High-throughput coherence control and hardware messaging in Everest

by A. K. Nanda
A.-T. Nguyen
M. M. Michael
D. J. Joseph

Everest is an architecture for high-performance cache coherence and message passing in partitionable distributed shared-memory systems that use commodity shared multiprocessors (SMPs) as building blocks. The Everest architecture is intended for use in designing future IBM servers using either PowerPC® or Intel® processors. Everest provides high-throughput protocol handling in three dimensions: multiple protocol engines, split request–response handling, and pipelined design. It employs an efficient directory subsystem design that matches the directory-access throughput requirement of high-performance protocol engines. A new directory design called the complete and concise remote (CCR) directory, which contains roughly the same amount of memory as a sparse directory but retains the benefits of a full-map directory, is used. Everest also supports system partitioning and provides a tightly integrated facility for secure, high-performance communication between partitions. Simulation results for both technical and commercial applications exploring some of the Everest design space are presented. The results show that the features of the Everest architecture can have significant impact on the performance of distributed shared-memory servers.

1. Introduction

Large shared-memory machines typically use smaller commodity SMPs as building blocks to provide economy of scale [1–3]. Two primary factors influence the use of SMP nodes as building blocks as opposed to single-processor nodes. First, the industry tends to build as large an SMP as possible using a fast bus and avoiding nonuniform memory latencies. Second, it is also considered desirable to amortize coherence controller and network interface costs as well as memory and I/O controller costs over several processors. The use of larger per-processor L2 caches and faster processor-memory

0018-8646/01/$5.00 © 2001 IBM

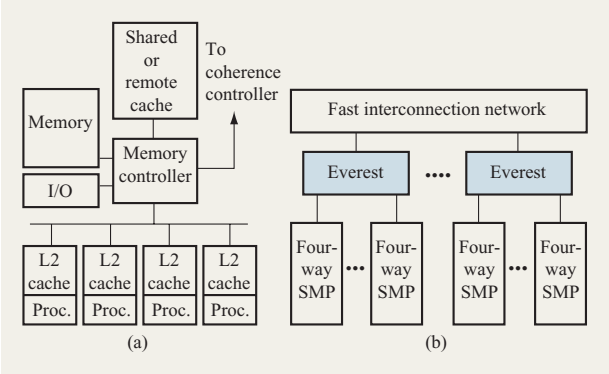

Figure 1

Shared-memory system environment: (a) Typical four-way SMP design; (b) typical Everest-based system.

buses has helped to alleviate the bus bandwidth problem in connecting several very-high-speed processors in an SMP. To connect these SMPs in order to build a distributed shared-memory machine, high-speed network designs have begun to appear [1, 3] that match the bandwidth of the SMP bus.

It is imperative that the coherence controllers attached to each SMP node (or to multiple SMP nodes) in such machines also be designed to handle the high memory-access traffic and the associated coherence transactions that appear on the bus. Current IBM Intel**-based and PowerPC*-based servers both incorporate SMP-based scalable shared-memory architectures. These servers will experience a bottleneck unless the coherence controllers provide high-throughput protocol processing. The Everest architecture was developed to provide high-throughput protocol handling for this class of machines, using parallelism in three dimensions: 1) multiple protocol engines, 2) split request–response handling, and 3) pipelined design, in addition to an efficient directory subsystem design that matches the directory-access throughput requirement of the high-performance protocol engines (PEs).

The multiple protocol engines in the Everest architecture are assigned to non-overlapping memory regions to provide parallelism at the highest functionality level. Each PE can be optionally pipelined and can have up to two split request–response units, one of which handles protocol request transactions while the other handles protocol response transactions. The Everest architecture uses a new directory design called the complete and concise remote (CCR) directory, which contains roughly the same amount of memory as a sparse directory [4] but retains the benefits of a full-map directory [5]. Because of its smaller size, the CCR directory can be designed using relatively faster memory to provide lower directory-access latency. Average directory-access time is further reduced significantly by the use of directory caches [6].

With Everest, the SMP building blocks can be organized into partitions, each of which is a complete shared-memory multiprocessor with its own operating system. The communication links between the partitions can then be exploited for message passing. Everest includes an interpartition communication facility (IPC) that provides high-performance message passing between partitions. The IPC supports secure, connection-oriented communication for potentially thousands of user-space processes. Address-translation support in the IPC allows user applications to specify zero-copy transfers directly to or from their own virtual address space.

This paper also presents the results of an empirical investigation into coherence-controller throughput in the context of the Everest architecture. The results show that using parallelism in any of the three dimensions or combinations of them has significant impact on the performance of applications with high communication requirements. The performance benefits of parallelism become more prominent when the number of processors handled by the coherence controller increases. The results on relative performance gain while adding incremental parallelism in one or more dimensions can be used by designers to determine performance/complexity tradeoffs on specific Everest implementations. We have presented performance tradeoffs in high-throughput coherence controllers in a recent paper [7].

The rest of the paper is organized as follows. Section 2 presents the Everest architecture. Simulation results are discussed in Section 3, and Section 4 presents conclusions.

2. The Everest architecture

The Everest architecture is intended to meet varying degrees of coherence-controller (CC) throughput requirements in wide-ranging node designs for future IBM servers employing distributed shared-memory systems. It achieves high throughput by providing aggressive parallelism in protocol processing. The amount of parallelism actually implemented in a specific Everest coherence controller, however, will depend on the coherence throughput required by a particular design and workload. The coherence throughput requirement is primarily a function of the number of processors connected to the CC and the processor and bus speeds. A representative shared-memory system environment is shown in **Figure 1**, with a typical SMP design shown in part (a) and a typical shared-memory system using Everest, each memory handling one or more SMPs, shown in part (b). Contemporary commodity SMP nodes do not scale well beyond four processors on a single bus because

of loading and the speed limit of the bus. Each SMP has a shared cache or a remote cache that is shared by all processors in the SMP. Future shared-memory designs will probably use one of these structures in an SMP node because of their high-performance potential. In a shared-cache configuration, both remote memory and local memory lines are cached. (Local cache lines with respect to a node are cache lines that belong to the main memory of that node. Remote cache lines with respect to a node are cache lines that belong to the main memory of another node.) The remote cache caches only those remote lines that are being used by the processors in the SMP. In this paper we assume the presence of a large remote cache in each SMP. One or more SMPs are connected to Everest, which maintains coherence among the SMPs connected to it as well as the remote SMPs connected through a fast network. Multiple network ports could be connected to the CC, depending on the network speed.

To extract parallelism at the highest level of functionality, the Everest architecture uses one or more protocol engines (PEs) operating independently, as shown in **Figure 2**. The study in [8] used one PE for handling requests to locations resident on remote memory (RPE) and another for handling requests to locations in the local home memory (LPE). Everest goes a step further by providing multiple local and remote engines. The amount of parallelism in a particular implementation of the Everest architecture can be expressed by the tuple $L_xR_yP_uS_v$, meaning that there are x LPEs, y RPEs, u stages in the pipeline of each PE, and v split request–response units in each pipeline. The simplest coherence controller includes one LPE ($x = 1$), no RPEs ($y = 0$), one pipeline stage ($u = 1$), and one execution unit ($v = 1$), and the LPE handles both remote and local requests. Parallelism in any of the dimensions can be added one element at a time or simultaneously depending on the coherence throughput requirement of the system. In this study we explore only that part of the design space of the tuple $L_xR_yP_uS_v$ which seems feasible.

