On-Chip Interconnection Architecture of the Tile Processor

iMesh, the Tile Processor Architecture's on-chip interconnection network, connects the multicore processor's tiles with five 2D mesh networks, each specialized for a different use. Taking advantage of the five networks, the C-based iLib interconnection library efficiently maps program communication across the on-chip interconnect. The Tile Processor's first implementation, the TILE64, contains 64 cores and can execute 192 billion 32-bit operations per second at 1 GHz.

David Wentzlaff
Patrick Griffin
Henry Hoffmann
Liewei Bao
Bruce Edwards
Carl Ramey
Matthew Mattina
Chyi-Chang Miao
John F. Brown III
Anant Agarwal
Tilera

As the number of processor cores integrated onto a single die increases, the design space for interconnecting these cores becomes more fertile. One manner of interconnecting the cores is simply to mimic multichip, multiprocessor computers of the past. Following past practice, simple bus-based shared-memory multiprocessors can be integrated onto a single piece of silicon. But, in taking this well-traveled route, we squander the unique opportunities afforded by single-chip integration. Specifically, buses require global broadcast and do not scale to more than about 8 or 16 cores. Some multicore processors have used 1D rings, but rings do not scale well either, because their bisection bandwidth does not increase with the addition of more cores.

This article describes the Tile Processor and its on-chip interconnect network, iMesh, which is a departure from the traditional bus-based multicore processor. The Tile Processor is a tiled multicore architecture developed by Tilera and inspired by MIT's Raw processor.[1,2] A tiled multicore architecture is a multiple-instruction, multiple-data (MIMD) machine consisting of a 2D grid of homogeneous, general-purpose compute elements, called cores or tiles. Instead of using buses or rings to connect the many on-chip cores, the Tile Architecture couples its processors using five 2D mesh networks, which provide the transport medium for off-chip memory access, I/O, interrupts, and other communication activity.

Having five mesh networks leverages the on-chip wiring resources to provide massive on-chip communication bandwidth. The mesh networks afford 1.28 terabits per second (Tbps) of bandwidth into and out of a single tile, and 2.56 Tbps of bisection bandwidth for an 8×8 mesh. By using mesh networks, the Tile Architecture can

0272-1732/07/$20.00 © 2007 IEEE Published by the IEEE Computer Society.

support anywhere from a few to many processors without modifications to the communication fabric. In fact, the amount of in-core (tile) communications infrastructure remains constant as the number of cores grows. Although the in-core resources do not grow as tiles are added, the bandwidth connecting the cores grows with the number of cores.

However, having a massive amount of interconnect resources is not sufficient if they can't be effectively utilized. The interconnect must be flexible enough to efficiently support many different communication needs and programming models. The Tile Architecture's interconnect provides communication via shared memory and direct user-accessible communication networks. The direct user accessible communication networks allow for scalar operands, streams of data, and messages to be passed between tiles without system software intervention. The iMesh interconnect architecture also contains specialized hardware to disambiguate flows of dynamic network packets and sort them directly into distinct processor registers. Hardware disambiguation, register mapping, and direct pipeline integration of the dynamic networks provide register-like intertile communication latencies and enable scalar operand transport on dynamic networks. The interconnect architecture also includes Multicore Hardwall, a mechanism that protects one program or operating system from another during use of directly connected networks.

The Tile Architecture also benefits from devoting each of its five separate networks to a different use. By separating the usage of the networks and specializing the interface to their usage, the architecture allows efficient mapping of programs with varied requirements. For example, the Tile Architecture has separate networks for communication with main memory, communication with I/O devices, and user-level scalar operand and stream communication between tiles. Thus, many applications can simultaneously pull in their data over the I/O network, access memory over the memory networks, and communicate among themselves. This diversity provides a natural way to utilize additional bandwidth, and separates traffic to avoid interference.

Taking advantage of the huge amount of bandwidth afforded by the on-chip integration of multiple mesh networks requires new programming APIs and a tuned software runtime system. This article also introduces iLib, Tilera's C-based user-level API library, which provides primitives for streaming and messaging, much like a lightweight form of the familiar sockets API. iLib maps onto the user-level networks without the overhead of system software.

Tile Processor Architecture overview

The Tile Processor Architecture consists of a 2D grid of identical compute elements, called tiles. Each tile is a powerful, full-featured computing system that can independently run an entire operating system, such as Linux. Likewise, multiple tiles can be combined to run a multiprocessor operating system such as SMP Linux. Figure 1 is a block diagram of the 64-tile TILE64 processor. Figure 2 shows the major components inside a tile.

As Figure 1 shows, the perimeters of the mesh networks in a Tile Processor connect to I/O and memory controllers, which in turn connect to the respective off-chip I/O devices and DRAMs through the chip's pins. Each tile combines a processor and its associated cache hierarchy with a switch, which implements the Tile Processor's various interconnection networks. Specifically, each tile implements a three-way very long instruction word (VLIW) processor architecture with an independent program counter; a two-level cache hierarchy; a 2D direct memory access (DMA) subsystem; and support for interrupts, protection, and virtual memory.

TILE64 implementation

The first implementation of the Tile Processor Architecture is the TILE64, a 64-core processor implemented in 90-nm technology, which clocks at speeds up to 1 GHz and is capable of 192 billion 32-bit operations per second. It supports subword arithmetic and can achieve 256 billion 16-bit operations per second (ops), or half a teraops for 8-bit operations. The TILE64

