TITLE

Buses and Crossbars

BYLINE

Rajeev Balasubramonian
School of Computing
University of Utah
Salt Lake City, UT, USA
rajeev@cs.utah.edu

Timothy M. Pinkston
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA, USA
tpink@usc.edu

SYNONYMS

Bus: shared interconnect, shared channel, shared-medium network
Crossbar: point-to-point switch, centralized switch, switched-medium network

DEFINITION

Bus: A bus is a shared interconnect used for connecting multiple components of a computer on a single chip or across multiple chips. Connected entities either place signals on the bus or listen to signals being transmitted on the bus, but signals from only one entity at a time can be transported by the bus at any given time. Buses are popular communication media for broadcasts in computer systems.

Crossbar: A crossbar is a non-blocking switching element with N inputs and M outputs used for connecting multiple components of a computer where, typically, $N = M$. The crossbar can simultaneously transport signals on any of the N inputs to any of the M outputs as long as multiple signals do not compete for the same input or output port. Crossbars are commonly used as basic switching elements in switched-media network routers.

DISCUSSION

Introduction

Every computer system is made up of numerous components such as processor chips, memory chips, peripherals, etc. These components communicate with each other via interconnects. One of the simplest interconnects used in computer systems is the *bus*. The bus is a shared medium (usually

a collection of electrical wires) that allows one sender at a time to communicate with all sharers of that medium. If the interconnect must support multiple simultaneous senders, more scalable designs based on switched-media must be pursued. The *crossbar* represents a basic switched-media building block for more complex but scalable networks.

In traditional multi-chip multiprocessor systems (circa 2000), buses were primarily used as off-chip interconnects, e.g., front-side buses. Similarly, crossbar functionality was implemented on chips that were used mainly for networking. However, the move to multi-core technology has necessitated the use of networks even within a mainstream processor chip to connect its multiple cores and cache banks. Therefore, buses and crossbars are now used within mainstream processor chips as well as chip sets. The design constraints for on-chip buses are very different from those of off-chip buses. Much of this discussion will focus on on-chip buses which continue to be the subject of much research and development.

Basics of Bus Design

A bus comprises a shared medium with connections to multiple entities. An interface circuit allows each of the entities either to place signals on the medium or sense (listen to) the signals already present on the medium. In a typical communication, one of the entities acquires ownership of the bus (the entity is now known as the *bus master*) and places signals on the bus. Every other entity senses these signals and, depending on the content of the signals, may choose to accept or discard them. Most buses today are *synchronous*, *i.e.*, the start and end of a transmission are clearly defined by the edges of a shared clock. An *asynchronous* bus would require an acknowledgment from the receiver so the sender knows when the bus can be relinquished.

Buses often are collections of electrical wires,[1] where each wire is typically organized as "*data*", "*address*", or "*control*". In most systems, networks are used to move messages among entities on the *data* bus; the *address* bus specifies the entity that must receive the message; and the *control* bus carries auxiliary signals such as arbitration requests and error correction codes. There is another nomenclature that readers may also encounter. If the network is used to implement a cache coherence protocol, the protocol itself has three types of messages: (i) *DATA*, which refers to blocks of memory, (ii) *ADDRESS*, which refers to the memory block's address, and (iii) *CONTROL*, which refers to auxiliary messages in the protocol such as acknowledgments. Capitalized terms as above will be used to distinguish message types in the coherence protocol from signal types on the bus. For now, it will be assumed that all three protocol message types are transmitted on the *data* bus.

Arbitration Protocols

Since a bus is a shared medium that allows a single master at a time, an arbitration protocol is required to identify this bus master. A simple arbitration protocol can allow every entity to have ownership of the bus for a fixed time quantum, in a round-robin manner. Thus, every entity can make a local decision on when to transmit. However, this wastes bus bandwidth when an entity has nothing to transmit during its turn.

