IBM @server

POWER4 System Microarchitecture

Technical White Paper

POWER4 introduces a new microprocessor organized in a system structure that includes new technology to form systems. POWER4 as used in this context refers to not only a chip, but also the structure used to interconnect chips to form systems. In this paper, we describe the processor microarchitecture as well as the interconnection architecture employed to form systems up to a 32-way symmetric multiprocessor.

Joel M. Tendler, Steve Dodson, Steve Fields,
Hung Le, Balaram Sinharoy
IBM Server Group

October 2001

1. Introduction

IBM has a long history of leveraging technology to build high performance, reliable systems. Over the last several years, we have been increasingly applying this expertise to UNIX servers. The IBM @server pSeries 680 continued this heritage with industry leading performance across a broad spectrum of industry standard benchmarks. The same microprocessor is used in the IBM @server iSeries 840. That system is also at the head of the list in a number of industry standard benchmarks. The IBM RS/6000 SP system incorporating the POWER3-II microprocessor is currently used in the world's largest supercomputer at the Lawrence Livermore National Laboratory.

Several years ago, we set out to design a new microprocessor that would insure we would leverage IBM strengths across many different disciplines to deliver a server that would redefine what was meant by the term server. POWER4 is the result. It was developed by over 300 engineers in several IBM development laboratories.

With POWER4, the convergence of iSeries and pSeries microprocessors will reach a new level. POWER4 was designed from the outset to satisfy the needs of both of these systems. Additionally, projected technology improvements drove significant enhancements to system structure. The resulting design leverages technology advances in circuit densities and module packaging to achieve high levels of performance suitable for server types of applications. We have publicly talked about POWER4 for over 2 years prior to its formal announcement. We were confident we would achieve its objectives in terms of performance, schedule, and content. And we have.

In this paper, we first describe what drove the design. We then take a closer look at the components of the resultant systems from a microarchitecture perspective. We start by first discussing the POWER4 chip. One of our major strengths is system expertise and the ability to design multiple parts in a consistent and synergistic manner. In that light, POWER4 cannot be considered only a chip, but rather an architecture of how a set of chips are designed together to realize a system. As such, POWER4 can be considered a technology in its own right. In that light, we discuss how systems are built by interconnecting POWER4 chips to form up to 32-way symmetric multiprocessors (SMPs). The interconnect topology, referred to as a Distributed Switch, is new to the industry. Finally, no system discussion would be complete without some view of the reliability, availability and serviceability (RAS) features and philosophy incorporated into POWER4 systems. The RAS design is pervasive throughout the system and is as much a part of the design as is anything else.

2. Guiding Principles

POWER4 was intended from the outset to be a server. In designing POWER4, a set of principles guided the Engineering team. These included:

- **SMP optimization:** The server must be designed for high throughput, multi-tasking environments. The system must be optimized for SMP operation where a large number of transistors could be used to improve total system performance. Servers of their very nature process disparate information often with multiple processes active concurrently.

- **Full system design approach:** To optimize the system, we started with the full design in mind up front. This marriage with the process technology, packaging, and microarchitecture was designed to allow software to exploit them. The processor core was designed to fit effectively in this environment. We designed the entire system, from the processor, to memory and I/O bridge chips, together. A new high performance processor needs a new subsystem to effectively feed it.

- **Very high frequency design:** To maintain a leadership position, we planned, from the outset, to deliver best-of-breed operating frequencies. We revamped our chip design with new transistor level tools and have transformed complex control logic into regular data flow constructs. Additionally, we have designed the system to permit system balance to be preserved as technology improvements become available allowing even higher processor frequencies to be delivered.

- **Leadership reliability, availability and serviceability:** Servers have evolved to continuous operation. Outages of any kind are not tolerated. We had already begun designing into our systems RAS attributes previously only seen in mainframe systems. With POWER4, we accepted the principle that one does not achieve high levels of reliability by only choosing reliable parts and building error correction codes (ECC) into major arrays, but an approach that eliminated outages and provided redundancy where errors could not be eliminated was required. Where possible, if an error occurred, we worked to transform hard machine stops (checkstops) into synchronous machine interrupts to software to allow it to circumvent problems if it could.

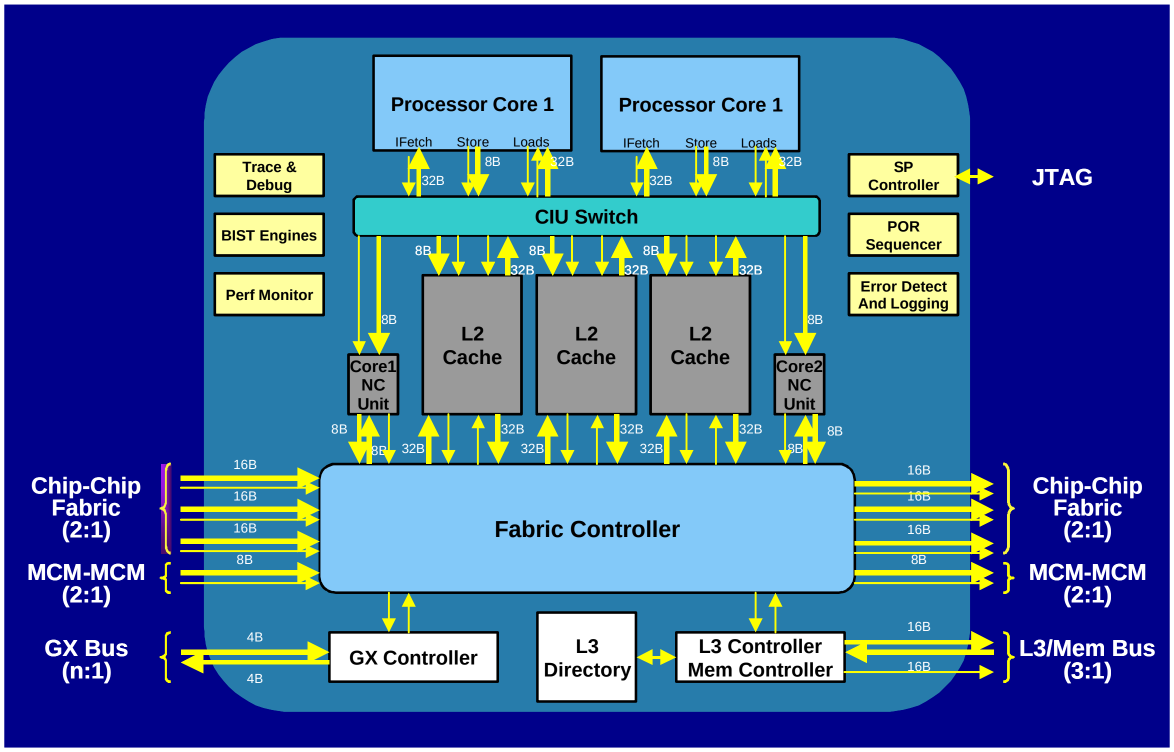
- **Designed to execute both commercial and technical applications supporting both IBM @server iSeries and pSeries systems with high performance:** As we balanced the system, we made sure the design can handle a varied and robust set of workloads. This is especially important as the e-business world evolves and data intensive demands on systems merge with commercial requirements. The need to satisfy high performance computing requirements with its historical high bandwidth demands and commercial requirements with its data sharing and SMP scaling requirements dictated a single design to address both environments. This would also allow us to meet the needs of what became IBM @server pSeries and iSeries with a single design.

- **Maintain binary compatibility for both 32-bit and 64-bit applications with prior PowerPC and PowerPCAS systems**: Several internal IBM task forces in the first half of the 1990s had concluded that the PowerPC architecture did not have any technical impediments to allow it to scale up to significantly higher frequencies with excellent performance. With no technical reason to change, in order to keep our customers software investment in tact, we accepted the absolute requirement of maintaining binary compatibility for both 32-bit and 64-bit applications, from a hardware perspective.

3. POWER4 Chip

The components of the POWER4 chip are shown in Figure 1. The chip has two processors on board. Included in what we are referring to as the processor are the various execution units and the split first level instruction and data caches. The two processors share a unified second level cache, also onboard the chip, through a Core Interface Unit (CIU) in Figure 1. The CIU is a crossbar switch between the L2, implemented as three separate, autonomous cache controllers, and the two processors. Each L2 cache controller can operate concurrently and feed 32 bytes of data per cycle. The CUI connects each of the three L2 controllers to either the data cache or the instruction cache in either of the two processors. Additionally, the CUI accepts stores from the processors across 8-byte wide buses and sequences them to the L2 controllers. Each processor has associated with it a Noncacheable (NC) Unit, the NC Unit in Figure 1, responsible for handling instruction serializing functions and performing any noncacheable operations in the storage hierarchy. Logically, this is part of the L2.

Figure 1: POWER4 Chip Logical View



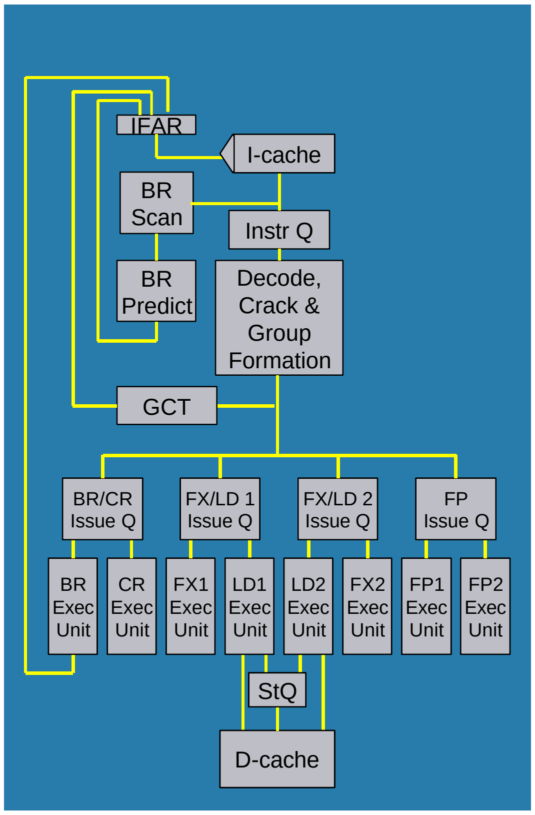
The directory for a third level cache, L3, and logically its controller are also located on the POWER4 chip. The actual L3 is on a separate chip. A separate functional unit, referred to as the Fabric Controller, is responsible for controlling data flow between the L2 and L3 controller for the chip and for POWER4 communication. The GX controller is responsible for controlling the flow of information in and out of the system. Typically, this would be the interface to an I/O drawer attached to the system. But, with the POWER4 architecture, this is also where we would natively attach an interface to a switch for clustering multiple POWER4 nodes together.