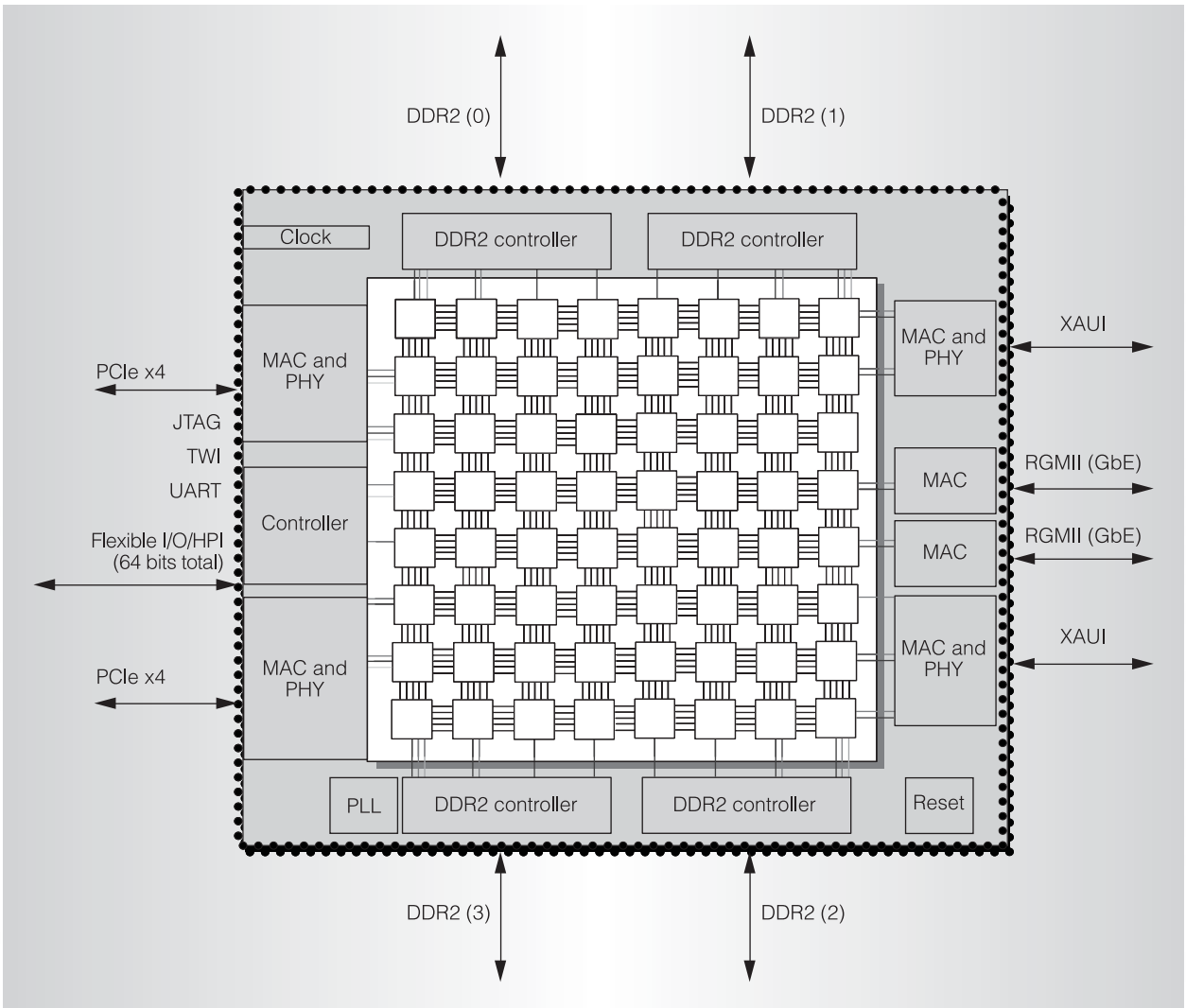

Figure 1. Block diagram of the TILE64 processor with on-chip I/O devices (MAC: media access controller; PHY: physical interface; XAUI: 10-Gbit Ethernet attachment unit interface; RGMII: reduced Gbit media-independent interface; Flexible I/O/HPI: Flexible general-purpose I/O/host port interface; JTAG: test access port; UART: universal asynchronous receiver/transmitter; PLL: phase-locked loop; PCIe: PCI Express; TWI: two-wire interface.

processor consists of an 8 × 8 grid of tiles. The chip includes 4.8 Mbytes of on-chip cache distributed among the processors; per-tile translation look-aside buffers (TLBs) for instructions and data; and 2D DMA between the cores and between main memory and the cores. The Tile Processor provides a coherent shared-memory environment in which a core can directly access the cache of any other core using the on-chip interconnects. The cores provide support for virtual memory and run SMP Linux, which implements a paged memory system.

To meet the power requirements of embedded systems, the TILE64 employs extensive clock gating and processor-napping modes. To support its target markets of intelligent networking and multimedia, TILE64 implements all the required memory and I/O interfaces on the SoC. Specifically, it provides off-chip memory bandwidth up to 200 Gbps using four DDR2 interfaces, and I/O bandwidth in

excess of 40 Gbps through two full-duplex ×4 PCIe interfaces, two full-duplex XAUI ports, and a pair of gigabit Ethernet interfaces. The high-speed I/O and memory interfaces are directly coupled to the on-chip mesh interconnect through an innovative, universal I/O shim mechanism.

TILE64 processor chips have been running in our laboratory for several months. In addition to SMP Linux, they run off-the-shelf shared-memory pthreads-based programs and embedded applications using the iLib API.

Interconnect hardware

The Tile Architecture provides ample on-chip interconnect bandwidth through the use of five low-latency mesh networks. The Tile Architecture designers chose a 2D mesh topology because such topologies map effectively onto 2D silicon substrates. The networks are not toroidal in nature, but rather simple meshes. Although it is possible to map 2D toroids onto a 2D substrate, doing so increases costs in wire length and wiring congestion, by a factor of approximately 2.

The five networks are the user dynamic network (UDN), I/O dynamic network (IDN), static network (STN), memory dynamic network (MDN), and tile dynamic network (TDN). Each network connects five directions: north, south, east, west, and to the processor. Each link consists of two 32-bit-wide unidirectional links; thus, traffic on a link can flow in both directions at once.

Each tile uses a fully connected crossbar, which allows all-to-all five-way communication. Figure 3 shows a grid of tiles connected by the five networks, and Figure 4 shows a single dynamic switch point in detail.

Four of the Tile Architecture's networks are dynamic; these provide a packetized, fire-and-forget interface. Each packet contains a header word denoting the x and y destination location for the packet along with the packet's length, up to 128 words per packet. The dynamic networks are dimension-ordered wormhole-routed. The latency of each hop through the network is one cycle when packets are going straight, and one extra cycle for route calculation when a packet must make a turn at a switch. Because the networks are wormhole-routed, they use minimal in-network buffering. In fact, the network's only buffering comes from small, three-entry FIFO buffers that serve only to cover the link-level flow-control cost.

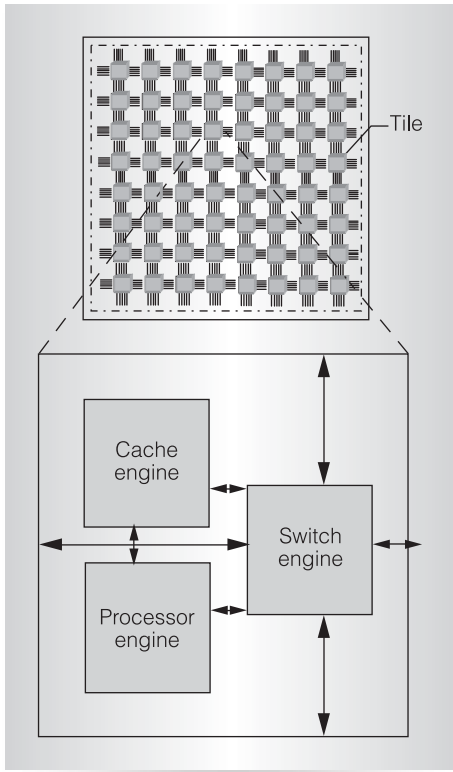

Figure 2. The view inside a tile.

The dynamic networks preserve ordering of messages between any two nodes, but do not guarantee ordering between sets of nodes. A packet is considered to be atomic and is guaranteed not to be interrupted at the receiving node. The dynamic networks are flow controlled and guarantee reliable delivery. As we discuss later, the dynamic networks support scalar and stream data transport.

