Selected papers from:

IBM RISC System/6000 Technology: Volume II

Sept. 23, 1993

Prior to First Edition 1993

Copies of this book might be obtainable from the local branch office. When the book is complete, IBM employees should be able to order copies from Mechanicsburg.

NOTE: These selected papers are being published here prior to the final editing process. Some additional changes may occur prior to publication in the complete version of *IBM RISC System/6000 Technology: Volume II*.

Please direct comments regarding the clarity, grammar, correctness, or audience of this early release to the editors at:

VNET: WHITE at AUSTIN
Internet: steve@hwperform.austin.ibm.com
Mail: Steve White, Zip 9440, IBM,
11400 Burnet Road, Austin, Tx 78750

Trademarks

The following trademarks apply to this book. They are listed in alphabetically order, not in the order in which they appear in this book.

- AIX is a registered trademark of International Business Machines Corporation.
- AIX/6000 and AIXwindows are trademarks of International Business Machines Corporation.
- Alpha is a trademark of Digital Equipment Corporation.
- ANSYS is a registered trademark of Swanson Analysis Systems.
- Apollo is a trademark of Apollo Computer, Inc.
- Apple and Macintosh are registered trademarks of Apple Computer Corporation.
- AXP is a trademark of Digital Equipment Corporation.
- C Set ++ is trademark of International Business Machines Corporation.
- Cray is a registered trademark of Cray Research, Inc.
- DB2/6000 is a trademark of International Business Machines Corporation.
- DEC and DECstation are registered trademarks of Digital Equipment Corporation.
- Ethernet is a trademark of XEROX Corporation.
- graPHIGS is a trademark of International Business Machines Corporation.
- Groupe Bull is a trademark of Groupe Bull.
- HP is a registered trademark of Hewlett–Packard Company.
- IBM is a registered trademark of International Business Machines Corporation.
- IBM Shared Memory System is a trademark of International Business Machines Corporation.
- IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers.
- InfoExplorer is a trademark of International Business Machines Corporation.
- Intel is a registered trademark of Intel Corporation.
- Internet is a registered trademark of Internet, Inc.
- IRIS Graphics Library is a trademark of Silicon Graphics, Inc.
- KAP is a trademark of Kuck and Associates, Inc.
- Kaleida is a trademark of Kaleida.
- MECHANICA is a registered trademark of Rasna Corporation.
- Micro Channel is a registered trademark of International Business Machines Corporation.
- Microsoft is a registered trademark of Microsoft Corporation.
- MIPS is a registered trademark of MIPS Computer Systems, Inc.
- MIT is a registered trademark of Massachusetts Institute of Technology.
- Motorola and Motorola 88110 are trademarks of Motorola, Inc.
- MSC is a registered trademark of The MacNeal–Schwendler Corporation.
- NASTRAN is a registered trademark of the National Aeronautics and Space Administration.
- Network File System is a trademark and NFS is a registered trademark of Sun Microsystems, Inc.
- OpenGL is a trademark of Silicon Graphics, Inc.
- OSF, Motif, and OSF/Motif are registered trademarks of Open Software Foundation.
- Perfect Benchmarks is a trademark of the Center for Supercomputing Research and Development, Univ. of Ill. at Urbana–Champaign.
- POWER/4, POWER Architecture, POWER2, POWER, PowerOpen, POWERstation, POWERserver, POWER GXT100, POWER GTX150, RGB 530, PowerPC, PowerPC Architecture, PowerPC 601, PowerPC 603, PowerPC 604, PowerPC 620, Presentation Manager, and RISC System/6000 are trademarks of International Business Machines Corporation.

- PS/2 is a registered trademark of International Business Machines Corporation.
- Silicon Graphics is a trademark of Silicon Graphics, Inc.
- SPEC, SPECmark, SPECfp92, and SPECint92 are trademarks of Standard Performance Evaluation Corporation.
- Spectrum is a trademark of the CENTRIC Engineering Systems, Inc.
- Taligent is a trademark of Taligent.
- TPC Benchmark, TPC–A, TPC–B, and TPC–C are trademarks of the Transaction Processing Performance Council.
- UNIX is a registered trademark of UNIX Systems Laboratories.
- VAST is a trademark of Pacific–Sierra Research Corporation.
- X Window System is a registered trademark of Massachusetts Institute of Technology.

