

## *A First Look at the Intel® QuickPath Interconnect*

**ABSTRACT:** A modern microprocessor-based computing system is comprised of a collection of sophisticated high performance components. Processors, memory devices, and input/output controllers are the building blocks from which the computer is constructed. These devices are very tangible evidence of the rapid ongoing advances in silicon technology and manufacturing methods. What is less easily discerned is the underlying fabric that ties together these elements. As the demands placed upon computing systems have continued to go up, the demands upon this fabric, or system interconnect, have also increased, especially in systems composed of multiple processors which requires architectural advances to allow the system to reach its full potential. As we shall see in this article, the Intel® QuickPath Interconnect (Intel® QPI) is one of those architectural advances.

### Evolution of Microprocessor Interconnect

The invention of the microprocessor started a revolution in computing. The Intel 4004, the first commercial single chip microprocessor, was very basic in its design and architecture with very modest instruction and data rates. These rates matched well with the memory technology of the day, and very usable and balanced systems could be constructed by directly connecting the processor to memory. A system bus served as the interconnect mechanism and carried the data between the memory subsystem and the processor.

The microprocessor has evolved over time, driven along by improvements in microarchitecture, manyfold increases in clock rates, and advancements in silicon design and manufacturing techniques. This increase in computing power demanded greater data rates from the memory system. Corresponding improvements were made in DRAM technology and the requirements for data were readily met by the memory systems of the day. The technology of signaling also improved and the data rate of the system interface kept up with the rest of the devices.

Things changed in the 1980s with the advent of more sophisticated, pipelined microarchitectures, such as the Intel 80486. The DRAMs could no longer keep pace with the data rates required and cache memory was introduced into microprocessor based designs. This high speed memory is built with fast SRAM technology. It has low latency and the high data rates required to keep the processor operating without stalling while waiting for data from the relatively slower DRAMs. However, cache memory devices are generally considerably more expensive and power hungry than slower DRAMs, so cache sizes tend to be small in order to provide the greater speed at a reasonable system cost. As a result,

caches store only a small portion of the data from the main system memory. They rely on the facts that typical programs are composed of small sequences of straight line code that loop back to repeatedly execute the same code many times, and that programs tend to operate on the same set of data entities for a while before moving on to the next set of data. In other words, caches take advantage of the idea of spatial and temporal locality, the idea that if you access data, you are highly likely to access nearby data.

A cache memory exploits these characteristics to provide a significant reduction in latency and to achieve high data throughput. The cache sub-system does so by fetching sequences of data from system memory beyond that which is immediately requested by the processor and storing it for fast access when needed. The system interconnect evolved to serve as the interface between the cache and the system memory and typically carries sequential bursts of data entities called *cache lines*. A cache line can be anywhere from 16 to 128 bytes of sequentially addressed data depending upon the design of the cache. The system memory, typically built with DRAM devices is well suited to provide such a burst of contiguously addressed data at high speed. The signaling technology of the interconnect also evolved to run at higher data rates, keeping up with the needs of the faster processors.

The next change in computers and interconnect systems occurred in the early 1990s with the introduction of multiprocessor systems. Two or more processors were connected to a common, shared memory system over the system interconnect. These processors operated in concert to share the overall workload and provided higher performance when compared to a single processor system. The system interconnect had to evolve yet again to efficiently handle requests from all the processors in the system. This was achieved by fairly handling requests between all the processors and efficiently pipelining their requests to memory. The signaling technology was updated to handle the electrical loads of multiple processors on a single bus at high transfer rates. The system interconnect was also enhanced to properly share the information cached in the processors.

The Intel® Pentium® Pro microprocessor was the first Intel architecture microprocessor to provide a system interconnect, the Front Side Bus (also abbreviated as the FSB), which supported symmetric multiprocessing. The FSB can connect up to four processors, a memory controller, and an I/O controller. The FSB can pipeline up to eight transactions for high throughput. The FSB also provides mechanisms to ensure that all the processors' caches share data from system memory properly and do not use stale data that has been modified in another processor's cache. Sharing data coherently between caches is a key capability required in any high performance system interconnect.