The most common arbitration protocol employs a central arbiter; entities must send their bus requests to the arbiter and the arbiter sends explicit messages to grant the bus to requesters. If the requesting entity is not aware of its data bus occupancy time beforehand, the entity must also send a bus release message to the arbiter after it is done. The request, grant, and release signals are part of the control network. The request signal is usually carried on a dedicated wire between an entity and the arbiter. The grant signal can also be implemented similarly, or as a shared bus that carries the ID

[1] Alternatively, buses can be a collection of optical waveguides over which information is transmitted photonically [10].

of the grantee. The arbiter has state to track data bus occupancy, buffers to store pending requests, and policies to implement priority or fairness. The use of pipelining to hide arbitration delays will be discussed shortly.

Arbitration can also be done in a distributed manner [5], but such methods often incur latency or bandwidth penalties. In one example, a shared arbitration bus is implemented with wired-OR signals. Multiple entities can place a signal on the bus; if any entity places a "one" on the bus, the bus carries "one", thus using wires to implement the logic of an *OR* gate. To arbitrate, all entities place their IDs on the arbitration bus; the resulting signal is the OR of all requesting IDs. The bus is granted to the entity with the largest ID and this is determined by having each entity sequentially drop out if it can determine that it is not the largest ID in the competition.

Pipelined Bus

Before a bus transaction can begin, an entity must arbitrate for the bus, typically by contacting a centralized arbiter. The latency of request and grant signals can be hidden with pipelining. In essence, the arbitration process (that is handled on the *control* bus) is overlapped with the data transmission of an earlier message. An entity can send a bus request to the arbiter at any time. The arbiter buffers this request, keeps track of occupancy on the *data* bus, and sends the grant signal one cycle before the *data* bus will be free. In a heavily loaded network, the *data* bus will therefore rarely be idle and the arbitration delay is completely hidden by the wait for the *data* bus. In a lightly loaded network, pipelining will not hide the arbitration delay, which is typically at least three cycles: one cycle for the request signal, one cycle for logic at the arbiter, and one cycle for the grant signal.

Case Study: Snooping-Based Cache Coherence Protocols

As stated earlier, the bus is a vehicle for transmission of messages within a higher level protocol such as a cache coherence protocol. A single *transaction* within the higher level protocol may require multiple messages on the bus. Very often, the higher-level protocol and the bus are co-designed to improve efficiency. Therefore, as a case study, a snooping bus-based coherence protocol will be discussed.

Consider a single-chip multiprocessor where each processor core has a private L1 cache, and a large L2 cache is shared by all the cores. The multiple L1 caches and the multiple banks of the L2 cache are the entities connected to a shared bus (Figure 1). The higher-level coherence protcol ensures that data in the L1 and L2 caches is kept coherent, i.e., a data modification is eventually seen by all caches and multiple updates to one block are seen by all caches in exactly the same order.

A number of coherence protocol operations will now be discussed. When a core does not find its data in its local L1 cache, it must send a request for the data block to other L1 caches and the L2 cache. The core's L1 cache first sends an arbitration request for the bus to the arbiter. The arbiter eventually sends the grant signal to the requesting L1. The arbitration is done on the *control* portion of the bus. The L1 then places the ADDRESS of the requested data block on the *data* bus. On a synchronous bus, we are guaranteed that every other entity has seen the request within one bus cycle. Each such "snooping" entity now checks its L1 cache or L2 bank to see if it has a copy of the requested block. Since every look-up may take a different amount of time, a wired-AND signal is provided within the *control* bus so everyone knows that the snoop is completed. This is an example of bus and protocol co-design (a protocol CONTROL message being implemented on the bus' *control* bus). The protocol requires that an L1 cache respond with data if it has the block in "modified" state, else, the L2 cache responds with data. This is determined with a wired-OR signal; all L1 caches place the outcome of their snoop on this wired-OR signal and the L2 cache accordingly determines if it must respond. The responding entity then fetches data from its arrays and places it on the *data* bus. Since the bus is

