POWER3: The next generation of PowerPC processors

by F. P. O'Connell
S. W. White

The POWER3 processor is a high-performance microprocessor which excels at technical computing. Designed by IBM and deployed in various IBM RS/6000® systems, the superscalar RISC POWER3 processor boasts many advanced features which give it exceptional performance on challenging applications from the workstation to the supercomputer level. In this paper, we describe the microarchitectural features of the POWER3 processor, particularly those which are unique or significant to the performance of the chip, such as the data prefetch engine, nonblocking and interleaved data cache, and dual multiply–add-fused floating-point execution units. Additionally, the performance of specific instruction sequences and kernels is described to quantify and further illuminate the performance attributes of the POWER3 processor.

Introduction

The IBM POWER3 processor is a 64-bit symmetric-multiprocessing-enabled superscalar RISC microprocessor which is the heart of a new line of RS/6000* workstations and server products. Designed to run at frequencies up to one-half gigahertz, the POWER3 processor supports even the most challenging technical computing applications. This paper describes the motivation for the creation of the POWER3 processor, the challenges that it addresses, the highlights of its microarchitecture that are significant to performance, and its key performance traits.

Product motivation and technological challenges

The POWER3 processor is the successor to the POWER2 processor; it was designed primarily to meet the demands for technical computing which come from a wide variety of customers in nearly every major sector of the marketplace: automotive, aerospace, pharmaceutical, weather prediction, energy, defense, electronics, chemical processing, bioengineering, environmental, and many areas of research. As with most modern microprocessor design efforts, the POWER3 processor required an enormous investment involving many highly skilled and talented professionals. Such an investment is prudent if and only if it addresses a large and growing market need, as is the case with modern technical computing.

- *Demand for faster processors*

There is an insatiable demand for faster computing from practitioners in engineering and scientific fields. This

©**Copyright** 2000 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/00/$5.00 © 2000 IBM

demand arises from two facts: The universe in which we live is dauntingly complex, and humans are constantly seeking to understand and master that complexity. Often the individual human drive to discover is heightened by national security pressures as well as corporate competition.

It is fascinating to note that the preparation for the digital computing revolution actually began thousands of years ago. Archimedes first proposed a successive approximation to the calculation of the constant π around 330 B.C., almost 2300 years before his mathematics would find its true utility with the aid of a digital computer. Building upon the mathematical foundations laid by mathematicians over millennia, the field of numerical methods quickly flourished with the advent of the digital computer.

Just since the 1960s, when digital computers began to become generally available, there has been an amazing amount of progress in engineering and scientific computation. Beginning with relatively simple linear, one-dimensional differential equations, the push began to add more and more realism to analyses, causing the amount of knowledge and insight gained from digital computer simulations, as well as the demand for higher-speed computing, to grow commensurately. At the same time, advances in computer technology spurred the development of new mathematical and algorithmic approaches to problems which were previously unsolvable, further increasing the demand for high-speed computing.

Contemporary computing challenges often involve the solution of partial differential equations involving three spatial dimensions (i.e., the dimensions of physical space) as well as time, and often one or more mathematical nonlinearities. Modeling three dimensions with sufficiently fine resolution requires large amounts of high-speed storage (i.e., main memory), usually one or several orders of magnitude more than the corresponding two-dimensional approximation. The finer spatial resolution often imposes shorter time steps, resulting in a greater number of time-step iterations for a given solution. The mathematical nonlinearities that are introduced cause the computation to iterate as it searches for an acceptable solution.

Examples of such challenging, three-dimensional, nonlinear analysis can be found in such diverse areas as airplane and automobile design, weather modeling, molecular simulation, and nuclear weapon stewardship. An interesting example of the successful application of high-performance computing, which in turn promotes heavy demand, is car crashworthiness. Although automobiles are useful machines, they are also the leading cause of accidental death in modern society. Hence, there is significant social and regulatory pressure to design safer cars. The finite-element software for performing simulations of automobiles during collisions was born in part from the field of nuclear weapon design. Today, car designs can be tested computationally for occupant safety with significant advantages over physical testing in terms of time, cost, and knowledge gained. But the computation involves many complexities, including the detailed three-dimensional model of the car design, models of the materials involved, models of the bending, rotating, and crushing behavior of structural components of the car, accounting for the contact between different parts of the car as it deforms, and models of the human occupants and their interaction with the car itself, including seat belts, the steering wheel, and airbags. All of these complexities produce a large, highly nonlinear computational mechanics simulation, which in turn requires many calculations and many computer cycles to complete. And although there are often many approximations in these simulations, computations such as these still require days of computing time to complete; simulations on the order of a few hours would greatly increase productivity and time to market.

- *Major design constraints and challenges*

The success of the personal computer has profoundly influenced the market for high-performance systems, with the main effect being an extreme sensitivity to price. Since much of the cost of a system is often associated with memory, a decision to use anything other than commodity DRAM—which enjoys the benefits of large-scale manufacturing due to the personal computer boom—would drastically increase the price of adequately configured systems. But DRAM-based memories present a difficult challenge to designers attempting to build systems in which performance scales with increasing processor frequency. As processor frequencies have soared, memory latency has decreased only modestly, with the result being that memory latency, in processor cycles, has grown. At the same time, the demand for large parallel machines has increased physical bus length, protocol, and contention. The burden for overcoming these problems falls mainly on the microprocessor.

Technical computing applications present computer designers with additional formidable challenges which center on floating-point and fixed-point computational speed, scalability, load/store bandwidth, and cache capacity. The POWER3 processor was designed to meet the rigors of technical computing applications, as well as the more general requirements of the high-performance computing marketplace, including reliability, ease of programming, addressability, and power and space restrictions.

Overview of the POWER3 processor

The POWER3 processor is a CMOS-based superscalar RISC microprocessor which conforms to the PowerPC

Architecture*[1] [1]. Its two most fundamental architectural features—symmetric multiprocessing (SMP) systems and 64-bit effective addressing—provide the basics necessary for the contemporary challenges of technical computing. Multiprocessor systems not only increase the computing capacity of the system under a single image; they also allow applications to improve performance by exploiting shared-memory parallelism. The high level of performance that multiple POWER3 processors can produce, along with 64-bit program addressability, allows customers to solve larger three-dimensional simulations that were previously impractical. Because the POWER3 processor is a 64-bit implementation, it supports both the 32-bit and 64-bit modes provided by PowerPC*.

