnects relies on a centralized ordering point, which imposes greater *indirection*[1] and *serialization latency*[2] as the number of network nodes increases. The dependence on the centralized ordering point prevents architects from providing global message ordering guarantee on scalable but unordered networks.

To tackle the problem above, we propose the SCORPIO network architecture. We eliminate the dependence on the centralized ordering point by decoupling message ordering from message delivery using two physical networks:

Main network. The main network is an unordered network and is responsible for broadcasting actual coherence requests to all other nodes and delivering the responses to the requesting nodes. Since the network is unordered, the broadcast coherence requests from different source nodes may arrive at the network interface controllers (NIC) of each node in any order. The NICs of the main network are then responsible for forwarding requests in global order to the cache controller, assisted by the notification network.

Notification network. For every coherence request sent on the main network, a notification message encoding the source node's ID (SID) is broadcast on the notification network to notify all nodes that a coherence request from this source node is in-flight and needs to be ordered. The notification network microarchitecture will be detailed later in Section 3.3; Essentially, it is a bit vector where each bit corresponds to a request from a source node, so broadcasts can be merged by OR-ing the bit vectors in a contention-less manner. The notification network thus has a fixed maximum network latency bound. Accordingly, we maintain synchronized time windows, greater than the latency bound, at each node in the system. We synchronize and send notification messages only at the beginning of each time window, thus guaranteeing that all nodes received the same set of notification messages at the end of that time window. By processing the received notification messages in accordance with a *consistent* ordering rule, all network interface controllers (NIC) determine *locally* the *global* order for the actual coherence requests in the main network. As a result, even though the coherence requests can arrive at each NIC in any order, they are serviced at all nodes in the same order.

Network interface controller. Each node in the system consists of a main network router, a notification router, as well as a network interface controller or logic interfacing the core/cache and the two routers. The NIC encapsulates the coherence requests/responses from the core/cache and injects them into the appropriate virtual networks in the main network. On the receive end, it forwards the received coherence requests to the core/cache in accordance with the global order, which is

[1] Network latency of a message from the source node to ordering point.
[2] Latency of a message waiting at the ordering point before it is ordered and forwarded to other nodes.

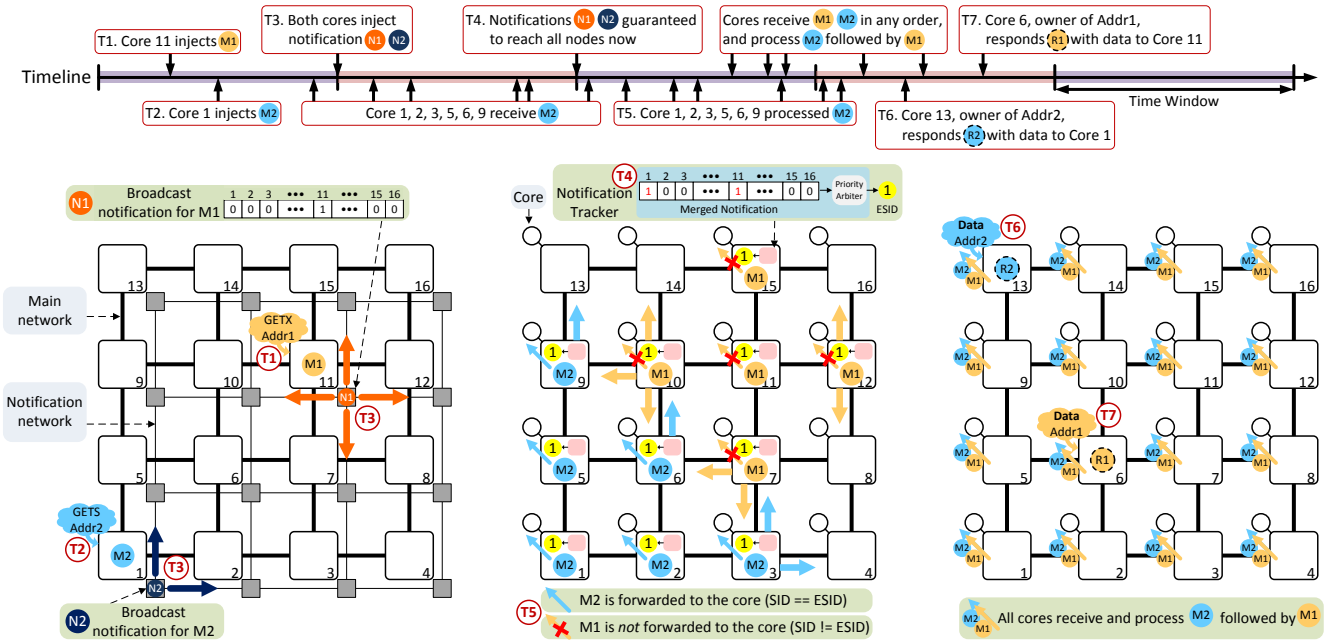


Figure 1: SCORPIO Ordered Network 16-Node Walkthrough Example

determined using the received notification messages at the end of each time window. The NIC uses an *Expected Source ID* (ESID) register to keep track of and informs the main network router which coherence request it is waiting for. For example, if the ESID stores a value of 3, it means that the NIC is waiting for a coherence request from node 3 and would not forward coherence requests from other nodes to the core/cache. Upon receiving the request from node 3, the NIC updates the ESID and waits for the next request based on the global order determined using the received notification messages. The NIC forwards coherence responses to the core/cache in any order.

3.1. Walkthrough Example

The walkthrough example in Figure 1 demonstrates how two messages are ordered.

1. At times T1 and T2, the cache controllers inject cache miss messages M1, M2 to the NIC at cores 11, 1 respectively. The NICs encapsulate these coherence requests into single flit packets, tag them with the SID of their source (11, 1 respectively), and broadcast them to all nodes in the main network.
2. At time T3, the start of the time window, notification messages N1 and N2 are generated corresponding to M1 and M2, and sent into the notification network.
3. Notification messages broadcast at the start of a time window are guaranteed to be delivered to all nodes by the end of the time window (T4). At this stage, all nodes process the notification messages received and perform a local but consistent decision to order these messages. In SCORPIO, we use a rotating priority arbiter to order messages according to increasing SID – the priority is updated each time window ensuring fairness. In this example, all nodes decide to process M2 before M1.
4. The decided global order is captured in the ESID register in NIC. In this example, ESID is currently 1 – the NICs are waiting for the message from core 1 (*i.e.* M2).
5. At time T5, when a coherence request arrives at a NIC, the NIC performs a check of its source ID (SID). If the SID matches the ESID then the coherence request is processed (*i.e.* dequeued, parsed and handed to the cache controller) else it is held in the NIC buffers. Once the coherence request with the SID equal to ESID is processed, the ESID is updated to the next value (based on the notification messages received). In this example, the NIC has to forward M2 before M1 to the cache controller. If M1 arrives first, it will be buffered in the NIC (or router, depending on the buffer availability at NIC) and wait for M2 to arrive.
6. Cores 6 and 13 respond to M1 (at T7) and M2 (at T6) respectively. All cores thus process all messages in the same order, *i.e.* M2 followed by M1.