Also included on the chip are functions we logically call Pervasive function. These include trace and debug facilities used for First Failure Data Capture, Builtin Self Test (BIST) facilities, Performance Monitoring Unit, an interface to the Service Processor (SP) used to control the overall system, Power On Reset (POR) Sequencing logic, and Error Detection and Logging circuitry.

Four POWER4 chips can be packaged on a single module to form an 8-way SMP. Four such modules can be interconnected to form a 32-way SMP. To accomplish this, each chip has five primary interfaces. To communicate to other POWER4 chips on the same module, there are logically four 16-byte buses. Physically, these four buses are implemented with six buses, three on and three off, as shown in Figure 1. To communicate to POWER4 chips on other modules, there are two 8-byte buses, one on and one off. Each chip has its own interface to the off chip L3 across two 16-byte wide buses, one on and one off, operating at one third processor frequency. To communicate with I/O devices and other compute nodes, two 4-byte wide GX buses, one on and one off, operating at one third processor frequency, are used. Finally, each chip has its own JTAG interface to the system service processor.
All of the above buses, except for the JTAG interface, scale with processor frequency. POWER4 systems will be offered at more than one frequency. It is also anticipated that over time, technological advances will allow us to increase processor frequency. As this occurs, bus frequencies will scale proportionately allowing system balance to be maintained.

POWER4 Core: Figure 2 shows a high level block diagram of a POWER4 core. Both cores on a chip are identical and provide a 2-way SMP model to software. The internal microarchitecture of the core is a speculative superscalar outoforder execution design.
Up to eight instructions can be issued each cycle, with a sustained completion rate of five instructions. Register rename pools and other outoforder resources coupled with the pipeline structure allow the design to have over 200 instructions in flight at any given time.
In order to exploit instruction level parallelism there are eight execution units, each capable of being issued an instruction each cycle. Two identical floating-point execution units, each capable of starting a fused multiply and add each cycle, i.e., a maximum 4 floating-point operations (FLOPs) per cycle per core, are provided. In order to feed the dual floating-point units, two loadstore units, each capable of performing address generation arithmetic, are provided.

Figure 2: POWER4 Core



Additionally, dual fixed point execution units, a branch execution unit and an execution unit to perform logical operations on the condition register exist.

Branch Prediction: To help mitigate the effects of the long pipeline necessitated by the high frequency design, POWER4 invests heavily in branch prediction mechanisms. In each cycle, up to eight instructions are fetched from the direct mapped 64 KB instruction cache. The branch prediction logic scans the fetched instructions looking for up to two branches each cycle. Depending upon the branch type found, various branch prediction mechanisms engage to help predict the branch direction or the target address of the branch or both. Branch direction for unconditional branches are not predicted. All conditional branches are predicted, even if the condition register bits upon which they are dependent are known at instruction fetch time. Branch target addresses for the PowerPC branch to link register (*bclr*) and branch to count register (*bcctr*) instructions can be predicted using a hardware implemented link stack and count cache mechanism, respectively. Target addresses for absolute and relative branches are computed directly as part of the branch scan function.

As branch instructions flow through the rest of the pipeline, and ultimately execute in the branch execution unit, the actual outcome of the branches are determined. At that point, if the predictions were found to be correct, the branch instructions are simply completed like all other instructions. In the event that a prediction is found to be incorrect, the instruction fetch logic causes the mispredicted instructions to be discarded and starts refetching instructions along the corrected path.

POWER4 uses a set of three branch history tables to predict the direction of branch instructions. The first table, called the local predictor, is similar to a traditional branch history table (BHT). It is a 16K entry array indexed by the branch instruction address producing a 1-bit predictor that indicates whether the branch direction should be *taken* or *nottaken*. The second table, called the global predictor, predicts the branch direction based on the actual path of execution to reach the branch. The path of execution is identified by an 11-bit vector, one bit per group of instructions fetched from the instruction cache for each of the previous eleven fetch groups. This vector is referred to as the global history vector. Each bit in the global history vector indicates whether the next group of instructions fetched are from a sequential cache sector or not. The global history vector captures this information for the actual path of execution through these sectors. That is, if there is a redirection of instruction fetching, some of the fetched group of instructions are discarded and the global history vector is immediately corrected. The global history vector is hashed, using a bitwise exclusive or with the address of the branch instruction. The result indexes into a 16K entry global history table to produce another 1-bit branch direction predictor. Similar to the local predictor, this 1-bit global predictor indicates whether the branch should be predicted to be *taken* or *nottaken*.   Finally, a third table, called the selector table, keeps track of which of the two prediction schemes works better for a given branch and is used to select between the local and the global predictions. The 16K entry selector table is indexed exactly the same way as the global history table to produce the 1-bit selector. This combination of branch prediction tables has been shown to produce very accurate predictions across a wide range of workload types.

If the first branch encountered in a particular cycle is predicted as not taken and a second branch is found in the same cycle, POWER4 predicts and acts on the second branch in the same cycle. In this case, the machine will register both branches as predicted, for subsequent resolution at branch execution, and will redirect the instruction fetching based on the second branch.

As branch instructions are executed and resolved, the branch history tables and the other predictors are updated to reflect the latest and most accurate information. Dynamic branch prediction can be overridden by software. This is useful in cases where software can predict better than the hardware. It is accomplished by setting two previously reserved bits in conditional branch instructions, one to indicate a software override and the other to predict the direction. When these two bits are zero (suggested use for reserved bits), hardware branch prediction is used. Since only reserved bits are used for this purpose, 100% binary compatibility with earlier software is maintained.

POWER4 uses a link stack to predict the target address for a branch to link instruction that it believes corresponds to a subroutine return. By setting the hint bits in a branch to link instruction, software communicates to the processor whether a *branch to link* instruction represents a subroutine return or a target address that is likely to repeat or neither.

When instruction fetch logic fetches a *branch and link* instruction (either conditional or unconditional) predicted as *taken*, it pushes the address of the next instruction onto the link stack. When it fetches a *branch to link* instruction with *taken* prediction and with hint bits indicating a subroutine return, the link stack is popped and instruction fetching starts from the popped address.

In order to preserve the integrity of the link stack in the face of mispredicted branch target link instructions, POWER4 employs extensive speculation tolerance mechanism in its link stack implementation to allow recovering the link stack under most circumstances.

The target address of a *branch to count* instruction is often repetitive. This is also true for some of the *branch to link* instructions that are not predictable through the use of the link stack, (because they do not correspond to a subroutine return). By setting the hint bits appropriately, software communicates to the hardware whether the target address for such branches are repetitive. In these cases, POWER4 uses a 32 entry, tagless, directmapped cache, called a count cache, to predict the repetitive targets, as indicated by the software hints. Each entry in the count cache can hold a 62-bit address. When a *branch to link* or *branch to count* instruction is executed, for which the software indicates that the target is repetitive and therefore predictable, the target address is written in the count cache. When such an instruction is fetched, the target address is predicted using the count cache.

Instruction Fetch: Instructions are fetched from the instruction cache (I-cache) based on the contents of the Instruction Fetch Address Register (IFAR). The IFAR is normally loaded with an address determined by the branch prediction logic. As noted earlier, on cases where the branch prediction logic is in error, the branch execution unit will cause the IFAR to be loaded with the corrected address of the instruction stream to be fetched. Additionally, there are other factors that can cause a redirection of the instruction stream, some based on internal events, others on interrupts from external events. In any case, once the IFAR is loaded, the I-cache is accessed and retrieves up to 8 instructions per cycle. Each line in the I-cache can hold 32 PowerPC instructions, i.e., 128 bytes. Each line is divided into four equal sectors. Since I-cache misses are infrequent, to save area, the I-cache has been designed to have a single port that can be used to read or write one sector per cycle. The I-cache directory (IDIR) is indexed by the effective address and contains 42 bits of real address per entry.

On an I-cache miss, instructions are returned from the L2 in four 32-byte beats. The L2 normally returns the critical sector, the sector containing the specific word address that references the cache line, in one of the first two beats. Instruction fetch logic forwards these demand-oriented instructions into the pipeline as quickly as possible. In addition, the cache line is written into one entry of the instruction prefetch buffer so that the I-cache itself is available for successive instruction fetches. The instructions are written to the I-cache during cycles when instruction fetching is not using the I-cache as will occur when another I-cache miss occurs. In this way, writes to the I-cache are hidden and do not interfere with normal instruction fetching.

The PowerPC architecture specifies a translation lookaside buffer (TLB) and a segment lookaside buffer (SLB) to translate from the effective address (EA) used by software and the real address (RA) used by hardware to locate instructions and data in storage. As these translation mechanisms take several cycles, once translated the {EA,RA} pair is stored in a 128 entry, 2-way setassociative array, called the effective to real address translation table (ERAT). POWER4 implements separate ERATs for instruction cache (IERAT) and for data cache (DERAT) accesses. Both ERATs are indexed using the effective address. A common 1024entry 4-way set associative TLB is implemented for each core.

When the instruction pipeline is ready to accept instructions, the IFAR content is sent to the I-cache, IDIR, IERAT and the branch prediction logic. The IFAR is updated with the address of the first instruction in the next sequential sector. In the next cycle, instructions are received from the I-cache and forwarded to an instruction queue from which the Decode, Crack and Group Formation logic shown in Figure 2, pulls instructions. This is done even before it is known that there is a I-cache hit. Also received in the same cycle are the RA from the IDIR and the {EA,RA} pair from the IERAT and the branch direction prediction information. The IERAT is checked to insure that it has a valid entry and that its RA matches what's in the IDIR. If the IERAT has an invalid entry, the EA must be translated from the TLB and SLB. Instruction fetching is then stalled. Assuming the IERAT is valid, if the RA from the IERAT matches what's in the IDIR, an I-cache hit is validated. Using the branch prediction logic, the IFAR is reloaded and the process is repeated. Filling the instruction queue in front of the Decode Crack and Group Formation logic allows instruction fetching to get ahead of the rest of the system and queue work for the remainder of the system. In this way, when there is an I-cache miss, there will hopefully be additional instructions in the instruction queue to be processed and thereby not freeze the pipeline. In fact, this is what is frequently observed to occur.