Foreword

In February 1990, IBM introduced the RISC System/6000 product line. The announcement raised the performance bar beyond our customers' and competitors' expectations. Along with the initial announcement, IBM provided a collection of related papers in the form of a book – *IBM RISC System/6000 Technology*. This popular book became the de facto reference guide for the RISC System/6000 family.

Over the years, the product line was enhanced with faster processor clocks, larger caches, more memory, new AIX and compiler releases, and new levels of integration and function of adapters.

Over time, even the performance metrics used to compare the systems became obsolete. At the time of the first announcement, the Drystone benchmark was popular. Gradually, more robust benchmarks, such as the SPECfp89 and SPECint89 suites, became the prominant industry standard comparisons. Even those suites have recently been replaced by the SPEC CFP92 and CINT92 suites.

In the fall of 1993, IBM announced new POWER2–based and PowerPC–based systems, new graphics capabilities, and new software capabilities. In an effort to update the book to cover the new offerings, as well as the changes over the years, this second volume of the RISC System/6000 Technology book has been created.

In the tradition of the original volume, this book contains the works of many RISC System/6000 developers. Its primary purpose is to provide background and insight into the RISC System/6000 family. It is intended to convey technical information about RISC System/6000 products which is not otherwise generally available.

We thank Phil Hester, Mary Ann Messina, and others for sharing their thoughts which helped guide the selection of topics. We are grateful to the authors, who in addition to bringing exceptional products to market, found time in their schedules to provide the reader with their view of the products.

In an effort to make the information available as quickly as possible, thirteen of the papers are being made available in this "condensed" version of the second volume. The table of contents includes papers for the complete volume; page numbers are included for papers which are part of this condensed version.

Article reviewers were selected both to provide a technical review and to improve overall readability. We thank Kate Stewart, Maurice Franklin, Jack Kemp, Michael Amerson, and Donald Tsang for providing many useful comments which have greatly improved this condensed version of the book.

In some cases, papers sumbitted to the IBM Journal of Research and Development covered areas which are key to this book. We thank the Journal editors for granting us permission to reprint some portions of the submissions covering POWER2.

We thank Phil Hester, Chuck Michalik, and Rich Anderson for supporting this effort and for providing the technical community a forum in which to present their material.

Steve White and John Reysa, Editors

IBM RISC System/6000 Technology: Volume II

Contents

POWER2: Next Generation of the RISC System/6000 Family

Steven W. White and Sudhir Dhawan

Introduction

In 1990, IBM announced the RISC System/6000 (RS/6000) family of highly concurrent superscalar workstations and servers supporting clock rates ranging from 20 MHz to 30 MHz [1]. All models included an 8KB instruction cache and either a 32KB or 64KB data cache. The performance levels achieved by the 25 MHz model 530 exceeded that of many of its contemporaries (Sun 4/200, DECstation 3100, Apollo DN10000, and MIPS M/2000) by more than 40% on a variety of benchmarks (Dhrystones 1.1, Whetstones, Linpack(dp), Livermore Loops (geometric mean) and SPECmark) [2]. This first implementation of the IBM POWER Architecture is referred to as POWER.

Over the years, incremental improvements to the initial offerings have been made. The cycle time has been increased to 62.5 MHz on desktop, desk–side, and rack systems. A 32K–byte instruction cache is standard on more than ten models. Additional compiler capability, especially in the area of restructuring data access patterns, has improved benchmarks and customer codes. Changes in the I/O area have increased Micro Channel bandwidth from 40MB/S to 80 MB/S peak.

While these changes were taking place, the competition also improved. A dichotomy in design philosophies became apparent. The RS/6000 systems and compilers aggressively exploit superscalar capabilities. This philosophy is shared in part by other designs such as Sun SuperSPARC and Motorola 88110. These superscalar capabilities involve not only multiple functional units but also hardware complexity which allows the units to function relatively autonomously. Some argue that the complexity makes high clock rates difficult to achieve, and that more performance can be achieved by clock rate than by aggressive instruction–level parallelism. Examples of this alternative philosophy are the DEC 21064 (a.k.a. Alpha), the HP PA7100, and the MIPS R4000. The debate about the advantages of each approach appears on electronic forums and in articles and editorials. The popularity of the topic has lead to the coining of catchy synonyms for the two approaches such as the ”Speed Demons” (high clock rate) versus the ”Brainiacs” (complexity) [3].