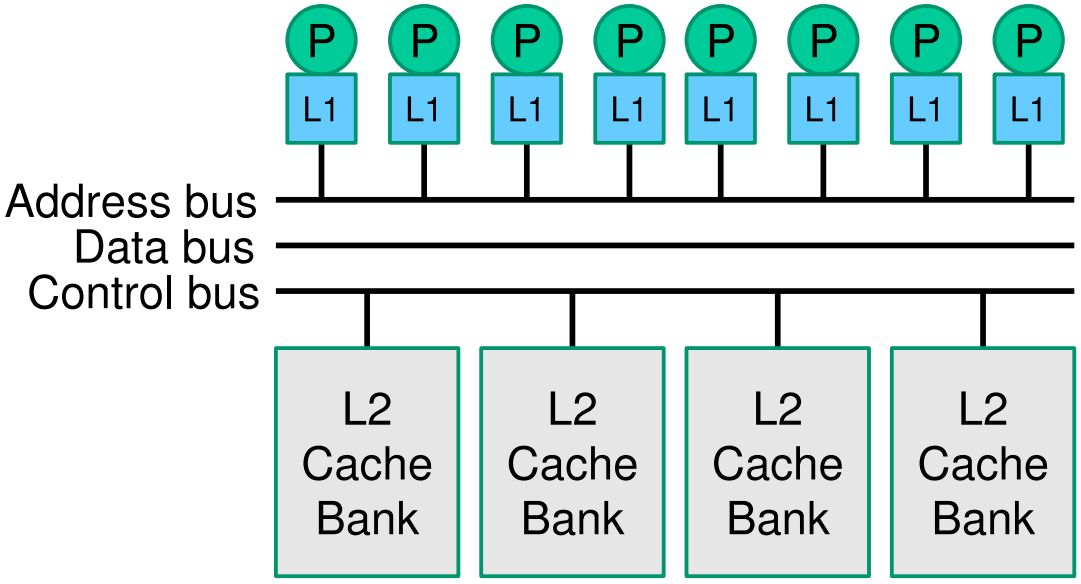

Figure 1: Cores and L2 cache banks connected with a bus. The bus is composed of wires that handle data, address, and control.

not released until the end of the entire coherence protocol transaction, the responder knows that the *data* bus is idle and need not engage in arbitration (another example of protocol and bus co-design). *Control* signals let the requester know that the data is available and the requester reads the cache block off the bus.

The use of a bus greatly simplifies the coherence protocol. It serves as a serialization point for all coherence transactions. The timing of when an operation is visible to everyone is well known. The broadcast of operations allows every cache on the bus to be self-managing. Snooping bus-based protocols are therefore much simpler than directory-based protocols on more scalable networks.

As described, each coherence transaction is handled atomically, *i.e.*, one transaction is handled completely before the bus is released for use by other transactions. This means that the *data* bus is often idle while caches perform their snoops and array reads. Bus utilization can be improved with a *split transaction bus*. Once the requester has placed its request on the *data* bus, the *data* bus is released for use by other transactions. Other transactions can now use the *data* bus for their requests or responses. When a transaction's response is ready, the *data* bus must be arbitrated for. Every request and response must now carry a small tag so responses can be matched up to their requests. Additional tags may also be required to match the wired-OR signals to the request.

The split transaction bus design can be taken one step further. Separate buses can be implemented for ADDRESS and DATA messages. All requests (ADDRESS messages) and corresponding wired-OR CONTROL signals are carried on one bus. This bus acts as the serialization point for the coherence protocol. Responders always use a separate bus to return DATA messages. Each bus has its own separate arbiter and corresponding control signals.

Bus Scalability

A primary concern with any bus is its lack of scalability. First, if many entities are connected to a bus, the bus speed reduces because it must drive a heavier load over a longer distance. In an electrical bus, the higher capacitive load from multiple entities increases the RC-delay; in an optical bus, the reduced photons received at photodetectors from dividing the optical power budget among