The POWER3 processor has also been designed to span several advances in CMOS technology, allowing it to more than double its initial product frequency of 200 MHz over its product lifetime. To date, POWER3 processors have been shipped in RS/6000 products at frequencies ranging up to 450 MHz. Applications which are optimized to the POWER3 platform can immediately take advantage of upgrades to faster POWER3 processors. A die photograph of the POWER3 processor is shown in **Figure 1**.

The POWER3 processor is partitioned into seven functional blocks (**Figure 2**):

- Instruction processing unit (IPU).
- Instruction flow unit (IFU).
- Fixed-point unit (FXU).
- Floating-point unit (FPU).
- Load/store unit (LSU).
- Data cache unit (DCU).
- Bus interface unit (BIU).

These functional units are discussed in the following sections.

- *Instruction processing unit and instruction flow unit*

Processor performance begins with the task of fetching the instructions for an application, partially decoding them, and dispatching them to the proper execution unit. The IPU and IFU are responsible for fetching, caching, and managing the flow of instructions during their tenure in the microprocessor (the tenure of a given instruction begins when it is dispatched to an execution unit and ends when it is completed).

Logically, instructions are fetched from memory; however, for performance reasons, the IPU implements a 32-kilobyte (KB) instruction cache and a cache reload buffer (CRB). The instruction cache holds 256 cache lines, each of which is 128 bytes in length, and is organized as two 128-way set-associative arrays. The instruction cache provides single-cycle access. The CRB holds the most recent cache line transferred from memory. To provide support for virtual storage, a 256-entry two-way set-

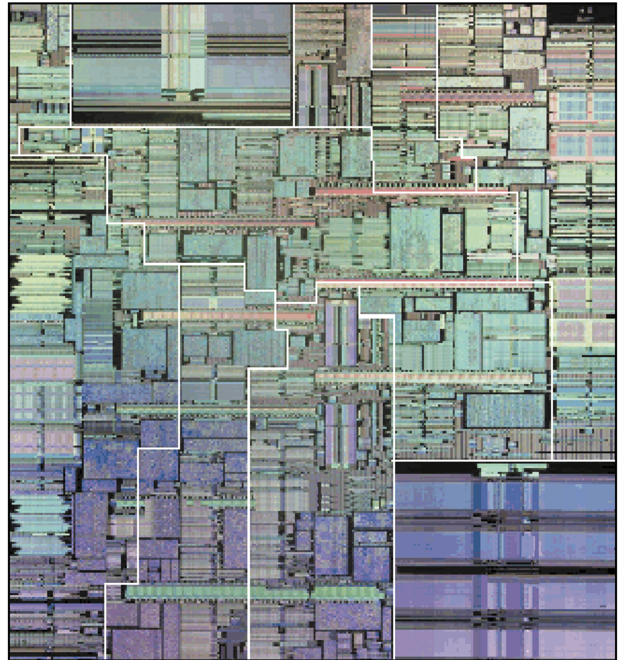

Figure 1

POWER3 processor die photo.

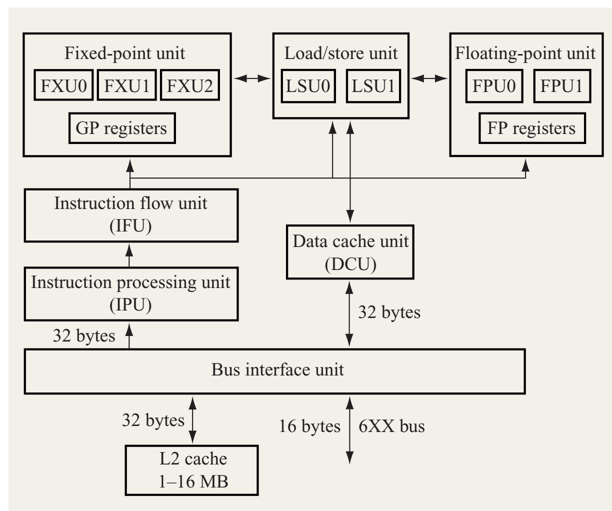

Figure 2

POWER3 processor functional unit block diagram.

[1] In this paper, the term *architecture* refers to the instruction set architecture specification defined in [1], whereas *microarchitecture* refers to the implementation details of a specific design, e.g., the POWER3 processor.

associative instruction translation lookaside buffer (ITLB) and a 16-entry instruction segment lookaside buffer (ISLB) are also implemented.

The IFU attempts to keep as many instructions as possible executing in parallel in the machine, maximizing instruction throughput. Up to eight instructions are fetched per cycle, up to four are dispatched per cycle, and up to four instructions per cycle can be completed. To improve throughput, instructions are dispatched in order, most are allowed to *execute* and *finish* out of order, and then all instructions *complete* in order. (Architectural registers are updated only when an instruction targeting them *completes*.) Executing and finishing instructions out of order increases the degree of instruction-level parallelism by allowing subsequent operations to execute both in parallel with logically prior long-running operations and ahead of operations which are delayed because of cache misses.

Instructions are dispatched to the various functional unit instruction queues and are tracked with an entry in the 32-entry completion queue. These unit instruction queues ensure each functional unit an adequate supply of instructions from which to select for execution; they also provide a place for the instruction flow unit to place instructions so that a stalled instruction does not block dispatching of subsequent instructions. In many designs, dispatch bandwidth is a frequent bottleneck. The robust implementation of the POWER3 processor greatly reduces the likelihood that performance will be affected by dispatch restrictions. Since operand availability is not a requirement for dispatch, availability of space in the instruction queues and in the completion queue are the two primary restrictions on dispatch. The completion block ensures that the architectural state of the processor is always correct, enforcing in-order completion of committed instructions and ensuring that exceptions and interrupts are handled properly and in order.