While it is desirable to pursue both approaches, the goals are often in conflict [4]. For a given technology there is likely to be sets of clock rate / instruction–level parallelism pairs which provide near optimal performance. Although a comparison is also clouded by many factors including compiler optimization and chip and system designer abilities, measurements on implementations is the generally accepted method of judging the trade–offs. Benchmarks clearly illustrate that the optimal design point is very application specific.

By tracking performance of various systems for the past few years, two points become apparent. First, performance improves at a healthy pace in the workstation and server markets; without continual improvements, leaders lose their position. Second, performance by vendor is a stair–step function. Often the competition is close, with several vendors jockeying for the lead position. While at any instant there may be a dominant system, leaders change frequently.

This paper describes the next generation of implementations of the POWER Architecure, POWER2 processors and systems. The

This paper has been submitted to the IBM Journal of Research and Development.

initial three models include the 55 MHz Model 58H, the 66.5 MHz Model 590, and the 71.5 MHz Model 990. With the arrival of POWER2–based systems, the RS/6000 family again moves into the lead on many industry standard benchmarks with a combination of increased clock rate, exploitation of architectural enhancements, doubled functional units, and increased cache capacity. The enhanced superscalar capability of POWER2 further widens the gap between instruction–level parallelism and clock rate approaches.

This paper consists of four major sections. The architecture section discusses enhancements to the programmer's view of the hardware, primarily new instructions which improve storage reference bandwidth, allow hardware square root, and speed floating–point to integer conversion. The implementation section includes a description of the POWER2 processor including the functional units, caches, and Translation Look–aside Buffers (TLBs.) The third section describes the fabrication technology aspects of system. The performance section examines how these changes affect performance and compares the resulting POWER2 performance to that of several competitive systems on a variety of industry standard benchmarks. The performance results demonstrate that superscalar capabilities are an attractive alternative to aggressive clock rates.

Architecture

The RS/6000 systems are implementations of a Reduced Instruction Set Computer (RISC) architecture. As is characteristic of many RISC architectures, loads and stores provide the only storage access; arithmetic instructions use only register operands. Several additions to the classical RISC architecture have been made to enhance performance opportunities. Three of these are 1) Floating–Point Multiply–Add (FMA), 2) Branch on CounT (BCT), and 3) update forms of storage references.

FMA is a compound instruction consisting of a floating–point multiply and a dependent add. On most RS/6000 implementations, the FMA operation performs the multiply and the dependent add with a total latency of only two cycles. Independent FMA instructions can start every cycle. The FMA operation, which allows a peak MFLOPS rate equal to two times the MHz rate while using a single functional unit, is credited by many as a key component of the RS/6000's outstanding floating–point performance. The HP PA7100 has a similar compound operation which allows a floating–point multiply and an <u>independent</u> add. The FMA in the POWER architecture combines a multiply and a <u>dependent</u> add. Simple coding of common constructs such as inner product or "**daxpy**" involve dependent pairs of operations.

The BCT form of a conditional branch allows a special purpose register, the count register, to be decremented and tested to determine the outcome of the branch. The count register can be loaded with an iteration count for a loop. Often a loop closing branch can be coded using the BCT form; the count register can be decremented and tested entirely within the branch unit without involving the fixed–point unit. In many other architectures, the iteration count would be held in a General Purpose Register (GPR) and the fixed–point unit would perform the decrement and test. The test results would be forwarded to the branch unit in the form of a condition code result. The BCT instruction and the count register are examples of architectural separation of resources which enhance the implementer's ability to exploit instruction–level parallelism. The loop count decrement operation can be off–loaded from the FXU and the branch unit can accurately determine the fetch path without FXU synchronization.

Both addressing forms of storage references, indexed and displacement, support an "update form". This pre–update of the base register (with the effective address) greatly decreases the need for explicit address arithmetic. The multiple operations which comprise each of the FMA, the BCT, and the update forms allow the designer the opportunity to provide instruction–level parallelism beyond the number of functional units and dispatch bandwidth.

POWER2 supports a super–set of the original architecture. Instructions have been added to increase performance opportunity: quad–word floating–point storage references, square root, and convert to integer. Virtual address translation has been modified to improve performance and to add capability. Hardware performance monitoring has also been added to the architecture. Upward compatibility for programs is maintained.

New Instructions

A class of high–performance floating–point storage access instructions have been added. The Load Quad and Store Quad instructions include all addressing forms previously supported for double–precision storage references: indexed and displacement, with and without update forms. These quad–word (128 bits) accesses move two adjacent double–precision storage operands into two adjacent floating–point registers.