The static network, which allows static communications, does not have a packetized format but rather allows static configuration of the routing decisions at each switch point. Thus, with the STN, an application can send streams of data from one tile to another by simply setting up a route

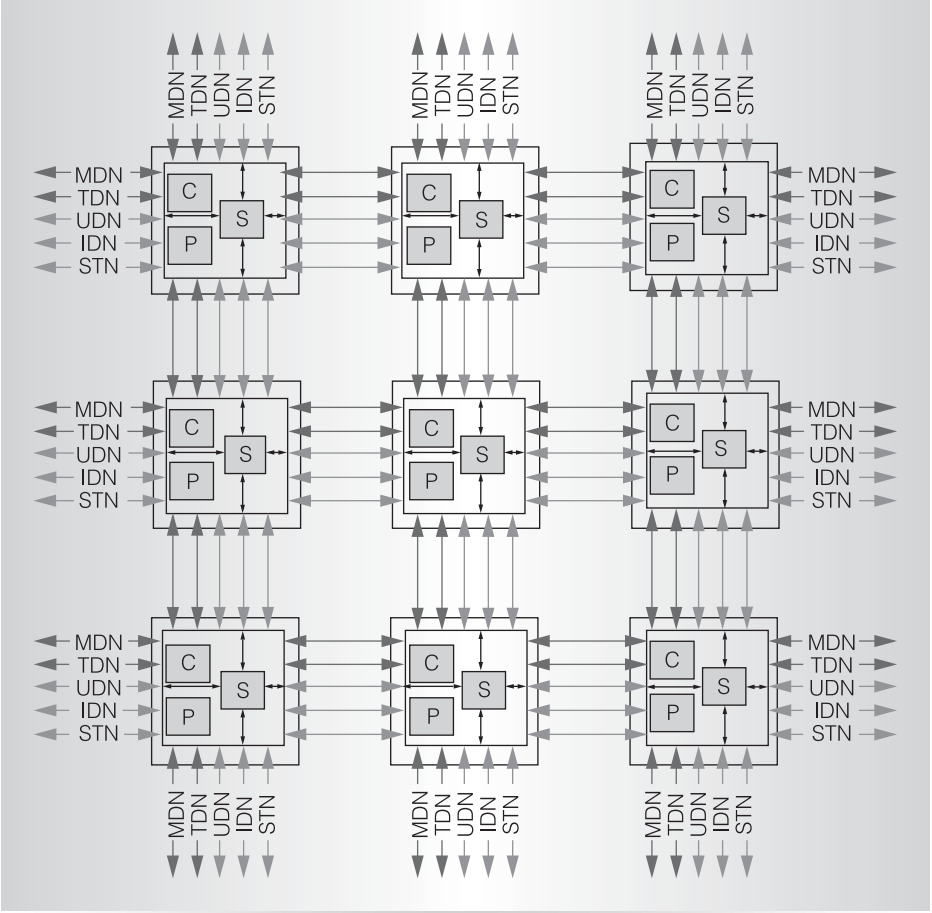

Figure 3. A 3 × 3 array of tiles connected by networks. (MDN: memory dynamic network; TDN: tile dynamic network; UDN: user dynamic network; IDN: I/O dynamic network; STN: static network.)

through the network and injecting a stream of data. The data stream then traverses the already set-up routers until the data reaches the destination tile. This allows a circuit-switched communications channel, ideal for streaming data. The static network also contains an auxiliary processor for reconfiguring the network in a programmatic manner. This functionality is an improved form of the Raw Processor's static network processor functionality.[3]

Network uses

The UDN primarily serves userland processes or threads, allowing them to communicate flexibly and dynamically, with low latency. This is a departure from the typical computer architecture, in which the only userland communication between threads is through shared-memory communication. By providing an extremely low-latency user-accessible network, streams of data, scalar operands, or messages can be directly communicated between threads running in parallel on multiple tiles without involving the operating system. The Tile Architecture also supplies userland interrupts to provide a fast mechanism for notifying userland programs of data arrival.

The Tile Architecture contains no unified bus for communication with I/O devices. Instead, I/O devices connect to the networks just as the tiles do. To support direct communication with I/O devices and allow system-level communications, the architecture uses the IDN, which connects to each tile processor and extends beyond the fabric of processors into I/O devices. Both control

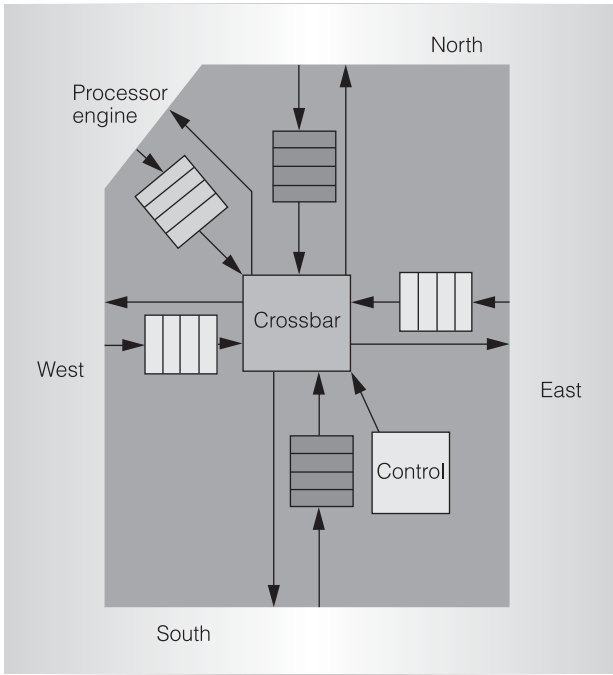

Figure 4. A single network crossbar.

information and streams of I/O data travel over the IDN. The IDN also serves for operating-system and hypervisor-level communications. It is important to have a dedicated network for I/O and system-level traffic to protect this traffic from userland code.

The caches in each tile use the MDN to communicate with off-chip DRAM. The TILE64 has four 64-bit DDR2/800 DRAM controllers on chip, which connect to the MDN at the edge of the tile arrays. These memory controllers allow glueless interfacing to DRAM. The MDN provides a mechanism for every tile to communicate with every RAM controller on the chip. When the cache needs to retrieve backing memory from off-chip DRAM, the in-tile cache controller constructs a message that it sends across the MDN to the DRAM controller. After servicing the transaction, the memory controller sends a reply message across the MDN. A buffer preallocation scheme ensures that the MDN runs in a deadlock-free manner.

The TDN works in concert with the MDN as a portion of the memory system. The Tile Architecture allows direct tile-to-tile cache transfers. The request portion of tile-to-tile cache transfers transit the TDN, and responses transit the MDN. To prevent deadlock in the memory protocol, tile-to-tile requests do not go over the MDN. Thus, this task required an independent communications channel.

The STN is a userland network. Userland programs are free to map communications channels onto the STN, thereby allowing an extremely low-latency, high-bandwidth channelized network—great for streaming data.

Logical versus physical networks

When designing a multicore processor with communication networks, it is often desirable to have multiple independent logical networks. Having multiple logical networks allows for privilege isolation of traffic, independent flow control, and traffic prioritization. The Tile Architecture's five different networks could have been implemented as logical or virtual channels over one large network, or as independent physical networks.

The choice to implement the TILE64's five logical networks with five physically independent networks runs contrary to much previous work and is motivated by how the relative costs of network design change for implementations on a single die. The first surprising realization is that network wiring between tiles is effectively free. Modern-day multilayer fabrication processes provide a wealth of wiring as long as that wiring is nearest-neighbor and stays on chip. If tiles shrink enough, the tile perimeter could eventually be so small that it would make wiring a challenge, but for our small tile this issue was nowhere in sight.