The FSB served the needs of the Intel Pentium Pro family of processors and was then enhanced to meet the needs of the Intel Pentium 4 processors that followed. Improvements in bus bandwidth were achieved by increasing the data transfer rates. A modified form of Gunning transceiver logic (called GTL+) signaling is used by some products to operate the FSB at clock rates of 400 MHz, and therefore deliver quad-pumped information transfer rates of up to 1.6 gigatransfers per second (GT/s). Part of the bandwidth improvement was made possible by reducing the number of processors connected to a single bus down to two (dual independent bus) and eventually to one (dedicated high speed bus). Reducing the number of devices connected to a FSB mitigates some board routing and signal quality concerns and therefore removes some obstacles of getting to higher data rates. Thus a four processor system required four FSBs to connect the four processors to a single central memory controller. This proved to be an expensive solution in that the memory controller hub (MCH), the chipset providing the memory and I/O interfaces on the platform, required over a thousand pins just to accommodate the four FSBs. More MCH pins were needed to handle the interface to memory. Moreover a single memory controller may become a bottleneck in the system that limits both the memory size and bandwidth that could be built into a system.

The architects of the Intel QuickPath Interconnect started by taking a fresh look at the needs of the entire system, striving to provide a complete solution that addressed future growth requirements. Intel QPI is intended to provide a very capable system interconnect fabric that is at the heart of scalable, high performance systems. Of course, due to its usage in very high production volumes, Intel QPI had to provide these functions while also keeping system and silicon costs as low as possible. The Intel QuickPath Architecture has been designed for future generations of Intel processors and provides plenty of headroom for growth in performance and features. Intel QPI achieves these goals through the use of narrow, packetized, cache-coherent, message based, high speed point-to-point links that use less than half the number of signals of the FSB. This results in lower package and routing costs and yet provides over fifty percent more bandwidth for higher performance. Intel QPI is also very flexible, allowing one to build highly scalable systems around multiple processors, memory controllers, and I/O controllers. Systems can choose to integrate the memory controller and I/O controllers onto the same die with the processor. Such systems can be upgraded in a modular fashion. Every processor added to a system will bring additional memory bandwidth and capacity with its integrated memory controller, and provide a very cost effective upgrade that offers a well balanced system. The Intel QPI cache coherence protocol also provides an efficient means to resolve cache coherence issues involving multiple processors.

Let us look at the issues created by multiple caches in a system and ways to keep them coherent so that all processors will always get the most up-to-date

information. We will then describe the high performance cache coherence mechanisms in Intel QPI, and how these mechanisms can handle systems containing a large number of devices such as processors and intelligent I/O controllers.

## Solving the Cache Coherency Problem

Whenever multiple processors, each with its own cache, cooperate to access and modify data in a shared memory system, they run the risk of accessing stale or outdated data that may have been modified by one of the other processors. This cache coherency problem can be best illustrated by an example from the real world.

Let us say that Mary, an attorney in a law firm, has to draw up a legal contract. She pulls together a team of experts, Robert, Janice, Patty, and Tom, who will all help to create the final document. Mary starts with a boilerplate form of the contract and creates a table of contents assigning page numbers to each section. She shares this with her team. She then prints out the boilerplate document and places it in a central location accessible to the entire team.

Robert decides he needs to study pages seven through ten and makes copies of those pages and takes them to his office for his own use. Similarly, Janice decides she needs pages eight through ten and copies and takes them. Similarly Patty and Tom make copies of pages that are of interest to them and the end result is that all four team members now have copies of document pages in their own respective offices (caches). Multiple copies of any page can exist locally, and each individual can refer to his or her own copy.

If Robert decides that he has no further use for page nine he can destroy his copy of that page. This ability to cache copies of pages works well as long as no member makes any changes to the pages in his or her office. However if Robert decides to change the contents of page ten then he must take steps to ensure that Janice is not working with obsolete information on her copy of that page. If Janice also decides to update page ten, she makes the problem even worse. This is the basis of the problem of cache coherency.

The team must institute a set of rules on how to handle updates in order to ensure that they have a graceful way of collaborating to produce an accurate document containing all the latest changes. This set of rules can range from something very simple but restrictive and with much overhead, to one that is more sophisticated and lets each team member work much more autonomously. Let us look at two ways of keeping the caches coherent.

## Write-Through Caching

The team can follow a simple set of rules whenever anyone decides to update a page. In our example above when Robert is ready to update page ten he tells the other team members that he is doing so, giving them the page number. They all check to see if they have copies of the pages and if so, every team member except Robert will destroy their copies. Robert then makes the change to page ten and places it at the central location. Robert can choose to keep a copy for himself, if he desires. If Janice now decides to make a change to the same page, she must go to the central location for the latest copy as she destroyed her copy of the page when Robert announced his intention to update page ten. She too must announce to all that she is about to change page ten and they all, including Robert, must destroy their copies of that page. Once she makes her update, she must then place a copy of the updated page ten in the central location. In case both Robert and Janice decided to make the change simultaneously they can toss a coin to decide who goes first. The other team member will then have to fetch the updated copy from the central location to merge in his or her updates. This mechanism, in computer architecture referred to as *write-through caching*, is the simplest form of a cache coherence mechanism for handling updates. A simple and efficient mechanism is required to announce which page is being changed. Each team member must always take the time to put the most up-to-date copy of the page in the central location after every update, for all the others to use.