3.2. Main Network Microarchitecture

Figure 2 shows the microarchitecture of the three-stage main network router. During the first pipeline stage, the incoming flit is buffered (BW), and in parallel arbitrates with the other virtual channels (VCs) at that input port for access to the crossbar's input port (SA-I). In the second stage, the winners of SA-I from each input port arbitrate for the crossbar's output ports (SA-O), and in parallel select a VC from a queue of free VCs (VS) [20]. In the final stage, the winners of SA-O traverse the crossbar (ST). Next, the flits traverse the link to the adjacent router in the following cycle.

Single-cycle pipeline optimization. To reduce the network latency and buffer read/write power, we implement looka-

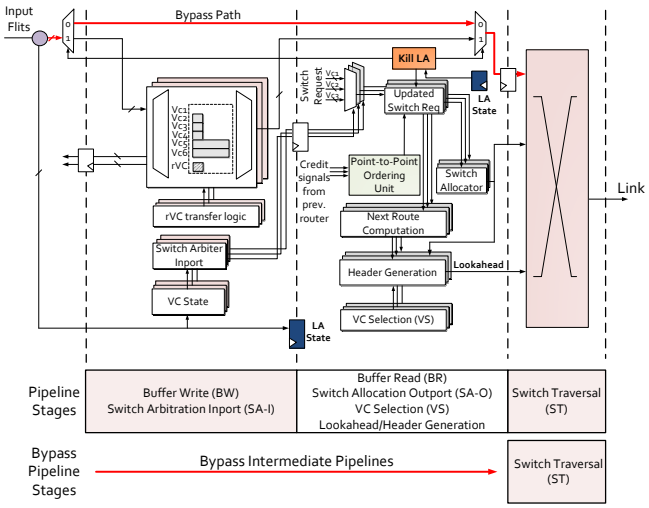


Figure 2: Router Microarchitecture

head (LA) bypassing [19, 27]; a lookahead containing control information for a flit is sent to the next router during that flit's ST stage. At the next router, the lookahead performs route-computation and tries to pre-allocate the crossbar for the approaching flit. Lookaheads are prioritized over buffered flits[3] – they attempt to win SA-I and SA-O, obtain a free VC at the next router, and setup the crossbar for the approaching flits, which then bypass the first two stages and move to ST stage directly. Conflicts between lookaheads from different input ports are resolved using a static, rotating priority scheme. If a lookahead is unable to setup the crossbar, or obtain a free VC at the next router, the incoming flit is buffered and goes through all three stages. The control information carried by lookaheads is already included in the header field of conventional NoCs – destination coordinates, VC ID and the output port ID – and hence does not impose any wiring overhead.

Single-cycle broadcast optimization. To alleviate the overhead imposed by the coherence broadcast requests, routers are equipped with single-cycle multicast support [27]. Instead of sending the same requests for each node one by one into the main network, we allow requests to fork through multiple router output ports in the same cycle, thus providing efficient hardware broadcast support.

Deadlock avoidance. The snoopy coherence protocol messages can be grouped into network requests and responses. Thus, we use two message classes or virtual networks to avoid protocol-level deadlocks:

- **Globally Ordered Request (GO-REQ):** Delivers coherence requests, and provides global ordering, lookahead-bypassing and hardware broadcast support. The NIC processes the received requests from this virtual network based on the order determined by the notification network.
- **Unordered Response (UO-RESP):** Delivers coherence responses, and supports lookahead-bypassing for unicasts. The NIC processes the received responses in any order.

[3]Only buffered flits in the reserved VCs, used for deadlock avoidance, are an exception, prioritized over lookaheads.

The main network uses XY-routing algorithm which ensures deadlock-freedom for the UO-RESP virtual network. For the GO-REQ virtual network, however, the NIC processes the received requests in the order determined by the notification network which may lead to deadlock; the request that the NIC is awaiting might not be able to enter the NIC because the buffers in the NIC and routers enroute are all occupied by other requests. To prevent the deadlock scenario, we add one reserved virtual channel (rVC) to each router and NIC, reserved for the coherence request with SID equal to ESID of the NIC attached to that router.

Proof: Suppose there is a deadlock in the network and the highest priority flit, flit earliest in global order, is unable to make progress. Let flit F be the highest priority flit, stuck at router R with $ESID = E$. If the flit is unable to make progress it implies either *(a)* F is unable to go up to the NIC at router R, or *(b)* F is unable to proceed to a neighboring router S.

Since F is the highest priority flit, it must have SID equal to ESID of the router R because a lower priority ESID is only obtained if the higher priority flit has been received at the NIC. Since a rVC is available for F in the NIC, flit F can be sent to the NIC attached to router R.

Flit F can't proceed to router S if the rVC and other VCs are full. The rVC is full if router S has an ESID with a higher priority than E. This is not possible because F is the highest priority flit which implies any flit of higher priority has already been received at all nodes in the system. For $E1$ with lower or same priority as E, the rVC is available and flit F can make progress. Thus, there is a contradiction and we can ensure that the requests can always proceed toward the destinations.

Point-to-point ordering for GO-REQ. In addition to enforcing a global order, requests from the *same* source also need to be ordered with respect to each other. Since requests are identified by source ID alone, the main network must ensure that a later request does not overtake an earlier request from the same source. To enforce this in SCORPIO, the following property must hold: *Two requests at a particular input port of a router, or at the NIC input queue cannot have the same SID.* At each output port, a SID tracker table keeps track of the SID of the request in each VC at the next router.

Suppose a flit with SID = 5 wins the north port during SA-O and is allotted VC 1 at the next router in the north direction. An entry in the table for the north port is added, mapping (VC 1) $\rightarrow$ (SID = 5). At the next router, when flit with SID = 5 wins all its required output ports and leaves the router, a credit signal is sent back to this router and then the entry is cleared in the SID tracker. Prior to the clearance of the SID tracker entry, any request with SID = 5 is prevented from placing a switch allocation request.