The second trade-off in on-chip network design is the amount of buffer space compared to wire bandwidth. In traditional off-chip networks, the wire bandwidth is at a premium, but on-chip buffering is relatively inexpensive. With on-chip networks, the wiring bandwidth is high, but the buffer space takes up silicon area, the critical resource. In the TILE64 processor, each network accounts for approximately 1.1 percent of a tile's die area, of which

more than 60 percent is dedicated to buffering. Because buffering is the dominating factor, the TILE64 reduces it to the absolute minimum needed to build efficient link-level flow control. If virtual channels were built with the same style of network, each virtual channel would need additional buffering equal to that of another physical network. Thus, on-chip network design reduces the possible area savings of building virtual-channel networks because virtual-channel networks do not save on the 60 percent of the silicon area dedicated to buffering. If the design used buffers larger than the minimum needed to cover the flow-control loop, a virtual-channel network would have opportunities to share buffering that do not exist in the multiple-physical-channel solution.

Another aspect of on-chip networks that affects buffering is that they are more reliable than interchip networks. This reliability mitigates the need for a lot of buffering, which less-reliable systems use to manage link failure. Finally, building multiple physical networks via replication simplifies the design and provides more intertile communication bandwidth.

Network-to-tile interface

To reduce latency for tile-to-tile communications and reduce instruction occupancy, the Tile Architecture provides access to the on-chip networks through register access tightly integrated into the processor pipeline. Any instruction executed in the processor within a tile can read or write to the UDN, IDN, or STN. The MDN and TDN are connected to each tile, but connect only to the cache logic and are only indirectly used by cache misses. There are no restrictions on the number of networks that can be written or read in a particular instruction bundle. Reading and writing networks can cause the processor to stall. Stalling occurs if read data is not available or a network write is writing to a full network. Providing register-mapped network access can reduce both network communication latency and instruction occupancy. For example, if an *add* must occur and the result value must be sent to another tile, the *add* can directly target the register-mapped network without the additional occupancy of a network move.

Receive-side hardware demultiplexing

Experience has shown that the overhead associated with dynamic networks is rarely in the networks themselves; rather, it resides in the receive-side software logic that demultiplexes data. In a dynamic network, each node can receive messages from many other nodes. (A node can be a process on a tile, or even a specific channel port within a process on a given tile). For many applications, the receive node must quickly determine for any data item that it receives which node sent the data.

One software-only solution to receive-side demultiplexing is to augment each message with a tag. When a new message arrives, the receiving node takes an interrupt. The interrupt handler then inspects the tag and determines the queue in memory or cache into which the message should be enqueued. Then, when the node wants to read from a particular sending node, it looks into the corresponding queue stored in memory and dequeues from the particular queue into which the data was sorted. Although this approach is flexible, the cost associated with taking an interrupt and implementing the sorting on the basis of a tag in software can be quite expensive. Also, reading from memory on the receive side is more costly than reading directly from a register provided by register-mapped networks. For these reasons, we look to hardware structures to accelerate packet demultiplexing at a receiving node.

To address this problem, the Tile Architecture contains hardware demultiplexing based on a tag word associated with each network packet. The tag word can signify the sending node, a stream number, a message type, or some combination of these characteristics. The UDN and IDN implement receive-side hardware demultiplexing. Figure 5 shows an example use of the demultiplexing hardware, with two neighboring tiles communicating via the UDN.

The Tile Architecture implements hardware demultiplexing by having several independent hardware queues with settable

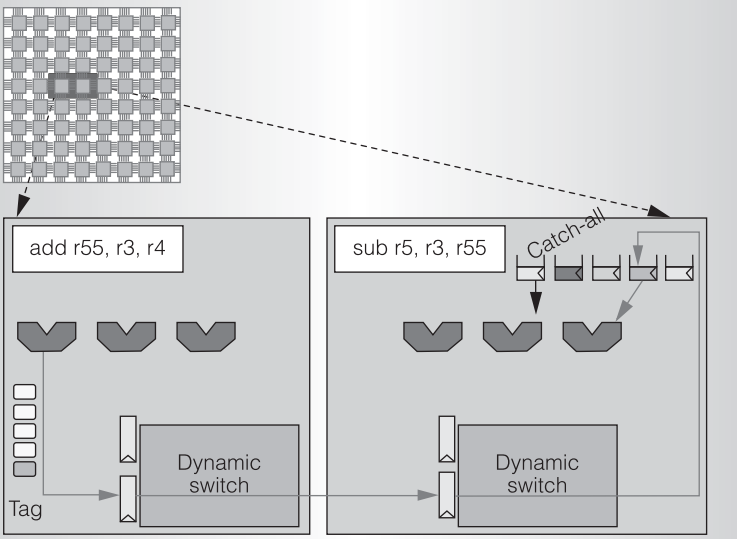

Figure 5. Demultiplexing overview.

tags into which data can be sorted at the receive endpoint. The receiving node sets the tags to some value. When the node receives a message, a tag check occurs. If the tag matches one of the tags set by the receiving node as one of the tags that it is interested in, the hardware demultiplexes the data into the appropriate queue. If an incoming tag does not match any of the receive-side set tags, a tag miss occurs; the data goes into a catch-all queue, and a configurable interrupt can be raised. By having a large tag space and a catch-all queue, the physical queues can be virtualized to implement a very large stream namespace. In effect, the implemented queues serve as a cache of the most recent streams a receive node has seen. Figure 6 shows the implementation of the hardware demultiplexing logic.

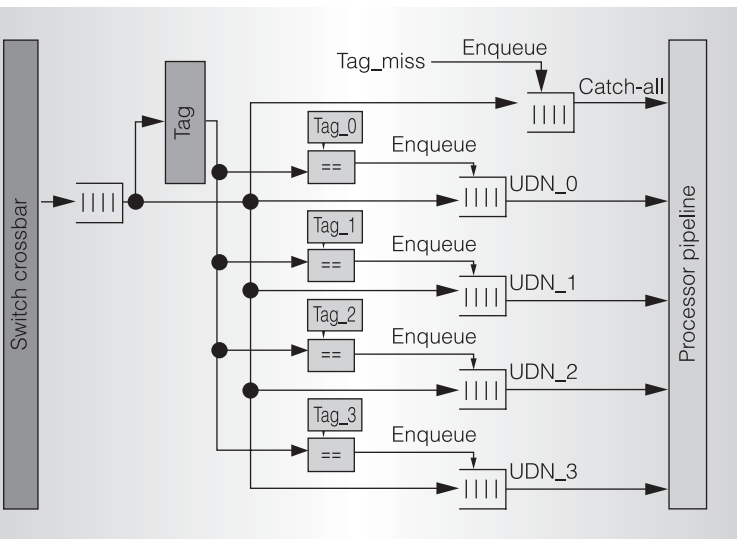

Figure 6. Receive-side demultiplexing hardware with tags.

The UDN contains four hardware demultiplexing queues and a catch-all queue, whereas the IDN contains two demultiplexing queues and a catch-all queue. The TILE64 implementation contains 128 words (512 bytes) of shared receive-side buffering per tile, which can be allocated between the different queues by system software.

Together, the demultiplexing logic and the streamlined network-to-tile interface let the Tile Processor support operations that require extremely low latency, such as scalar operand transport, thus facilitating streaming.

Flow control

It is desirable that on-chip networks be a reliable data transport mechanism. Therefore, all the networks in the Tile Architecture contain link-level forward and reverse flow control. The TILE64 implementation uses a three-entry, credit-based flow-control system on each tile-to-tile boundary. Three entries is the minimum buffering needed to maintain full-bandwidth, acknowledged communications in the design. The architecture also uses this link-level flow control to connect the switch to the processor and memory system.