If there is an I-cache miss, several different scenarios are possible. First, the instruction prefetch buffers are examined to see if the requested instructions are there, and, if so, it will steer these instructions into the pipeline as though they came from the I-cache and also write the critical sector into the I-cache. If the instructions are not found in the instruction prefetch buffer, a demand fetch reload request is sent to the L2. The L2 processes this reload request with high priority. When it is returned from the L2, an attempt will be made to write it into the I-cache.

In addition to these demand-oriented instruction fetching mechanisms, POWER4 prefetches instruction cache lines that might be referenced soon into its instruction prefetch buffer capable of holding four entries of 32 instructions each. The instruction prefetch logic monitors demand instruction fetch requests and initiates prefetches for the next one (if there is a hit in the prefetch buffer) or two (if there is a miss in the prefetch buffer) sequential cache lines after verifying they are not already in the I-cache. As these requests return cache lines, they are stored in the instruction prefetch buffer so that they do not pollute the demand-oriented I-cache. The I-cache contains only cache lines that have had a reference to at least one instruction.

Decode, Crack and Group Formation: As instructions are executed outoforder, it is necessary to remember the program order of all instructions in flight. In order to minimize the logic necessary to track a large number of in flight instructions, groups of instructions are formed. The individual groups are tracked through the system. That is, the state of the machine is preserved at group boundaries, not at an instruction boundary within a group. Any exception causes the machine to be restored to the state of the oldest group prior to the exception.

A group contains up to five internal instructions referred to as IOPs. In the decode stages the instructions are placed sequentially in a group, the oldest instruction is placed in slot 0, the next oldest one in slot 1, and so on. Slot 4 is reserved for branch instructions only. If required, noops are inserted to force the branch instruction to be in the fourth slot. If there is no branch instruction then slot 4 contains a noop. Only one group of instructions can be dispatched in a cycle, and all instructions in a group are dispatched together. By dispatch we mean the movement of a group of instructions into the issue queues. Groups are dispatched in program order. Individual IOPs are issued from the issue queues to the execution units out of program order.

Results are committed when the group completes. A group can complete when all older groups have completed and when all instructions in the group have finished execution. Only one group can complete in a cycle.

Internal instructions, in most cases, are architected instructions. However, in some cases, instructions are split into one or more internal instructions. In order to achieve high frequency operation, we have limited instructions to read at most 2 registers and write at most one register. (Some floating-point operations do not obey this restriction, for performance reasons, e.g., the *fused multiply and add* series of instructions are handled directly in the floating-point unit though they require three sources.) If this is not the case then the instruction is split to satisfy this requirement. As an example, the load with update instructions that load one register and increment an index register is split into a load and an add instruction. Similarly, the *load multiple word* instruction is implemented with multiple *load word* instructions. As far as group formation, we differentiate two classes. If an instruction is split into 2 instructions, such as *load with update*, we consider that *cracking*. If an instruction is split into more than 2 IOPs then we term this a millicoded instruction. Cracked instructions flow into groups as any other instructions with one restriction. Both IOPs must be in the same group. If both IOPs cannot fit into the current group, the group is terminated and a new group is initiated. The instruction following the cracked instruction may be in the same group as the cracked instruction, assuming there is room in the group. Millicoded instructions always start a new group. The instruction following the millicoded instruction also initiates a new group.

For correct operation, certain instructions are not allowed to execute speculatively. To ensure that the instruction executes nonspeculatively, it is not executed until it is next to complete. This mechanism is called completion serialization. To simplify the implementation, such instructions form single instruction groups. Examples of completion serialization instructions include loads and stores to guarded space and context synchronizing instructions such as *mtmsr*.

In order to implement outoforder execution, many of the architected registers are renamed, but not all. To insure proper execution of these instructions, any instruction that sets a non renamed register terminates a group.

Instructions performing logical operations on the Condition Register appear in lower frequency than other instructions. As such, a single execution is dedicated to this function. With the instruction issue rules described in the next section, this required restricting this class of instructions to only two of the four slots, specifically, slots 0 and 1.

Group Dispatch and Instruction Issue: Instruction groups are dispatched into the issue queues one group at a time. As a group is dispatched, control information for the group is stored in the Group Completion Table (GCT). The GCT can store up to 20 groups. The GCT entry contains the address of the first instruction in the group. As instructions finish execution, that information is registered in the GCT entry for the group. Information is maintained in the GCT until the group is retired, i.e., either all of its results are committed or the group is flushed from the system.

Each instruction slot feeds separate issue queues for the Floating-Point Units, the Branch Execution Unit, the CR Execution Unit, the Fixed Point Execution Units and the Load/Store Execution Units. The Fixed Point and Load/Store Execution Units share common issue queues. Table 1 summarizes the depth of each issue queue and the number of queues available for each type of queue. For the Floating-Point Issue Queues and the common issue queues for the Fixed Point and Load/Store Units, the issue queues fed from instruction slots 0 and 3 hold instructions to be executed in one of the execution units while the issue queues fed from instruction slots 1 and 2 feed the other execution unit. The CR Execution Unit draws its instructions from the CR Logical Issue Queue fed from instruction slots 0 and 1.

Table 1: Issue Queues

Queue Type	Entries per Queue	Number of Queues
Fixed Point and Load/Store Units	9	4
Floating-Point	5	4
Branch Execution	12	1
CR Logical	5	2

Instructions are dispatched into the top of an issue queue. As they are issued from the queue, the remaining instructions trickle down. In the case of two queues feeding a common execution unit, the two queues are interleaved. The oldest instruction that has all of its sources set in the common interleaved queue is issued to the execution unit.

Before a group can be dispatched all resources to support that group must be available. If they are not, the group is held until the necessary resources are available. To successfully dispatch, the following resources are assigned:

- **GCT entry:** One GCT entry is assigned for each group. It is released when the group retires.

- **Issue Queue slot:** An appropriate issue queue slot must be available for each instruction in the group. It is released when the instruction in it has successfully been issued to the execution unit. Note that in some cases this is not known until several cycles after the instruction has been issued. As an example, a fixed point operation dependent on an instruction loading a register can be speculatively issued to the fixed point unit before we know if the load instruction resulted in an L1 data cache hit. Should the load instruction miss in the cache, the fixed point instruction is effectively pulled back and sits in the issue queue until data is successfully loaded into the register.

- **Rename Register:** For each register that is renamed and set by an instruction in the group, a corresponding rename resource must be available. Table 2 summarizes the rename resources available to each POWER4 core. The rename resource is released when the next instruction writing to the same logical resource is committed.

Table 2: Rename resources

Resource Type	Logical Size	Physical Size
GPRs	32 (36)	80
FPRs	32	72
CRs	8 (9) 4-bit fields	32
Link/Count Registers	2	16
FPSCR	1	20
XER	4 fields	24

As a result of cracking and millicode used in forming groups, in some situations it is necessary to use additional logical registers. Four additional GPRs and one additional 4-bit CR field are required. The condition register is architected to be 1 32-bit register comprised as 8 4-bit fields. Each field is renamed. Not all of the XER is renamed. Only four of the XER fields are renamed.

- **Load Reorder Queue (LRQ) entry:** An LRQ entry must be available for each load instruction in the group. It is released when the group completes. The LRQ has 32 entries.

- **Store Reorder Queue (SRQ) entry:** An SRQ entry must be available for each store instruction in the group. They are released when the result of the store is successfully sent to the L2, after the group completes. The SRQ has 32 entries.

The operation of the LRQ and the SRQ is described in the section on the Load/Store Unit.

As noted previously, certain instructions require completion serialization. Groups so marked will not be issued until that group is the next to complete, i.e., all prior groups have successfully completed. Additionally, instructions that read a nonrenamed register cannot be executed until we are sure all writes to that register have completed. To simplify the implementation, any instruction that writes to a nonrenamed register sets a switch that is reset when the instruction finishes execution. If the switch is set, then this blocks dispatch of an instruction that reads a nonrenamed register. Writes to a nonrenamed register are guaranteed to be in program order by making them completion serialization.

Since instruction progression through the machine is tracked in groups, when a particular instruction within a group needs to signal an interrupt it is achieved by flushing all of the instructions (and results) of the group and then redispatching the instructions into single instruction groups. A similar mechanism is used to ensure that the fixed point exception register summary overflow bit is correctly maintained.

Load/Store Unit Operation: The Load/Store Unit requires special attention in an outoforder execution machine in order to insure memory consistency. As we cannot be sure that the result of a store operation will be committed at the time it is executed, a special mechanism is employed. Associated with each SRQ entry is a Store Data Queue (SDQ) entry. The SDQ entry maintains the desired result to be stored until the group containing the store instruction is committed. Once committed, the data maintained in the SDQ is written to the caches. Additionally, three particular hazards need to be guarded against.