3.3. Notification Network Microarchitecture

The notification network is an ultra-lightweight bufferless mesh network consisting of 5 N-bit bitwise-OR gates and

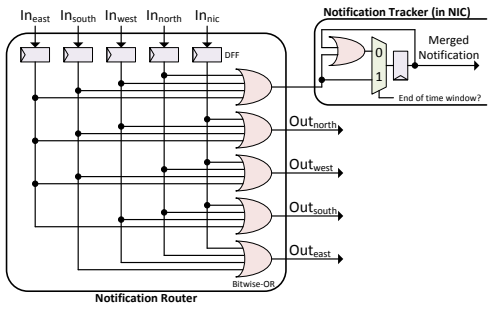


Figure 3: Notification Router Microarchitecture

5 N-bit latches at each "router" as well as N-bit links connecting these "routers", as shown in Figure 3, where N is the number of cores. A notification message is encoded as a N-bit vector where each bit indicates whether a core has sent a coherence request that needs to be ordered. With this encoding, the notification router can merge two notification messages via a bitwise-OR of two messages and forward the merged message to the next router. At the beginning of a time window, a core that wants to send a notification message asserts its associated bit in the bit-vector and sends the bit-vector to its notification router. Every cycle, each notification router merges received notification messages and forwards the updated message to all its neighbor routers in the same cycle. Since messages are merged upon contention, messages can always proceed through the network without being stopped, and hence, no buffer is required and network latency is bounded. At the end of that time window, it is guaranteed that all nodes in the network receive the same merged message, and this message is sent to the NIC for processing to determine the global order of the corresponding coherence requests in the main network.

For example, if node 0 and node 6 want to send notification messages, at the beginning of a time window, they send the messages with bit 0 and bit 6 asserted, respectively, to their notification routers. At the end of the time window, all nodes receive a final message with both bits 0 and 6 asserted. In a 6×6 mesh notification network, the maximum latency is 6 cycles along the X dimension and another 6 cycles along Y, so the time window is set to 13 cycles.

Multiple requests per notification message. Thus far, the notification message described handles one coherence request per node every time window, *i.e.* only one coherence request from each core can be ordered within a time window. However, this is inefficient for more aggressive cores that have more outstanding misses. For example, when the aggressive core generates 6 requests at around the same time, the last request can only be ordered at the end of the 6th time window, incurring latency overhead. To resolve this, instead of using only 1 bit per core, we dedicate multiple bits per core to encode the number of coherence requests that a core wants to order in this time window, at a cost of larger notification message size. For example, if we allocate two bits instead of 1 per core in the notification message, the maximum number of coherence

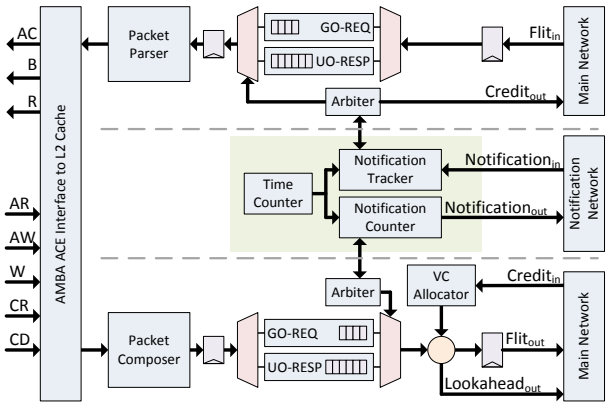


Figure 4: Network Interface Controller Microarchitecture

requests can be ordered in this time window can be increased to three[4]. Now, the core sets the associated bits to the number of coherence requests to be ordered and leaves other bits as zero. This allows us to continue using the bitwise-OR to merge the notification messages from other nodes.

3.4. Network Interface Controller Microarchitecture

Figure 4 shows the microarchitecture of the NIC, which interfaces between the core/cache and the main and notification network routers.

Sending notifications. On receiving a message from core/cache, the NIC encapsulates the message into a packet and sends it to the appropriate virtual network. If the message is a coherence request, the NIC needs to send a notification message so that the coherence request can be ordered. Since the purpose of the notification network is to decouple the coherence request ordering from the request delivery, the NIC can always send the coherence requests to the main network whenever possible and send the corresponding notification messages at the beginning of later time windows. We use a counter to keep track of how many pending notification messages still remain to be sent. The counter can be sized arbitrarily for expected bursts; when the maximum number of pending notification messages, represented by this counter, is reached, the NIC blocks new coherence requests from injecting into the main network.

Receiving notifications. At the end of every time window, the NIC pushes the received merged notification message into the notification tracker queue. When the notification tracker queue is not empty and there is no previously read notification message being processed, the head of the queue is read and passed through a rotating priority arbiter to determine the order of processing the incoming coherence requests (*i.e.* to determine ESIDs). On receiving the expected coherence request, the NIC parses the packet and passes appropriate information to the core/cache, and informs the notification tracker to up-

[4]The number of coherence requests is encoded in binary, where a value of 0 means no request to be ordered, 1 implies 1 request, while 3 indicates 3 requests to be ordered (maximum value that a 2-bit number can represent).

date the ESID value. Once all the requests indicated by this notification message are processed, the notification tracker reads the next notification message in the queue if available and re-iterate the same process mentioned above. The rotating priority arbiter is updated at this time.

If the notification tracker queue is full, the NIC informs other NICs and suppresses other NICs from sending notification messages. To achieve this, we add a "stop" bit to the notification message. When any NIC's queue is full, that NIC sends a notification message with the "stop" bit asserted, which is also OR-ed during message merging; consequently all nodes ignore the merged notification message received; also, the nodes that sent a notification message this time window will resend it later. When this NIC's queue becomes non-full, the NIC sends the notification message with the "stop" bit de-asserted. All NICs are enabled again to (re-)send pending notification messages when the "stop" bit of the received merged notification message is de-asserted.

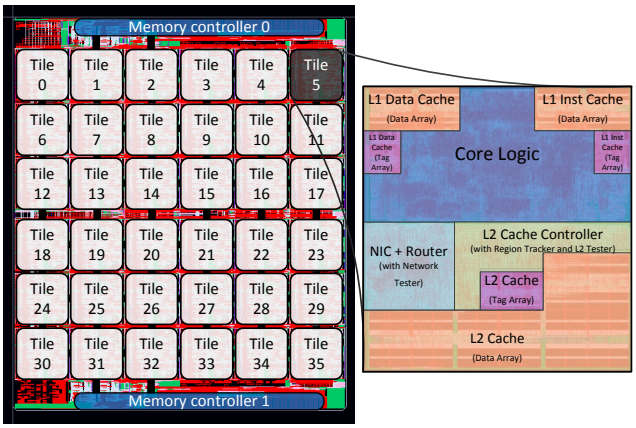


Figure 5: 36-Core chip layout with SCORPIO NoC

4. 36-Core Processor with SCORPIO NoC

The 36-core fabricated multicore processor is arranged in a grid of 6×6 tiles, as seen in Figure 5. Within each tile is an in-order core, split L1 I/D caches, private L2 cache with MOSI snoopy coherence protocol, L2 region tracker for destination filtering [26], and SCORPIO NoC (see Table 1 for a full summary of the chip features). The Freescale e200 core simply assumes a bus is connected to the AMBA AHB data and instruction ports, cleanly isolating the core from the details of the network and snoopy coherence support. Between the network and the processor core IP is the L2 cache with AMBA AHB processor-side and AMBA ACE network-side interfaces. Two Cadence DDR2 memory controllers attach to four unique routers along the chip edge, with the Cadence IP complying with the AMBA AXI interface, interfacing with Cadence PHY to off-chip DIMM modules. All other IO connections go through an external FPGA board with the connectors for RS-232, Ethernet, and flash memory.