Although link-level flow control provides a reliable base to work with, dynamic on-chip networks require higher-level flow control to prevent deadlocks on a shared network and to equitably share network resources. Even though the TILE64's four dynamic networks are replicas of each other, they use surprisingly varied solutions for end-to-end flow control. The strictest flow control is enforced on the MDN, which uses a conservative deadlock-avoidance protocol. Every node that can communicate with a DRAM memory controller is allocated message storage in the memory

controller. Each node guarantees that it will never use more storage than it has been allocated. Acknowledgments are issued when the DRAM controller processes a request. The storage space is assigned such that multiple in-flight memory transactions are possible to cover the latency of acknowledgments. Because all in-flight traffic has a preallocated buffer at the endpoint, no traffic can ever congest the MDN.

The other memory network, the TDN, uses no end-to-end flow control, but relies solely on link-level flow control. The TDN is guaranteed not to deadlock, because its forward progress depends solely on that of the MDN.

The two software-accessible dynamic networks, IDN and UDN, both implement mechanisms to drain and refill the networks. Thus, in the case of an insufficient-buffering deadlock, the networks can be drained and virtualized, using the off-chip DRAM as extra in-network buffering. In addition to this deadlock-recovery mechanism, the IDN uses preallocated buffering with explicit acknowledgments when communicating with IDN-connected I/O devices. Communications on the UDN use different end-to-end flow control, depending on the programming model used. Buffered channels and message passing use software-generated end-to-end acknowledgments to implement flow control. For applications using raw channels (described later), only the demultiplex buffering is used, and it is up to the programmer to orchestrate usage.

Protection

The Tile Architecture has novel features not typically found in conventional multicore processors. Because it has directly accessible networks, and particularly because it has user-accessible networks, usability and modularity require that it also protect programs from one another. It is not desirable for one program to communicate with another program unrestrained. Likewise, it is not desirable for a userland program to directly message an I/O device or another operating system running on another set of tiles. To address these problems, the Tile Architecture implements a mechanism called Multicore Hardwall.

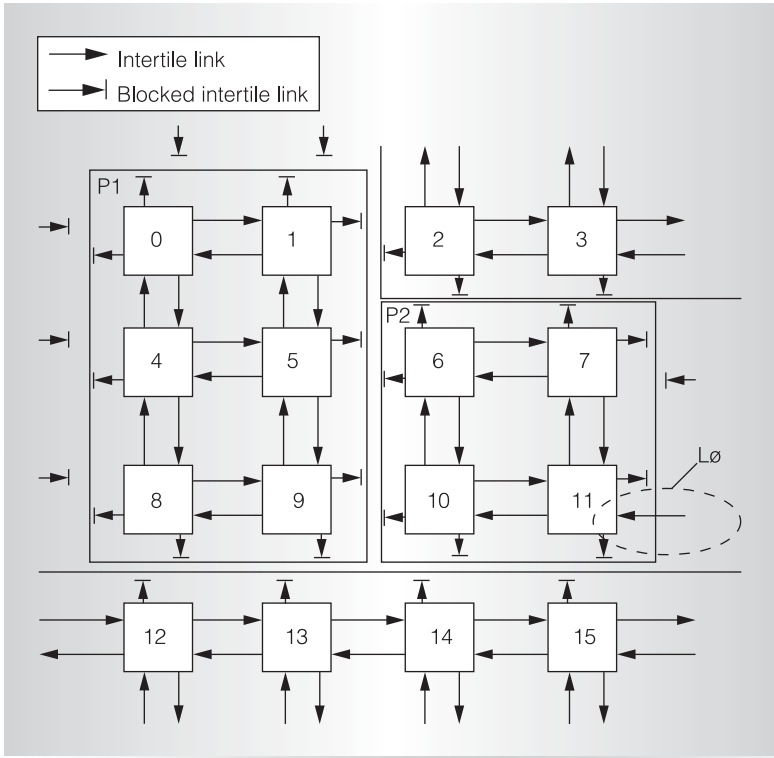

Figure 7. Protection domains on a dynamic network.

Multicore Hardwall is a hardware protection scheme by which individual links in the network can be blocked from having traffic flow across them. Multicore Hardwall protects every link on the UDN, IDN, and STN, whereas traffic on the MDN and TDN are protected by standard memory protection mechanisms through a TLB. Figure 7 shows multiple protection domains between tiles for a 4×4 fabric of tiles. If the processor attempts to send traffic over a hardwalled link, the Multicore Hardwall mechanism blocks traffic and signals an interrupt to system software, which can take appropriate action. Typically, the system software kills the process, but, because Multicore Hardwalling can also serve to virtualize larger tile fabrics, the system software might save the offending message and play it back later on a different link. The network protection is implemented on outbound links; thus, it is possible to have unidirectional links in the network on which protection is set up in only one direction. Figure 7 shows an example of this arrangement at link L0.

Shared-memory communication and ordering

When designing single-chip tiled processors, communication costs are low enough that designers can consider alternative approaches to traditional problems such as shared memory. The TILE64 processor uses neighborhood caching to provide an on-chip distributed shared cache. Neighborhood caching functions by homing data in a single tile's on-chip cache. This homing decision is made by system software and is implemented on a page basis via a tile's memory management unit. If a memory address is marked local, the data is simply retrieved from the local cache or, in the event of a cache miss, from main memory. If an address is marked as remote, a message is sent over the TDN to the home tile to retrieve the needed data. Coherency is maintained at the home tile for any given piece of data, and data is not cached at non-home tiles. The tiles' relative proximity and high-bandwidth networks allow neighborhood caching to achieve suitable performance. Read-only data can be cached throughout the system.

For communication with multiple networks or networks and shared memory, the question of ordering arises. The Tile Architecture guarantees that network injection and removal occur in programmatic order. However, there are no inter-network guarantees within the network; thus, synchronization primitives are constructed by software out of the unordered networks. When an application uses both memory and network, a memory fence instruction is required to make memory visible if memory is used to synchronize traffic that flows over the networks. Likewise, the memory fence instruction is used when network traffic is used to synchronize data that is passed via shared memory.

Interconnect software

The Tile Architecture's high-bandwidth, programmable networks let software implement many different communication interfaces at hardware-accelerated speeds. Tilera's C-based iLib library provides programmers with a set of commonly used communication primitives, all implemented via on-chip communication on the UDN. For example, iLib provides lightweight socket-like streaming channels for streaming algorithms, and provides an MPI-like message-passing interface for ad hoc messaging.

By providing several different communication primitives, iLib lets the programmer use whichever communication interface is best for the problem at hand. The UDN network design—in particular, the demux hardware—lets iLib provide all these communication interfaces simultaneously, using some demux queues for channel communication and others for message passing. As we demonstrate later, the high-bandwidth, scalable network enables efficient burst communication for applications that require data reorganization between computation phases.

Communication interfaces

There are two broad categories of UDN-based communication in iLib: socket-like channels and message passing.