Due to the implicit register updates available in storage reference instructions and the BCT branch, most RS/6000 floating–point loops simply consist of storage references, floating–point arithmetic, and a branch. The fixed–point units (which executes all storage reference instructions) and floating–point unit operate fairly autonomously. Therefore, the number of cycles required to execute an iteration of a floating–point loop is usually limited either by the number of storage references or the number of arithmetic instructions. When the performance of a loop is limited by storage references, the Load Quad and Store Quad instructions can provide improvement.

The Linpack benchmark [5] shown in Figure 1 illustrates the quad–word benefit. The top code block represents the inner loop after inlining but without unrolling. The pseudo–assembly code shows three storage reference instructions, the performance limiter for this code on a POWER processor.

The bottom code block represents the code after unrolling (by a factor of two) the stride–1 inner loop. The pseudo–assembly code shows three pairs of storage references. In each pair, two adjacent storage locations and two adjacent floating–point registers are involved. A quad–word reference can replace each pair, resulting in roughly the same number of cycles per iteration as in the upper loop. However, the number of floating–point operations per iteration has doubled. Since the total number of iterations is halved, quad–word storage reference capability almost doubles the performance on the Linpack benchmark.

In addition to the quad–word storage reference instructions, a hardware Square Root instruction has been added. On previous RS/6000s, the square root function is provided by a library routine call. By replacing the call with a single instruction, the number of cycles per operation drops from about 50 to roughly 25. In the SPEC CFP92 suite, hardware square root provides a substantial gain on the ORA benchmark, which spent about 50% of its time in the library square root routine. Application areas which will exhibit performance gains from Square Root instruction include computational physics and graphics.

```
DO 100 i=1,n                          loop:  lfd   fpr2,8(r11)          # a(i,j)
  a(i,j) = a(i,j) + a(i,k)*atemp             lfdu  fpr4,8(r12)          # a(i,k)
 ENDDO                                       fma   fpr6,fpr2,fpr4,fpr8  # fpr8=atemp
                                             stfdu fpr6,8(r11)          # a(i,j)
                                             bct   loop

       ******************************************************************

                                     loop:   lfd   fpr2,8(r11)        # a(i,j)
                                             lfd  fpr3,16(r11)  # a(i+1,j)
DO 100 i=1,n,2                               – – – – –
 a(i,j) = a(i,j) + a(i,k)*atemp              lfd  fpr4,8(r12) # a(i,k)
 a(i+1,j)=a(i+1,j) + a(i+1,k)*atemp          lfdu  fpr5,16(r12) # a(i+1,k)
ENDDO                                        – – – – –
                                             fma  fpr6,fpr2,fpr4,fpr8   # fpr8=atemp
                                             fma  fpr7,fpr3,fpr5,fpr8
                                             – – – – –
                                             stfd  fpr6,8(r11)           # a(i,j)
                                             stfdu  fpr7,16(r11)         # a( i+1,j)
                                             bct  loop
```

Figure 1 Quad–word storage reference benefit on Linpack benchmark

Additional new instructions allow more efficient conversion of a floating–point value to an integer value. FCIR and FCIRZ provide the conversion with default rounding and with round toward zero, respectively. They improve random number generation where the seed is a floating–point value but the modulo arithmetic calculations require integer inputs. Table look–up approaches, which input a floating–point value but need an integer offset to index a table, provide another example. During interpolation, floating–point to integer conversion can be used to determine the two adjacent grid points (integer indices) which surround a calculated point on a grid. The Fortran INT intrinsic function can use FCIRZ.

Enhanced Translation

In addition to the newly added instructions, users obtain a performance gain from the modifications to the virtual address translation process. As in most virtual memory systems, the operating system manages a set of architecturally defined pages tables which maintain a mapping of virtual pages to real pages. To increase the efficiency of this process, translation information for recently accessed pages is cached in Translation Look–aside Buffers (TLBs). When a storage reference requires translation information which is not available in the TLBs, a TLB miss occurs and hardware searches the page tables for the translation information and either updates the TLBs or signals that a page fault has occurred.