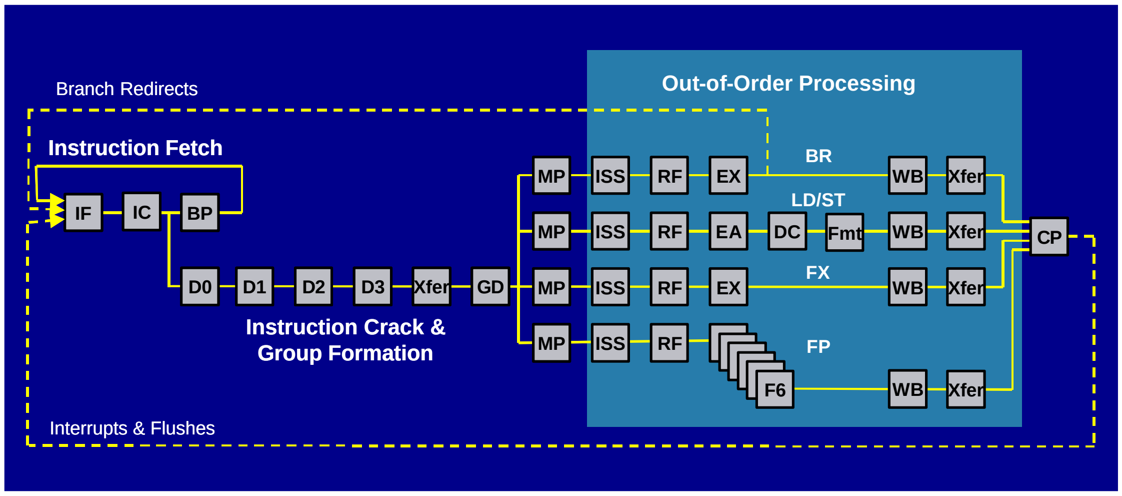
- **Load Hit Store:** A younger load that executes before an older store to the same memory location has written its data to the caches must retrieve the data from the SDQ. As a result, as loads execute they check the SRQ to see if there is any older store to the same memory location with data in the SDQ. If one is found, the data is forwarded from the SDQ rather than from the cache. If the data cannot be forwarded, as will be the case if the load and store instructions operate on overlapping memory locations and the load data is not the same as or contained within the store data, the group containing the load instruction is *flushed*, that is, it and all younger groups are discarded and refetched from the instruction cache. If we can tell that there is an older store instruction that will write to the same memory location but has yet to write its result to the SDQ, the load instruction is *rejected* and reissues again waiting for the store instruction to execute.

- **Store Hit Load:** If a younger load instruction executes before we have had a chance to recognize that an older store will be writing to the same memory location, the load instruction has gotten stale data. To guard against this, as a store instruction executes it checks the LRQ and if it finds a younger load that has executed and loaded from memory locations the store is writing to, the group containing the load instruction and all younger groups are *flushed* and refetched from the instruction cache. To simplify the logic, all groups following the store are *flushed*. If the offending load is in the same group as the store instruction, the group is flushed and all instructions in the group form single instruction groups.

- **Load Hit Load:** Two loads to the same memory location must observe the memory reference order and guard against a store to the memory location from another processor between the intervening loads. If the younger load obtains old data the older load must not obtain new data. This requirement is called *sequential load consistency*. To guard against this, LRQ entries for all loads include a bit, that, if set, indicates a snoop has occurred to the line containing the loaded data for that entry. When a load instruction executes, it compares its load address against all addresses in the LRQ. A match against a younger entry which has been snooped indicates that a sequential load consistency problem exists. To simplify the logic all groups following the older load instruction are flushed. If both load instructions are in the same group then the flush request is for the group itself. In this case each instruction in the group when refetched form single instruction groups so as to avoid this situation the second time around.

Instruction Execution Pipeline: Figure 3 shows the POWER4 instruction execution pipeline for the various pipelines. The *IF*, *IC* and *BP* cycles correspond to the instruction fetching and branch prediction cycles. The *D0* through *GD* cycles are the cycles during which instruction decode and group formation occur. The *MP* cycle is the *Mapper* cycle where all dependencies are determined, resources assigned and the group dispatched into the appropriate issue queues. During the *ISS* cycle the IOP is issued to the appropriate execution unit, reads the appropriate register to retrieve its sources during the *RF* cycle and executes during the *EX* cycle writing its result back to the appropriate register during the *WB* cycle. At this point the instruction has finished execution but has not yet been completed. It cannot complete for at least 2 more cycles, the *Xfer* and *CP* cycle, assuming all older groups have completed and all other instructions in the same group have also finished.

Figure 3: POWER4 Instruction execution pipeline



Instructions waiting in the Instruction Queue after being fetched from the instruction cache wait prior to the *D1* cycle. This will be the case if instruction are fetched (up to 8 per cycle) faster than they can be formed into groups. Similarly, instructions can wait prior to the *MP* cycle if resources are not available to dispatch the entire group into the issue queues. Instructions wait in the issue queues prior to the *ISS* cycle. Similarly, they can wait to complete prior to the *CP* cycle.

Though not shown, the CR Logical Execution Unit, the unit responsible for executing logical operations on the Condition Register, is identical to the Fixed Point Execution pipeline, shown as the *FX* pipeline in the figure. The Branch Execution Unit pipeline is shown as the *BR* pipeline in the figure. If a branch instruction is mispredicted, either direction or target, then there is at least a 12 cycle branch mispredict penalty, depending on how long the mispredicted branch needed to wait to be issued.

The pipeline for the two Load/Store Units is identical and is shown as the *LD/ST* pipeline in the figure. After accessing the register file, load and store instructions generate the effective address in the *EA* cycle. The DERAT and for load instructions the Data Cache Directory and the data cache, are all accessed during the *DC* cycle. If a DERAT miss should occur the instruction is *rejected*, i.e., it is kept in the issue queue. Meanwhile a request is made to the TLB to reload the DERAT with the translation information. The rejected instruction reissues again a minimum of 7 cycles after it was first issued. If the DERAT still does not contain the translation information the instruction is rejected again. This process continues until the DERAT is reloaded. If a TLB miss occurs, i.e., we do not have translation information in the TLB, the translation is initiated speculatively. However, the TLB is not updated until we are sure the instruction will be executed, i.e., the TLB is updated when the group the instruction failing translation is in becomes the next group to complete. Two different page sizes are supported, 4-KB and 16-MB.

In the case of loads, if the directory indicates the L1 data cache contains the cache line, the requested bytes from the returned data are formatted, the *fmt* cycle, and written into the appropriate register. They are also available for use by dependent instructions during this cycle. In anticipation of a data cache hit, dependent instructions are issued so that their *RF* cycle lines up with the load instructions writeback cycle. If a cache miss is indicated, then a request is initiated to the L2 to retrieve the line. Requests to the L2 are stored in the Load Miss Queue (LMQ). The LMQ can hold up to eight requests to the L2. If the LMQ is full, the load instruction missing in the data cache is rejected, reissues again in 7 cycles and the process is repeated. If there is already a request to the L2 for the same line from another load instruction, the second request is merged into the same LMQ entry. If this is the third request to the same line, the load instruction is rejected and processing continues as above. All reloads from the L2 check the LMQ to see if there is an outstanding request yet to be honored against a just returned line. If there is, the requested bytes are forwarded to the register to complete the execution of the load instruction. After the line has been reloaded, the LMQ entry is freed for reuse.

In the case of store instructions, rather than write data to the data cache, the data is stored in the SDQ as described above. Once the group containing the store instruction is completed, an attempt is made to write the data in the data cache. If the cache line containing the data is already in the L1 data cache, the changed data is written to the data cache. If it is not, the line is not reloaded from the L2. In both cases, the changed data is written to the L2. The coherency point for POWER4 is the L2 cache. Additionally, all data in the L1 data cache is also in the L2 cache. If data needs to be cast out of the L2, the line is marked invalid in the L1 data cache if it is resident there.

The pipeline for the two fixed point execution units is shown as the *FX* pipe in Figure 3. Two fixed point instructions, with one dependent on the other, cannot issue on the successive cycles. There must be at least one dead cycle between their issue cycles.

The two floating-point execution units follow the same pipeline and is shown as the *FP* pipe in the figure. Floating-point instructions require 6 execution cycles. But, both pipes are fully piped, that is, two instructions can be issued to the floating-point pipes each cycle.

4. Storage Hierarchy

The POWER4 storage hierarchy consists of three levels of cache and the memory subsystem. The first and second levels of the hierarchy are on board the POWER4 chip. The directory for the third level cache, the L3, is on the chip, but the actual cache is off chip. Table 3 shows capacities and organization of the various levels of the hierarchy on a per chip basis.

Table 3: Storage hierarchy organization and size

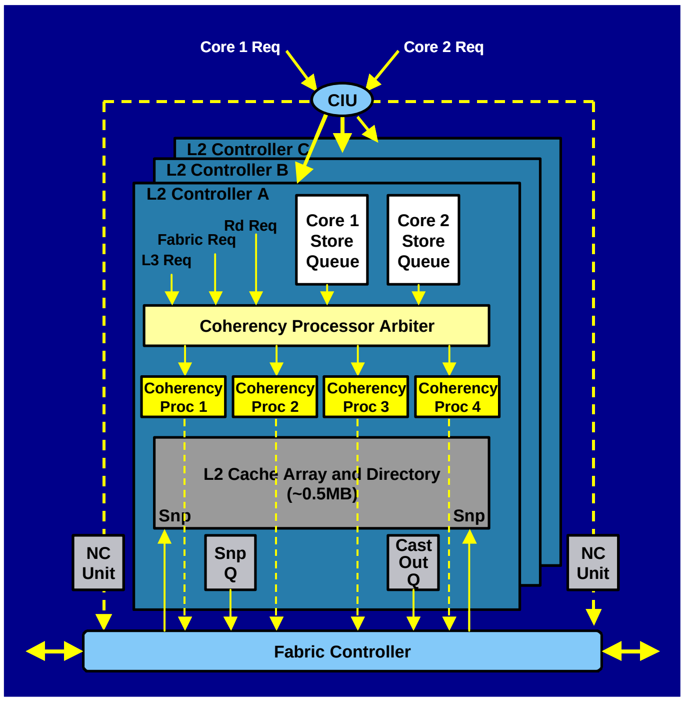
Component	Organization	Capacity per Chip
L1 Instruction Cache	Direct map, 128-byte line managed as 4 32-byte sectors	128 KB (64 KB per processor)
L1 Data Cache	2-way, 128-byte line	64 KB (32 KB per processor)
L2	8-way, 128-byte line	~ 1.5 MB
L3	8-way, 512-byte line managed as 4 128-byte sectors	32 MB
Memory	---	0-16 GB

L1 Caches: The L1 instruction cache is single ported capable of either one 32-byte read or write each cycle. The store through L1 data cache is triple ported capable of two 8-byte reads and one 8 byte write per cycle with no blocking. L1 data cache reloads are 32-bytes per cycle. The L1 caches are parity protected. When attempting to read data that exhibits a parity error, the faulty line is invalidated in the cache and retrieved from the L2. All data stored in the L1 data cache is available in the L2 cache guaranteeing no data loss. Data in the L1 can be in one of two states:

- **I (invalid state):** The data is invalid.
- **V (valid state):** The data is valid.