Table 1: SCORPIO chip features

Process	IBM 45 nm SOI
Dimension	11×13 mm^2
Transistor count	600 M
Frequency	833 MHz
Power	28.8 W
Core	Dual-issue, in-order, 10-stage pipeline
ISA	32-bit Power Architecture™
L1 cache	Private split 4-way set associative write-through 16 KB I/D
L2 cache	Private inclusive 4-way set associative 128 KB
Line Size	32 B
Coherence protocol	MOSI (O: forward state)
Directory cache	128 KB (1 owner bit, 1 dirty bit)
Snoop filter	Region tracker (4KB regions, 128 entries)
NoC Topology	6×6 mesh
Channel width	137 bits (Ctrl packets – 1 flit, data packets – 3 flits)
Virtual networks	1. Globally ordered – 4 VCs, 1 buffers each 2. Unordered – 2 VCs, 3 buffers each
Router	XY routing, cut-through, multicast, lookahead bypassing
Pipeline	3-stage router (1-stage with bypassing), 1-stage link
Notification network	36-bits wide, bufferless, 13 cycles time window, max 4 pending messages
Memory controller	2× Dual port Cadence DDR2 memory controller + PHY
FPGA controller	1× Packet-switched flexible data-rate controller

4.1. Processor Core and Cache Hierarchy Interface

While the ordered SCORPIO NoC can plug-and-play with existing ACE coherence protocol controllers, we were unable to obtain such IP and hence designed our own. The cache subsystem comprises L1 and L2 caches and the interaction between a self-designed L2 cache and the processor core's L1 caches is mostly subject to the core's and AHB's constraints.

The core has a split instruction and data 16/,KB L1 cache with independent AHB ports. The ports connect to the multiple master split-transaction AHB bus with two AHB masters (L1 caches) and one AHB slave (L2 cache). The protocol supports a single read or write transaction at a time, hence there is a simple request or address phase, followed by a response or data phase. Transactions, between pending requests from the same AHB port, are not permitted thereby restricting the number of outstanding misses to two, one data cache miss and one instruction cache miss, per core. For multilevel caches, snooping hardware has to be present at both L1 and L2 caches. However, the core was not originally designed for hardware coherency. Thus, we added an invalidation port to the core allowing L1 cachelines to be invalidated by external input signals. This method places the inclusion requirement on the caches. With the L1 cache operating in write-through mode, the L2 cache will only need to inform the L1 during invalidations and evictions of a line.

4.2. Coherence Protocol

The standard MOSI protocol is adapted to reduce the writeback frequency and to disallow the blocking of incoming snoop requests. Writebacks cause subsequent cacheline accesses to go off-chip to retrieve the data, degrading performance, hence we retain the data on-chip for as long as possible. To achieve this, an additional O_D state instead of a dirty bit per line is added to permit on-chip sharing of dirty data. For

example, if another core wants to write to the same cacheline, the request is broadcast to all cores resulting in invalidations, while the owner of the dirty data (in M or O_D state) will respond with the dirty data and change itself to the Invalid state. If another cores wants to read the same cacheline, the request is broadcast to all cores. The owner of the dirty data (now in M state), responds with the data and transitions to the O_D state, and the requester goes to the Shared state. This ensures the data is only written to memory when an eviction occurs, without any overhead because the O_D state does not require any additional state bits.

When a cacheline is in a transient state due to a pending write request, snoop requests to the same cacheline are stalled until the data is received and the write request is completed. This causes the blocking of other snoop requests even if they can be serviced right away. We service all snoop requests without blocking by maintaining a forwarding IDs (FID) list that tracks subsequent snoop requests that match a pending write request. The FID consists of the SID and the request entry ID or the ID that matches a response to an outstanding request at the source. With this information, a completed write request can send updated data to all SIDs on the list. The core IP has a maximum of 2 outstanding messages at a time, hence only two sets of forwarding IDs are maintained per core. The SIDs are tracked using a N bit-vector, and the request entry IDs are maintained using 2N bits. For larger core counts and more outstanding messages, this overhead can be reduced by tracking a smaller subset of the total core count. Since the number of sharers of a line is usually low, this will perform as well as being able to track all cores. Once the FID list fills up, subsequent snoop requests will then be stalled.

The different message types are matched with appropriate ACE channels and types. The network interface retains its general mapping from ACE messages to packet type encoding and virtual network identification resulting in a seamless integration. The L2 cache was thus designed to comply with the AMBA ACE specification. It has five outgoing channels and three incoming channels (see Figure 4), separating the address and data among different channels. ACE is able to support snoop requests through its Address Coherent (AC) channel, allowing us to send other requests to the L2 cache.

4.3. Functional Verification

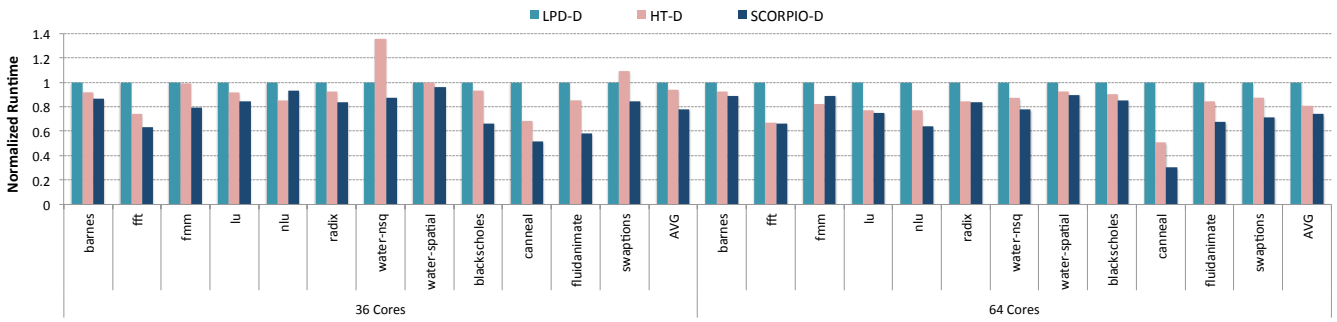
We ensure correct functionality of the SCORPIO RTL using a suite of regression tests that verify the entire chip. Since the core is verified commercial IP, our regression tests focus on verifying integration of various components, which involves (1) load/store operations on both cacheable and non-cacheable regions, (2) lock and barrier instructions, (3) coherency between L1s, L2s and main memory, and (4) software-triggered interrupts. The tests are written in assembly and C, and we built a software chain that compiles tests into machine code.