The POWER hardware TLB miss search consists of hashing bits from the virtual address, indexing a Hash Anchor Table (HAT) to obtain a pointer into the Page Frame Table (PFT), and following the pointers through a chain of PFT entries to find a match. Since the translation information is non–cacheable, this process requires a minimum of two cache misses per TLB miss, one for the HAT entry and one for the PFT entry. Since an inverted page table is used, entries along a PFT chain tend to be scattered through the table; each additional search step along a PFT chain usually incurs an additional cache miss. Another attribute of an inverted page table is that only one virtual page can be mapped to a given real page. Support for aliasing at the page level results in ping–ponging of PFT entries.

The POWER2 translation scheme, which is described in more detail in [6], essentially places the translation information of the first PFT entry, pointed to by the HAT entry, into the HAT entry, reducing the number of storage references. The POWER2 PFT entries are cacheable. To exploit the cacheability of the PFT entries, the structure of the PFT is defined so that eight adjacent entries in the PFT (which fit in a cache line) contain entries from the same PFT chain. As a result, the table walk associated with a TLB miss often requires at most one cache miss, shortening the translation process. The new definition of the PFT structure (away from an inverted PFT) simplifies page level aliasing.

Performance Monitor

While the above architectural enhancements aim at improving performance, a set of performance monitors are added to observe many processor aspects which affect performance. This facility allows the measurement of instruction/data cache/TLB misses, functional unit utilization, instruction distribution by functional unit, and number of instructions executed from a particular class or group. The performance monitor facility is described in [7]. This facility provides a wealth of information which allows detection of performance bottlenecks. Examples of use of this monitor facility on applications are shown in [8].

Implementation

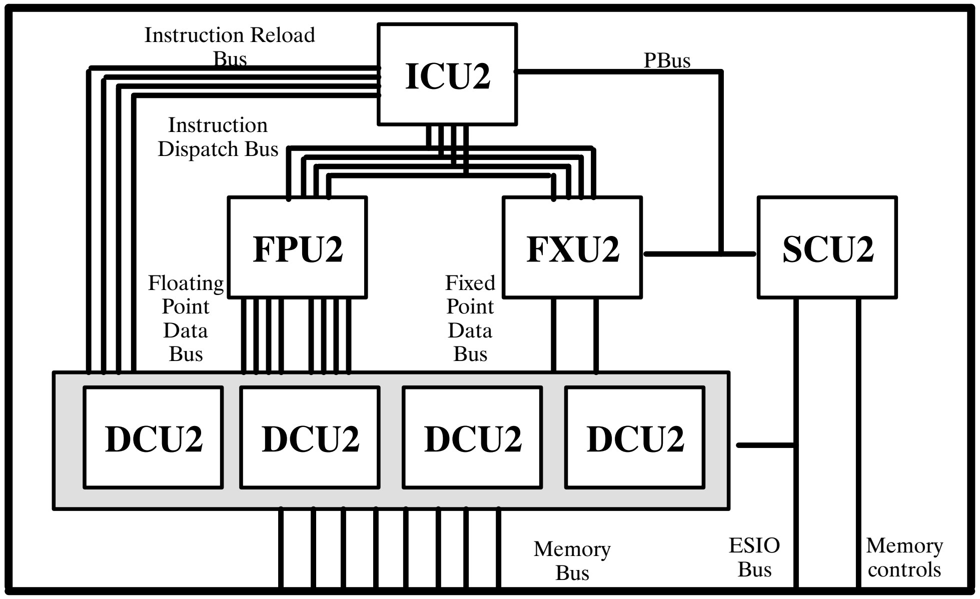
As shown in Figure 2, the POWER2 processor complex consists of eight semi–custom chips partitioned in a fashion similar to POWER: an Instruction Cache Unit (ICU), a Fixed–Point Unit (FXU), a Floating–Point Unit (FPU), four Data Cache Units (DCU) and a Storage Control Unit (SCU). The FXU and FPU each contain two execution units. To support additional execution units, the memory, instruction reload, and instruction dispatch buses, as well as the interface from the DCU to the FXU and FPU, have been widened. The POWER2 implementation can execute 6 instructions (branch, condition register, two fixed–point, and two floating–point) per cycle. Like POWER, the POWER2 FPU supports a compound multiply–add instruction which increases the peak execution rate to 8 operations per cycle. The interfaces between the Multi–Chip Module (MCM) and the memory and I/O units are compatible with those of POWER. The POWER2 processor chip set offers two system configurations: a 4–word memory bus with 128KB of Data Cache and an 8–word memory bus with 256KB of Data Cache. The Instruction Cache (I–cache) is 32KB for both configurations. An 8–word memory system is shown in Figure 3.