L2 Cache: The unified second level cache is shared across the two processors on the POWER4 chip. Figure 4 shows a logical view of the L2 cache. The L2 is implemented as three identical controllers. Cache lines are hashed across the three controllers.

Figure 4: L2 logical view



Each L2 controller consists of 4 SRAM partitions, each capable of supplying 16 bytes of data every other cycle. The four partitions can supply 32 bytes per cycle, taking four consecutive cycles to transfer a 128-byte line to the processor. The data arrays are ECC protected (single error correct, double error detect). Both wordline and bitline redundancy are implemented.

The L2 cache directory is implemented in two redundant 8-way set associative parity protected arrays. The redundancy, in addition to providing a backup capability, also provides two nonblocking read ports to permit snoops to proceed without causing interference to load and store requests.

A pseudo LRU replacement algorithm is implemented as a standard 7-bit tree structure. As the L1 is a store through design, store requests to the L2 are at most 8 bytes per request. The L2 implements two 4-entry 64-byte queues for gathering individual stores and minimizing L2 requests for stores.

The majority of control for L2 cache management is handled by four coherency processors in each controller. For each request to the L2 from the processors, either L1 data cache reload requests or instruction fetches, or from one of the store queues, a coherency processor is assigned to handle the request. Each coherency processor has associated with it a castout processor to handle deallocation of cache lines to accommodate L2 reloads on L2 misses. The coherency processor does the following:

- Controls the return of data from the L2 (hit) or from the fabric controller (miss) to the requesting core via the CIU;

- Updates the L2 directory as needed;

- Issues fabric commands for L2 misses on fetch requests and for stores that do not hit in the L2 in the M, Me or Mu state (described below);

- Controls writing into the L2 when either reloading due to fetch misses in the L2 or stores from the processors; and,

- Initiates back invalidates to a processor via the CIU resulting from a store from one core that hits a cache line marked as resident in the other processor's L1 data cache.

Included in each L2 controller are four snoop processors responsible for managing coherency operations snooped off of the fabric. When a fabric operation hits on a valid L2 directory entry, a snoop processor is assigned to take the appropriate action. Depending on the type of operation, the inclusivity bits in the L2 directory, and the cache line's coherency state, one or more of the following actions may result:

- Send a back invalidate request to the core(s) to invalidate a cache line in its L1 data cache;

- Read the data from the L2 cache;

- Update the cache line's directory state;

- Issue a *push* operation to the fabric to write modified data back to memory; and,

- Source data to another L2 from this L2.

In addition to dispatching a snoop processor, the L2 provides a snoop response to the fabric for all snooped operations. When a fabric operation is snooped by the L2, the directory is accessed to determine if the targeted cache line is resident in the L2 cache and, if so, what its coherency state is. Coincident with the snoop directory lookup, the snooped address is compared with the addresses of any currently active coherency, castout and snoop processors to detect address collision scenarios. The address is also compared to the per core reservation address registers. Based upon all of this information, the snoop response logic determines the appropriate snoop response to send back.

The L2 cache controller also acts as the reservation station for the two cores on the chip in support of the *lwarx/ldarx* and *stwcx/stdcx* instructions. One address register for each core is used to hold the reservation address. The reservation logic maintains a reservation flag per core to indicate when a reservation is set. The flag is set when a *lwarx* or *ldarx* instruction is received from the core, and is reset when a *stwcx* or *stdcx* instruction succeeds, or when certain invalidating type operations are snooped, including a store to the reservation address from other cores in the system.

The L2 cache implements an enhanced version of the MESI coherency protocol supporting seven states as follows:

- **I (invalid state):** The data is invalid. This is the initial state of the L2 entered from a Power OnReset or a Snoop invalidate hit.

- **SL (shared state, can be source to local requesters):** The data is valid. The cache line may also be valid in other L2 caches. From this state, the data can be sourced to another L2 on the same module via intervention. This state is entered as a result of a core L1 data cache load request or instruction fetch request that misses in the L2 and is sourced from another cache or from memory when not in other L2s.

- **S (shared state):** The data is valid. The cache line may also be valid in other L2 caches. In this state, the data cannot be sourced to another L2 via intervention. This state is entered when a snoop read hit from another processor on a chip on the same module occurs and the data and tag were in the SL state.

- **M (modified state):** The data is valid. The data has been modified and is exclusively owned. The cache line cannot be valid in any other L2. From this state the data can be sourced to another L2 in a chip on the same or remote module via intervention. This state results from a store operation performed by one of the cores on the chip.

- **Me (exclusive state):** The data is valid. The data is not considered modified but is exclusive to this L2. The cache line cannot be valid in any other L2. Castout of an Me line only requires invalidation of the tag, i.e., data does not have to be written back to memory. This state is entered as a result of one of the cores on the chip asking for a reservation via the *lwarx* or *ldarx* instruction when data is sourced from memory or for a cache line being prefetched into the L2 that was sourced from memory. (Sourcing data from the L3 in O state is equivalent to sourcing it from memory.)

- **Mu (unsolicited modified state):** The data is valid. The data is considered to have been modified and is exclusively owned. The cache line cannot be valid in any other L2. This state is entered as a result of one of the cores on the chip asking for a reservation via the *lwarx* or *ldarx* instruction when data is sourced from another L2 in

M state or for a cache line being prefetched into the L2 that was sourced from another L2 in M state.

- **T (tagged state):** The data is valid. The data is modified with respect to the copy in memory. It has also been sourced to another cache, i.e., it was in the M state at sometime in the past, but is not currently exclusively owned. From this state, the data will not be sourced to another L2 via intervention until the combined response is received and it is determined that no other L2 is sourcing data, i.e., if no L2s have the data in SL state. This state is entered when a snoop read hit occurs while in the M state.

The L2 state is maintained in the L2 directory. Table 4 summarizes the L2 states and possible L1 data cache state and possible state in other L2s. The directory also includes bits to indicate whether or not the data may be contained in one or both of the core's L1 data caches. Whenever a core requests data to be loaded into its L1 data cache a bit corresponding to that processor is set. This bit is not set for instruction fetches. This indication is imprecise as it is not reset if the data is replaced by the L1.

Table 4: Valid l2 states

L2 State	L1 Data Cache	Sate in Other L2s
I	I	Any
S_L	I, V	I, S, S_L, T
S	I, V	I, S, T
M, Me or Mu	I, V	I
T	I, V	I, S, S_L

Included within the L2 subsystem are two Noncacheable Units (NCU), one per core, labeled the NC Units in Figure 4. The NCUs handle noncacheable loads and stores, as well as cache and synchronization operations. Each NCU is partitioned into two parts: the NCU Master and the NCU Snooper. The NCU Master handles requests originating from cores on the chip while the NCU Snooper handles the snooping of *tlbie* and *icbi* operations from the fabric.

The NCU Master includes a 4-deep FIFO queue for handling cache inhibited stores, including memory mapped I/O store operations, and cache and barrier operations. It also contains a 1-deep queue for handling cache inhibited load operations.

The return of data for a noncacheable load operation is via the L2 controller using the same reload buses as for cacheable load operations. Cache inhibited stores are routed through the NCU in order to preserve execution ordering of noncacheable stores with respect to each other.

Cache and synchronization operations originating in a core on the chip are handled in a similar manner as are cache inhibited stores except that they do not have any data associated with them. These operations are issued to the fabric. Most will be snooped by an L2 controller. Included in this category are *icbi, tlbie, tlbsync, eieio, sync, ptesync, lsync, dcbf, dcbi* and a core acknowledgment that a snooped *TLB* has completed.

The NCU Snooper snoops *icbi* and *tlbi*e operations from the fabric propagating them upstream to the cores. These snoops are sent to the core via the L2 controller's reload buses. It also snoops *sync, ptesync, lsync,* and *eieio*. These are snooped as they may need to be retried due to an *icbi* or *TLB* that has not yet completed to the same processor.

L3 Cache: Figure 5 shows a logical view of the L3 cache. The L3 consists of two components, the L3 controller and the L3 data array. The L3 controller is on the POWER4 chip and contains the tag directory, and the queues and arbitration logic to support the L3 and the memory behind it. The data array is stored in two 16 MB eDRAM chips mounted on a separate module. A separate memory controller can be attached to the backside of the L3 module.

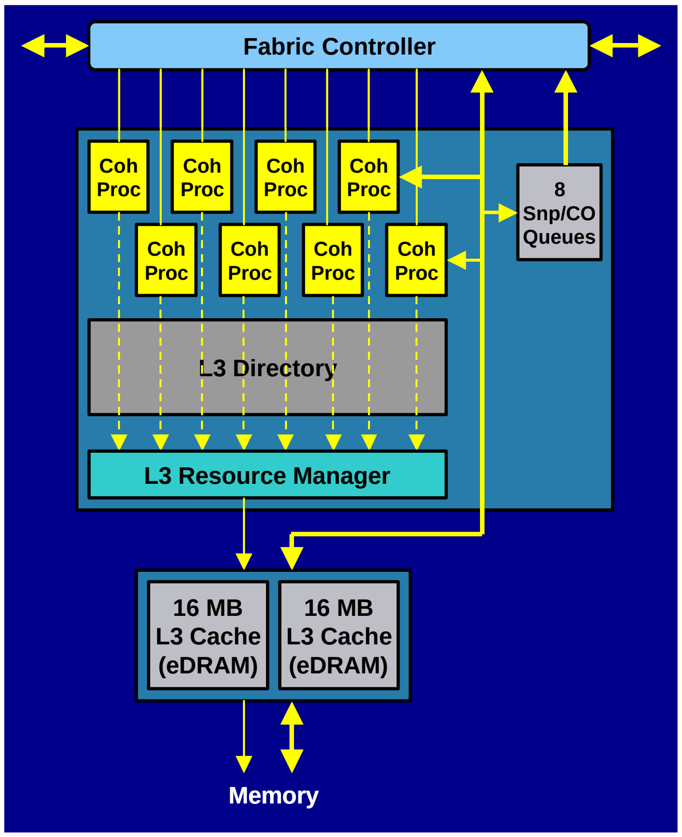


Figure 5: L3 logical view
To facilitate physical design and minimize bank conflicts, the embedded DRAM on the L3 chip is organized as 8 banks at 2 MB per bank, with banks grouped in pairs to divide the chip into four 4 MB quadrants. The L3 Controller is also organized in quadrants. Each quadrant contains two coherency processors to service requests from the Fabric, perform any L3 cache and/or memory accesses, and update the L3 tag directory. Additionally, each quadrant contains 2 processors to perform the memory castouts, invalidate functions and DMA writes for I/O operations. Each pair of quadrants shares one of the two L3 tag directory SRAMs.