The POWER3 processor uses two mechanisms to improve branch-prediction accuracy. First, by tracking all outstanding condition-code-setting instructions, the CPU can determine when the branch outcome is known at dispatch, obviating the need to guess the direction of a branch. For branches that are unresolved at dispatch, the outcome is guessed and instructions are dispatched speculatively. If it is found that the branch was guessed incorrectly when the condition-code-setting instruction finishes, all instructions beyond the associated branch are canceled, and the correct instructions are then dispatched.

The primary method for branch prediction for unresolved branches uses a branch-history table (BHT) containing 2048 prediction fields, each with a two-bit branch-history entry. The two-bit prediction field is a saturating up–down counter with 0 corresponding to strongly not-taken and 3 corresponding to strongly taken. When branches are resolved, the prediction field for that entry is incremented or decremented depending upon whether the branch was taken or not taken, respectively, except when the field is already saturated.

- *Fixed-point execution units*

The POWER3 processor contains three fixed-point execution units: two single-cycle units and one multicycle unit. The single-cycle units execute all single-cycle instructions (arithmetic, shift, logical, compare, trap, and count leading zero) with a single-cycle latency (this means that instructions dependent upon the result can execute in the next cycle). All other fixed-point instructions, such as multiply and divide, are handled by the multicycle unit. Since the POWER3 processor is a 64-bit microprocessor, this includes 64-bit as well as 32-bit integer operands. The two single-cycle fixed-point units share a six-entry instruction queue, while the multicycle unit includes a three-entry instruction queue.

In contrast to the POWER2 processor [2], which included two symmetric units that executed both fixed-point and load/store instructions, the POWER3 design includes two dedicated load/store units in addition to the three fixed-point units. The independence of the fixed-point execution units and the load/store execution units is obviously a large performance benefit for calculations that are predominately integer in nature, such as Monte Carlo simulations. But even in floating-point calculations, this separation can be important. An example of this occurs in a sparse-matrix-vector multiply, which involves address indirection, whereby an integer index must be converted to byte-offset by a fixed-point instruction before it is used by a subsequent floating-point load operation.

- *Floating-point execution units*

The floating-point unit (FPU) contains two symmetrical floating-point execution units which implement a fused multiply–add pipeline conforming to the PowerPC Architecture. All floating-point instructions pass through both the multiply stage and the add stage. For floating-point multiplies, 0 is used as the add operand, and for floating-point adds, 1 is used as the multiplicand.

Each floating-point execution unit supports three-cycle data forwarding for dependent instructions within the same execution unit when the target of the first instruction feeds either the FRB or the FRC operand of the dependent instruction, where the operation is FRT $\leftarrow$ [(FRA) $\times$ (FRC)] + (FRB). In the case of data forwarding between execution units, or when, on the same execution unit, the first instruction is feeding the FRA operand of the dependent instruction, the latency is four cycles. It is worth noting that, for achieving frequency targets, the pipeline of floating-point register-to-register

instructions is broken up into ten stages (Fetch, Dispatch/Decode, Register Access, Execute 1, 2, 3, and 4, Finish, Complete, and Writeback), but only the first three Execute stages are exposed for dependent instructions.

Most floating-point instructions have single-cycle throughput. Since the POWER3 processor can execute two floating-point multiply–add instructions per cycle, the peak floating-point rate of the machine is four floating-point operations per processor cycle. The floating-point arithmetic operations that are not pipelined are square root (*fsqrt* and *fsqrts*) and divide (*fdiv* and *fdivs*). These operations can use either of the execution units and are assisted by additional logic to handle their numerical algorithms.

The POWER3 processor implements the optional PowerPC instructions *fres* (single-precision floating-point reciprocal estimate) and *frsqrte* (floating-point reciprocal square-root estimate). These are often useful for boosting performance in applications that do not need the full accuracy provided by the divide and square-root instructions (e.g., some graphic routines). These fast-estimate instructions also provide the seed values for iterative divide and square-root routines. The sPPM example in this paper describes software vector versions of these routines, which perform significantly faster than the hardware divide and square-root instructions.

The optional floating-point select instruction, *fsel*, is also implemented to provide for a floating-point conditional instruction with no branching. While this eliminates the chance of incurring the penalty for a mispredicted branch, the more significant advantage in eliminating the branch is the increased flexibility provided to the compiler in scheduling a group of instructions that includes an *fsel*.

The FPU also includes 32 64-bit floating-point registers and 24 64-bit physical rename registers or "buffers." All target results of floating-point load and arithmetic instructions are placed in rename registers until the instruction completes (i.e., until the completion stage of the instruction). This method of using rename registers is vital to executing out of order, executing speculatively, and breaking false register dependencies. However, stalls may occur at some point if all rename registers are allocated. The POWER3 processor optimizes the use of its floating-point rename registers, which consume a large piece of premium silicon area on the chip. Typically, for microprocessors which implement register renaming, rename registers are allocated when instructions are dispatched. While the POWER3 processor does allocate from its pool of 32 "virtual" rename registers during the dispatch cycle, it delays allocation of the physical rename registers until the cycle for which they are needed, typically the execute or finish stages. This technique makes better use of the physical rename registers and prevents them from becoming the source of a performance bottleneck. The result is that the POWER3 processor is able to sustain near-peak performance on key application kernels such as *rank-n update* and *matrix–matrix multiply*, which press the execution and completion rate of floating-point instructions to the maximum.

A central FPU instruction queue above the twin floating-point units can hold up to eight floating-point instructions, helping to maintain a steady flow of work for the FPU. The execution units can pull instructions from the queue in an out-of-order fashion, allowing logically later instructions whose operands are available to bypass other instructions which are waiting for operands. This flexibility provides a performance advantage when executing legacy code scheduled for other microarchitectures, or for variable delays such as stalls resulting from cache misses. In addition to out-of-order issue, out-of-order finish capability allows faster but independent instructions to bypass slower ones. A common example is an instruction stream containing a floating-point divide followed by a series of FMAs which are independent of the divide; while one execution unit is executing the divide, the other instructions execute and finish in parallel in the other execution unit.