5. Architecture Analysis

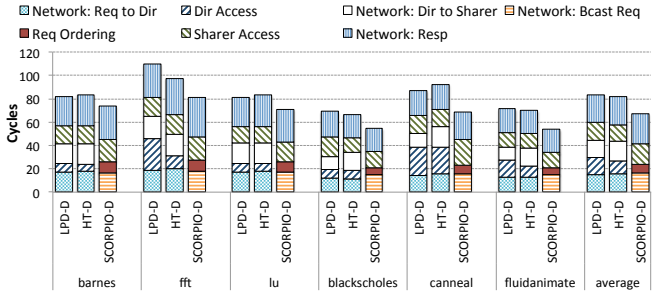
Modeled system. For full-system architectural simulations of SCORPIO, we use Wind River Simics [6] extended with the GEMS toolset [24] and the GARNET [9] network model. The SCORPIO and baseline architectural parameters as shown in Table 1 are faithfully mimicked within the limits of the GEMS and GARNET environment:

- GEMS only models in-order SPARC cores, instead of SCORPIO's Power cores.
- L1 and L2 cache latency in GEMS are fixed at 1 cycle and 10 cycles. The prototype L2 cache latency varies with request type and cannot be expressed in GEMS, while the L1 cache latency of the core IP is 2 cycles.
- The directory cache access latency is set to 10 cycles and DRAM to 80 cycles in GEMS. The off-chip access latency of our chip prototype is unknown as it depends on the PCB board and packaging, which is still being designed. The directory cache access was approximated from the directory cache parameters, but will also vary depending on request type for the chip.
- The L2 cache, NIC, and directory cache accesses are fully-pipelined in GEMS.
- Maximum of 16 outstanding messages per core in GEMS, unlike our chip prototype which has a maximum of two outstanding messages per core.

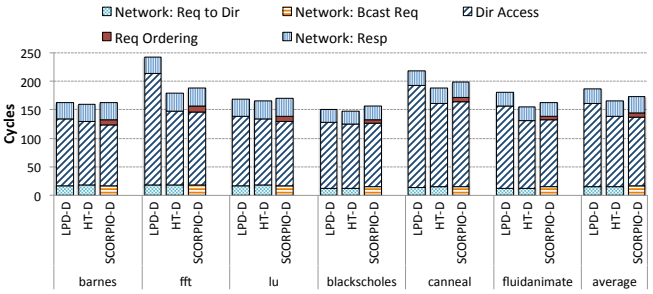
Directory baselines. For directory coherence, all requests are sent as unicasts to a directory, which forwards them to the sharers or reads from main memory if no sharer exists. SCORPIO is compared with two baseline directory protocols. The *Limited-pointer directory* (LPD) [8] baseline tracks when a block is being shared between a small number of processors, using specific pointers. Each directory entry contains 2 state bits, log N bits to record the owner ID, and a set of pointers to track the sharers. We evaluated LPD against full-bit directory in GEMS 36 core full-system simulations and discovered almost identical performance when approximately 3 to 4 sharers were tracked per line as well as the owner ID. Thus, the pointer vector width is chosen to be 24 and 54 bits for 36 and 64 cores, respectively. By tracking fewer sharers, more cachelines are stored within the same directory cache space, resulting in a reduction of directory cache misses. If the number of sharers exceeds the number of pointers in the directory entry, the request is broadcast to all cores. The other baseline is derived from *HyperTransport* (HT) [14]. In HT, the directory does not record sharer information but rather serves as an ordering point and broadcasts the received requests. As a result, HT does not suffer from high directory storage overhead but still incurs on-chip indirection via the directory. Hence for the analysis only 2 bits (ownership and valid) are necessary. The ownership bit indicates if the main memory has the ownership; that is, none of the L2 caches own the requested line and the data should be read from main memory. The valid bit is used to indicate whether main memory has received the writeback data. This



(a) Normalized runtime for 36 and 64 cores



(b) Served by other caches (36 cores)



(c) Served by directory (36 cores)

Figure 6: Normalized Runtime and Latency Breakdown

is a property of the network, where the writeback request and data may arrive separately and in any order because they are sent on different virtual networks.

Workloads. We evaluate all configurations with SPLASH-2 [5] and PARSEC [12] benchmarks. Simulating higher than 64 cores in GEMS requires the use of trace-based simulations, which fail to capture dependencies or stalls between instructions, and spinning or busy waiting behavior accurately. Thus, to evaluate SCORPIO's performance scaling to 100 cores, we obtain SPLASH-2 and PARSEC traces from the Graphite [25] simulator and inject them into the SCORPIO RTL.

Evaluation Methodology. For performance comparisons with baseline directory protocols and prior in-network coherence proposals, we use GEMS to see the relative runtime improvement. The centralized directory in HT and LPD adds serialization delay at the single directory. Multiple distributed directories alleviates this but adds on-die network latency between the directories and DDR controllers at the edge of the chip for off-chip memory access, for both baselines. We evaluate the distributed versions of LPD (LPD-D), HT (HT-D), and SCORPIO (SCORPIO-D) to equalize this latency and specifically isolate the effects of indirection and storage overhead. The directory cache is split across all cores, while keeping the total directory size fixed to 256 KB. Our chip prototype uses 128KB, as seen in Table 1, but we changed this value for baseline performance comparisons only so that we don't heavily penalize LPD by choosing a smaller directory cache.

The SCORPIO network design exploration provides insight into the performance impact as certain parameters are varied. The finalized settings from GEMS simulations are used in the fabricated 36-core chip NoC. In addition, we use behavioral RTL simulations on the 36-core SCORPIO RTL, as well as 64 and 100-core variants, to explore the scaling of the uncore to high core counts. For reasonable simulation time, we replace the Cadence memory controller IP with a functional memory model with fully-pipelined 90-cycle latency. Each core is replaced with a memory trace injector that feeds SPLASH-2 and PARSEC benchmark traces into the L2 cache controller's AHB interface. We run the trace-driven simulations for 400 K cycles, omitting the first 20 K cycles for cache warm-up.

We evaluate the area and power overheads to identify the practicality of the SCORPIO NoC. The area breakdown is obtained from layout. For the power consumption, we perform gate-level simulation on the post-synthesis netlist and use the generated value change dump (VCD) files and Synopsys PrimeTime PX. To reduce the simulation time, we use trace-driven simulations to obtain the L2 and network power consumption. We attach a mimicked AHB slave, that responds to memory requests in a few cycles, to the core and run the Dhrystone benchmark to obtain the core power consumption.

5.1. Performance

To ensure the effects of indirection and directory storage are captured in the analysis, we keep all other conditions equal. Specifically, all architectures share the same coherence protocol and run on the same NoC (minus the ordered virtual network GO-REQ and notification network).