Everest uses a new coherency directory design called the complete and concise remote (CCR) directory. The CCR directory preserves the properties of a full-map directory [1, 5] and is comparable in size to a sparse directory [4]. The CCR directory design provides higher bandwidth than conventional full-map directories without losing the benefits of such directories. Everest provides an independent directory cache [6] in each LPE to reduce the average latency of directory lookups. The RPEs do not have to access the directory.

The interpartition communication facility is tightly coupled to the protocol engines, which provide it with high-bandwidth, low-latency access to memory. User applications in different system partitions can establish

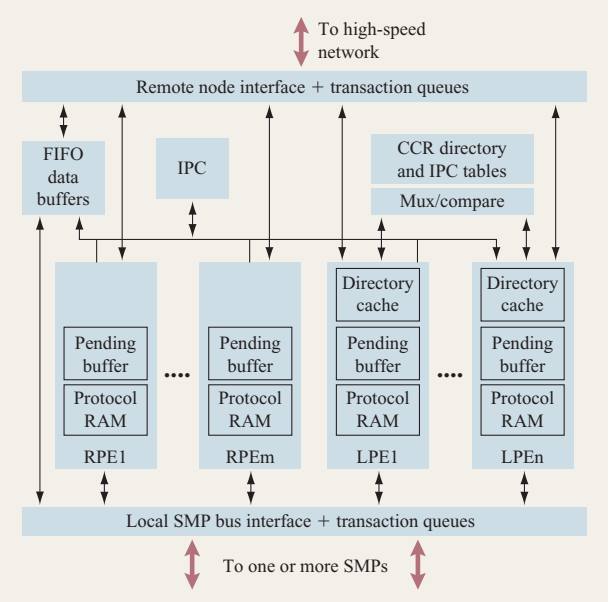

Figure 2

The Everest architecture.

communication channels through a trusted kernel agent. Once established, send/receive (send/recv) or remote direct memory access (RDMA)-type transfers can be initiated in user space. The user application creates lists of instructions in its own address space and then signals the IPC facility that it has work for the IPC to do. The IPC facility fetches the instructions from memory, performs the requested communication operations asynchronously, and reports status back to the user when communication is complete.

The tables used by the IPC for maintaining channel state and an address translation cache are located in the same memory as the CCR directory. Both the CCR and the IPC tables demand high-bandwidth access to a relatively large amount of memory, which may best be provided by embedded on-chip DRAM. External double-data-rate SDRAM is a practical alternative.

Multiple protocol engines

A protocol engine in Everest is a self-contained protocol processing unit that handles protocol activities independently of other PEs and shares as little resource as possible with another PE. The RPEs do not access the CCR directory. The LPEs may share the same CCR directory, but each of them has associated with it a large directory cache which minimizes the contention at the CCR directory due to the LPEs. Each PE also has its own exclusive pending buffer (PB), which is used as a scratch pad for protocol operations in progress.

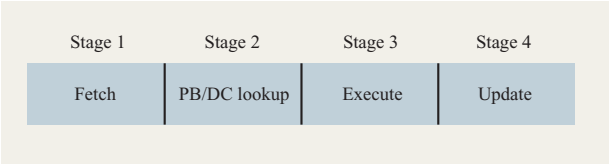

Figure 3

Protocol engine pipeline.

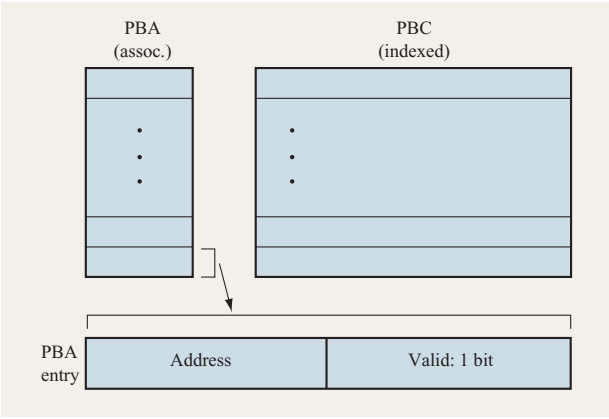

Figure 4

Split pending buffer.

Each of the multiple RPEs or multiple LPEs handles a non-overlapping region of physical memory. The multiple PEs can be assigned to interleaved memory regions by using address bits at any position in the physical address. However, low-order memory interleaving assignment of the PEs intuitively results in less contention, as consecutive memory accesses go to different PEs. The simulation results presented here use low-order memory interleaving. If there are two LPEs, for example, one of them handles the odd memory lines and the other handles the even memory lines.

Pipelined protocol handling

The operations that process a coherence transaction, such as arbitration among incoming queues, pending buffer and directory lookup, and directory updates, would take several cycles if performed in a single pipeline stage. Protocol processing operations can be broken into low-latency pipeline stages to increase coherence-controller throughput. The Everest pipeline consists of four stages, as shown in **Figure 3**. In the first stage, a transaction is fetched from an input queue and pre-decoded to determine whether it is a coherence request or response. In the second stage, the directory and pending buffer are read. If the transaction is a request, lookups are performed in both the directory cache and the pending buffer. However, if the transaction is a response, only the pending buffer is read. In the third stage, the transaction plus the results of the directory cache and pending buffer lookup are presented to a protocol-handling unit for execution (in the split request–response case, there are multiple protocol-handling units in each protocol engine). In the fourth stage, the pending buffer and directory cache are updated, and one or more new transactions are issued to the transport layer of the appropriate output network port.

The directory cache plays a central role in this decision space for the Everest architecture. The deeper the pipeline and the smaller the directory cache, the higher the controller clock rate. However, this is generally achieved at the expense of lower directory cache hit rates and longer transaction latencies. Directory cache capacity and hit rates can be increased by increasing the cycle time, but this increases the occupancy of the entire controller. Consequently, we restrict our consideration to single-cycle directory caches in Everest.

Split request–response handling

Within each PE, two independent protocol-handling units can be used concurrently, one for protocol request transactions and one for response transactions. The motivation for separating request and response handling arises from the observation that they access the pending buffer differently. When a request is processed, an associative lookup of the pending buffer is needed for collision[1] detection, and a pending buffer[2] entry is allocated if no collision is detected. The index of the allocated pending buffer entry is assigned to all downstream transactions generated as a result of processing the request. The transient directory state is also saved in the allocated pending buffer entry. Each response carries a pending buffer index that is used to retrieve the content of the corresponding pending buffer entry when it arrives back at the coherence controller from which the corresponding request was generated. No collision detection is necessary in the case of responses, since any possible collision would have been resolved when the corresponding request first came in.

Noting this distinction, we separate the pending buffer into associative and indexed halves, as illustrated in **Figure 4**. The associative half (PBA, where A stands for address)

[1] Collision occurs when an incoming request to a coherence controller is for a cache line that has another request associated with it still in progress in the same coherence controller.
[2] A pending buffer holds transient state and other information associated with a request for a cache line to a coherence controller for as long as the request is considered still in progress from the point of view of the coherence controller, according to the cache coherence protocol.

contains the address of the transaction in progress and a valid bit. The indexed half (PBC, where C stands for content) maintains the state of the pending transaction. Supporting multiple ports on the non-associative PBC portion is relatively easy and allows simultaneous accesses by request and response handlers. The PBC is multi-ported, with two read ports and two write ports: one read–write pair for the request stream and another pair for the response stream. The PBA is single-ported and supports associative lookup for requests.