## Write-Back Caching

The team members decide that writing through to the central location is unnecessary overhead. Each team member should be able to keep the pages they have modified, as they are likely to make several more updates to them. However any team member that has made changes to a page, would have to forward that modified page to other members when that page is requested by another team member. This is *write-back caching* and is the cache coherence mechanism used on modern microprocessors.

Let us see how our legal team would work under the rules of write-back caching. Robert starts by getting copies of pages seven through ten. He announces to all the others that he has done so. Next Janice get pages eight through ten and announces to all members that she has done so. Robert, Patty, and Tom check their copies of pages to see if they have a local copy of any of the ones Janice has fetched. Robert sees that he does and makes a notation on his copies of pages eight through ten that they are shared, and he also lets Janice know that he has copies of those pages. Janice now marks her copies of pages eight through ten as shared with someone else. Note that Robert is the only one with a copy of page seven, and it is exclusively in his office (cache) as long as no one else fetches a copy.

When Robert is ready to make his updates and starts with page ten, he sees that it is shared with someone else. So he announces his intention to modify page ten and then must wait for a response from the rest of the team. All the others check their copies of the pages and destroy their copies, as none of them has made any changes to their local versions. So Janice throws away her copy of page ten. Robert now makes the change to page ten and keeps it with him. At this point he has the most up-to-date contents of that page, so he marks it as having been modified. This modified page is the only version of that page that is up to date in the system, there are no other valid copies at this point. When Janice is ready to make a change to page ten, she goes to the central location to get a copy but also announces to all other team members about her desire to change page ten. Robert, seeing that his modified copy of page ten is the most up-to-date, informs Janice of this. He then gives her his modified copy of page ten, leaving Robert without a copy. Now Janice has the only up-to-date copy of page ten and she can make additional changes to it, or store it in her cache. However she must not destroy it as this is the only copy of page ten that is up-to-date. If Robert and Janice had both decided to update the page simultaneously, then either one of them could have gone first and then handed the modified page to the other for further updates.

Recapping, each team member can hold local copies of pages as long as they keep track of the state of the page in the system. The page can be:

- **Shared** with one or more members of the team. The team member can destroy their copy of this page at any time if he or she no longer needs it and does not need to inform anyone of that action.
- **Exclusive** in only one cache. The owner can destroy her copy if she no longer needs it, as this page is up to date in the central location. This exclusive copy of the page has not been updated at this point, but could be updated by the owner of that page without informing anyone else of the change.
- **Modified** - this page can exist in only one cache. The owner can make further changes to it at will without informing the other team members. The owner must forward the page to anyone else who needs it. If the owner no longer needs the page, then she or he must put it back in the central location as it is the only up-to-date copy.

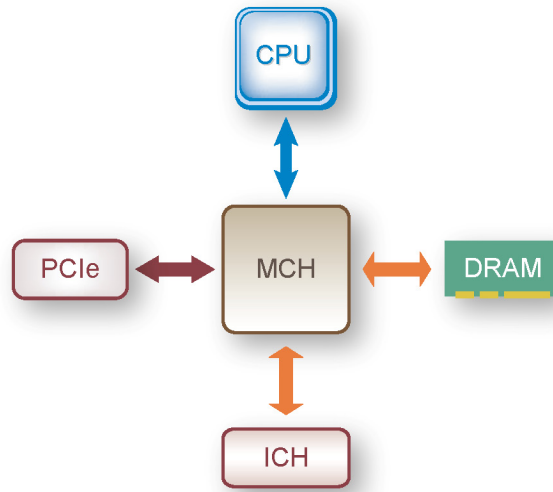
The team members must communicate with each other to properly share the pages of their document. All of the communication for this purpose is termed *coherency traffic*. However, the team members may also send messages to each other for other purposes. For example, if Janice decides to take a coffee break and invites Robert to join her, messages between them about breaking for coffee would have no bearing on the shared document and would be termed as *non-coherent traffic* in computer parlance.