Figure 6 shows the normalized full-system application run-

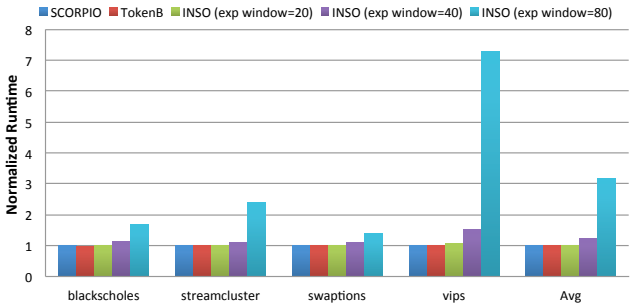


Figure 7: Comparison with TokenB and INSO

time for SPLASH-2 and PARSEC benchmarks simulated on GEMS. On average, SCORPIO-D shows 24.1% better performance over LPD-D and 12.9% over HT-D across all benchmarks. Diving in, we realize that SCORPIO-D experiences average L2 service latency of 78 cycles, which is lower than that of LPD-D (94 cycles) and HT-D (91 cycles). The average L2 service latency is computed over all L2 hit, L2 miss (including off-chip memory access) latencies and it also captures the internal queuing latency between the core and the L2. Since the L2 hit latency and the response latency from other caches or memory controllers are the same across all three configurations, we further breakdown request delivery latency for three SPLASH-2 and three PARSEC benchmarks (see Figure 6). When a request is served by other caches, SCORPIO-D's average latency is 67 cycles, which is 19.4% and 18.3% lower than LPD-D and HT-D, respectively. Since we equalize the directory cache size for all configurations, the LPD-D caches fewer lines compared to SCORPIO-D and HT-D, leading to a higher directory access latency which includes off-chip latency. SCORPIO provides the most latency benefit for data transfers from other caches on-chip by avoiding the indirection latency.

As for requests served by the directory, HT-D performs better than LPD-D due to the lower directory cache miss rate. Also, because the directory protocols need not forward the requests to other caches and can directly serve received requests, the ordering latency overhead makes the SCORPIO delivery latency slightly higher than the HT-D protocol. Since the directory only serves 10% of the requests, SCORPIO still shows 17% and 14% improvement in average request delivery latency over LPD-D and HT-D, respectively, leading to the overall runtime improvement.

To compare SCORPIO's performance with TokenB and INSO, we ran a subset of benchmarks on a 16 core system in GEMS. Figure 7 shows the normalized runtime when keeping all conditions equal besides the ordered network. It was found that SCORPIO's runtime is 19.3% and 70% less than INSO with an expiration window of 40 and 80 cycles, respectively. TokenB's performance is similar to SCORPIO because we do not model the behavior of TokenB in the event of data races where retries and expensive persistent requests affect it significantly. Thus, SCORPIO performs as well as TokenB without persistent requests and INSO with an impractical expiration window size of 20 cycles.

5.2. NoC Design Exploration for 36-Core Chip

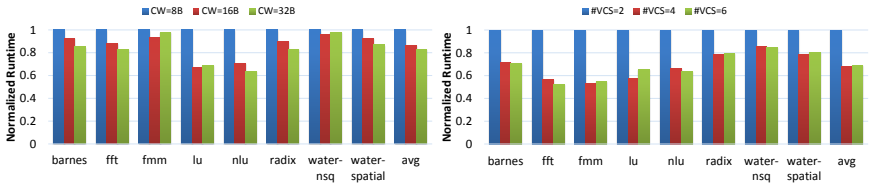
In GEMS, we swept several key SCORPIO network parameters, channel-width, number of VCs, and number of simultaneous notifications, to arrive at the final 36-core fabricated configuration. Channel-width impacts network throughput by directly influencing the number of flits in a multi-flit packet, affecting serialization and essentially packet latency. The number of VCs also affects the throughput of the network and application runtimes, while the number of simultaneous notifications affect ordering delay. Figure 8 shows the variation in runtime as the channel-width and number of VCs are varied. All results are normalized against a baseline configuration of 16-byte channel-width and 4 VCs in each virtual network.

Channel-width. While a larger channel-width offers better performance, it also incurs greater overheads – larger buffers, higher link power and larger router area. A channel-width of 16 bytes translates to 3 flits per packet for cache line responses on the UO-RESP virtual network. A channel-width of 8 bytes would require 5 flits per packet for cache line responses, which degrades the runtime for a few applications. While a 32 byte channel offers a marginal improvement in performance, it expands router and NIC area by 46%. In addition, it leads to low link utilization for the shorter network requests. The 36-core chip contains 16-byte channels due to area constraints and diminishing returns for larger channel-widths.

Number of VCs. Two VCS provide insufficient bandwidth for the GO-REQ virtual network which carries the heavy request broadcast traffic. Besides, one VC is reserved for deadlock avoidance, so low VC configurations would degrade runtime severely. There is a negligible difference in runtime between 4 VCs and 6 VCs. Post-synthesis timing analysis of the router shows negligible impact on the operating frequency as the number of VCs is varied, with the critical path timing hovering around 950ps. The number of VCs indeed affects the SA-I stage, but it is off the critical path. However, a tradeoff of area, power, and performance still exists. Post-synthesis evaluations show 4 VCs is 15% more area efficient, and consumes 12% less power than 6 VCs. Hence, our 36-core chip contains 4 VCs in the GO-REQ virtual network. For the UO-RESP virtual network, the number of VCs does not seem to impact run time greatly once channel-width is fixed. UO-RESP packets are unicast messages, and generally much fewer than the GO-REQ broadcast requests. Hence 2 VCs suffices.

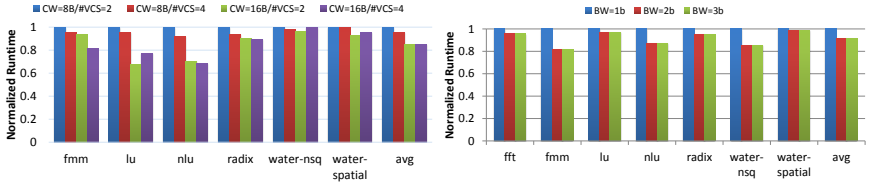
Number of simultaneous notifications. The Freescale e200 cores used in our 36-core chip are constrained to two outstanding messages at a time because of the AHB interfaces at its data and instruction cache miss ports. Due to the low injection rates, we choose a 1-bit-per-core (36-bit) notification network which allows 1 notification per core per time window.

We evaluate if a wider notification network that supports more notifications each time window will offer better performance. Supporting 3 notifications per core per time window, will require 2 bits per core, which results in a 72-bit notification network. Figure 8d shows 36-core GEMS simulations of



(a) Channel-widths

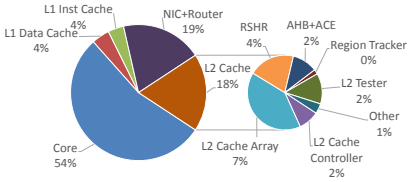
(b) GO-REQ VCs



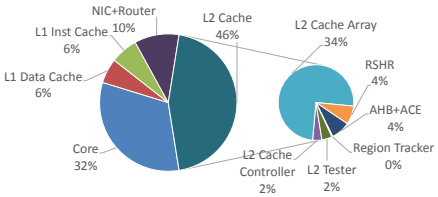
(c) UO-RESP VCs

(d) Simultaneous notifications

Figure 8: Normalized Runtime with Varying Network Parameters



(a) Tile power breakdown



(b) Tile area breakdown

Figure 9: Tile Overheads

SCORPIO achieving 10% better performance for more than one outstanding message per core with a 2-bit-per-core notification network, indicating that bursts of 3 messages per core occur often enough to result in overall runtime reduction. However, more than 3 notifications per time window (3-bit-per-core notification network) does not reap further benefit, as larger bursts of messages are uncommon. A notification network data width scales as $O(m \times N)$, where m is the number of notifications per core per time window. Our 36-bit notification network has $< 1\%$ of tile area and power overheads; Wider data widths only incurs additional wiring which has minimal area and power compared to the main network and should not be challenging given the excess wiring space remaining in the our chip.