Figure 5 shows a local protocol engine with split request–response handling. There are two request queues on the input and two on the output of the protocol engine. One is for transactions that travel for one "hop" in the network and the first requests of two-hop transactions. The other is for second requests of two-hop transactions. Second-hop requests have priority over first-hop requests in the case of multi-hop transactions. This is done for the purpose of avoiding common deadlock conditions that can arise if first-hop requests are allowed to block second-hop requests. Note that the Everest NUMA protocol assumes a network that provides a minimum of three virtual channels, one for responses and two for requests. Virtual channels use the same physical network to emulate multiple separate networks, thus preventing traffic on one virtual channel from blocking traffic on other virtual channels indefinitely. These channels correspond to the input and output queues shown in Figure 5.

The fetch unit retrieves a request or a response and dispatches it to the lookup unit. If it is a request, the lookup unit launches reads to the PBA and the directory cache. If there is a hit in the directory cache and no collision in the pending buffer, the transaction is forwarded to the request protocol-handling unit for execution and an entry in the pending buffer is allocated. If there is a collision in the pending buffer, the content of the existing entry is forwarded to the request protocol-handling unit to handle the collision.

If the directory cache misses (and there is no collision), the content of the directory cache set that missed is passed along to the directory cache controller, which determines whether an eviction is required. If so, the directory controller launches a writeback to the directory. The request protocol-handling unit is notified of the directory cache miss and responds by enqueuing the transaction in the sleep queue. When the response from the directory is received, the directory controller reactivates the transaction in the sleep queue. The fetch unit will eventually fetch the transaction from the sleep queue and reprocess it. The request protocol-handling unit processes the transaction and updates the pending buffer. It also generates new request(s) and/or response(s), which it inserts into the output queues.

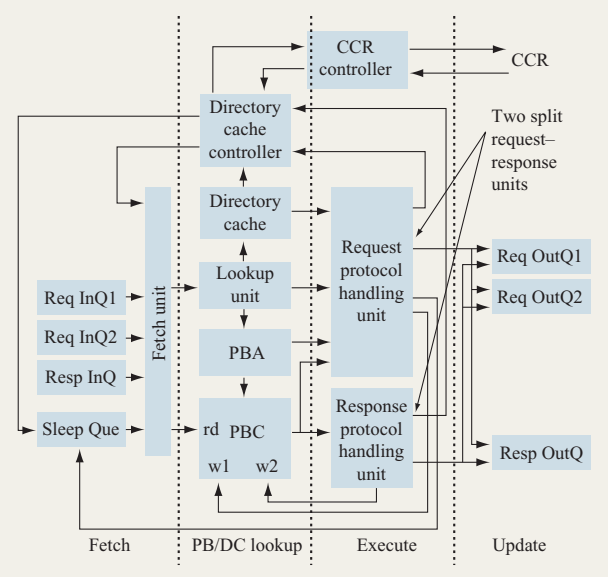

Figure 5

Pipelined, split request–response local protocol engine.

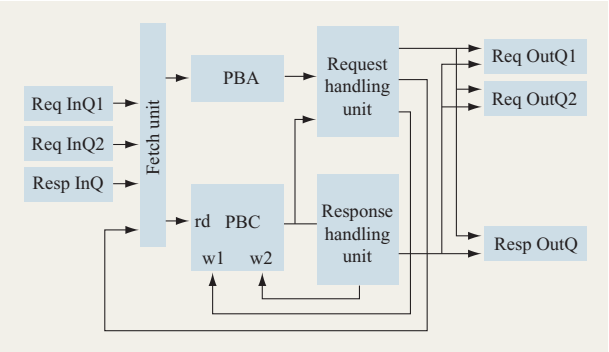

Figure 6

Pipelined, split request–response remote protocol engine.

Response processing is somewhat less involved than request processing. As noted, in the lookup phase the PBC is read instead of the PBA. Since there is no directory cache lookup, the transaction is passed deterministically to the response protocol-handling unit for execution. The response protocol-handling unit may generate new responses, which it launches by inserting them into output queues. It also updates the PBC on a dedicated write port and schedules a directory cache update, if applicable, through the directory controller.

Remote protocol engines (**Figure 6**) are very similar to local protocol engines, except that they do not contain a

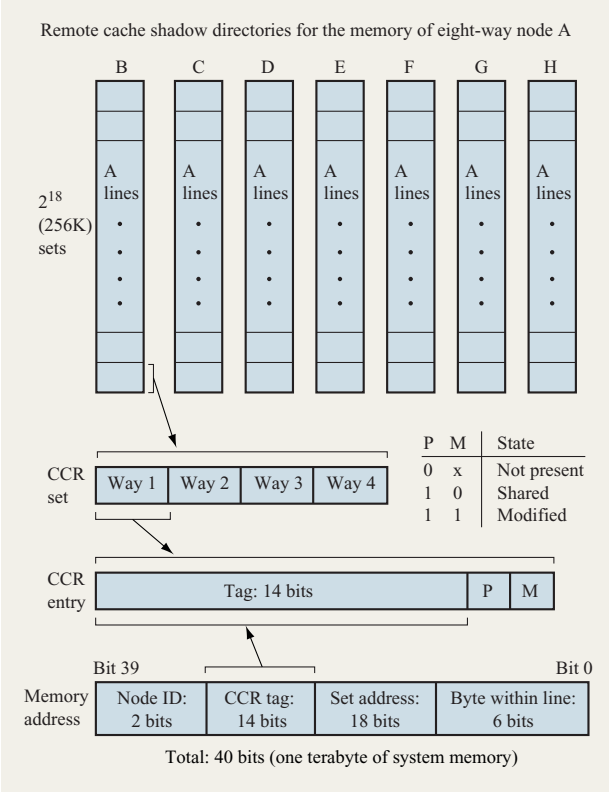

Figure 7

Organization of the CCR directory.

directory cache or a sleep queue. Consequently, they operate much more deterministically than local protocol engines. Also, the coherence protocol is very different for remote memory transactions and local memory transactions. Therefore, separate local and remote protocol engines not only increase the degree of parallelism in the controller, but also simplify protocol engine design.

CCR directory

The complete and concise remote (CCR) directory in Everest maintains state information on the memory lines belonging to the local home memory that are cached in the remote nodes. This is done by keeping a shadow of each shared-cache directory or remote-cache directory in the system [except for the shared cache(s) in the local node] in the CCR directory of the local node. As an example, the CCR directory in a 64-way system built using eight-way nodes per coherence controller consists of seven shadow directories in each coherence controller. **Figure 7** shows the organization of the CCR directory for this configuration. The shared cache or the remote cache in each SMP node is assumed to be 64MB, four-way set-associative with 64-byte lines. Each shared or remote cache has 256K sets.

A shadow in the CCR directory also contains 256K sets, each set containing state bits for four cache lines. Even though a shadow has enough space to keep the state information on *all* of the lines in the remote cache it represents, it keeps the state information on *only the lines in the remote cache that belong to the local home memory*. For example, in Figure 7 the shadow directory C contains the state bits for the lines belonging to home memory A that are currently in remote cache C. However, the lines in remote cache C belonging to memories C through H are not represented in the shadow of C in the CCR directory of node A.