The L3 cache is 8-way setassociative organized in 512 byte blocks, with coherence maintained on 128-byte sectors for compatibility with the L2 cache. Five coherency states are supported for each of the 128-byte sectors as follows:

- **I (invalid state):** The data is invalid.

- **S (shared state):** The data is valid. In this state, the L3 can only source data to L2s that it is caching data for.

- **T (tagged state):** The data is valid. The data is modified relative to the copy stored in memory. The data may be shared in other L2 or L3 caches.

- **Trem (remote tagged state):** This is the same as the T state, but the data was sourced from memory attached to another chip.

- **O (prefetch data state):** The data in the L3 is identical to the data in memory. The data was sourced from memory attached to this L3. The status of the data in other L2 or L3 caches is unknown.

Each L3 coherency processor supports one random cache or memory access. For sequential accesses the L3 coherency processors can support up to four concurrent load/store requests within a 512-byte L3 cache block. This allows the L3 to support increased cache and memory throughput for many common technical workloads to take advantage of the bandwidth capability available with the high speed buses in POWER4 systems.

The L3 is designed to be used as a standalone 32 MB L3 cache, or to be combined with other L3s on the same processor module in pairs or groups of four to create a larger, address interleaved L3 cache of 64 MB or 128 MB. Combining L3s into groups not only increases the L3 cache size, but also scales the available L3 bandwidth. When combined into groups, L3s and the memory behind them are interleaved on 512 byte granularity. The fabric bus controller controls which quadrant of which L3 a particular real address maps to, and the selected L3 controller adjusts the mapping from real address to L3 index and tag to account for the increase in the effective cache size. Table 5 shows the mapping of real address bits to L3 index and tag, as well as the algorithm for routing an address to the corresponding L3 controller and quadrant. The custom address flow logic is optimized for the 128-MByte combined case. To handle the index/tag adjustment for the smaller L3 cache sizes, the appropriate bits are swapped as the L3 controller receives an address from the Fabric Bus Controller. This approach causes the index bits to appear in a nonintuitive order, but avoids the need for the custom address flow logic to shift all of the address bits to make this adjustment. All address bit ranges in Table 5 assume that the full 42-bit address is denoted as bits 22:63. Bits 55:56 are the sector ID bits, and bits 57:63 are the offset within the 128-byte coherence granule.

Table 5: Mapping of real address bits to access L3 depending on L3 size

Logical L3 Size	L3 Index	L3 Tab	L3 Chip Select	L3 Quadrant Select
32 MB	51:52, 42:50	22:41	--	53:54
64 MB	51,41:50	22:40	54	52:53
128 MB	40:50	22:39	53:54	51:52

The L3 caches data, either from memory that resides beneath it or that resides elsewhere in the system, on behalf of the processors attached to its processor module. When one of its processors issues a load request that misses the L3 cache, the L3 controller allocates a copy of the data in S (shared) state. Inclusivity with the L1 and L2 is not enforced. Hence, when the L3 deallocates data it does not invalidate any L1 or L2 copies. The L3 enters T or Trem state when one of its local L2 caches does a castout from M or T state. An address decode is performed at snoop time to determine whether the address maps to memory behind the L3 or elsewhere in the system, and this causes the L3 to transition to T or Trem state as appropriate. This design point was chosen to avoid the need for a memory address range decode when the L3 performs a castout operation. The L3 can use the T/Trem distinction to determine whether the data can be written to the attached memory controller, or whether the castout operation must be issued as a Fabric Bus transaction.

When in T or Trem state, the L3 sources data to any requestor in the system. However, when in S state, the L3 will only source data to its own L2s. This minimizes data traffic on the buses between processor modules, since whenever possible, data is sourced by an L3 cache on the requesting processor module. When in O state, the L3 sources data to any requestor using the same rules that determine when it is permitted to send data from its attached memory controller, i.e., no cache is sourcing data and no snooper retried the request.

The L3 tag directory is ECC protected to support single bit error correct and double bit error detect. Uncorrectable errors result in a system checkstop. If a directory access results in a correctable error, the access is stalled while the error is corrected. After correction the original access takes place. When an error is corrected, a Recovered Attention is sent to the Service Processor for thresholding purposes.

The L3, memory address and control buses have parity bits for singlebit error detection. The L3 and memory data buses, as well as the L3 cache embedded DRAMs, have ECC to support singlebit error correct and doublebit error detect. Uncorrectable errors are flagged and delivered to the requesting processor with an error indication, resulting in a machine check interrupt. Correctable errors are corrected inline, and a Recovered Attention is sent to the Service Processor for thresholding purposes.

The L3 supports Cache Line Delete. The Cache Line Delete function is used to mask stuck faults in the L3 cache embedded DRAMs. Line Delete Control Registers allow the Service Processor to specify values of L3 index for which a particular member should not be used. When the L3 controller snoops a request that matches a specified L3 index, it masks off the tag directory compare for the member in question. The replacement algorithm also avoids the deleted member when choosing a victim in the specified congruence class. Cache Line Delete can be invoked at IPL time based upon results of poweron diagnostic testing, or it can be enabled dynamically due to a fault detected at runtime.

If an L3 tag directory develops a stuck fault, or L3 cache embedded DRAMs develop more stuck faults than can be handled with the Line Delete Control Registers, the L3 cache on the failing processor chip can be reconfigured and logically taken out of the system without removing other L3 caches in the system and without reconfiguring the memory attached to that L3. Memory accesses continue to pass through the reconfigured L3 module, but that L3 controller no longer performs cache operations.

Memory Subsystem: A logical view of the memory subsystem is shown in Figure 6. Each POWER4 chip can have an optional memory controller attached behind the L3 cache. Memory controllers are packaged two to a memory card and support two of the four processor chips on a module. A module can attach 2 memory cards, maximum. Memory controllers can have either one or two ports to memory.

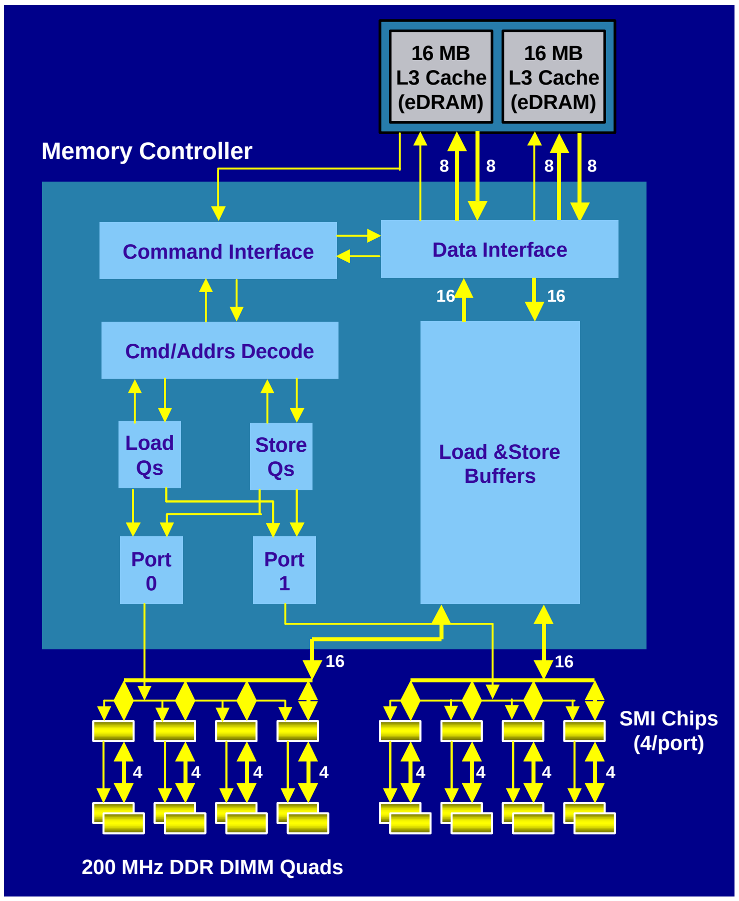


Figure 6: Memory subsystem logical view

The memory controller is attached to the L3 eDRAM chips, with each chip having two 8-byte buses, one in each direction, to the data interface in the memory controller. These buses operate at one-third processor speed using the Synchronous Wave Pipeline Interface to operate at high frequencies.

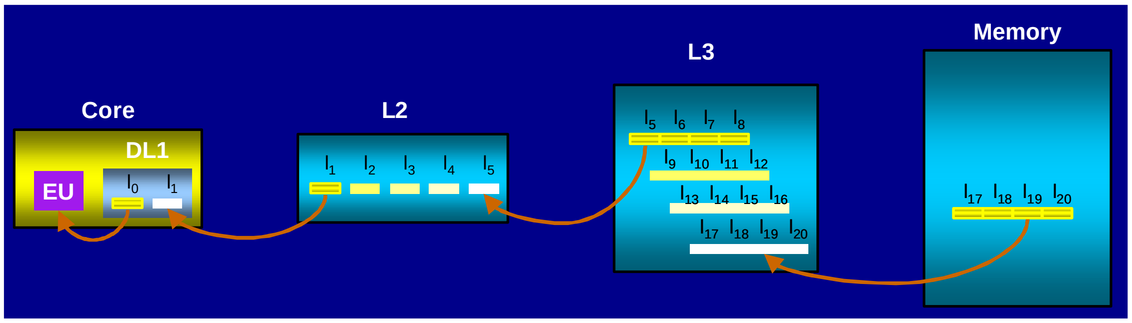
Each port to memory has four 4-byte bidirectional buses operating at 400 MHz connecting Load/Store buffers in the memory controller to four System Memory Interface (SMI) chips used to read and write data from memory. When two memory ports are available they each work on 512-byte boundaries. The memory controller has a 64-entry Read Command Queue, a 64-entry Write Command Queue and a 16-entry Write Cache Queue.