The iLib channels interface provides long-lived connections between processes or threads. Semantically, each channel is a first-in, first-out (FIFO) connection between two processes. A channel send operation always sends data to the same receiver. Generally, a channel provides a point-to-point connection between two processes (a sender and a receiver); this type of connection is often used to implement producer-consumer communication without worrying about shared-memory race conditions. The UDN's all-to-all connectivity also lets iLib provide sink channel topologies, in which many senders send to one receiver. In this case, the receive operation implicitly selects the next available packet from any of the incoming FIFO buffers. Sink channels are often used to collect results from process-parallel programs, in which many workers must forward their results to a single process for aggregation.

iLib actually provides several channel APIs, each optimized for different communication needs. *Raw channels* allow very low-overhead communication, but each FIFO buffer can use only as much storage as is available in the hardware demux buffer in each tile. *Buffered channels* have slightly higher overhead but allow arbitrary amounts of storage in each FIFO buffer by virtualizing the storage in memory.

Thus, buffered channels are appropriate for applications that require a large amount of buffering to decouple burstiness in a producer-consumer relationship, whereas raw channels are appropriate for finely synchronized applications that require very low-latency communication.

The message-passing API is similar to MPI.[4] Message passing lets any process in the application send a message to any other process in the application at any time. The message send operation specifies a destination tile and a message key to identify the message. The message receive operation lets the user specify that the received message should come from a particular tile or have a particular message key. The ability to restrict which message is being received simplifies cases in which several messages are sent to the same tile simultaneously; the receiving tile can choose the order in which it receives the messages, and the iLib runtime saves the other messages until the receiver is ready to process them. The ability to save messages for later processing makes message passing the most flexible iLib communication mechanism: Any message can be sent to any process at any time without any need to establish a connection or worry about the order in which the messages will be received. Figure 8 presents example code for the raw channels, buffered channels, and messaging iLib APIs.

Raw channels:

```
int a = 2, b = 3;
ilib_rawchan_send_2(send_port, a, b);

int a_in, b_in;
ilib_rawchan_receive_2(receive_port, a_in, b_in);
```

Buffered channels:

```
char packet[SIZE];
ilib_bufchan_send(send_port, packet, sizeof(packet));

size_t size_in;
char* packet_in = ilib_bufchan_receive(receive_port, &size_in);
```

Messages:

```
char message[SIZE];
ilib_msg_send(ILIB_GROUP_SIBLINGS, receiver_rank, TAG,
      message, sizeof(message));

char message_in[SIZE];
ilibStatus status;
ilib_msg_receive(ILIB_GROUP_SIBLINGS, sender_rank, TAG,
      message_in, sizeof(message_in), &status);
```

Figure 8. iLib code examples.

Table 1 shows the relative performance and flexibility of the different iLib communication APIs. Raw channels achieve single-cycle occupancy by sending and receiving via register-mapped network ports. Buffered channels are more flexible, letting the user create FIFO connections with unlimited amounts of in-channel buffering (the amount of buffering is determined at channel creation time), but they incur more overhead because they use an interrupt handler to drain data from the network. Message passing is the most flexible, but also the most expensive, interface. It provides unlimited, dynamically allocated buffering and out-of-order delivery of messages with different message keys, but at the expense of greater interrupt overhead. The data transmission bandwidth for buffered channel messages is actually the same as for raw channels (they all use the same UDN), but the interrupt and synchronization overhead has a significant performance impact.

Implementation

The Tile interconnect architecture lets the iLib communication library implement many different forms of communication using the same network. This makes the

Table 1. Performance and ordering properties of different UDN communication APIs.

Mechanism	Latency (cycles)	Occupancy (cycles)	Bandwidth (bytes/cycle)	Buffering	Ordering
Raw channels	9	3 send, 1 receive	3.93	Hardware	FIFO
Buffered channels	150	100	1.25	Unlimited, static	FIFO
Message passing	900	500	1.0	Unlimited, dynamic	Out of order; FIFO by key

Tile Architecture a more flexible, more programmable approach to parallel processing than SoC designs that provide interconnects only between neighboring cores. In particular, each tile's demux queue and demux buffer let iLib efficiently separate different flows of incoming traffic and handle each flow differently.

For example, the implementation of raw channels lets the programmer reserve a demux queue for a single channel's incoming data. To connect a raw channel, iLib allocates one of the receiving tile's four UDN demux queues and assigns its tag value to a per-channel unique identifier. The sending process or processes then send data packets to that receiving tile, specifying the same unique identifier that was mapped to the now-reserved demux queue. Thus, all the packets for a particular raw channel are filtered into a particular demux queue. The receive operation then simply reads data directly from that queue. Because raw-channel packets can be injected directly into the network, the send operation requires only a few cycles. Similarly, the receive operation requires only a single register move instruction because the incoming data is directed into a particular register-mapped demux queue. In fact, because the receive queue is register mapped, the incoming data can be read directly into a branch or computation instruction without any intermediate register copy.

Users of raw channels must implement flow control to manage the number of available buffers without overflowing them. Often, the necessary flow control is implemented as an ''acked channel,'' in which a second FIFO buffer is connected to transmit acknowledgments in the reverse direction from the receiver to the sender. In such an implementation, the channel receiver begins by sending several credit packets to the sender. When the sender needs to send, it first blocks on the receive of a credit packet and then sends the data packet. When the receiver dequeues a data packet, it sends a credit packet back to the sender.

The implementation of raw channels demonstrates how iLib can reserve a demux queue to separate out traffic for a particular point-to-point channel. However, some algorithms require considerable buffering in the semantic FIFO buffer between the sender and receiver. The required buffering could be significantly larger than the amount of storage in the UDN demux buffer. In such cases, a demux queue can be reserved for buffered-channel traffic. When using buffered channels, the demux is configured to generate an interrupt when the demux buffer fills with data, allowing the interrupt handler to drain the incoming traffic into a memory buffer associated with each buffered channel. The receive operation then pulls data from the memory buffer instead of from a register-mapped demux queue. Two key features of the Tile Architecture allow efficient implementation of these virtualized, buffered channels: configurable interrupt delivery on demux buffer overflow and low-latency interrupts. Configurable interrupt delivery allows delivery of interrupts for buffered-channel data but not for raw-channel data, and low-latency interrupts let the data be rapidly drained into memory. In fact, an optimized interrupt handler can interrupt the processor, save off enough registers to do the work, and return to the interrupted code in less than 30 cycles.