In order to maintain an exact shadow of the remote caches, the CCR directory requires the remote-cache controller to inform the home-coherence controller containing the shadow when the remote cache evicts a line corresponding to the memory of that home node. Since the degree of associativity of the shadow in the CCR directory is the same as the degree of associativity of the corresponding remote cache, and the CCR directory is informed about the evictions from the remote cache, it is guaranteed that a CCR set in the shadow will always have a slot available when the remote cache has to allocate a new line in that set. Therefore, a directory entry is never evicted from the CCR unless the line is also being evicted in the corresponding remote cache.

The division of the address fields for accessing a CCR directory assuming a 40-bit system-wide physical address is shown in Figure 7. Each entry in a shadow keeps a 14-bit tag and two state bits. The presence bit P tells whether the line is present in the corresponding remote cache, and the modified bit M tells whether the line is modified in that cache. The P bit in all of the CCR directory entries is initialized to 0 at system reset. The states of a line in the corresponding remote cache interpreted from the P and M bits are shown in the table in Figure 7.

CCR directory vs. sparse and full-map directories

By keeping the information on the exact number of remotely cached lines, the CCR directory provides a dynamic full-map directory [5] of currently shared lines. Consequently, the CCR directory offers all the advantages of a full-map directory. In contrast, a sparse directory keeps the state information on only a subset of the memory lines that could have been remotely cached in a full-map directory scheme, which leads to inferior performance and a complex protocol compared to a full-map directory.

It is theoretically possible to modify the original sparse directory scheme to keep information equivalent to the CCR directory. The enhanced sparse directory would also receive evict information from the remote caches and

would have sufficient space to shadow the remote caches. In that case, the sparse directory would have to have an associativity of $n \times w$ in a system with n remote caches which are w-way set-associative, and would require a huge multiplexor to obtain the presence-bit vector when there is a hit. On the other hand, the CCR directory contains n w-way shadows that would require only small muxes to obtain the directory information. Gathering the presence-bit information from the n possible hits is a simple logic operation. Thus, the CCR directory would avoid the extra latency penalty of a large mux of the enhanced sparse directory. **Figure 8** compares the two schemes for the example system of Figure 7.

The amounts of memory required to implement various directory schemes for the same example system as in Figure 7 are compared in **Table 1**, which makes the following assumptions. A conventional four-way sparse directory would have to have 256K to 2M sets to obtain a good hit rate. Each entry in the conventional sparse directory or the enhanced sparse directory would require about two bytes for tag and state and one byte for presence vector (a total of three bytes per entry). Each of the CCR directory entries would require two bytes for tag and state. A full-map directory entry requires 1.25 bytes to store the presence vector and state. For this example configuration, the CCR directory is two orders of magnitude smaller than the full-map directory and is comparable in size to the conventional and enhanced sparse directories.

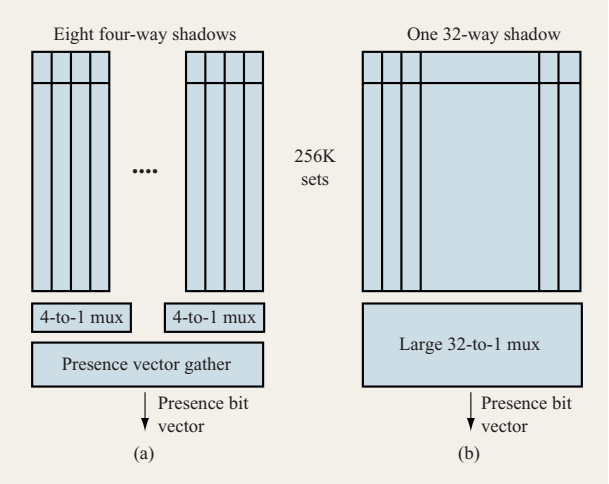

Figure 8

Comparing CCR and sparse directory designs: (a) CCR directory; (b) enhanced sparse directory.

Table 1 Memory requirement for various directory schemes.

	Conventional sparse	*Enhanced sparse*	*CCR*	*Full map*
Directory size	3–24 MB	24 MB	16 MB	2500 MB

Combining property of the CCR directory

Theoretically, a system can have only one CCR directory, and to maintain coherence among all the nodes, every single CCR directory has the same size as the total remote-cache directory in the system. If there is only one SMP (and hence one remote cache) per coherence controller, the system will have as many CCR directories as the number of remote caches. However, if there are two SMPs per coherence controller, the number of coherence controllers, and hence the number of CCR directories, is half the number of remote caches. In other words, a single CCR is capable of representing multiple memory nodes at the home coherence controller. Thus, the CCR directory size per SMP is inversely proportional to the number of SMPs connected to the coherence controller.

Directory cache

Everest uses one directory cache per LPE to provide high-throughput directory access. The use of a directory cache has been studied in [6]. In this paper we present the design of a directory cache specific to the CCR directory design, which is not discussed in [6]. The state information on a cache line in the directory cache is equivalent to the state information on that cache line in the CCR directory, but the formats of the state information are different in each case. The directory cache keeps the state information in a more compact form than the CCR. The directory cache in each LPE has a one-cycle lookup time and is as large as can be accommodated on the controller chip without exceeding the one-cycle access time. Although we do not use a second-level directory cache in our study, a second-level directory cache may prove beneficial if there is space on the controller chip. **Figure 9** shows how an incoming memory address is used to access a directory-cache entry for 8K entries in a four-way set-associative directory cache with two entries per directory-cache line.

An entry in the directory cache contains two sets of presence vectors, two valid (V) bits, two modified (M) bits, and two change (C) bits corresponding to the two cache lines it represents. The seven presence bits correspond to the seven remote caches. The V bit determines whether the information on the corresponding cache line in the directory cache is valid. The V bits of all directory-cache entries are initialized to 0 at reset. The M bit indicates whether the line is modified in one of the remote caches. The C bit indicates whether the directory entry has been changed since being cached in the directory cache.

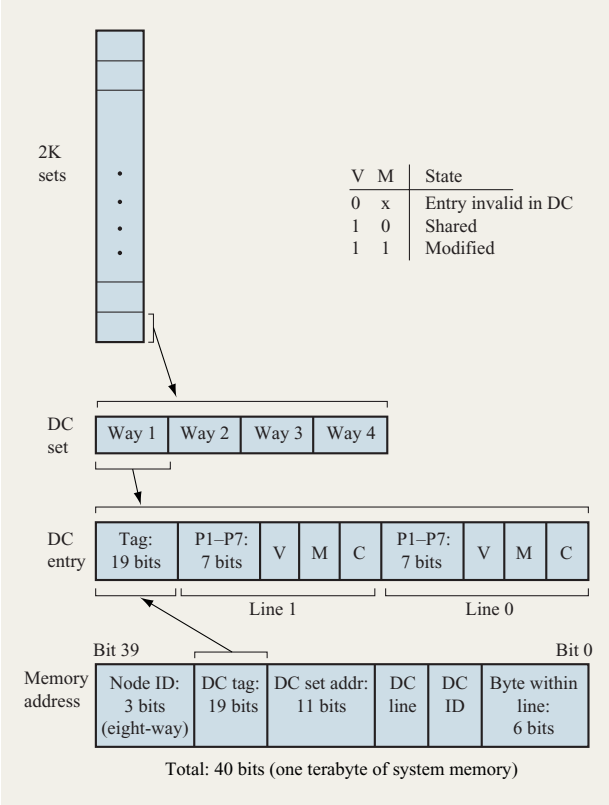

Figure 9

Directory cache organization.