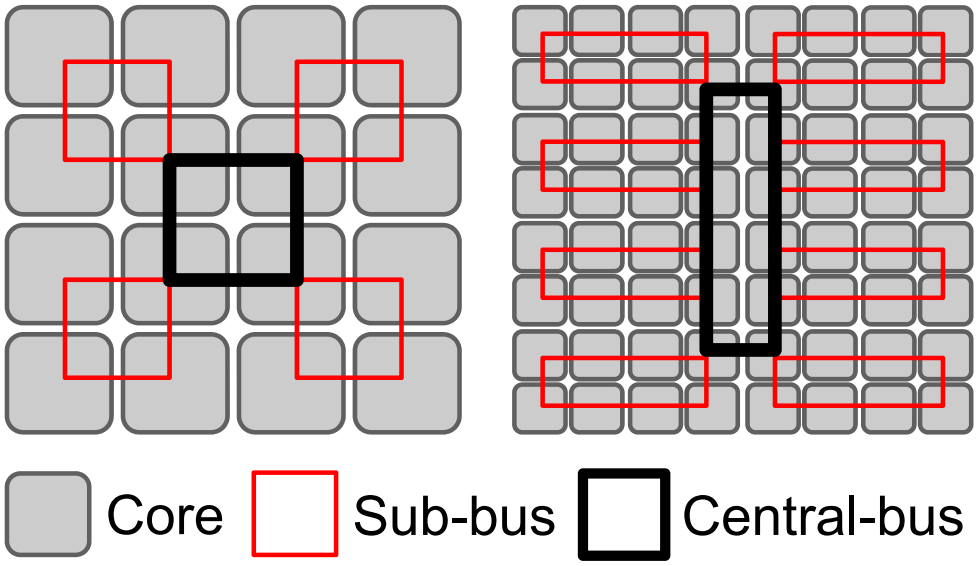

Figure 2: A hierarchical bus structure that localizes broadcasts to relevant clusters.

multiple entities likewise increases the time to detect bus signals. Second, with many entities competing for the shared bus, the wait-time to access the bus increases with the number of entities. Therefore, conventional wisdom states that more scalable switched-media networks are preferred when connecting much more than 8 or 16 entities [4]. However, the simplicity of bus-based protocols (such as the snooping-based cache coherence protocol) make it attractive for small- or medium-scale symmetric multiprocessing (SMP) systems. For example, the IBM POWER7^{TM} processor chip supports 8 cores on its SMP bus [8]. Buses are also attractive because, unlike switched-media networks, they do not require energy-hungry structures such as buffers and crossbars. Researchers have considered multiple innovations to extend the scalability of buses, some of which are discussed next.

One way to scale the number of entities connected using buses is, simply, to provide multiple buses, e.g., dual-independent buses or quad-independent buses. This mitigates the second problem listed above regarding the high rate of contention on a single bus, but steps must still be taken to maintain cache coherency via snooping on the buses. The Sun Starfire multiprocessor [3], for example, uses four parallel buses for ADDRESS requests wherein each bus handles a different range of addresses. Tens of dedicated buses are used to connect up to 32 IBM POWER7^{TM} processor chips in a coherent SMP system [8]. While this option has high cost for off-chip buses because of pin and wiring limitations, a multi-bus for an on-chip network is not as onerous because of plentiful metal area budgets.

Some recent works have highlighted the potential of bus-based on-chip networks. Das et al., [6] argue that buses should be used within a relatively small cluster of cores because of their superior latency, power, and simplicity. The buses are connected with a routed mesh network that is employed for communication beyond the cluster. The mesh network is exercised infrequently because most applications exhibit locality. Udipi et al., [14] take this hierarchical network approach one step further. As shown in Figure 2, the intra-cluster buses are themselves connected with an inter-cluster bus. Bloom filters are used to track the buses that have previously handled a given address. When coherence transactions are initiated for that address, the Bloom filters ensure that the transaction is broadcasted only to the buses that may find the address relevant. Locality optimizations such as page coloring help ensure that bus broadcasts do not travel far, on average. Udipi et al., also employ multiple buses and low-swing wiring to further extend bus scalability in terms of performance and energy.