- *Load/store execution units*

All loading and storing of data is handled by two load/store execution units. Load instructions transfer data from memory to a fixed- or floating-point rename register; store instructions do just the opposite, transferring data from a register to memory. (Since the POWER3 processor is cache-based, data from a load may be found in the L1 or L2 cache, or a cache-line transfer from memory may be initiated as a result of the load.) The performance of store instructions is enhanced by the presence of a store buffer, which is 16 entries deep. Store instructions can finish executing if they have obtained their data; they do not have to wait until the data is written into the data cache.

The POWER3 processor also implements load and store update form instructions, which update the general-purpose register containing the load or store address as part of the instruction, eliminating the need for a separate add instruction. Using load update forms allows the compiler to generate concise code which maximizes the computational work performed within the four-instruction completion rate per cycle. A common example is a matrix-vector multiplication, which primarily consists of two *lfdu* instructions and two *fma* instructions per cycle; this sequence will run at the peak flop rate of the processor when its data operands are contained in the L1 cache.

The two load/store execution units share a six-entry instruction queue. The out-of-order LSU also permits load instructions to bypass store instructions while keeping

track of any data dependencies that might exist, further enhancing performance and instruction scheduling flexibility. Order among store instructions is always maintained in both the execute stage and the store queue.

- *Data cache unit*

The data cache unit (DCU) consists primarily of the data memory management unit (MMU), the primary (L1) data cache, and the data prefetch unit.

Memory management unit

The MMU is primarily responsible for address translation of data requests. It includes a 16-entry segment lookaside buffer (SLB) and two mirrored 256-entry two-way set-associative data translation lookaside buffers (DTLB) in support of the dual load/store execution units. For translation misses, the instruction and data share hardware to search the page table for the correct translation. The MMU supports one terabyte (2^{40} bytes) of physical memory, 64-bit effective addressing, and 80-bit virtual addressing.

L1 data cache

The L1 data cache is 64 KB in size and is designed to provide single-cycle access to the load/store units. The data cache consists of 512 128-byte cache lines organized into four banks. Each bank is 128-way set-associative; that is, any line with the two address bits (A55/A56) set to select a particular bank may reside in any of the 128 cache slots in that bank. Within each bank, there are two sub-banks, one for all even doublewords and one for all odd doublewords (selected by address bit A60). The data cache can return the operands for two loads in the same cycle provided they are not both in the same bank *and* they are also not in the same even or odd doubleword sub-bank (i.e., if A55, A56, and A60 all match for two loads, a conflict exists). While 128-way set associativity provides performance advantages, it makes a least-recently-used (LRU) replacement algorithm impractical to implement. Hence, the POWER3 processor implements a round-robin replacement scheme for both the L1 instruction cache and the L1 data cache.

As data is transferred between memory and the processor, buses of various widths and frequencies are used. To provide a place to reconstruct a cache line, and to accommodate differences in frequencies and bus widths, linewidth buffers are often inserted in the transfer paths. Between the L1 data cache and the BIU, four cache-reload buffers (CRBs) are used to stage data into the L1 data cache (each data cache bank has a dedicated CRB). Outgoing data from the data cache is staged through four cache-storeback buffers (CSBs), one per bank. Load hits to data in the CRB are satisfied directly, rather than delayed until the cache line is reloaded into the L1 cache array. The CSB provides a 64-byte interface to the BIU that also facilitates a highly parallel and efficient DCU.

The data unit can support up to four outstanding cache misses, providing the basis for reducing the effective latency to memory. When a load misses the L1 data cache, the instruction is placed in a six-entry load-miss queue while the BIU fetches the data. Subsequent loads can continue to execute. If a subsequent load hits the cache, its data is returned immediately. If a subsequent load misses the cache, it is also placed in the load-miss queue. Since the BIU and memory subsystem can overlap the requests to memory for the missed cache lines, the average latency of memory per miss is reduced when there is more than one cache miss outstanding. Only after there are four outstanding misses in the load-miss queue and a fifth miss is encountered does load execution stall until one of the load instructions in the LMQ is serviced. Loads that hit the cache while there are four outstanding cache misses will continue to execute.

Data prefetching

One of the most effective and innovative features of the POWER3 processor is its hardware data prefetch capability. The POWER3 processor prefetches data by monitoring data cache-line misses and detecting patterns. When a pattern, herein called a *stream*, is detected, the POWER3 processor speculatively prefetches cache lines in anticipation of their use. With memory latency improving at a slower pace than processor cycle time, data prefetching is extremely advantageous in hiding the memory latency in order to achieve adequate bandwidth for data-hungry applications.

Prefetched streams have data storage patterns that reference consecutive cache lines, either in order of increasing addresses or decreasing addresses. It has been observed by the authors and by others [3] that a high percentage of data reference patterns in engineering/scientific applications conform to this pattern. Because of the economies of cache-based processors, new and rewritten applications give preference to such consecutive data access patterns. Even many so-called sparse data structures store the bulk of data in a stride-1 fashion, while the indirect addressing associated with the sparsity is contained within a cacheable region. Cache-miss patterns that are random or at a stride (in cache lines) greater than one (e.g., every fourth line) will not cause hardware prefetches, and attempting to prefetch the latter case would greatly increase the complexity of the hardware; such patterns are already handled adequately by the multiple outstanding miss capability of the POWER3 processor (since each access would be a distinct cache line).

The POWER3 processor prefetch engine includes a ten-entry stream filter and a stream prefetcher (**Figure 3**). The

purpose of the stream filter is to observe all data cache-line misses, in the form of real addresses (RA in Figure 3) and to detect possible streams to prefetch. The stream filter records data cache-line miss information; the real address of the cache line is incremented or decremented (depending upon the offset within the line corresponding to load operand), and this "guess" is placed in the FIFO filter. As new cache misses occur, if the real address of a new cache miss matches one of the guessed addresses in the filter queue, a stream has been detected. If the stream prefetcher has fewer than four streams active, the stream is installed, and a prefetch to the line anticipated next in the stream is sent out via the BIU. Once placed in the stream prefetcher, a stream remains active until it is aged out. Normally a stream is aged out when the stream reaches its end and other cache misses displace its entry in the stream filter.