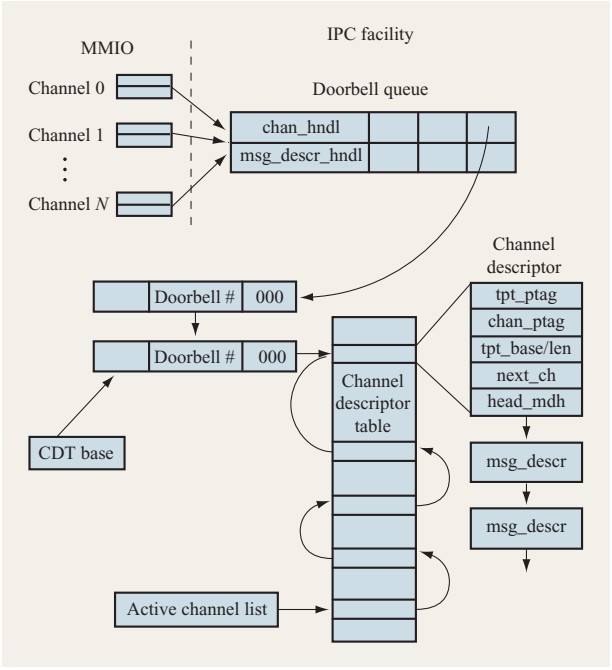

Figure 10

Doorbells and channel descriptors.

Interpartition communication facility

The IPC facility in Everest is designed to support direct user-space access to hardware in a secure manner, thus providing low-latency communication. High bandwidth is an independent goal, which is supported by the tight coupling of the IPC to the coherence protocol engines. The IPC supports a secure, connection-oriented type of communication. A user establishes interpartition channels through a trusted kernel agent. Once established, send/recv-type transfers can be initiated in user space over the channels on scatter/gather buffer lists. The IPC has an address translation capability that provides the user with a means of specifying buffer addresses in its own virtual address space. Transfers are cache-coherent within each partition. Delivery is reliable and in order. Failures do not propagate between partitions.

The IPC supports multiple protected bidirectional communication channels. Each channel consists of two endpoints, and an endpoint consists of a pinned channel buffer and a doorbell. A channel buffer is a memory area that contains the data a user wishes to transfer, which can be read or written by the IPC and the user process. The IPC also requires a pinned memory area for message descriptors, through which a user describes the transfers it wishes the IPC to perform on its behalf. The channel buffers and descriptor area are typically not contiguous in real address space. A doorbell is an address in memory-mapped I/O space where the user signals the IPC that it has work for it to do on a particular channel. Typically, an IPC facility on a node is used to support a channel with an endpoint associated with memory at that node. However, this is only a performance optimization.

Channel buffers are specified to the IPC in a pseudovirtual address form. This address form is generated for the user by a trusted kernel agent that takes as input a user-space address of a virtually contiguous memory area. The kernel agent determines the corresponding physical addresses of the associated memory pages, then pins and maps them into a translation-and-protection table (TPT) in main memory accessible to the IPC. Each entry in the TPT maps one page. The pseudovirtual address handed back to the user consists of an index into the TPT and an offset in the page mapped by a TPT entry. Multiple contiguous TPT entries are used to map multi-page buffers.

Figure 10 illustrates how the doorbell mechanism works. Each channel endpoint created by the kernel agent for a user process consumes one OS page table entry (i.e., one page of virtual address space). This ensures that one user

will never be able to ring another user's doorbell. To simplify the task of identifying a memory-mapped I/O (MMIO) to a doorbell, the pages are restricted to a contiguous physical address range. A simple range check is then adequate. Once the MMIO is identified as a doorbell ring, both the doorbell address and the data are stored in a doorbell queue if space is available. The address is interpreted by the IPC as a *channel handle* and the data fields as a *message descriptor handle*. If the queue is full when software issues a write to a doorbell, the command will not be enqueued, and software must explicitly retry the write. To ascertain whether the command was accepted into the command queue, software reads the doorbell location after the write. The read returns the last descriptor handle enqueued in the doorbell queue. If the handle returned does not match the one written, software assumes that it was not accepted into the doorbell queue. The IPC drops the command if the queue is full or if the write that submitted the command is followed by any doorbell access other than a read to the same doorbell address. In this case, an invalid descriptor handle is returned.

When the IPC fetches a doorbell from the queue, it uses the channel handle to locate a channel descriptor in a channel attribute table (CHAT). To locate a channel descriptor in the channel attribute table, the IPC substitutes the upper bits of the CHAT base address for the upper bits of the doorbell address, as shown in Figure 10. A channel descriptor contains the base address and length of an address translation table (tpt_base/len) associated with the channel, a memory protection tag (tpt_ptag) that facilitates secure sharing of TPT tables by multiple channels, a channel protection tag (chan_ptag) that facilitates secure sharing of a single destination-channel endpoint by multiple source-channel endpoints, a descriptor handle that points to the head of a list of descriptors associated with the channel, and a chaining pointer (next_ch) used to chain channels with pending work into an active channel list. The active channel list is used by the IPC to allocate network bandwidth to channels. Each channel is given a quantum of bandwidth on the network, as the IPC traverses the active channel list. When all of the transfers specified in the message descriptor list of a channel are complete, the channel is deleted from the active list.

Once the IPC has a copy of the channel descriptor, it can locate the address translation table and the message descriptor list associated with the doorbell ring. For this, the IPC uses the message descriptor handle provided in the doorbell ring or from the channel descriptor (head_mdh). When processing a doorbell, the IPC first checks to see whether the channel is already in the active list. If it is, the channel descriptor need not be enqueued there, and the IPC uses the head message descriptor

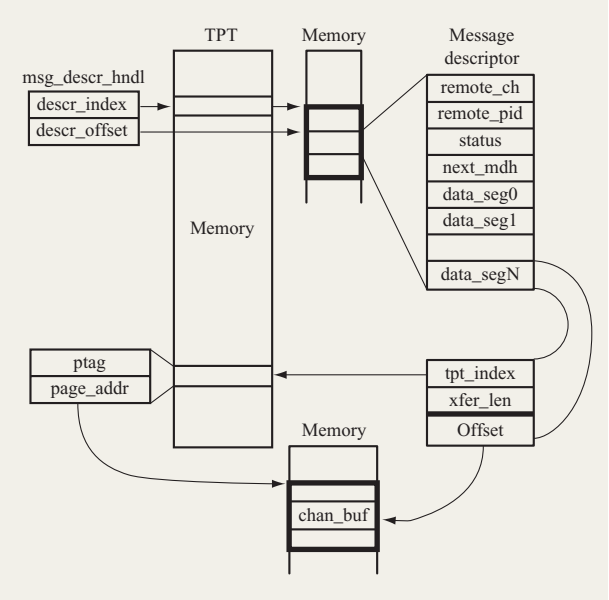

Figure 11

DMA descriptor and channel buffer lookup.

handle from the channel descriptor to locate the first message descriptor in the list. If the channel descriptor is not in the active list, the IPC inserts it there and uses the message descriptor handle from the doorbell to locate the head of the message descriptor list. If, when the channel's bandwidth quantum is consumed, there are still pending message descriptors in the list, the next one to be processed on the next pass around the active channel list is recorded in the channel descriptor (head_mdh) as the new head of the descriptor list.