The memory is protected by a singlebit error correct, doublebit error detect ECC. Additionally, memory scrubbing is used in the background to find and correct soft errors.

Each memory extent has an extra DRAM to allow for transparent replacement of one failing DRAM per group of four DIMMs using chip kill technology. Redundant bit steering is also employed.

Hardware Data Prefetch: POWER4 systems employ hardware to prefetch data transparently to software into the L1 data cache. When load instructions miss sequential cache lines, either ascending or descending, the prefetch engine initiates accesses to the following cache lines before being referenced by load instructions. In order to insure the data will be in the L1 data cache, data is prefetched into the L2 from the L3 and into the L3 from memory. Figure 7 shows the sequence of prefetch operations. Eight such streams per processor are supported.

Figure 7: POWER4 hardware data prefetch


Memory
L3
Core
L2
DL1
EU
l_0 l_1
l_1 l_2 l_3 l_4 l_5
l_5 l_6 l_7 l_8
l_9 l_{10} l_{11} l_{12}
l_{13} l_{14} l_{15} l_{16}
l_{17} l_{18} l_{19} l_{20}
l_{17} l_{18} l_{19} l_{20}

Once a sequential reference stream is recognized, whenever a load instruction initiates a request for data in a new cache line the prefetch engine starts staging the next sequential line into the L1 data cache from the L2. At the same time it initiates a request to the L3 to stage a line into the L2. However, as latencies to load the L2 from the L3 are longer than the latency to load the L1 from the L2, rather than prefetch the second cache line, the fifth is prefetched, as shown in Figure 7. Prior references, or the initial ramp up on stream initiation, has already staged the second through fourth lines from the L2 to the L1 data cache. Similarly, a line is replaced in the L3 from memory. In order to minimize processing required to retrieve data from memory into the L3, a 512-byte line is prefetched. This needs to be done only every fourth line referenced. In the case shown in the figure, lines 17 through 20 are prefetrched from memory to the L3.

As memory references are based on real addresses, whenever a page boundary is crossed the prefetching must be stopped as we do not know the real address of the next page. Towards this end, POWER4 implements two page sizes, 4KB and 16MB. In addition to allowing the prefetch to continue for longer streams, it also saves translation time. This is especially useful for technical applications where it is common to sequentially reference large amounts of data.

In order to guard against prematurely installing a stream to be prefetched by the hardware, POWER4 ramps up the prefetches slowly requiring an additional 4 sequential cache misses to occur before the entire sequence of cache lines are in various stages of prefetch to the L1 data cache. However, software can often tell that a prefetch stream should be initiated. Towards this end, the *data cache block touch (dcbt)* instruction has been extended using a previously reserved bit to indicate to the hardware that a prefetch stream should be installed immediately without waiting for confirmation.

Special logic to implement data prefetching exists in the core's Load/Store Unit (LSU) and in the L2 and L3. The direction to prefetch, up or down, is determined by the actual load address within the line that causes the cache miss. If the load address is in the bottom _ of the line then the guessed direction is up. If the actual load address is in the top _ of the line then the guessed direction is down. The prefetch engine initiates a new prefetch when it detects a reference to the line it guessed will be used. If the initial guess on the direction is not correct the subsequent access will not confirm to the prefetch engine that it had a stream. The incorrectly initiated stream will eventually be deallocated. The corrected stream will be installed as a new stream.

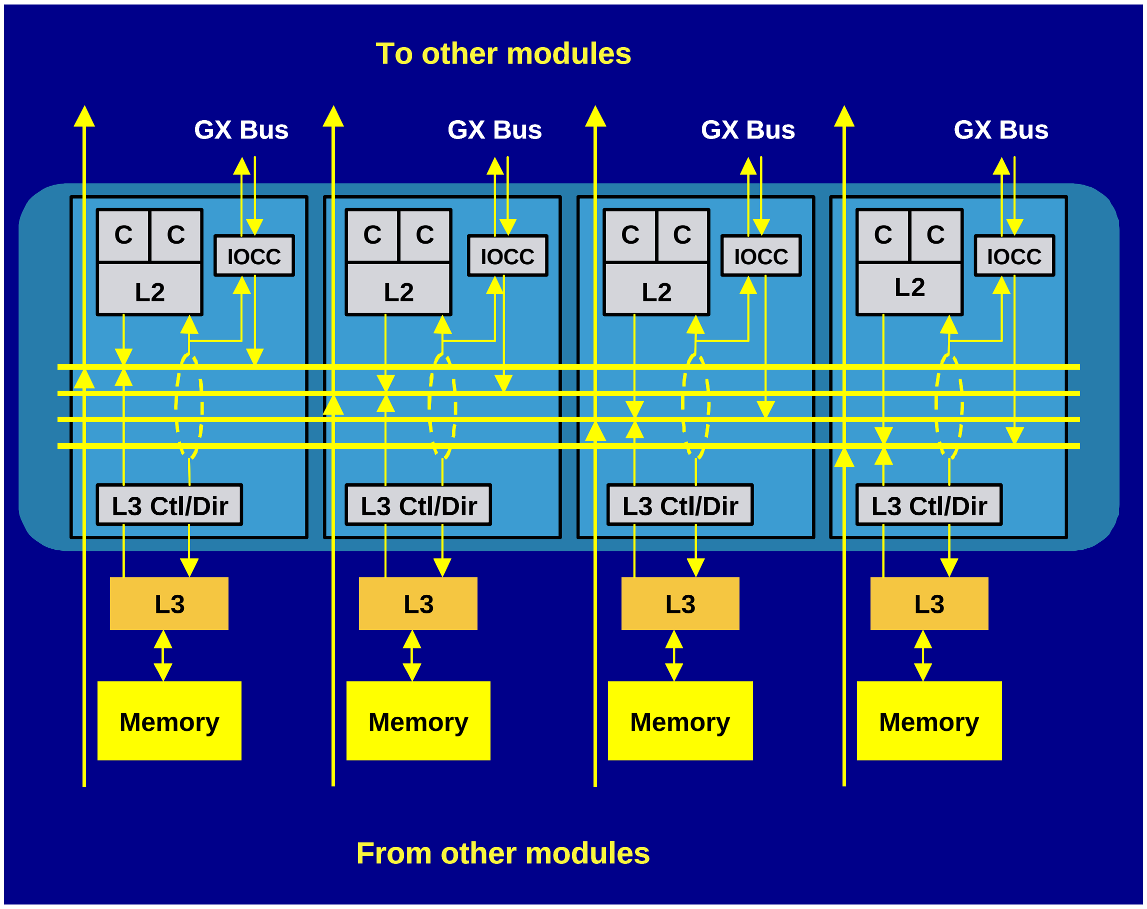
5. Interconnecting Chips to Form Larger SMPs

The basic building block is a multichip module (MCM) with four POWER4 chips to form an 8-way SMP. Multiple MCMs can be further interconnected to form 16, 24 and 32-way SMPs.

4-chip, 8-way SMP Module: Figure 8 shows the logical interconnection of four POWER4 chips across four logical buses to form an 8-way SMP. Each chip writes to its own bus arbitrating between the L2, I/O controller, and the L3 controller for the bus. Each of the four chips snoop all of the buses and if a transaction is presented that it needs to act on it takes the appropriate action. Request for data from an L2 are snooped by all chips to (a) see if it is in their L2 and in a state that it can source it to the requesting chip, or (b) see if it is in its L3 or in memory behind its L3 cache based on the real address of the request. Assuming it is, the sourcing chip returns the requested data to the requesting chip on its bus.

The interconnection topology appears like a bus-based system from a single chip's perspective. From the module's perspective it appears like a switch.

Figure 8: POWER4 multi-chip module with four chips



Multiple Module Interconnect: Figure 9 shows the interconnection of multiple 4-chip MCMs to form larger SMPs. From 1 to 4 MCMs can be interconnected. When interconnecting multiple MCMs, the intermodule buses act as repeaters moving requests and responses from one module to another module in a ring topology. As with the single MCM configuration, each chip always sends requests/commands and data on its own bus but snoops all buses.

12. L3 Memory Configurations: As noted earlier, each MCM can have from zero to two memory cards. In the case of two memory cards, there is no requirement that they be of equal size. In the case of no memory cards or two equal size memory cards connected to an MCM, the four L3s attached to the module act as a single 128 MB L3. In a single MCM system, each L3 caches data sourced from the memory attached behind its L3 cache. In the case of multiple MCMs and data being sourced from memory attached to another module, an attempt is made to cache the returned data on the requesting module. The particular L3 chosen is the L3 attached to the chip controlling the bus on which the data is returned on.

However, if the L3 is busy servicing requests, it is not cached. Also, data is not cached on the sourcing module if it is being sourced to a chip on another module.