When a stream is being prefetched, the prefetcher tries to stay two cache lines ahead of the current line (i.e., the line whose elements are currently being accessed by a load). The cache line that is one line ahead of the current line is prefetched into the L1 cache, and the line which is two ahead of the current line is prefetched into a special prefetch buffer in the BIU. Hence, the prefetch engine can concurrently prefetch four streams, each of which may be up to two lines ahead of the current line, for a total of eight prefetches per processor. The prefetch engine monitors all load addresses from the LSU (EA0 and EA1 in Figure 3). When the LSU finishes with the current line and advances to the next line (which is already in the L1 cache because of prefetching), the prefetch hardware transfers the line which is in the prefetch buffer to the L1 and prefetches the next line into the buffer. In this way, the prefetching of lines is automatically paced by the rate at which elements in the stream are consumed.

Bus interface unit

The bus interface unit (BIU) provides the interface between the processor bus and the other processor units: the instruction processing unit, the data cache unit, the prefetch engine, and the L2 cache. Its memory bus interface is 128 bits (16 bytes) wide and supports a variety of processor-to-bus clock ratios.

L2 cache

The POWER3 processor supports a private, either direct-mapped or set-associative, unified instruction and data secondary (L2) cache with sizes from 1 MB to 16 MB. The private bus to the L2 is 32 bytes wide, and cache-line transfers of the supported 128-byte line are performed in a burst operation of four cycles. For the 375-MHz RS/6000 44P Model 270, which runs the L2 interface at a ratio of 3:2 with the processor, this produces a burst rate of 8 GB/s. The 43P Model 270 also has a load-use latency

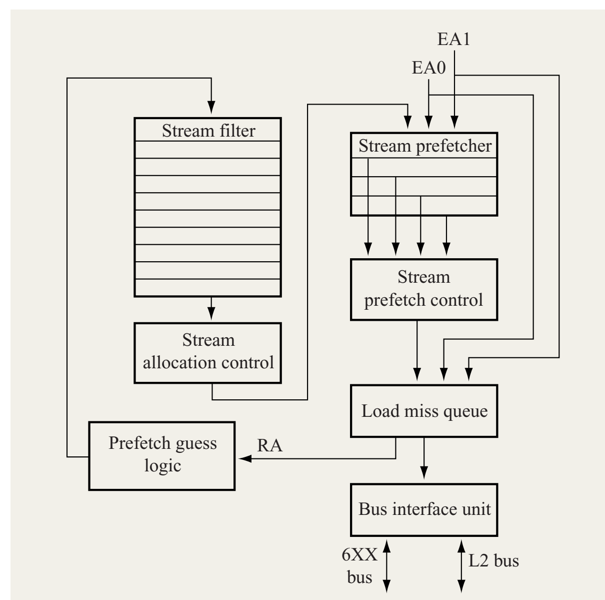

Figure 3

The prefetch engine.

(L1 miss, L2 hit instance) of approximately twelve cycles. The wide high-speed L2 cache interface provides ample bandwidth for processor requests as well as for snoop traffic from other processors or node controllers in the system.

Performance

While a complete characterization of the performance of the POWER3 processor is well beyond the scope of this paper, a good understanding of its performance can be gained by examining a set of basic loops and kernels, since even complicated applications can often be broken down into or related to more simple loops and kernels. In this section, this building block approach is taken, beginning with very basic loops and proceeding to some fundamental kernels. Finally, the sPPM application from the Accelerated Strategic Computing Initiative (ASCI) project is examined.

- *Fundamental loop-level performance*

Table 1 shows the measured performance for a set of fundamental loops. In all cases, the data is contained within the L1 data cache, and an outer repetition loop is used to obtain an accurate timing. The inner loop is often unrolled by the compiler to minimize branch instructions and to allow more latitude in scheduling instructions for maximum performance. To illustrate: Loop 2 before unrolling would be simply *lfd*, *stfd*, *bct*, and after unrolling four times would be *lfd*, *stfd*, *lfd*, *stfd*, *lfd*, *stfd*, *lfd*, *stfd*, *bct*.

Table 1 Performance of various fundamental loops.

ID	Kernel	Floating-point operations	Loads	Stores	Average no. of cycles per iteration
1	x(i)=s	0	0	1	1.0
2	x(i)=y(i)	0	1	1	1.0
3	x(i)=x(i)+s*y(i)	2	2	1	1.5
4	x(i)=x(i)+y(i)	1	2	1	1.5
5	s=s+x(i)	1	1	0	0.6
6	s=s+x(i)*y(i)	2	2	0	1.0
7	x(i)=sqrt(y(i))	1	1	1	11.0
8	x(i)=1/y(i)	1	1	1	9.2
9	s=s+y(i)*a(ix(i))	2	3	0	1.7
10	if(a(i).gt.s) s=a(i)	1	1	0	1.3
11	x(i)=a(i)+x(i−1)	1	2	1	3.0

All floating-point data is double precision (8 bytes). Inner loop count is 512.

Of course, unrolling requires that the compiler account for any iterations that are not a multiple of the level of unrolling.