## Tying It All Together

Multiple processor systems operate under very similar rules for sharing data described in our example above. The caches, represented by the team members above, can hold multiple cache lines. Each cache line is typically composed of sixty-four bytes of data and is the smallest entity handled and tracked by the cache—akin to the document page in our example. The cache controller tracks the state of each cache line and marks it as Modified, Exclusive, or Shared and responds accordingly. If a cache line is no longer up to date in a cache, the controller marks it as Invalid. These, taken together, are referred to by their initials as the MESI states (pronounced “messy”). The central memory in a computer system is the common repository of the document in our example above. A typical computer system has at least one memory controller that interfaces with the banks of DRAM memory.

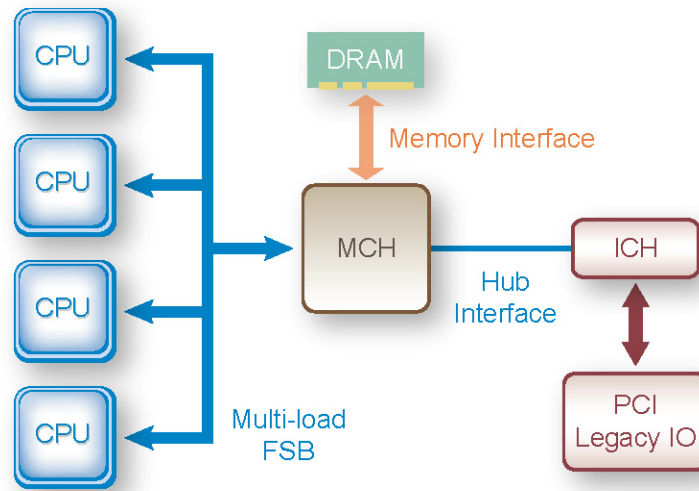
## Anatomy of a Multiprocessor System

A typical single processor system is composed of a processor, a memory controller hub (MCH), and an I/O controller hub (ICH). The processor, with its internal cache, is connected to the memory controller with a system interface bus. The memory controller in turn is connected to the I/O controller using another bus designed for that purpose. The system interface bus carries the requests from the processor to the memory controller and the I/O controller. Each request consists of a command or action to be performed, an address of the location in memory or I/O, and optionally data if it is an operation to write out data. This typical single microprocessor based computer system is illustrated in Figure 1.



**Figure 1** Single Processor Computer System Based on the Front Side Bus

The Intel Pentium Pro multiprocessor system, introduced in the mid-1990s, connected as many as four processors on the system interface bus to the memory controller, as shown in Figure 2. This FSB added the capability to manage cache coherence between the processor caches using the write-back protocol.



**Figure 2** Four Processor Computer System Based on the Front Side Bus

The system interface between the CPUs and the MCH operates in a manner very similar to our example of the legal team described earlier. The details of a simple request for data are as follows:

A processor requests data from memory over the Front Side Bus. All the other processors observe this request go by as the FSB connects them all together. These processors check their caches to see if any one of them has a copy of the cache line being requested. This operation of checking the caches of adjacent processors is dubbed a *snoop* in the cache coherence protocol. The term *probe* is also used to describe this operation.

The snoop can produce one of several results depending upon the status of the line in that cache. If the cache does not have a copy of the line being requested, then the cache takes no action. The memory controller will provide the data to the requesting processor. If none of the caches in the system have the data, then the requesting processor places the data in its cache and marks it as Exclusive.

If the requested cache line is in any cache in the Shared or Exclusive state, the cache with the line in that state signals as such on the FSB. If a cache has the line in the Exclusive state, that cache changes its state to Shared. The memory controller returns the cache line data to the requesting processor, which places the cache line into its cache in the Shared state because one or more caches have indicated that they also have copies of that data.

If the cache line is in the Modified state in the snooped cache, then it signals as such on the FSB. The memory controller recognizes this signal and does not return data. Instead the cache that has the Modified line places the data on to the bus and sends it to the processor requesting the cache line. That cache then marks its own copy of the line as Shared. The receiving processor puts the data in its cache and also marks it Shared. The memory controller simultaneously takes a copy of the data and writes it into system memory.

The rules of operation define similar sequences for all caches to follow where they want to modify data in a cache line or evict the line from the cache to system memory if the line has been updated. These rules are identical to those followed by the legal team when they needed to change the contents of a document page.

It is important to point out that if the rules of operation are correctly defined, and all devices properly follow those rules, then certain characteristics will be ensured. In particular any given snoop will result in all caches responding with either:

- no hits - no Shared, Exclusive, or Modified copies
- a single hit - a single Exclusive or Modified copy
- multiple hits - all Shared
- or the final option, at most one hit-Modified.