Figure 11 illustrates how the IPC identifies the location of a message descriptor and its associated channel buffer in memory. A message descriptor handle consists of two parts. The first part (descr_index) is an index into the TPT table. This gives the location of the physical memory page where the message descriptor is located. The second part (descr_offset) indicates where in that page the descriptor is located. A message descriptor specifies the remote partition (remote_pid), the channel endpoint in the remote partition (remote_ch), transfer status (status), a scatter/gather list of data segments (data_seg), and a chaining pointer (next_mdh) that the user uses to create message descriptor lists. The remote-channel endpoint field is used by the kernel agent to connect channel endpoints in two different partitions to form a connection-oriented channel. The status field is used by the IPC to report status back to the user on completion of the transfer requested in the message descriptor. The number of data segments in a descriptor varies by channel, but is

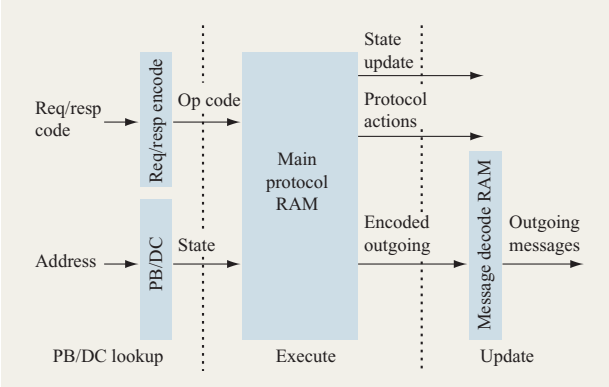

Figure 12

Three pipeline stages of protocol RAM.

fixed within a channel. A data segment consists of three parts: an index into the TPT (tpt_index) indicating the physical memory page where the start of the channel buffer is located, the offset in that page of the start of the channel buffer, and the size of the segment (xfer_len). Multiple contiguous TPT entries (up to one page) may be indicated to span the transfer length of any given segment. A TPT entry consists of two parts: a protection tag (ptag) and the physical memory page address (page_addr). For each memory reference, the memory protection tag (tpt_tag) in the channel descriptor is compared against the one in the TPT entry. If they do not match, an error is reported and the transfer is aborted.

To summarize: A send operation on an established channel begins with a user process creating a list of descriptors in a preregistered memory area. The user then rings his channel doorbell (i.e., an MMIO write to the location associated with the channel). The IPC ascertains the location of the corresponding channel descriptor in IPC memory, which indicates the location in main memory of an address translation table (TPT) and possibly the head of a message descriptor list. The IPC translates the channel buffer addresses specified in the message descriptors, performs a direct memory access (DMA) to obtain their content from memory, packages it into network packets, and transmits the packets to the remote channel endpoint specified in the message descriptor (remote_pid, remote_ch). The channel protection tag (chan_ptag) is sent along in each packet and checked against the channel protection tag registered at the remote endpoint. A protection tag violation aborts the transfer. When all of the packets associated with a message have been successfully received, the remote-channel endpoint sends a notification back to the sender. When the sending endpoint receives this notification, it reports status back to the user in the corresponding message descriptor. Not shown are controls in the message descriptor that can be set by the user to request an interrupt upon completion. Completion interrupts can be specified at either channel endpoint.

Figure 11 illustrates send-side processing. An analogous process is performed for receive-side processing. The remote channel ID and channel protection tag are sent in message packets to the destination node and are used by the IPC there to perform secure CHAT lookups. From that point, the procedure for identifying receive DMA descriptors and TPT entries is the same as for send-side processing, except that it is performed on the CHAT and TPT in the remote partition.

Note that only send/recv-style communication is supported by the IPC. Remote DMA (RDMA) is a desirable and rather straightforward extension. Essentially, an RDMA message descriptor type must be defined with data segments that specify both local- and remote-channel buffers (i.e., local and remote tpt_index and offset). These are sent along in the message packets and are used by the remote IPC to locate the channel buffer, rather than looking up a receive-message descriptor.

Protocol RAM

The Everest architecture employs a flexible, high-throughput implementation of the protocol finite-state machine (FSM) using a programmable protocol RAM. The operation of the protocol RAM spans three pipeline stages, as shown in **Figure 12**. The first stage involves a lookup in the request–response-encode RAM modules, which occurs in the PB/DC lookup stage of the Everest pipeline. These lookups produce an index. In the second stage, the index produced in the first stage is used for lookup in the main protocol RAM, resulting in a vector of encoded protocol outputs, such as outgoing messages and changes in state. In the third stage, the outgoing message component of the protocol output is decoded, using the outgoing-message-decode RAM, into messages compatible with the CC interfaces.

All three types of programmable protocol RAM—request–response encode, protocol, and outgoing-message decode—are direct-mapped with one-cycle access time. A typical size for the request–response-encode RAM and the outgoing-message-decode RAM is 32 entries with 8 bits each. The size of the main protocol RAM is dependent on the number of index bits used for accessing it. A typical range for the index size is 10–12 bits, resulting in protocol RAM size of 1–4K entries. Each entry is typically 40 bits, representing an encoding of protocol actions.

In comparison to fully hardwired protocol FSM, the presented implementation offers flexibility for postproduction changes to the coherence protocol without sacrificing performance, since the occupancy

Table 2 Benchmark types and datasets.

Application	*Type*	*Problem size*
FFT	FFT computation	256K complex doubles
Water-Nsq	Study of forces and potentials of water molecules in a 3D grid	512 molecules
TPC-C	Benchmark of on-line transaction processing workload	500MB database

of the protocol FSM remains at one cycle, with negligible (if any) increase in latency. In comparison to protocol processors, the presented implementation offers substantial advantages in performance, with an order-of-magnitude improvement in occupancy and latency without sacrificing flexibility.

3. Simulation results

In this section we present simulation results of the effect on system performance of three of the main features of the Everest architecture. The three throughput-enhancing features are multiple PEs, split request–response PEs, and pipelining. We investigate the synergy among these architectural features and their effect on CC performance.

Base system parameters

The base simulated system includes eight (four for commercial applications) 150-MHz CCs connected through a fast switch with 90-ns no-contention latency [7]. Attached to each CC is an SMP that includes four 600-MHz PowerPC processors with 32KB L1 and 1MB L2 four-way set-associative LRU caches with 64B cache lines, a 16MB remote cache, interleaved main memory, and a 150-MHz pipelined, split-transaction SMP bus. The base architecture includes a CC without any of the features under study. We use its performance as a comparison point for the performance of other configurations with more features. In our experiments we vary the number of SMP nodes connected to each CC and the total number of CCs in the system (while keeping the total number of processors fixed) to vary the load on the CCs.