Upgrade paths from POWER–based systems (e.g. from models 570 and 580 to the model 590) exist. The CPU planar, along with some miscellaneous parts such as an air dam, are replaced. S4.5 memory cards are required; however, the memory cards may be populated using 80 nsec. single in–line memory modules (SIMMs) from the upgraded system.

Instruction Cache Unit (ICU)

The ICU fetches and dispatches instructions. It contains a two–way set–associative 32KB I–cache with a 128–byte lines, the associated I–cache directories, and a 128–entry two–way set–associative Instruction TLB (I–TLB). The ICU control logic fetches instructions from the I–cache and places them in one of two instruction buffers depending on the expected path and the occurrence of conditional branches. The buffer for sequential path instructions has 16 elements. Unconditional branches (and instructions following these branches) are removed from the sequential buffer, the fetch path is altered, and instructions along the new path are appended to those remaining in the sequential buffer. Instructions from a conditional branch target path are placed in an 8–entry target buffer. The ICU processes instructions in the sequential buffer. It executes branch and condition–register instructions and dispatches the remaining instructions to the FXU and FPU. The ICU can fetch 8 instructions per cycle from the I–cache regardless of alignment. It can dispatch 6 instructions per cycle: two internally and four externally to the FXU and FPU.

Generally, the FXU and FPU receive an uninterrupted instruction stream and do not see the effect of branches (i.e. zero–cycle branches). The ICU dispatches the sequential path instructions to the FXU and FPU. Instructions beyond an unresolved conditional branch are dispatched conditionally; conditionally dispatched instructions are held in FXU and FPU instruction buffers until the branch is resolved. Then they are issued or cancelled.



*** Bus widths are as follows:**

Pbus	**1 single–word**
Instruction reload bus	**1 quad–word**
Instruction dispatch bus	**4 independent single–words**
Floating–Point Data Bus	**2 independent quad–words**
Fixed–Point Data Bus	**2 independent single–words**
ESIO bus	**1 double–word**
Memory data bus	**4 or 8 words**

Figure 2 POWER2 Processor Complex

To minimize conditional branch delay after an incorrect guess, the ICU also fetches the branch taken path from the I–cache and dispatches the target instructions onto the instruction bus. Until the branch is resolved, the FXU and FPU ignore these target path instructions. If the branch is resolved as ”not taken”, the instructions for the correct path are in the FXU/FPU instruction buffers and usually incur no penalty. If the branch is resolved as ”taken”, the conditionally dispatched instructions are cancelled and the target instructions (on the instruction bus) are loaded into the FXU and FPU buffers.

Unconditional branches, ”not taken” conditional branches, and ”taken but resolved” branches often cause no pipeline delay. Taken branches which are not resolved when they are processed often result in a single cycle delay in the FXU/FPU pipelines. More ICU details are in [9].

Fixed–Point Unit (FXU)
The FXU performs all storage references, all integer arithmetic, and logical operations. The FXU contains the GPRs, two fixed–point execution units, the data cache directory, the Data TLB (DTLB), and the TLB reload hardware. The 32 32–bit GPRs are implemented in two 8–port (4R/4W) register files, one for each execution unit. The register files implement full bypass to minimize delay between dependent operations. It also performs register score–boarding so that data cache accesses do not hold off subsequent independent register–register instructions.

The dual execution units, which can execute a total of two instructions per cycle, are fed by an 8–element (6–element in POWER) FXU instruction buffer that holds only fixed–point instructions. Each unit contains an adder and a logic functional unit. The second unit also contains a multiply and divide unit. The multiply / divide unit executes a multiply in 2 cycles (3–5 cycles in POWER) and a divide in 13 to 17 cycles (19–20 cycles in POWER).

When both functional units are ready to accept a new instruction, the (logically) first instruction is issued to the first unit. If a second FXU instruction has been dispatched, and its input operands are available, it is issued to the second unit. To minimize data dependencies, a three–input adder has been placed above the second unit. As a result, even if the source register(s) for the second FXU instruction is being updated by an add (or storage reference with update) type operation in the first unit, the second instruction can execute in parallel.