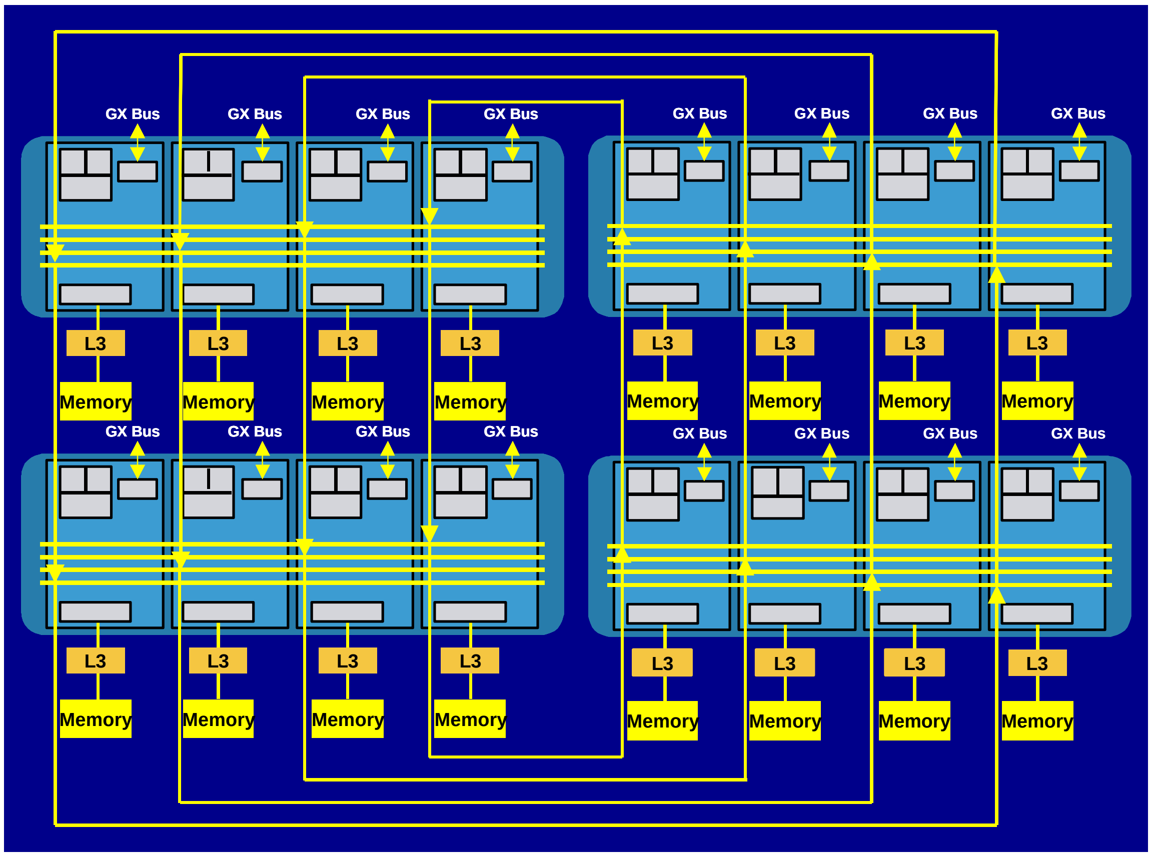


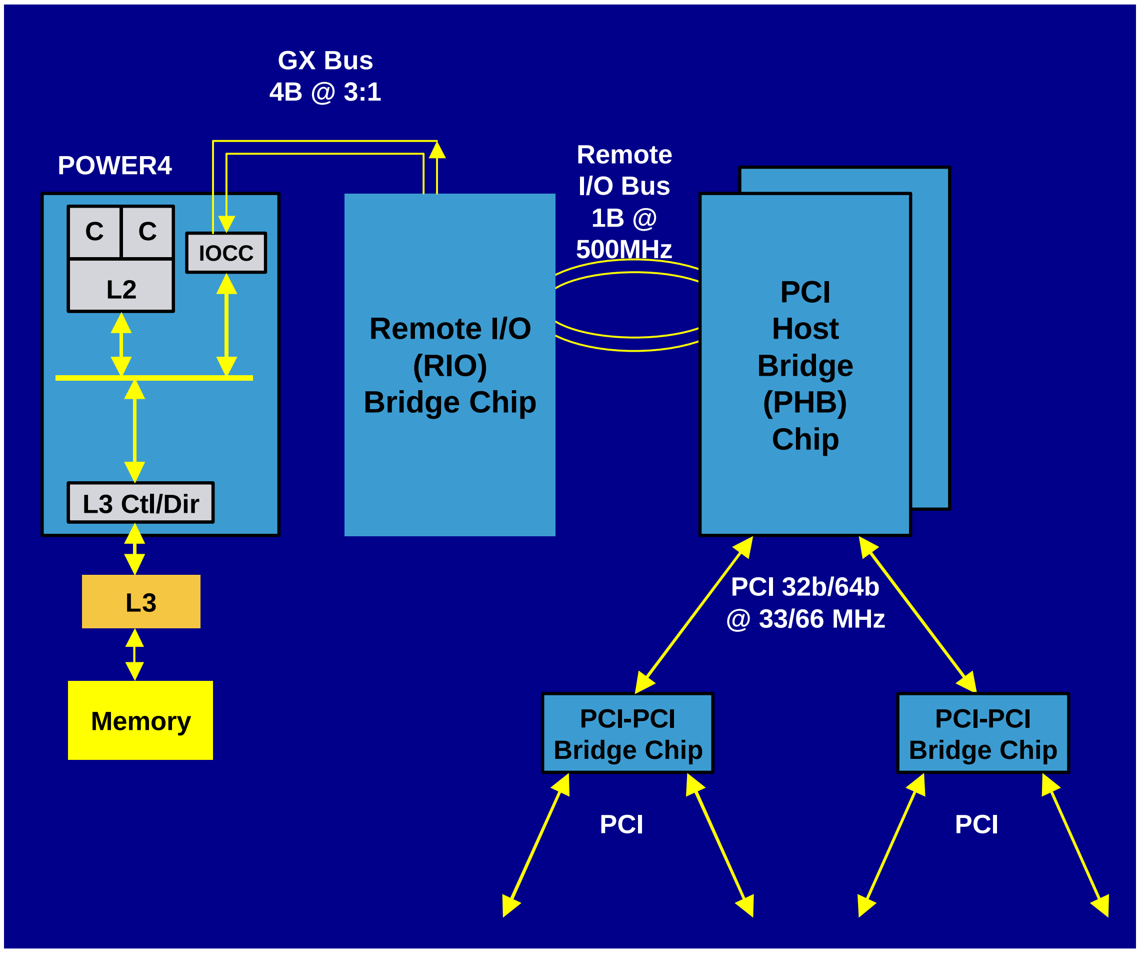
Figure 9: Multiple POWER4 multi-chip module interconnection

If one memory card or two unequal size memory cards are attached to a module, then the L3s attached to the module function as two 64 MB L3s. The two L3s that act in concert are the L3s that would be in front of the memory card. (Note that one memory card is attached to two chips.) The caching of requests to remote modules described above functions in this case in a comparable manner with the exception that the two L3s acting as a single L3 are considered to logically form a module boundary (for caching purposes).

6. I/O Structure

Figure 10 shows the I/O structure in POWER4 systems. Attached to a POWER4 GX bus is a Remote I/O (RIO) Bridge chip. This chip transmits the data across two 1-byte wide RIO buses to PCI Host Bridge (PHB) chips. Two separate PCI buses attach to PCIPCI bridge chips that further fan the data out across multiple PCI buses. When multiple nodes are

Figure 10: I/O logical view



interconnected to form clusters of systems, the RIO Bridge chip is replaced with a chip that connects with the switch. This provides increased bandwidth and reduced latency over switches attached via the PCI interface.

7. System Balance

POWER4 systems are designed to deliver balanced performance. As an example, as additional chips and MCMs are added to form larger SMP systems, additional resources are added, as can be seen from Figure 9. With the addition of each pair of POWER4 chips, the ability to add a memory card is provided. In addition to memory capacity, memory bandwidth is increased. Each additional POWER4 chip provides additional L3 resource.

All buses interconnecting POWER4 chips, whether or not on or off module, operate at half processor speed. As future technology is exploited allowing chip size to decrease and operating frequencies to increase, system balance is maintained, as bus speeds are no longer fixed but geared to processor frequency.

The multi-MCM configuration provides a worst case memory access latency of slightly over 10% more than the best case memory access latency maintaining the flat memory model simplifying programming.

The 8-way MCM is the building block for the system. It is only available with four chips, each with its attached L3. A single processor on a chip has all of the L3 resources attached to the module, and the full L2 onboard the chip. If this processor is the only processor executing, it would exhibit extremely good performance. If only one chip of the four on the module is active, the situation is similar, though both processors now share a common L2. They both have full access to all of the L3s attached to the module. When analyzing measurements comparing 1-way to 2-way to 4-way to 8-way performance one must account for the full L3 available in all of these configurations.

8. RAS Philosophy

It is not our intent here to describe the RAS mechanisms implemented in POWER4 systems. However, we would be remiss if we did not address the basic philosophy driving the design. Other white papers are planned to address this area in depth.

Simply stated, the RAS approach is to minimize outages as much as possible. This is achieved using several mechanisms. They manifest themselves by using redundant resources in some cases (as is the case with bit steering to allow spare memory bits to be used to correct faulty memory cells and by, after general availability, dynamically switching in spare resources such as processors, if a resource fails); by changing many checkstops into synchronous machine interrupts to allow software to take corrective action rather than halting execution; by failing only the affected partition in an LPAR environment and letting other partitions continue operation. If the POWER4 service processor detects a component beginning to exhibit errors above a predetermined threshold, the Field Replaceable Unit (FRU) containing the component, is scheduled to be replaced before it incurs a hard failure

9. Future Roadmap

Enhancements to the current POWER4 system in the coming years will take several directions.

We are in the process of leveraging newer technologies to allow us to increase frequency while further decreasing power. We have already stated we will exploit IBM’s low-k technology employed in their 0.13 _m lithography process. We will aggressively increase processor frequencies to the 2+ GHz range while maintaining the system balance our current design offers.

The current design introduces parallelism throughout the system so as to overcome the relatively speaking increasing memory latencies resulting from high frequency operations. The parallelism allows one to continue executing in the presence of cache misses. Future POWER4 systems will continue this design, increasing parallelism and providing larger caches.

We have already invested in insuring that software can exploit the increased performance levels POWER4 systems will be offering. But, this is a never ending endeavor. We will continue making system level enhancements so as to provide even greater performance increases over time.

10. Summary

POWER4 development has met our objectives in terms of performance and schedule as defined at the project start.

Enormous levels of bandwidth and concurrency contribute to superior performance across a broad range of commercial and high performance computing environments. These unprecedented performance levels are achieved by a total system design that exploits IBM's leading technologies.

As early as January 2000, the superiority of this design point was recognized by the publishers of the MicroProcessor Report. At that time they awarded POWER4 the 2000 MicroProcessor Technology Award in recognition of its innovations and technology exploitation.

We are well along in developing follow-on systems to the current POWER4 to further enhance its leadership. POWER4 is redefining what is meant by a server and how a server needs to be designed.

11. Notices