Discussion of loops

1. Two stores per cycle can complete in bursts, allowing leeway in the scheduling of stores within a loop. However, for a solid stream of store operations, such as this loop, the 16-entry store queue will fill up, and the performance will be limited by the single store port into the L1 cache.
2. The copy loop contains one load and one store per element, which the POWER3 processor can execute in one cycle per iteration provided the data is in the L1 cache and there are no bank conflicts in the L1 cache.
3. This loop, commonly known as DAXPY, is load/store-bound like the first two loops, and again achieves the maximum rate possible for the POWER3 microarchitecture: (2 loads + 1 store)/2 (load/stores per cycle) = 1.5 cycles per iteration.
4. Loop 4 is identical to DAXPY, but without the multiply. Performance is still determined by the load/store rate.
5. This loop is the sum reduction of a vector. The compiler unrolls the loop by eight and produces eight partial sums. This breaks the interdependence among add operations and allows the computation to proceed at maximum rate until the partial sums have to be added to complete the loop. The final partial sum operations, plus the constraint on the instruction completion rate, add to the per-iteration timing.
6. This loop, commonly known as DDOT, is load-bound (there are two loads for every compound floating-point instruction). Since two loads per cycle can be completed, the loop executes in an average of one cycle per iteration.
7. Loop 7 shows the average performance of floating-point double-precision square root. Since both floating-point execution units can work in parallel, and a floating-point double-precision square root normally takes 22 cycles, the average time is 11 cycles. The loads are essentially free because they are entirely overlapped with the arithmetic instructions.
8. Loop 8 shows the average performance of floating-point double-precision divide. Again, two divides execute in parallel, halving the average time to approximately nine cycles.
9. Loop 9 is an indirect DDOT in which one of the vectors is indirectly addressed using a vector of integer indices. This is the crux of the sparse-matrix-vector multiply, often used in iterative equation solvers. The maximum possible rate for three loads and two load/store units with cache-contained data is 1.5 cycles per iteration. One additional cycle is required nearly every unrolled iteration, where the loop is unrolled by four (i.e., 0.2 cycles per original iteration), most likely due to a cache access interleave conflict.
10. Loop 10 shows the compare and branch performance of the POWER3 processor in selecting the maximum element of an array. Each floating-point compare and conditional store takes only slightly more than one cycle because of the branch-prediction capabilities of the processor.
11. This loop exposes the three-cycle dependent operation latency in the floating-point execution unit. Since each operation requires the result from the previous operation, the execution time is limited by the effective pipeline depth of three. Note that even though the actual pipeline depth is much longer than three, the data-forwarding capabilities within the floating-point execution unit keep the dependent operation latency to just the three essential arithmetic stages.

• *Memory performance and the STREAM benchmark*

The fundamental loops discussed in the previous section are such that all data required for the steady-state execution of the loops is contained in the L1 cache, thus demonstrating the *core* performance of the processor. The *memory* performance of the system is also important in judging the overall performance of a system. To reveal the memory performance of the system, the working set of data loaded and stored must exceed the size of the largest cache.

STREAM [4] is a standard benchmark which was designed to demonstrate sustained memory performance. It consists of four very simple loops in which all data is retrieved from memory. The loops are listed along with their results for an RS/6000 43P Model 260 (200 MHz) in **Table 2**. The bandwidth as reported by the STREAM benchmark credits doubleword stores as eight-byte transfers; however, this understates the actual memory traffic for a typical cache-based system. Stores cause a read–modify–write transaction, meaning that the cache line must be fetched from memory before the modified data can be written back to memory. As a result, for each doubleword store, eight bytes are fetched from memory and eight bytes are stored to memory. The actual memory traffic accounting for this is shown in the last column of Table 2. The performance level achieved on the STREAM benchmark is primarily a demonstration of the prefetching engine of the POWER3 processor. In the above measurements, the *prefetch_by_load* compiler directive, available in Version 6.1.0.3 of the XL Fortran Compiler, is used in the source code to improve the performance (which requires that the results be placed in the *Experimental/Nonstandard Results* section of the STREAM repository). The directive inserts a dummy load (with the same address as the store) in the instruction stream prior to the store. This allows the prefetch engine, which monitors load misses, to detect the miss and prefetch the cache line, thus improving performance. XL Fortran Version 7.1 automatically generates these dummy loads at optimization level O4. Less than 40% of the stated performance could be attained without prefetching.

Table 2 STREAM benchmark performance.

Name	*Kernel*	*RS/6000 43P Model 260 Results* (MB/s)	
		*Reported**	*Actual*
COPY	c(j)=a(j)	942	1,413
SCALE	b(j)=s*c(j)	985	1,477
SUM	c(j)=a(j)+b(j)	1,096	1,461
TRIAD	a(j)=b(j)+s*c(j)	1,103	1,470

*Listed in the *Experimental/Nonstandard Results* section of the STREAM repository, since the code was compiled with a compiler directive.

• *Matrix–matrix multiply*

One of the most ubiquitous application kernels is the matrix–matrix multiply, or, more generally, the rank-n update. Mathematically this is, in matrix notation,

$C \leftarrow C + AB$ (rank-n update);

or, using indicinal notation,

$c_{ij} = c_{ij} + a_{ik}b_{kj}$,

where a repeated index implies summation. If implemented in source code in the most straightforward manner, this kernel consists of a three-nested loop structure, with a DAXPY as the innermost loop. When optimized for cache-based superscalar processors, however, the code often becomes a six-level nested loop structure with the innermost loop consisting of multiple DDOTs [5]. Two key optimizations are gained in this transformation. First, unrolling the loops associated with the product indices i and j four times (so called 4×4 unrolling), and making the loop associated with the index k the innermost loop, produces an inner kernel consisting of 16 *fma* instructions, eight *lfd* instructions, and one *bct* instruction. This transformed code reduces the average storage reference ratio to just one half *lfd* per *fma*, compared with the standard DAXPY implementation of one *stfd* and two *lfd* instructions per *fma*. The maximum execution rate on a POWER3 processor for the transformed code is thus limited by the 16 *fma* instructions. Hence, with two floating-point execution units, it would take a minimum of eight processor cycles to execute, achieving the peak rate of the machine of four floating-point operations per cycle. In fact, for a completely L1-cache-contained case, the POWER3 processor takes 8.06 cycles per iteration to perform this loop, coming as close as is practically possible to achieving the peak rate (that is, one extra cycle for every 16 iterations, or 129 cycles instead of a perfect 128 cycles, probably due to an L1 interleave conflict). Achieving this rate uses all of the physical rename registers and fully exploits the allocation/deallocation optimizations of rename registers discussed in the FPU section.

The second important optimization is cache blocking, which reduces the number of times the data must be brought into the caches to complete the computation, allowing even very large arrays to be computed at close to the peak core rate. Cache blocking effectively breaks up the matrix–matrix multiply into multiple matrix–matrix multiplies on subregions of the original matrices. Each of these smaller products has one matrix operand that fits effectively into the cache. This type of cache blocking also lends itself easily to parallel implementations.

Cache blocking and loop unrolling are implemented in DGEMM, an optimized general matrix multiply routine in

Table 3 Kernel performance for the RS/6000 44P Model 270 (375 MHz).