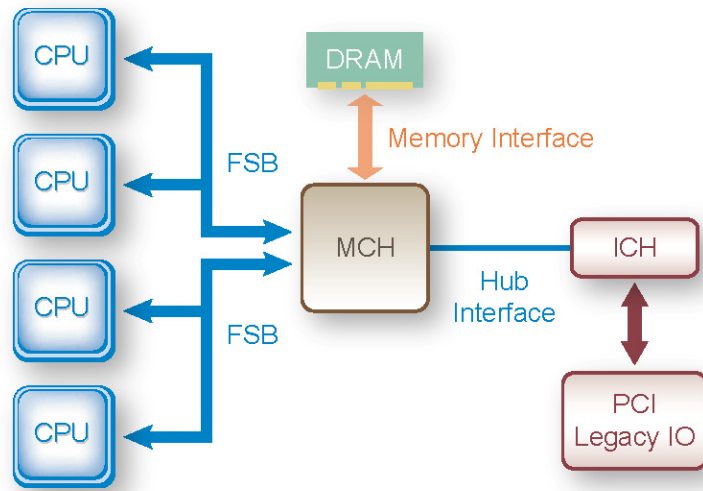
Under no circumstances would multiple caches signal a hit-Modified condition and

attempt to simultaneously write their Modified lines to the same location in memory.

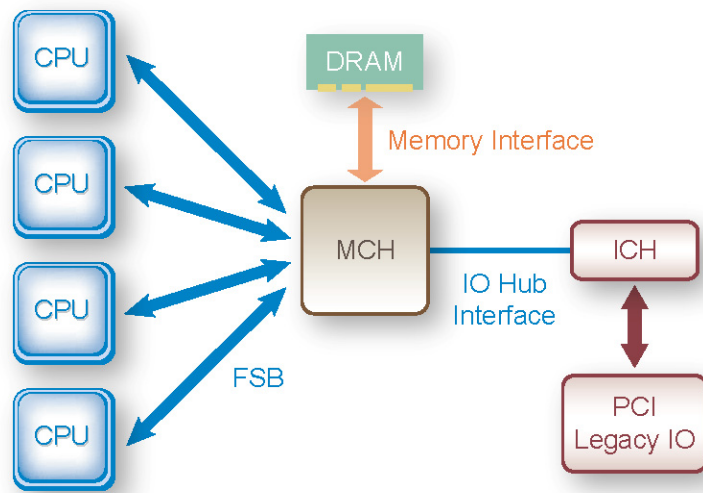
## Evolution to a Link based System

The Front Side Bus works well and offers a simple and elegant solution for multiple processor systems. As computer systems speed up, driven by improvements in processor architecture, silicon technology, and memory technologies, the FSB is operated at a faster rate in order to keep up with the data rates required for a balanced system. This approach has worked well for over a decade through at least five generations of processor evolution at Intel, starting with the Intel Pentium Pro, and continuing beyond the Intel Pentium 4 to the most recent (2009) Atom family of processors.

However, the approach of constantly increasing the data rates on the FSB will eventually encounter some practical considerations that result in diminishing returns for the efforts or costs involved in attaining the faster speeds. Pushing the FSB to operate at data rates faster than 800 MT/s, while still having five electrical loads (devices) on the bus, becomes very problematic. An alternative approach was to reduce the number of loads per bus, first to three and eventually to two. With this approach, the FSB rates were pushed up to 1000 and eventually 1600 MT/s (in certain products). Figures 3 and 4 show the system configurations through these successive steps. To support the four processor configuration of Figure 4, note that the memory controller is designed to support four electrically independent FSBs connected to it. One side effect of this is that the memory controller in these systems has over 1500 pins split across four FSBs with 175 signals each. Additional MCH device pins are needed to connect to the memory arrays. This is obviously driving the cost of the MCH device in the wrong direction. Another aspect of these two system configurations is that the entire system memory is behind that one MCH controller.



**Figure 3** Four Processor System with Two Frontside System Buses



**Figure 4** Four Processor System with Four Independent System Buses

## A System Architecture Transition

Several trends in the evolution of technology made it evident that a fresh look at the entire architecture of a microprocessor system was required. First Intel's latest generation of processors, with multiple cores on a single die, demand ever higher data bandwidth. Multiples of these processors in a system require more

memory bandwidth than a single memory controller can economically support. Second, thanks to Moore's Law, it is significantly more economical to integrate memory controllers into the same die as the processors and provide a very modular computational unit. These modular processor and memory units can be used to create powerful multiprocessor systems when connected together with an appropriate interface. The Front Side Bus could have been extended to serve this need but would require significant changes that would render it incompatible with the current systems. Once the architects recognized that the benefits of backward compatibility would have to be sacrificed they decided to take a fresh look at the needs of multiprocessor systems and design an interface with the future in mind.