Experimental methodology

We use execution-driven simulation based on a PowerPC version of the Augmint simulation tool kit [9]. Our simulator includes detailed contention models for SMP buses, memory controllers, interleaved memory banks, protocol engine pipelines, directory memory, and external point contention for the interconnection network.

We use both commercial and scientific benchmarks in the performance experiments. We select three applications out of the eight that we evaluated in our paper on high-throughput coherence controllers [7]: a commercial workload, a communication-intensive scientific application, and a computation-intensive scientific application. The commercial workload is a trace of 400 million instructions of TPC-C** [10] running on IBM DB2* [11] using 500MB memory-resident databases running on 32-processor systems. We collected the commercial trace using the COMPASS [12] environment. The scientific benchmarks are FFT and Water-Nsq from the SPLASH-2 suite [13] of parallel applications running on 64-processor systems. The dataset sizes we use for the SPLASH-2 applications are as given in **Table 2**. All benchmarks are written in C and compiled using the IBM XLC C compiler with optimization level O2. All experimental results reported in this paper are for the parallel phase only of these applications.

Performance results

To capture the effect of each feature on system performance, we ran experiments on the base system configuration and then added the throughput-enhancing features to the CCs to investigate their effect on system performance.

Figures 13, 14, and 15 show the performance impact of the features under study on various system configurations. In these experiments we keep the total number of processors in the system constant and vary the number of SMPs connected to each CC. For the FFT and Water-Nsq we use configurations of four CCs with four SMPs each, eight CCs with two SMPs each, and sixteen CCs with one SMP each. For the commercial workload we use a configuration of two CCs with four SMPs each, four CCs with two SMPs each, and eight CCs with one SMP each. In all of these cases, each SMP includes four processors.

Figure 13 shows the improvement in execution time of FFT for systems with various combinations of CC architectural features over a system without any features. The leftmost group of seven bars shows the improvement on a system with one SMP connected to each CC. The three leftmost bars of this group show the improvement when one of the three throughput-enhancing features is applied to the CCs. The execution time is reduced on

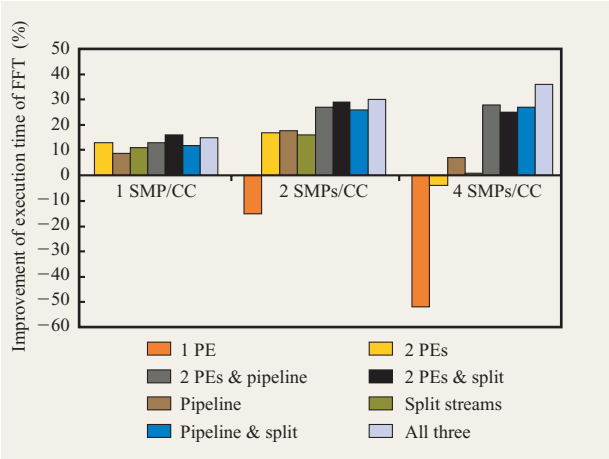

Figure 13

Improvement of execution time of FFT, normalized to the base configuration of one SMP/CC without throughput enhancement.

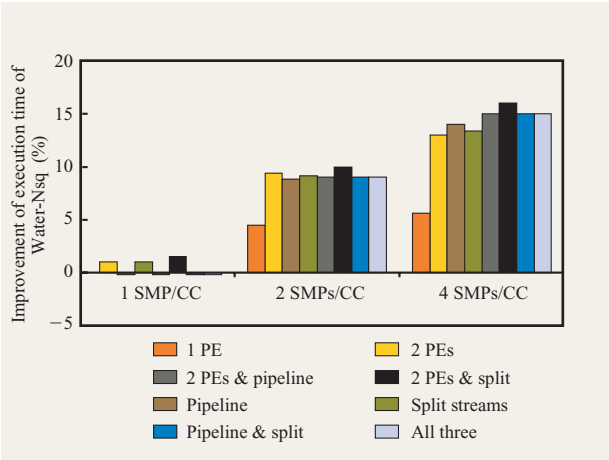

Figure 14

Improvement of execution time of Water-Nsq, normalized to the base configuration of one SMP/CC without throughput enhancement.

average by 10%. The next set of three bars shows the improvement in execution time when two of three features are used in the CCs. The rightmost bar shows the improvement when all three features are utilized in the CCs. Even though FFT is a communication-intensive application, applying two or three of the features does not offer significantly better improvement over the CCs with a single feature, because there is not much contention at the CCs for the configuration with one SMP per CC.

The collection of bars in the center of Figure 13 shows the improvement in execution time as we halve the number of CCs in the system by connecting two SMPs to each CC. The leftmost bar shows that the execution time of the system with no throughput-enhancing features underperforms the base system with one SMP per CC, because there are fewer CCs in the system for the same number of concurrent coherence transactions. For high-communication applications such as FFT, even though the portion of local data is greater in the systems with fewer CCs, the high contention at the CCs negates any benefit of the extra data locality. However, applying any of the parallel features allows the system to maintain its performance, but with fewer CCs. For instance, the second, third, and fourth bars show that when one of the three features is applied, the reduction in execution time is even better than for the configurations with one SMP per CC and one throughput-enhancing feature per CC. The fifth, sixth, and seventh bars show that the execution time can be further reduced by using two throughput-enhancing features, because contention is higher in the configurations with two SMPs per CC than in those with one SMP per CC. The last bar shows that using all three features does not offer any significant reduction in execution time because much of the contention is already greatly reduced with only two features.

The rightmost group of bars in Figure 13 shows the impact of the three features on performance in configurations with the number of CCs a quarter of the base configuration. The first bar shows that when no features are used, the significant contention at the CC reduces the performance to about half that of the base configuration. When a single feature is added to the CCs, the second to fourth bars show that the performance is comparable to that of the base configuration, even with a quarter of the number of CCs in the system. The next three bars show that when two features are used in the CCs, the execution time is reduced by about 27%. The last bar indicates that using all three features continues to provide benefits.

Figure 14 illustrates the impact of the three architectural features on Water-Nsq, which is a low-communication application. The left group of bars shows that applying a single feature to the CCs offers little improvement in execution time. Execution time for the pipelined PE can grow if there are not enough overlapping coherence transactions at the CCs to offset the extra latency of the pipelined PE (three cycles for pipelined vs. two cycles for non-pipelined). The second, fourth, sixth, and seventh bars illustrate the negative effect of pipelining on execution times for low-communication applications.

The second and third groups of bars show that, even with more SMPs per CC, there are not enough concurrent coherence activities to increase the contention at the CCs to a level that can negate the benefit of data locality. As

shown in the first bar of each of the two groups, the execution times of Water-Nsq actually improve over those of the base case because there are fewer accesses to remote nodes as the number of CCs is reduced. In fact, many requests are satisfied by caches on one of the two buses connected to the local CC or by the local memory module. The rest of the bars indicate that adding more features does not offer further benefit after the incorporation of one feature in the CCs, because contention at the CC is no longer significant.

Figure 15 shows the impact of the throughput-enhancing features on the performance of TPC-C. The chart shows a trend similar to that of Figure 14, although the degree of reduction in execution time is greater for TPC-C because it has higher-coherence traffic than Water-Nsq.