5.3. Scaling Uncore Throughput for High Core Counts

As core counts scale, if each core's injection rate (cache miss rate) remains constant, the overall throughput demand on the uncore scales up. We explore the effects of two techniques to optimize SCORPIO's throughput for higher core counts.

Pipelining uncore. Pipelining the L2 caches improves its throughput and reduces the backpressure on the network which may stop the NIC from de-queueing packets. Similarly, pipelining the NIC will relieve network congestion. The performance impact of pipelining the L2 and NIC can be seen in Figure 10 in comparison to a non-pipelined version. For 36 and 64 cores, pipelining reduces the average latency by 15% and 19%, respectively. Its impact is more pronounced as we increase to 100 cores, with an improvement of 30.4%.

Boosting main network throughput with VCs. For good scalability on any multiprocessor system, the cache hierarchy and network should be co-designed. As core count increases, assuming similar cache miss rates and thus traffic injection rates, the load on the network now increases. The theoretical throughput of a $k \times k$ mesh is $1/k^2$ for broadcasts, reducing from 0.027 flits/node/cycle for 36-cores to 0.01 flits/node/cycle for 100-cores. Even if overall traffic across the entire chip remains constant, say due to less sharing or larger caches, a 100-node mesh will lead to longer latencies than a 36-node mesh. Common ways to boost a mesh throughput include multiple meshes, more VCs/buffers per mesh, or wider channel.

Within the limits of the RTL design, we analyze the scalability of the SCORPIO architecture by varying core count and number of VCs within the network and NIC, while keeping the injection rate constant. The design exploration results show that increasing the UO-RESP virtual channels does not yield much performance benefit. But, the OREQ virtual channels matter since they support the broadcast coherent requests. Thus, we increase only the OREQ VCs from 4 VCs to 16 VCs (64 cores) and 50 VCs (100 cores), with 1 buffer per VC. Further increasing the VCs will stretch the critical path and affect the operating frequency of the chip. It will also affect area, though with the current NIC+router taking up just 10% of tile area, this may not be critical. A much lower overhead solution for boosting throughput is to go with multiple main networks, which will double/triple the throughput with no impact on frequency. It is also more efficient area wise as excess wiring is available on-die.

For at least 64 cores in GEMS full-system simulations, SCORPIO performs better than LPD and HT despite the broadcast overhead. The 100-core RTL trace-driven simulation

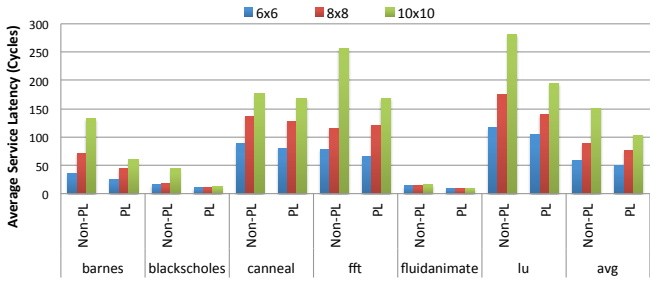


Figure 10: Pipelining effect on performance and scalability

Table 2: Comparison of multicore processors[5]

		Intel Core i7 [7]	AMD Opteron [1]	TILE64 [31]	Oracle T5 [4]	Intel Xeon E7 [3]	SCORPIO
Clock frequency		2–3.3 GHz	2.1–3.6 GHz	750 MHz	3.6GHz	2.1–2.7 GHz	1 GHz (833 MHz post-layout)
Power supply		1.0 V	1.0 V	1.0 V	-	1.0 V	1.1 V
Power consumption		45–130 W	115–140 W	15–22 W	-	130 W	28.8 W
Lithography		45 nm	32 nm SOI	90 nm	28 nm	32 nm	45 nm SOI
Core count		4–8	4–16	64	16	6–10	36
ISA		x86	x86	MIPS-derived VLIW	SPARC	x86	Power
Cache hierarchy	L1D	32 KB private	16 KB private	8 KB private	16 KB private	32 KB private	16 KB private
	L1I	32 KB private	64 KB shared among 2 cores	8 KB private	16 KB private	32 KB private	16 KB private
	L2	256 KB private	2 MB shared among 2 cores	64 KB private	128 KB private	256 KB private	128 KB private
	L3	8 MB shared	16 MB shared	N/A	8 MB	18–30 MB shared	N/A
Consistency model		Processor	Processor	Relaxed	Relaxed	Processor	Sequential consistency
Coherency		Snoopy	Broadcast-based directory (HT)	Directory	Directory	Snoopy	Snoopy
Interconnect		Point-to-Point (QPI)	Point-to-Point (HyperTransport)	5 8×8 meshes	8×9 crossbar	Ring	6×6 mesh

results in Figure 10 show that the average network latency increases significantly. Diving in, we realize that the network is very congested due to injection rates close to saturation throughput. Increasing the number of VCs helps push throughput closer to the theoretical, but is ultimately still constrained by the theoretical bandwidth limit of the topology. A possible solution is to use multiple main networks, which would not affect the correctness because of we decouple message delivery from ordering. Our trace-driven methodology could have a factor on the results too, as we were only able to run 20K cycles for warmup to ensure tractable RTL simulation time; we noticed that L2 caches are under-utilized during the entire RTL simulation runtime, implying caches are not warmed up, resulting in higher than average miss rates.

An alternative to boosting throughput is to reduce the bandwidth demand. INCF [10] was proposed to filter redundant snoop requests by embedding small coherence filters within routers in the network. We leave this for future work.

5.4. Overheads

Power. Overall, the aggregated power consumption of SCORPIO is around 28.8 W and the detailed power breakdown of a tile is shown in Figure 9a. The power consumption of a core with L1 caches is around 62% of the tile power, whereas the L2 cache consumes 18% and the NIC and router 19% of tile power. A notification router costs only a few OR gates; as a result, it consumes less than 1% of the tile power. Since most of the power is consumed at clocking the pipeline and state-keeping flip-flops for all components, the breakdown is not sensitive to workload.