Finally, the message-passing interface uses yet a third demux configuration option to implement immediate processing of incoming messages. When the message-passing interface is enabled, the catch-all demux queue is configured to interrupt the processor immediately when a packet arrives. To send a message, the ilib_msg_send( ) routine first sends a packet containing the message key and size to the receiver. The receiver is interrupted, and the messaging engine checks to see whether the receiver is currently trying to receive a message with that key. If so, a packet is sent back to the sender, telling it to transfer the message data. If no receive operation matches the message key, the messaging engine saves the notification and returns from the interrupt handler. When the receiver eventually issues an ilib_msg_receive( ) with the same message key, the messaging engine sends a packet back to the sender, interrupting it and telling it to transfer data. Thus, the ability to configure

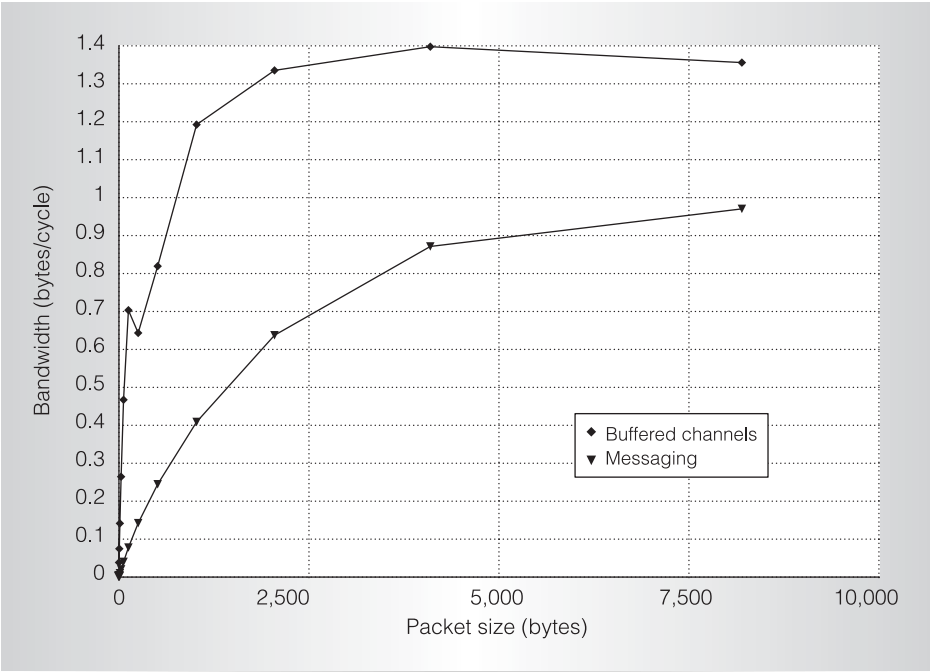

Figure 9. Bandwidth versus packet size for buffer channels and messaging.

a particular demux queue to interrupt the processor when packets arrive lets iLib implement zero-copy, MPI-style message passing.

We have shown how the Tile interconnect architecture allows iLib to separate out different traffic flows and handle each flow differently. Demux queues can be used to separate out individual channels' traffic and map it to register mapped queues that will only drain when the receiver reads from the network registers. Alternatively, a demux queue can be configured to interrupt the tile when the demux buffer is full of data, so that incoming traffic can be drained in large bursts. And as a third option, a demux queue can be configured to generate an interrupt whenever traffic arrives, so that the incoming packet can be processed promptly and a response generated. These different modes of operation can be used to implement raw channels, buffered channels, and message passing, respectively, all using the same hardware and running as needed by the application.

iLib characterization

Although software libraries provide ease of programming and flow control, they also introduce overhead on communication channels. iLib is no exception: We now turn to characterizing the performance of the different forms of iLib channels.

iLib communications flow over the UDN. The UDN hardware provides a maximum bandwidth of 4 bytes (one word) per cycle. UDN links consist of two unidirectional connections. The most primitive link type is raw channels. For raw channels, communicating data occurs at a maximum of 3.93 bytes per cycle. The overhead is due to header word injection and tag word injection cost. Figure 9 compares the performance of transferring different sizes of packets using buffered channels and the iLib messaging API. For the buffered-channels case, a decent amount of overhead relates to reading and writing memory. The messaging interface incurs additional overhead related to interrupting the receive tile. Both buffered channels and messaging use the same packets for bulk data transfer: an 18-word packet consisting of one header word, a tag word, and 16 words of data. Because buffered channels and messaging use the same messaging primitives, asymptotically, they can reach the same maximum bandwidth for large packet sizes.

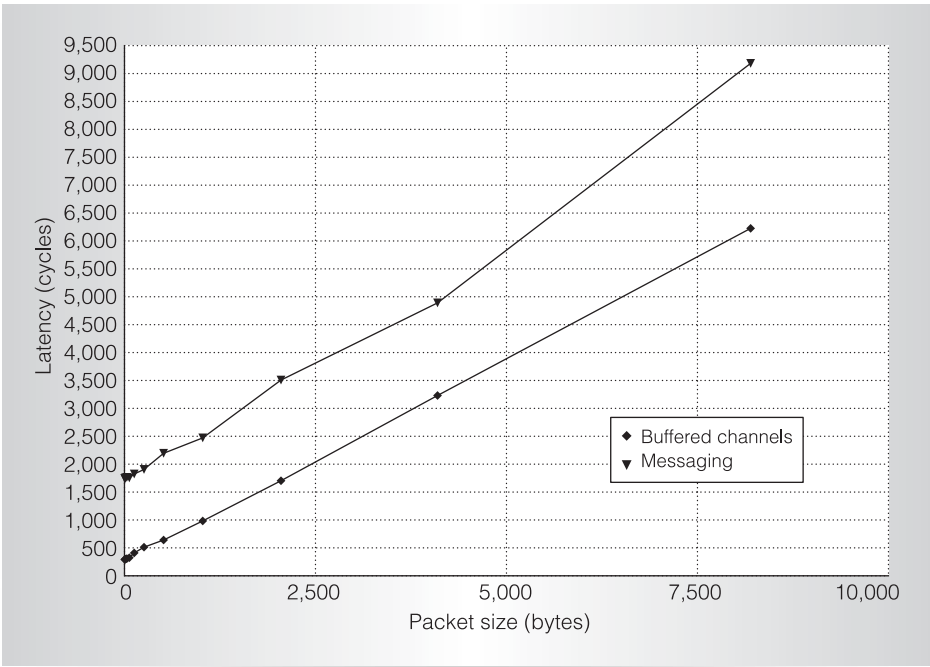

Figure 10. Latency versus packet size for buffered channels and messaging.

Figure 10 examines the latency of buffered channels and messaging as a function of packet size. We conducted this test by sending a packet of fixed size from one tile to a neighboring tile and then having that tile respond with a packet of the same size. The latency is the time taken to complete this operation, divided by 2. Regardless of packet size, messaging incurs approximately 1,500 cycles more overhead than buffered channels.

Applications

Several microbenchmarks are useful for characterizing a chip's interconnect. We implemented the same applications in multiple programming styles to demonstrate each communication mechanism's relative communication overhead.

Corner turn

To illustrate the benefits of Tile's flexible, software-accessible, all-to-all interconnect network, we examine a microbenchmark commonly seen between phases of DSP applications.

Image-processing applications operate on multidimensional data. The image itself is 2D, and computation often requires pixels from multiple images. Multicore processors generally distribute these multidimensional arrays across cores to exploit data parallelism.

Frequently, the most efficient data distribution for one stage of the application is inefficient for another. For example, a 2D frequency transform is implemented as a series of 1D transforms on rows of data and then a series of 1D transforms on columns of data. In this scenario, we would like each core to contain entire rows of the array during the first phase, and then entire columns during the second phase.

The process of reorganizing a distributed array from distribution in one dimension to distribution in another is known as a *corner turn*.[5] Implementing a corner turn requires each core to send a distinct message to every other core. Furthermore, these messages can be relatively small. To perform well on a corner turn, a multicore processor needs a high-bandwidth network with minimal contention and low message overhead.

To illustrate the performance and behavior of the various networks, we implemented corner turn in four different ways. Two factors distinguished these implementations: the network used to redistribute the data and the network used to synchronize

Table 2. Corner turn: performance comparison of shared memory versus raw channels.