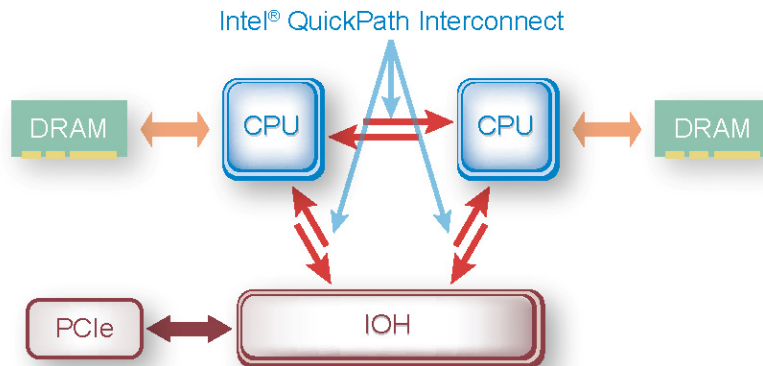
The key objectives of the new interface were:

- Significantly Improved Performance and Bandwidth. The goal was to create an interface that would make a radical improvement in system bandwidth and meet the latency goals of the new crop of processors being conceived. The interface also had to be readily scalable to meet future bandwidth demands.
- Software and Processor Family Compatible. The new interface had to be completely transparent to the operating system and user software once the system interface had been properly initialized. Moreover this interface had to be common for future platforms and support both the Itanium and Intel Architecture family of processors.
- Low Cost. The interface had to be economical to build on the silicon and in the system boards. An interface with high data rates would make the most efficient use of signals pins and traces.
- Scalable. The interface had to be architected with an eye on all the possible system topologies of the future that were reasonable to consider and provide the necessary capabilities to support their needs.
- Power Efficient. The interface had to be power efficient and meet all the requirements of power control and management capabilities of different platforms.
- Reliable and Serviceable. This interface was to be used in server systems where reliability and serviceability are key requirements. The interface had to provide the necessary features for robust, error-free operation of the system even in the event of signal interference and disruption. Moreover the interface needed capabilities to support more robust memory systems.
- Support new capabilities. The architecture of the interface had to provide

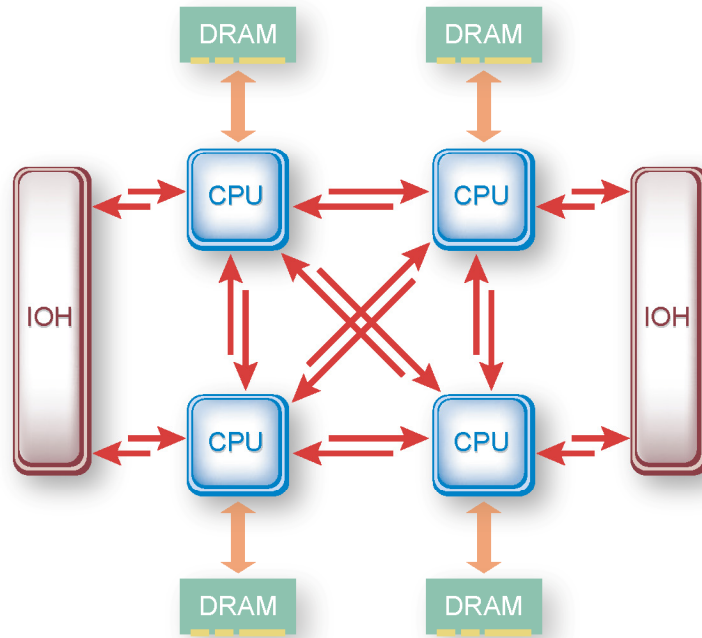
The Intel Quickpath Interface was created to meet all these objectives.

## Anatomy of an Intel® QuickPath Interconnect System

Figures 5 and 6 show some possible configurations of two- and four-processor systems respectively that can be built using the Intel QPI. Each processor typically has a memory controller on the same die, allowing systems to be more scalable in performance. However this is not essential and systems can have separate, discrete memory controllers. Similarly, the I/O subsystems can either be incorporated onto the same die as the processors or built as separate I/O hubs.