Kernel	*Size* (rows × columns)	*Size* (KB)	*No. of gigaFLOPS*
DGEMM	50 × 50	20	1.22
DGEMM	100 × 100	78	1.31
DGEMM	500 × 500	1,953	1.27
DGEF/DGES	500 × 500	1,953	1.17
DGEF/DGES	2000 × 2000	31,250	1.11
DGEF/DGES	5000 × 5000	195,313	1.15

Table 4 Linpack $n = 1000$ performance for the RS/6000 SP 375-MHz SMP Thin Node parallel processor.

	No. of gigaFLOPS	*Theoretical peak* (GFLOPS)
Single-threaded	1.2	1.5
Two-way parallel	2.2	3.0
Four-way parallel	3.7	6.0

the IBM Engineering Scientific Subroutine Library [6]. **Table 3** shows the performance on the RS/6000 44P Model 270 for various sizes of arrays. Note that actual performance achieves a high percentage of the peak performance, the latter being 1.5 GFLOPS for a 375-MHz POWER3 processor on matrices ranging from small (20 KB) to large (200 MB), demonstrating the total effectiveness of the microarchitecture.

The rank-n update is computationally the largest piece of work required to factorize a matrix, a task often employed in technical computing applications. Again, unrolling and blocking are important in achieving optimal performance on the POWER3 processor. Table 3 also shows the performance of DGEF and DGES (factor then solve) for a variety of matrix sizes. The performance of a 1000 × 1000 matrix (reported on the *Performance Database Server* site [7] for the "$n = 1000$" column of Linpack Performance), is often used as an indication of the technical computing performance of a system. **Table 4** lists the published performance of the RS/6000 SP 375-MHz SMP Thin Node parallel processor. Note that the efficiency, defined as the ratio of actual performance to theoretical peak performance, falls off as more processors are employed in the solution. This is because, for a given problem size ($n = 1000$), the percentage of the solution that runs in parallel remains fixed. The parallel piece speeds up when more processors take part in the computation, but the serial piece of the computation still requires the same amount of computing time. Hence, the efficiency declines; this is often referred to as Amdahl's law [5].

Another variant of the Linpack benchmark reported on the *Performance Database Server* site is the Linpack-Parallel benchmark. The rules of this benchmark differ from the "$n = 1000$" benchmark in that the matrix size to be factored and solved can be any size; thus, the ratio of parallel to serial computation can be kept large. On the ASCI White system (further described below), IBM measured 4.938 teraflops per second (TF) on a matrix of size 430 000 × 430 000 and a 464-node partition to set a new world record for this benchmark.

- *Example application: sPPM*

ASCI includes several milestone projects, including a three-teraflop-per-second (TF) ASCI Blue system in 1998, a 10-TF ASCI White system in 2000, and 30-TF and 100-TF systems in the near future. The ASCI White system uses the sixteen-way 375-MHz POWER3 SMP High Node as a building block for the 512-node IBM SP supercomputer that includes six trillion bytes (TB) of memory and 160 TB of IBM disk storage. The key performance benchmark for ASCI White is the sPPM (simplified Piecewise Parabolic Method) code of Woodward et al. [8]. IBM began early POWER3-targeted tuning of sPPM in 1997. The information gained from this tuning effort was invaluable to IBM's software developers seeking to optimize the code produced by IBM compilers for POWER-based systems.

Most of the POWER3 processor tuning falls into two categories: exploitation of new instructions and exploitation of the hardware prefetch mechanism. The new instructions important to sPPM are the estimate instructions and the select instruction.

The key routines in the sPPM application include a sufficient number of divides and square-root operations to produce a significant impact on the overall execution time. The first optimization step involved the hand distribution of loops to allow separation of these slow operations into dedicated loops, and then replacement of these loops with calls to IBM MASS [9] routines which accept "vector" arguments and length counts. While not advantageous for a single divide or square-root operation, software emulation of the steps performed by the divide/square-root hardware often provides a performance opportunity when ten or more operations are involved. Interleaving operations to exploit the inherent independence between distinct divide operations provides parallelism which is not present between the steps of a given divide operation.

The POWER3 implementation of the *fres* and *frsqrte* instructions allows an efficient seed guess for the start of an iterative algorithm. Using POWER3-specific "vector" routines, for lengths of 256 single-precision elements, when the arrays are cache-resident and outer loops are used for timing purposes, the per-element timing for a divide operation becomes 2.7 cycles (versus 9 cycles in

hardware). The per-element timing for square root becomes 4.9 cycles (versus 11 cycles in hardware). (Since the *frsqrte* instruction is double-precision, it is not used in generating the estimate for the vector single-precision square-root routine.) The XL Fortran Version 7.1 compiler automatically generates vector code at optimization level O4 for the following operations: divide, square root, log, exponent, reciprocal square root, sin, cos, and tan.

The POWER3 implementation of the select instruction *fsel* provides a significant performance advantage on sPPM, particularly on the large number of max( ) and min( ) operations. The select instruction takes three input operands, and depending on the sign of the first input operand, assigns either the second or third input operand to the result register. As a result, the compiler can replace the generic COMPARE/BRANCH/ASSIGN sequence for min/max with a SUBTRACT/SELECT pair, reducing the instruction count and often increasing performance as a result. [In the case of "if(a.gt.b*c)", the compiler fuses the multiply instruction and the new subtract instruction into a compound instruction, so the subtract operation is free.] However, the more obvious performance advantage of using *fsel* instructions comes from the elimination of a potential mispredicted branch and the resulting pipeline stalls. In many cases, the random relationship among floating-point values causes branch-prediction mechanisms (such as a branch history table) to be defeated, resulting in a high branch-misprediction average. Furthermore, the delay between floating-point compares and branch resolution is often greater than in the fixed-point case (because of relatively longer floating-point pipelines); hence, branch-misprediction penalties are higher. The use of a SELECT operation eliminates the possibility of a mispredicted branch for MIN/MAX operations. A more subtle, but significant, advantage which results from the replacement of the conditional branch by the select instruction is that a greater degree of instruction scheduling flexibility often results, leading to additional performance gains.