Tile configuration	Matrix size (words)	Achieved bandwidth (Gbps)			
		Shared memory, UDN sync	Shared memory, STN sync	Raw channels, UDN sync	Raw channels, STN sync
2 × 2	256 × 128	9.54	13.97	9.73	13.91
4 × 4	512 × 256	17.67	23.69	19.18	46.58
8 × 8	1,024 × 512	37.99	42.92	11.24	96.85

the tiles. Data was transmitted on either the TDN, using shared memory, or the UDN, using iLib's raw channels. For each of the data transmission methods, we implemented synchronization using either the STN or the UDN. The combination of data transmission methods and synchronization methods yielded four implementations. To measure an implementation's efficacy, we measured the achieved bandwidth of the data reorganization.

Table 2 shows the results for the four corner-turn implementations. The best-performing implementation was clearly the one transmitting data using raw channels and synchronizing using the STN. Raw channels provide direct access to the UDN and let each word of data be transmitted with minimal overhead. Furthermore, using the STN for synchronization keeps the synchronization messages from interfering with the data for maximum performance. The drawbacks of this implementation are that it requires the programmer to carefully manage the UDN and that it requires extra implementation time to fully optimize.

The corner-turn implementation that uses raw channels for data transmission and the UDN for synchronization performed far more poorly than the one using the STN for synchronization. This was due to the overhead of virtualizing the UDN for multiple logical streams (the data and the synchronization). When two logically distinct types of messages must share the same network, the user must add extra data to distinguish between the types of messages. Deadlock is avoided by sharing the available buffering between the two types of messages. Both of these issues add overhead to the implementation; in the 64-tile case, this overhead becomes very destructive to performance.

The shared-memory implementations of corner turn are far simpler to program, but their performance is also lower than for corner turn using raw channels and STN. Both the ease of implementation and the performance difference are due to the extra overhead of sending shared data on the TDN. For each data word that the user sends on the TDN, the hardware adds four extra header words. These extra words let the hardware manage the network and avoid deadlocks, so the program is far easier to write, but it is limited in performance. When data is sent over the TDN, the synchronization method makes less of a difference than for raw channels. This is because the TDN implementation keeps data and synchronization separate, despite the synchronization method.

Dot product

Dot product, another widely used DSP computation, takes two vectors of equal length, multiplies the elements pairwise, and sums the results of the multiplications, returning a scalar value. Dot product has wide applications in signal filtering, where a sequence of input samples are scaled by differing constants, and it is the basic building block for finite impulse response (FIR) filters.

To map a dot product across a multicore processor, the input vectors are evenly distributed across the array of processors. A processor then completes all of the needed multiplications and sums across the results. Finally, a distributed gather-and-reduction add is performed across all of the individual processor's results. To measure performance, we used a 65,536-element dot product. The input data set consisted of 16-bit values and computed a 32-bit result.

Table 3. Dot product: performance comparison of shared memory versus raw channels. Lower cycle count is faster.

Tiles (no.)	Shared memory (cycles)	Raw channels (cycles)
1	420,360	418,626
2	219,644	209,630
4	87,114	54,967
8	56,113	28,091
16	48,840	14,836
32	59,306	6,262
64	105,318	4,075

As with the corner turns, we mapped dot product using two communications mechanisms, comparing Tilera shared memory with iLib's raw channels. Both methods were optimized for the architecture. Table 3 compares the two communication mechanisms. The results indicate that the shared-memory implementation contains higher communication overhead and hence does not scale as well as the raw-channel implementation. Another noteworthy result is the performance jump from 2 to 4 tiles. Whereas the computation resources only double, the application exhibits superlinear speedup, because this is the point where the data sets completely fit in a tile's L2 cache.

In some cases, design decisions involving multicore interconnects fly in the face of conventional multi-chip multiprocessor wisdom. For example, when networks are integrated on chip, multiple physical networks can be superior to virtual channel networks. The Tile Processor uses an unconventional architecture to achieve high on-chip communication bandwidth. The effective use of this bandwidth is made possible by the synergy between the hardware architecture of the Tile Processor's on-chip interconnect and the software APIs that use the interconnect. The iLib communication library uses numerous directly accessible networks to provide flexible communication mechanisms to meet the needs of a variety of parallel applications. MICRO

Acknowledgments

We thank everyone on the Tilera team for their effort in designing the Tilera chip and accompanying software tools. iMesh, iLib, Multicore Hardwall, TILE64 and Tile Processor are trademarks of Tilera Corporation.

References

1. M.B. Taylor et al., ''Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 2-13.
2. E. Waingold et al., ''Baring It All to Software: Raw Machines,'' *Computer*, vol. 30, no. 9, Sept. 1997, pp. 86-93.
3. M.B. Taylor et al., ''Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures,'' *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 341-353.
4. *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, 1994; http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.
5. J.M. Lebak. *Polymorphous Computing Architectures (PCA) Example Application 4: Corner-Turn*, tech. report, MIT Lincoln Laboratory, 2001.

David Wentzlaff is a founder of Tilera and a member of the architecture team. His technical interests include designing parallel computing systems and investigating the OS-hardware interface. Wentzlaff has an SM in electrical engineering and computer science from Massachusetts Institute of Technology.

Patrick Griffin is a member of the software team at Tilera, where he designed and developed the iLib on-chip interconnect programming library. His research interests include parallel programming models and tools for performance evaluation of parallel programs. Griffin has a MEng in electrical engineering and computer science from MIT.

Henry Hoffmann is a member of the applications team at Tilera. His research interests include parallel algorithms and architectures. He has an SM in electrical engineering and computer science from MIT.

Liewei Bao is a member of the architecture team at Tilera. His research interests include multicore processing, networking applications, and memory interfaces. Bao has an MS in electrical engineering from Rose-Hulman Institute of Technology in Terra Haute, Indiana.

Bruce Edwards is a member of the architecture team at Tilera. He previously worked on processors at Symbolics and Digital Equipment Corporation and designed multiple generations of wireless network systems at Broadcom. His research interests include wireless systems and microprocessor architecture and design. Edwards has a BS in electrical engineering from MIT. He is a member of the IEEE.

Carl Ramey is a member of the architecture team at Tilera. His research interests include processor and I/O architecture. Ramey has an MS in electrical and computer engineering from Carnegie Mellon University.

Matthew Mattina is a member of the architecture team at Tilera. His research interests include parallel computing and computational genomics. Mattina has an MS in electrical engineering from Princeton University. He is a member of the IEEE.

Chyi-Chang Miao is a member of architecture team at Tilera. His research interests focus on microprocessor architecture and design. Miao has a PhD in physics from the University of Michigan at Ann Arbor.

John F. Brown III is a founder of Tilera and a member of the executive team. He has 25 years of experience in processor architecture and design. Brown holds bachelor's and master's degrees in electrical engineering from Cornell University.

Anant Agarwal is a professor of electrical engineering and computer science at MIT and an associate director of the Computer Science and Artificial Intelligence Laboratory. He is a founder of Tilera and a member of the executive team. His technical interests include parallel computer architecture and software systems. Agarwal has a PhD in electrical engineering from Stanford University.

Direct questions and comments about this article to David Wentzlaff, Tilera Corp., 1900 West Park Dr., Suite 290, Westborough, MA 01581; wentzlaf@tilera.com.