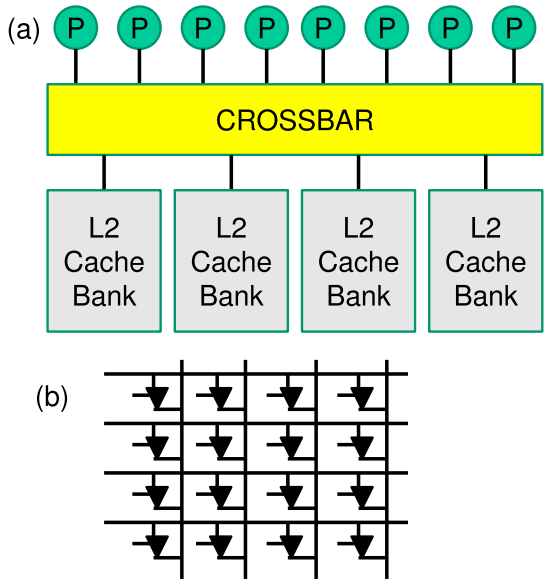

Figure 3: (a) A "dance-hall" configuration of processors and memory. (b) The circuit for a 4x4 crossbar.

Crossbars

Buses are used as a shared fabric for communication among multiple entities. Communication on a bus is always broadcast-style, *i.e.*, even though a message is going from entity-A to entity-B, all entities see the message and no other message can be simultaneously in transit. However, if the entities form a "dance hall" configuration (Figure 3(a)) with processors on one side and memory on the other side, and most communication is between processors and memory, a crossbar interconnect becomes a compelling choice. Although crossbars incur a higher wiring overhead than buses, they allow multiple messages simultaneously to be in transit, thus increasing the network bandwidth. Given this, crossbars serve as the basic switching element within switched-media network routers.

A crossbar circuit takes N inputs and connects each input to any of the M possible outputs. As shown in Figure 3(b), the circuit is organized as a grid of wires, with inputs on the left, and outputs on the bottom. Each wire can be thought of as a bus with a unique master, *i.e.*, the associated input port. At every intersection of wires, a pass transistor serves as a *crosspoint connector* to short the two wires, if enabled, connecting the input to the output. Small buffers can also be located at the crosspoints in *buffered crossbar* implementations to store messages temporarily in the event of contention for the intended output port. A crossbar is usually controlled by a centralized arbiter that takes output port requests from incoming messages and computes a viable assignment of input port to output port connections. This, for example, can be done in a crossbar *switch allocation* stage prior to a crossbar *switch traversal* stage for message transport. Multiple messages can be simultaneously in transit as long as each message is headed to a unique output and each emanates from a unique input. Thus, the crossbar is *non-blocking.* Some implementations allow a single input message to be routed to multiple output ports.

A crossbar circuit has a cost that is proportional to $N \times M$. The circuit is replicated W times, where W represents the width of the link at one of the input ports. It is therefore not a very scalable circuit. In fact, larger centralized switches such as Butterfly and Benes switch fabrics are constructed hierarchically from smaller crossbars to form *multistage indirect networks* or *MINs.* Such networks have a cost that is proportional to $NIog(M)$ but have a more restrictive set of messages that can be routed simultaneously without blocking.

A well-known example of a large-scale on-chip crossbar is the Sun Niagara processor [11]. The crossbar connects eight processors to four L2 cache banks in a "dance hall" configuration. A recent example of using a crossbar to interconnect processor cores in other switched point-to-point configurations is the Intel QuickPath Interconnect [7]. More generally, crossbars find extensive use in network routers. Meshes and tori, for example, implement a 5x5 crossbar in router switches where the five input and output ports correspond to the North, South, East, West neighbors and the node connected to each router. The mesh-connected Tilera Tile-Gx^{TM} 100-core processor is a recent example [13].