Since the FXU contains the GPRs which are used in addressing, the FXU executes all storage reference instructions, including the address translation and data cache directory search. It contains the sixteen

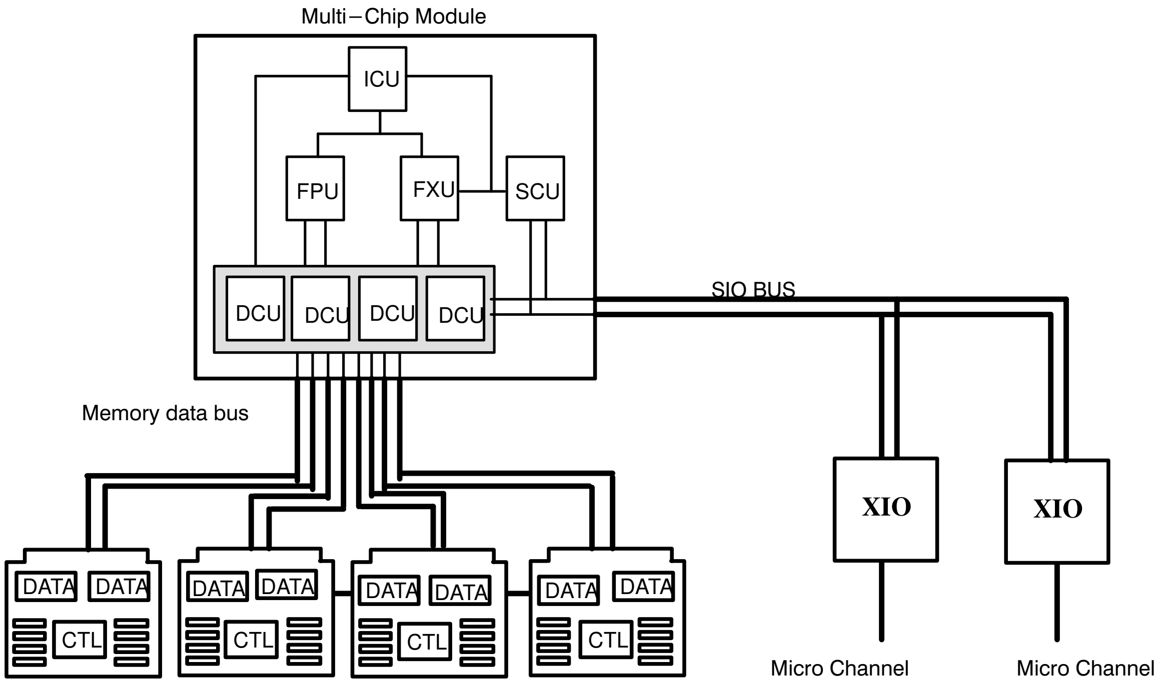


Figure 3 POWER2 8–word System

32–bit segment registers (effective to virtual) and a 512–entry (128–entry in POWER) two–way set–associative D–TLB (virtual to real) for address translation. It also has TLB reload hardware for processing both I–TLB and D–TLB misses. The two execution units are supported by dual ported data caches and dual ported directories. The cache supports first–order non–blocking accesses; subsequent load/store operations are overlapped with data cache misses.

Single element loads have the same load–use delay as POWER; one cycle separates the load from a dependent arithmetic. The dual execution units allow two single–element (including quad–word) storage references per cycle. Like POWER, hardware handles the most frequent cases of unaligned accesses. POWER2 extends hardware support for unaligned accesses to include quad–word accesses on any odd–word boundary. Multi–cycle load and store operations are handled using both units. The opcode is loaded into both units and two loads/stores are executed per cycle. The FXU implementation is more fully described in [6].

Floating–Point Unit (FPU)

The FPU includes the Floating–Point Registers (FPRs) and two double–precision (64–bit) execution units; two floating–point instructions can be executed per cycle. As in POWER, POWER2 implements FPR renaming to increases the FXU/FPU autonomy. The number of physically implemented registers exceeds the number (32) which are defined in the architecture. Mapping hardware selects a "free" physical register to hold the destination value for each floating–point load operation. Since loads do not overwrite the previous data in the architected register, they may be executed before the completion of a logically previous arithmetic instruction, even if it loads an FPR which is a source operand for the arithmetic. This is a substantial benefit on short loops where register renaming can not be done in software. Moving the loads further ahead allows the effects of the load–use delay to be minimized. To support the increased number of functional pipelines, the number of physical registers are increased (from 40 in POWER) to 54.