An opportunity identified in the sPPM application occurs in low-reuse cache-blocking efforts. In this case, the algorithm was a 2D out-of-place transpose written in Fortran (which stores arrays column-wise). The transpose operation, in indicinal notation, is simply

$$c_{ij} = b_{ji} .$$

In the straightforward coding of a transpose operation, one array is accessed stride-1 (handled well by the prefetch mechanism), while the other array is stride-n (the leading dimension of the array). In typical blocking attempts, the array is partitioned into cache-resident sub-blocks, so that misses occur along the top of the stride-n array sub-block during the initial row access, with the blocking advantage coming from cache hits for subsequent rows. The blocking is performed to reduce the number of columns in the sub-block in order to reduce thrashing, so that the subsequent rows are accessed while they are still in cache. Often, using the processor's registers, a mini-transform on an even smaller sub-block is used to allow all loads and stores to be part of short stride-1 strings of accesses.

However, this last step is not optimal for the POWER3 processor. Arranging the accesses for a 4 × 4 mini sub-block (to fill 16 registers) as four sets of four-element stride-1 accesses often results in a miss followed by three more misses to the same cache line. On the POWER3 processor, these redundant misses fill the load miss queue, inhibiting further load processing. As a result, the misses for adjacent columns are never overlapped. The optimal approach for the POWER3 processor uses less aggressive blocking and exploits the hardware prefetch mechanism to fetch the predominately stride-1 array while purposely clustering the misses in the stride-n direction. This particular arrangement of the accesses in the transposed direction results in four misses which are sent to memory almost in parallel, thereby decreasing the average miss penalty. Cache hits then result for the remaining elements upon return of the four cache lines.

Several other compiler improvements center around the POWER3 mechanisms for handling cache misses, either natural overlap (exploiting multiple outstanding misses) or hardware-initiated prefetch. Current work includes consideration of the hardware stream mechanism during loop distribution and fusion. For example, a two-statement loop with seven streams can often be split, resulting in two loops, each with four streams or fewer. In the original loop, four streams will be prefetched and three streams will not be prefetched. Once split, operands in each loop may be prefetched. For loops with more than ten streams, loop distribution can prevent the thrashing of the stream filter.

As a result of the tuning described above, the sPPM application increased its average performance by approximately 25% to achieve 3.8 TF on 495 nodes of the ASCI White system. This remarkable performance is a significant milestone in the Department of Energy's goal to simulate the aging and operation of a nuclear weapon.

Conclusion

The POWER3 processor is a complex but efficient superscalar microprocessor which excels at the varied demands of technical computing. It provides excellent performance using advanced features that maximize instruction-level parallelism and minimize delays due to memory latency. Designed with the demands of challenging engineering/scientific applications in mind, the POWER3 processor proves itself on key application

benchmarks, exhibiting high data bandwidth and sustained floating-point performance as well as flexibility required to handle the varied demands of modern computing. The POWER3 processor provides a solid, high-frequency building block for SMP system products, both as stand-alone and as IBM SP nodes in switch-connected massively parallel supercomputers.

Acknowledgments

The authors wish to thank Mike Mayfield, Steve Tung, and Dwain Hicks for helping us to understand the details of the POWER3 implementation. We also thank Frank Johnston for his assistance in measuring the Linpack performance, Bruce Curtis of the Lawrence Livermore National Laboratory for his work on sPPM optimization, and Roch Archambault for his compiler enhancements in the area of vector-intrinsic code generation.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, C. May, E. Silha, R. Simpson, and H. Warren, Eds., Morgan Kaufmann Publishers, Inc., San Francisco, 1994.
2. Steve White and John Reysa, *PowerPC and Power2: Technical Aspects of the New IBM RISC System/6000*, First Edition, IBM Order No. SA23-2737-00, 1994, available through IBM branch offices.
3. T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.
4. STREAM: Sustainable Memory Bandwidth in High Performance Computers, *http://www.cs.virginia.edu/stream/*.
5. Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, Society for Industrial and Applied Mathematics, Philadelphia, 1991.
6. *Engineering and Scientific Subroutine Library for AIX, Guide and Reference*, IBM Order No. SA22-7272, 1998; available online at *http://www.rs6000.ibm.com/resource/aix_resource/sp_books/essl/*.
7. LINPACK, *http://performance.netlib.org/performance/html/PDSbrowse.html*.
8. The sPPM benchmark code README file, *http://www.lcse.umn.edu/research/sppm*.
9. IBM Mathematical Acceleration SubSystem, *http://www.austin.ibm.com/resource/technology/MASS/*.

Received February 5, 2000; accepted for publication October 11, 2000

Frank P. O'Connell *IBM Enterprise Server Group, 11400 Burnet Road, Austin, Texas 78758 (oconnell@us.ibm.com).*

Mr. O'Connell is a Senior Technical Staff Member in the Future Processor Performance Department, where he has been a member of the RS/6000 high-performance processor development effort since 1992. For the past 15 years, he has focused on scientific and technical computing performance within IBM, including microprocessor and systems design, operating system and compiler performance, algorithm development, and application tuning, in the capacity of both product development and customer support. Mr. O'Connell received a B.S.M.E. degree from the University of Connecticut and an M.S. degree in engineering–economic systems from Stanford University.

Steven W. White *IBM Enterprise Server Group, 11400 Burnet Road, Austin, Texas 78758 (white@austin.ibm.com).*

Dr. White is a Distinguished Engineer in the IBM Enterprise Server Group. He received B.S., M.S., and Ph.D. degrees from Texas A&M University, where he also taught for the Electrical Engineering Department for three years. In 1982, he joined IBM to work on mainframe scientific and engineering processor development, architecture, and system design. He had a two-year assignment with the computational physics group at the Lawrence Livermore National Laboratory. Dr. White has authored or co-authored more than two dozen papers in a variety of areas including VLSI design, parallel processing, numerical algorithms, code optimization, system design, performance analysis, human genome research, and compilers. He was the primary editor for *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*. He has been granted twenty-two U.S. patents and has six pending applications. Dr. White was a recipient of the 1999 Gordon Bell Award for "highest sustained performance." He has also received an IBM Corporate Award, an IBM Outstanding Technical Achievement Award, two IBM Division Awards, an IBM Outstanding Innovation Award, seven IBM Invention Awards, three IBM supplemental patent awards, and several informal awards. He is a registered professional engineer.