**Figure 5** System with Two Processors Employing Intel® QuickPath Interconnect Technology



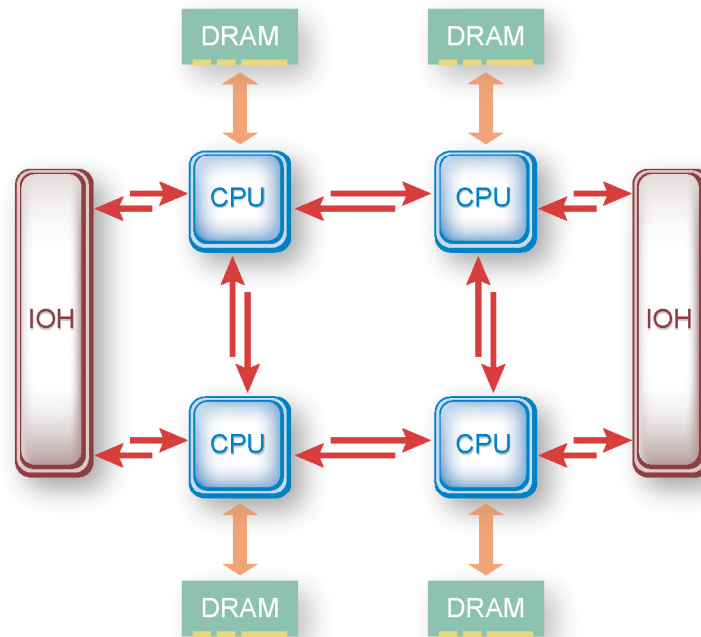
**Figure 6** System with Four Processors Employing Intel® QuickPath Technology

These systems have multiple Intel QPI links (indicated by the red arrow pairs in the figures) connecting the devices to each other. Each one of the links can operate independently of the other links. The performance of such systems can be very high, particularly if the processors are allocated independent tasks, working on data that are optimally distributed across the different memory controllers and close to their own processors. Most current operating systems do recognize such system configurations with Non-Uniform Memory Accesses (NUMA) from each processor and the OS places the data in the memory accordingly.

The systems shown in Figures 5 and 6 are fully connected, at least in that each processor has a direct link to every other processor in the system. However, it is possible to build systems using Intel QPI where the devices do not connect to all others. Figure 7 shows a four processor system where each processor only uses

two sets of links to connect to the other processors.