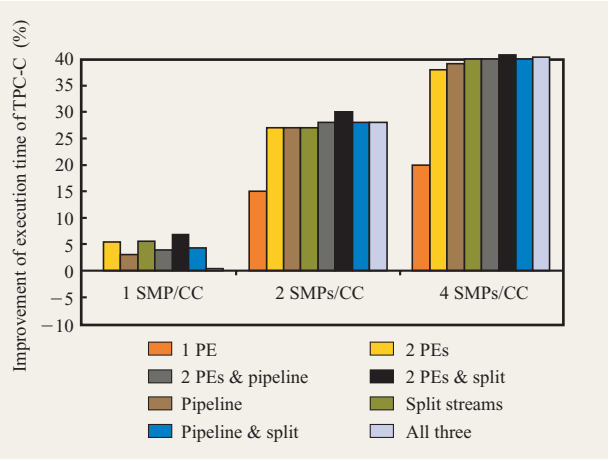

Figure 15

Improvement of execution time of TPC-C, normalized to the base configuration of one SMP/CC without throughput enhancement.

4. Related work

Everest provides the capability to process multiple requests and responses per coherence-controller cycle. In this section we discuss the throughput capabilities of other systems.

The Magic coherence controller of the Stanford FLASH [14] system uses a custom protocol processor to process protocol handlers. Each handler typically takes tens of cycles. Since each Magic chip contains only one protocol-processor core, it can process only one request or response at a time and can employ neither aggressive pipelining nor parallelism between requests and responses. Accordingly, Magic throughput cannot exceed one handler in tens of cycles.

The Sequent Sting [2] and Sun s3,mp [15] systems employ two microsequencers for processing protocol handlers. Each handler takes tens of cycles. Even after doubling throughput at best by employing two microsequencers, the throughput of these systems is one protocol handler in tens of cycles.

Systems that employ off-chip protocol processors such as Typhoon [16] can achieve only significantly lower throughput, even when employing multiple protocol processors.

Hardware-based coherence controllers in systems such as Dash [5] and Alewife [17] can achieve better performance than the above-mentioned system, but not close to that of Everest, since they do not employ any kind of parallelism.

5. Conclusions

In this paper we describe the Everest architecture. Everest achieves high-throughput protocol processing by providing parallelism in three dimensions—parallel protocol engines, pipelined execution, and split request–response handling. Everest uses a new directory design called the complete and concise remote (CCR) directory, which uses roughly the same amount of memory as a sparse directory but retains the benefits of a full-map directory. Everest also supports system partitioning and provides a tightly integrated facility for secure, high-performance communication between partitions.

The Everest design space was explored using both technical and commercial workloads. The results show that using parallelism in any of the three dimensions provides similar benefits. For high-communication applications such as FFT, combinations of the three dimensions provide additional improvements. However, applying more aggressive parallelism yields diminishing returns when the throughput of the CC is more than adequate.

We also examine the effect of multiple SMPs connected to each CC. The performance benefits become more prominent as the number of processors handled by the coherence controller increases. Using combinations of the three dimensions continues to bring significant performance gain for high-communication applications. With multiple SMPs sharing a single CC, the numbers of CC nodes in the system and the interconnection network are smaller. This can reduce the cost of the system, with a small increase in CC complexity as multiple PEs, pipeline, and split request–response handling are employed. The results on relative performance gain while adding incremental parallelism in one or more dimensions can be used by designers to determine performance/complexity tradeoffs on specific Everest implementations.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intel Corporation or Transaction Processing Performance Council.

References

1. J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 241–251.
2. T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pp. 308–317.
3. W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke, "The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 98–107.
4. W.-D. Weber, "Scalable Directories for Cache-Coherent Shared Memory Multiprocessors," Ph.D. Thesis, Stanford University, CA, 1993.
5. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH Multiprocessor," *IEEE Computer* **25,** 63–79 (March 1992).
6. M. M. Michael and A. Nanda, "Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors," *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, *HPCA-5*, IEEE, 1999, pp. 142–151.
7. A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph, "High-Throughput Coherence Controllers," *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, *HPCA-6*, IEEE, 2000, pp. 284–289.
8. M. M. Michael, A. K. Nanda, B.-H. Lim, and M. Scott, "Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 219–228.
9. A.-T. Nguyen, M. M. Michael, A. Sharma, and J. Torrellas, "The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures," *Proceedings of the International Conference on Computer Design*, October 1996, pp. 486–490.
10. Transaction Processing Performance Council, *TPC Benchmark D and TPC Benchmark C: Standard Specifications; http://www.tpc.org*.
11. IBM Corporation, *DATABASE 2 Information and Concepts Guide for Common Servers Version 2*, Order No. S20H-4664-00, May 1995.
12. A. K. Nanda, Y. Hu, M. Ohara, C. Benveniste, M. Giampapa, and M. M. Michael, "The Design of COMPASS: An Execution Driven Simulator for Commercial Applications Running on Shared Memory Multiprocessors," *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 503–509.
13. S. C. Woo, M. Ohara, E. Torri, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 24–36.
14. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH multiprocessor," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp. 302–313.
15. A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishi, "The S3.mp Scalable Shared Memory Multiprocessor," *Proceedings of the 1995 International Conference on Parallel Processing,* Vol. 1, August 1995, pp. 1–10.
16. Steven K. Reinhardt, James R. Larus, and David A. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proceedings of the 21st Annual International Symposium on Computer Architecture,* April 1994, pp. 325–336.
17. A. Agarwal, R. Binchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd International Symposium on Computer Architecture,* May 1995, pp. 2–13.

Received September 18, 2000; accepted for publication February 11, 2001

Ashwini K. Nanda *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (ashwini@us.ibm.com)*. Dr. Nanda is a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, where he also manages the Scalable Server Architecture group. Dr. Nanda's research interests include computer architecture and shared-memory system design and performance, with an emphasis on commercial applications. He is currently involved in several research projects at IBM, including MemorIES (Memory Instrumentation and Emulation System), High Throughput Coherence Controllers, and the Watson Commercial Server Performance Laboratory. He is also a member of the architecture and design team for IBM's next-generation NUMA-Q machines.

Anthony-Trung Nguyen *University of Illinois, Urbana–Champaign, 1304 West Springfield Avenue, Urbana, Illinois 61801 (anguyen@cs.uiuc.edu)*. Mr. Nguyen is a Ph.D. candidate in the Computer Science Department at UIUC. He previously had a two-year assignment with the Scalable Systems Group at the IBM Thomas J. Watson Research Center, where he worked on coherence controller architectures for scalable servers. His research interests are computer architecture and performance analysis. Mr. Nguyen received an S.B. degree from the Massachusetts Institute of Technology and an M.Eng. degree from Cornell University.

Maged M. Michael *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (magedm@us.ibm.com)*. Dr. Michael received the Ph.D. degree in computer science from the University of Rochester in 1997. He is currently a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. His research interests include multiprocessor architecture, cache coherence, theory of distributed computing, concurrent algorithms, multiprocessor synchronization, and multiprocessor simulation.

Douglas J. Joseph *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (djoseph@us.ibm.com)*. Dr. Joseph received a Ph.D. degree in computer science from the University of Colorado in 1997. He is currently at the IBM Thomas J. Watson Research Center, where he is involved in high-end scalable architecture. His interests include scalable cache coherence controller and high-performance SAN architectures for next-generation I/O and cluster messaging, communication library and operating system support for next-generation SANs, and dynamic partitioning.