RELATED ENTRIES

Broadcast
Cache Coherency
Communication in Parallel Computing
Direct Networks
Interconnection Networks
Multistage Networks
Network Interface
PCI-Express
Routing
Switch Architecture
Switching Techniques

BIBLIOGRAPHIC NOTES AND FURTHER READING

For more details on bus design and other networks, readers are referred to the excellent textbook by Dally and Towles [5]. Recent papers in the architecture community that focus on bus design include those by Udipi et al. [14], Das et al. [6], and Kumar et al. [12]. Kumar et al. [12] articulate some of the costs of implementing buses and crossbars in multi-core processors and argue that the network must be co-designed with the core and caches for optimal performance and power. A few years back, S. Borkar made a compelling argument for the widespread use of buses within multi-core chips that is highly thought-provoking [1, 2]. The paper by Charlesworth [3] on Sun's Starfire, while more than a decade old, is an excellent reference that describes considerations when designing a high-performance bus for a multi-chip multiprocessor. Future many-core processors may adopt photonic interconnects to satisfy the high memory bandwidth demands of the many cores. A single photonic waveguide can carry many wavelengths of light, each carrying a stream of data. Many receiver "rings" can listen to the data transmission, each ring contributing to some loss in optical energy. The Corona paper by Vantrease et al. [15] and the paper by Kirman et al. [10] are excellent references for more details on silicon photonics, optical buses, and optical crossbars.

The basic crossbar circuit has undergone little change over the last several years. However, given the recent interest in high-radix routers which increase the input/output-port degree of the crossbar used as the internal router switch, Kim et al. [9] proposed hierarchical crossbar and buffered crossbar organizations to facilitate scalability. Also, given the relatively recent shift in focus to energy-efficient on-chip networks, Wang et al. [16] proposed techniques to reduce the energy usage within crossbar circuits. They introduced a cut-through crossbar that is optimized for traffic that travels in a straight line through a mesh network's router. The design places some restrictions on the types of message turns that can be simultaneously handled. Wang et al. also introduce a segmented crossbar that

prevents switching across the entire length of wires when possible.

References

[1] S. Borkar. Networks for Multi-Core Chips – A Contrarian View, ISLPED Keynote, 2007. www.islped.org/X2007/BorkarISLPED07.pdf.

[2] S. Borkar. Networks for Multi-Core Chips – A Controversial View. In *Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN)*, 2006.

[3] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1), January 1998.

[4] W. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of DAC*, 2001.

[5] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 1st edition, 2003.

[6] R. Das, S. Eachempati, A. K. Mishra, N. Vijaykrishnan, and C. R. Das. Design and Evaluation of Hierarchical On-Chip Network Topologies for Next Generation CMPs. In *Proceedings of HPCA*, 2009.

[7] Intel Corp. An Introduction to the Intel QuickPath Interconnect. http://www.intel.com/technology/quickpath/introduction.pdf.

[8] Joel M. Tendler. POWER7 Processors: The Beat Goes On. http://www.ibm.com/developerworks/wikis/download/attachments/104533501/POWER7+-+The+Beat+Goes+On.pdf.

[9] J. Kim, W. Dally, B. Towles, and A. Gupta. Microarchitecture of a High-Radix Router. In *Proceedings of ISCA*, 2005.

[10] N. Kirman, M. Kyrman, R. Dokania, J. Martinez, A. Apsel, M. Watkins, and D. Albonesi. Leveraging Optical Technology in Future Bus-Based Chip Multiprocessors. In *Proceedings of MICRO*, 2006.

[11] P. Kongetira. A 32-Way Multithreaded SPARC Processor. In *Proceedings of Hot Chips 16*, 2004. (`http://www.hotchips.org/archives/`).

[12] R. Kumar, V. Zyuban, and D. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads, and Scaling. In *Proceedings of ISCA*, 2005.

[13] Tilera. TILE-Gx Processor Family Product Brief. http://www.tilera.com/sites/default/files/productbriefs/PB025_Gx_Processor_A_v3.pdf.

[14] A. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards Scalable, Energy-Efficient, Bus-Based On-Chip Networks. In *Proceedings of HPCA*, 2010.

[15] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. Beausoleil, and J.-H. Ahn. Corona: System Implications of Emerging Nanophotonic Technology. In *Proceedings of ISCA*, 2008.

[16] H.-S. Wang, L.-S. Peh, and S. Malik. Power-Driven Design of Router Microarchitectures in On-Chip Networks. In *Proceedings of MICRO*, 2003.