---

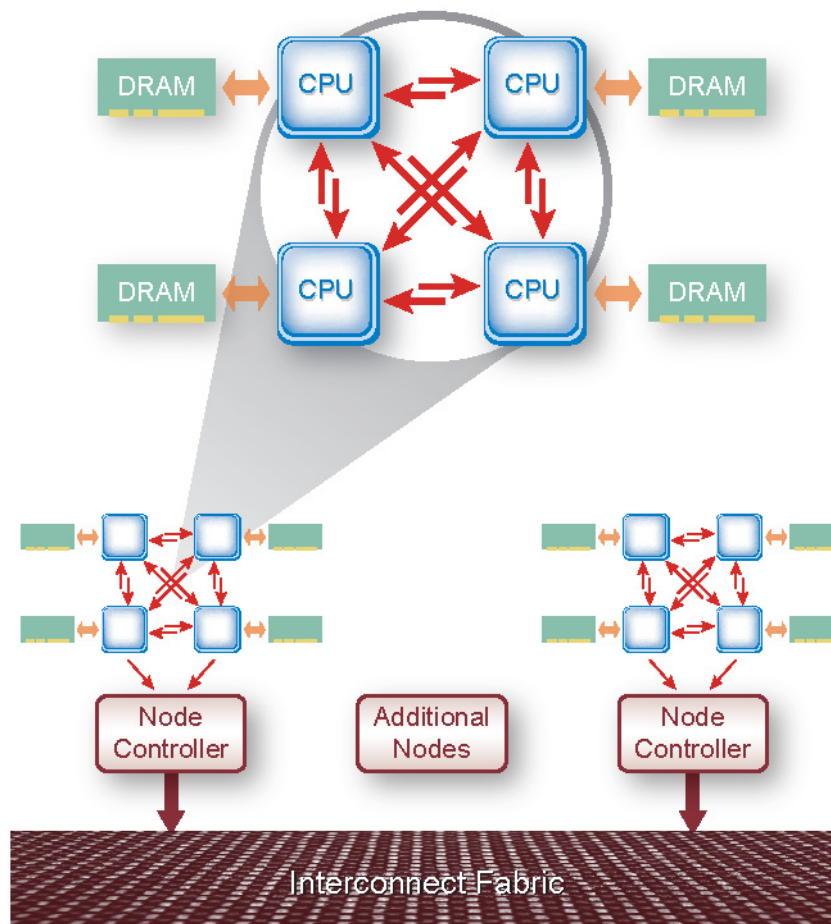


---

**Figure 7** Partially Connected System with Intel® QuickPath Interconnect

If processor A in Figure 7 needs to access the memory controller in processor C, it sends its request through either processors B or D, that must in turn forward that request on to the memory controller in C. Similarly, larger systems with eight or more processors can be built using processors with three links, and routing traffic through intermediate processors. Intel QPI routing mechanisms enable such systems to be built. However, systems that are partially connected cannot perform as well as fully connected systems due to the longer latency involved with routing through the intermediate processors, and possible bandwidth congestion on the fewer links in the system. Intel QPI also provides mechanisms to help improve the performance of such partially connected systems by reducing the amount of traffic that is created across the links.

Still larger systems can be created by connecting the processors in a hierarchy. Two or four processors could be connected to a *node controller*, forming a cache coherent *node*. Multiples of these cache coherent nodes are connected together using either Intel QPI or some other scalable, coherent interconnect. This approach enables creating systems with many more than 2, 4, or 8 processors. Figure 8 illustrates the organization of such systems, which are capable of very high performance on distributable programs. Clearly the operating system must play a large role in distributing tasks to each processor in order to get the highest performance from such systems.



**Figure 8** Hierarchical Multi-Processor System with Intel® QuickPath Interconnect, Using Node Controllers

## About the Authors

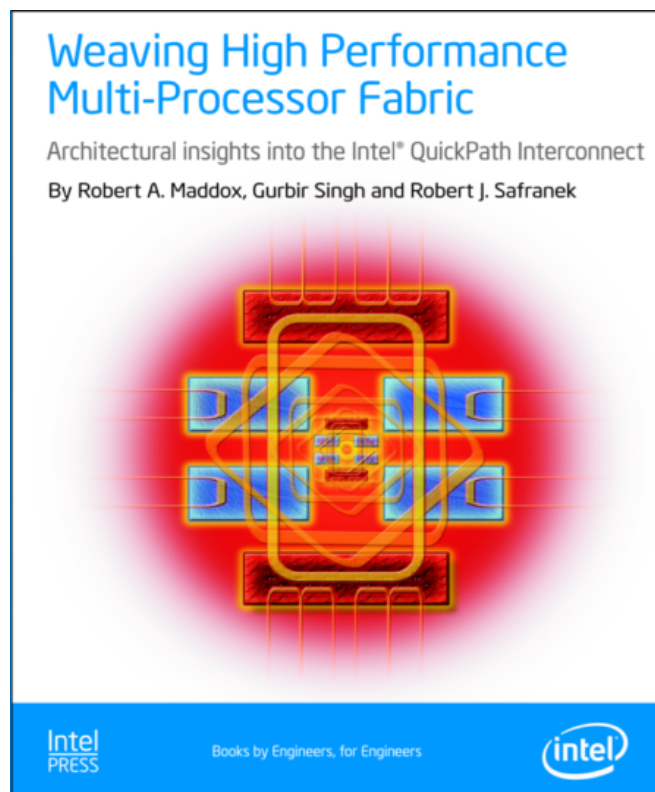
**Robert A. Maddox** joined Intel® Corporation in 1998. He is currently a Staff Technical Marketing Engineer in the Server Platforms Group, focused on working with the industry on enabling the use of the Intel® QuickPath Interconnect.

**Gurbir Singh** joined Intel® Corporation in 1984 and is now a Senior Principal Engineer in the Digital Enterprise Architecture and Planning group in Intel. Gurbir led the architecture team defining the Intel® QuickPath Interface. He holds thirty patents in the field of system interfaces and cache architecture.

**Robert J. Safranek** joined Intel® Corporation in 2000. He has been working on the definition of the Intel® QuickPath Interconnect since its inception and was the primary architect for the first IA-32 products based on it.

This article is based on material found in book *Weaving High Performance Multi-Processor Fabric* by Robert A. Maddox, Gurbir Singh and Robert J. Safranek. Visit the Intel Press web site to learn more about this book:

[http://www.intel.com/intelpress/sum\\_qpi.htm](http://www.intel.com/intelpress/sum_qpi.htm)



Copyright © 2009 Intel Corporation. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Publisher, Intel Press, Intel Corporation, 2111 NE 25 Avenue, JF3-330, Hillsboro, OR 97124-5961. E-mail: [intelpress@intel.com](mailto:intelpress@intel.com) .