Area. The dimension of the fabricated SCORPIO is $11 \times 13\ mm^2$. Each memory controller and each memory interface controller occupies around $5.7\ mm^2$ and $0.5\ mm^2$ respectively. Detailed area breakdown of a tile is shown in Figure 9b. Within a tile, L1 and L2 caches are the major area contributors, taking 46% of the tile area and the network interface controller together with router occupying 10% of the tile area.

6. Related Work

Multicore processors. Table 2 includes a comparison of AMD, Intel, Tilera, SUN multiprocessors with the SCORPIO chip. These relevant efforts were a result of the continuing challenge of scaling performance while simultaneously managing frequency, area, and power. When scaling from multi to many cores, the interconnect is a significant factor. Current industry chips with relatively few cores typically use bus-based, crossbar or ring fabrics to interconnect the last-level cache, but suffers from poor scalability. Bus bandwidth saturates with more than 8 to 16 cores on-chip [15], not to mention the power overhead of signaling across a large die. Crossbars have been adopted as a higher bandwidth alternative in several multicores [4, 13], but it comes at the cost of a large area footprint that scales quadratically with core counts, worsened by layout constraints imposed by long global wires to each core. From the Oracle T5 die photo, the 8-by-9 crossbar has an estimated area of 1.5X core area, hence about $23mm^2$ at 28nm. Rings are an alternative that supports ordering, adopted in Intel Xeon E7, with bufferless switches (called stops) at each hop delivering single-cycle latency per hop at high frequencies and low area and power. However, scaling to many cores lead to unnecessary delay when circling many hops around the die.

The Tilera TILE64 [31] is a 64-core chip with 5 packet-switched mesh networks. A successor of the MIT RAW chip which originally did not support shared memory [30], TILE64 added directory-based cache coherence, hinting at market support for shared memory. Compatibility with existing IP is not a concern for startup Tilera, with cache, directory, memory controllers developed from scratch. Details of its directory protocol are not released but news releases suggest directory cache overhead and indirection latency are tackled via trading off sharer tracking fidelity. Intel Single-chip Cloud Computer

[5]Core i7 Nehalem Architecture(2008) and Xeon E7-4800 Series(2011) values.

(SCC) processor [17] is a 48-core research chip with a mesh network that does not support shared memory. Each router has a four stage pipeline running at 2 GHz. In comparison, SCORPIO supports in-network ordering with a single-cycle pipeline leveraging virtual lookahead bypassing, at 1 GHz.

NoC-only chip prototypes. Swizzle [28] is a self-arbitrating high-radix crossbar that embeds arbitration within the crossbar to achieve single cycle arbitration. Prior crossbars require high speedup (crossbar frequency at multiple times core frequency) to boost bandwidth in the face of poor arbiter matching, leading to high power overhead. Area remains a problem though, with the 64-by-32 Swizzle crossbar taking up $6.65mm^2$ in 32nm process [28]. Swizzle acknowledged scalability issues and proposed stopping at 64-port crossbars, and leveraging these as high-radix routers within NoCs. There are several other stand-alone NoC prototypes that also explored practical implementations with timing, power and area consideration, such as the 1 GHz Broadcast NoC [27] that optimizes for energy, latency and throughput using virtual bypassing and low-swing signaling for unicast, multicast, and broadcast traffic. Virtual bypassing is leveraged in the SCORPIO NoC.

7. Conclusion

The SCORPIO architecture supports global ordering of requests on a mesh network by decoupling the message delivery from the ordering. With this we are able to address key coherence scalability concerns. While our 36-core SCORPIO chip is an academic chip design that can be better optimized , we learnt significantly through this exercise about the intricate interactions between processor, cache, interconnect and memory design, as well as the practical implementation overheads.

Acknowledgements

SCORPIO is a large project involving 6 students collaborating closely. Key contributions: Core integration(Bhavya,Owen), Coherence protocol design(Bhavya,Woo Cheol), L2 cache(Bhavya), Memory interface(Owen), High-level idea of notification network (Woo Cheol), Network architecture (all), Network RTL(Suvinay), DDR2 integration(Sunghyun,Owen), Backend(Owen), On-chip testers(Tushar), RTL verification(Bhavya,Suvinay), GEMS(Woo Cheol).

References

[1] "AMD Opteron 6200 Series Processors," http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx.

[2] "ARM AMBA," http://www.arm.com/products/system-ip/amba.

[3] "Intel Xeon Processor E7 Family," http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html.

[4] "Oracle's SPARC T5-2, SPARC T5-4, SPARC T5-8, and SPARC T5-1B Server Architecture," http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-024-sparc-t5-architecture-1920540.pdf.

[5] "SPLASH-2," http://www-flash.stanford.edu/apps/SPLASH.

[6] "Wind River Simics," http://www.windriver.com/products/simics.

[7] "First the tick, now the tock: Next generation Intel microarchitecture (Nehalem)," http://www.intel.com/content/dam/doc/white-paper/intel-microarchitecture-white-paper.pdf, 2008.

[8] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Computer Architecture News*, May 1988.

[9] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator," in *ISPASS*, 2009.

[10] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-network coherence filtering: Snoopy coherence without broadcasts," in *MICRO*, 2009.

[11] ——, "In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects," in *HPCA*, 2009.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," 2008.

[13] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Sneger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Fabric," *IEEE Micro*, vol. 32, no. 1, pp. 32–43, 2012.

[14] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture," *IEEE Micro*, vol. 27, pp. 10–21, 2007.

[15] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[16] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S. W. Keckler, and D. Burger, "On-chip interconnection networks of the trips chip," *IEEE Micro*, vol. 27, no. 5, pp. 41–50, 2007.

[17] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borakar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *ISSCC*, 2010.

[18] D. R. Johnson, M. R. Johnson, J. H. Kelm, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Rigel: A 1,024-core single-chip accelerator architecture," *IEEE Micro*, vol. 31, no. 4, pp. 30–41, 2011.

[19] T. Krishna, J. Postman, C. Edmonds, L.-S. Peh, and P. Chiang, "SWIFT: A SWing-reduced Interconnect For a Token-based Network-on-Chip in 90nm CMOS," in *ICCD*, 2010.

[20] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha, "A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator," in *ICCD*, 2007.

[21] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: A 1000-core cache-coherent processor with on-chip optical network," in *PACT*, 2010.

[22] M. M. Martin, M. D. Hill, and D. A. Wood, "Timestamp snooping: An approach for extending smps," in *ASPLOS*, 2000.

[23] ——, "Token coherence: Decoupling performance and correctness," in *ISCA*, 2003.

[24] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *Computer Architecture News*, 2005.

[25] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010.

[26] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *ISCA*, 2005.

[27] S. Park, T. Krishna, C.-H. O. Chen, B. K. Daya, A. P. Chandrakasan, and L.-S. Peh, "Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI," in *DAC*, 2012.

[28] K. Sewell, "Scaling high-performance interconnect architectures to many-core systems," Ph.D. dissertation, University of Michigan.

[29] K. Strauss, X. Shen, and J. Torrellas, "Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors," in *MICRO*, 2007.

[30] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[31] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.