The floating–point functional units are more symmetrical and autonomous than the fixed–point units. The FXU can only issue an instruction to the second functional unit if two instructions are ready when an instruction is issued to the first unit. In the FPU, whenever an instruction is ready to issue and a functional unit is available, the instruction is issued. Both FPU units perform all operations, including divide and square root. The execution units are fed by an eight–element instruction buffer that may contain 8 floating–point interruptible instructions. Arithmetic results conform to IEEE 754 binary floating–point standard. As in POWER, the POWER2 FPU supports the compound multiply–add instruction, (A X B) + C. The two operations are executed with a single rounding of the result, with the same latency as a single multiply or add instruction. The dual units enable the FPU to execute two double–precision multiply/add instructions every cycle, resulting in up to four floating–point operations per cycle.

Each of the FPU execution units can execute a divide or the new square root instruction. When loops contain multiple divides, or when single divide loops are unrolled, proper scheduling may allow divide operations to be performed in parallel, providing roughly a factor of two advantage over POWER. Since the square root instruction is roughly twice as fast as the POWER library sequence, dual execution of square root instructions gives POWER2 roughly a factor of four advantage over POWER for this function.

To exploit the bandwidth potential of the quad–word storage reference instructions, the dual FPU execution units are supported by a dual quad–word interface to the data cache. Up to four floating–point registers can be loaded per cycle.

The POWER2 FPU has a separate unit for normalizing store data, thus allowing stores to be executed in parallel with arithmetic operations. As a result, floating–point stores effectively require zero FPU cycles whereas they required one FPU cycle in POWER. Additional FPU details are provided in [10].

Data Cache Unit

POWER2 has a four–way set–associative dual–port 128K/256K–byte (32KB/64KB in POWER) data cache that is partitioned into four identical chips. Each chip has a 64K–byte array. The cache line size is 128/256 bytes (64/128 bytes in POWER). A store back cache is implemented to minimize memory bus traffic. The four data cache chips generate two single–word data buses to the FXU, two quad–word buses to the FPU, a quad–word Instruction Reload bus to the ICU, and a two–word System I/O bus to the I/O subsystem for DMA data (Figure x1). For increased reliability the DCUs support memory scrubbing, single–bit correction double–bit detection error checking and code (ECC), and bit steering.

The cache can be configured with a 256K–byte capacity (256–byte lines) supporting an 8–word memory bus. Alternatively, the DCUs can be configured to support a 4–word memory bus, 128K–byte data cache and 128–byte line size. Two, four, or eight memory cards are allowed. When the system contains two memory cards, it configures itself as 4–word memory system. If there are more than two memory cards, an 8–word memory system is configured.

<u>Storage Control Unit</u>
The Storage Control Unit contains the controls and configuration registers for memory, arbitrates for all communications between the CPU (ICU, FXU, DCU), memory and the System I/O bus. It generates the controls for the System I/O (SIO) bus and a data path for Programmed I/O. The SIO bus has been enhanced to prefetch DMA read data from memory to enable the system to sustain a DMA rate of over 150MB/sec using the Streaming Data protocol on the Micro Channel. The memory cards used in POWER2 systems can be used in all POWER systems and the POWER2 I/O unit can be used in POWER systems which support Extended Input/Output (XIO) (e.g. 570,580,970,980).

<u>I/O Unit</u>
The I/O Unit used with POWER2 is one used in the RS/6000 Model 580 and 980. The I/O Unit implements the 64–bit Streaming Data protocol on the Micro Channel at 10 MHz. Dual 64–byte buffers per DMA channel are implemented so that operations over the SIO bus and Micro Channel are fully overlapped. In addition, the protocol for the SIO bus was enhanced to include prefetch data commands from the I/O Unit so that the DMA data from memory is available to the I/O Unit with minimum delay.

For DMA transfers to memory, the I/O Unit keeps a record of every byte that has been modified in the 64–byte buffer. If all bytes are modified, it results in a write to memory. If only part of the bytes in the 64–byte buffer were modified, the I/O Unit does a read–modify–write. In the previous version of I/O Unit, a read–modify–write was performed for all transfers to memory. This implementation almost doubles the throughput for all DMA operations to memory. The I/O Unit, along with some logic on the I/O planar, reduces the arbitration time on the Micro Channel from 400ns to 100ns, thus improving bandwidth and bus utilization.

Chip	Transistor Count (thousands)		Die Size (mmxmm)	Signal I/O
	Logic	Memory		
FXU	583	848	12.7x12.7	473
FPU	1001	315	12.7x12.7	504
ICU	547	2277	12.7x12.7	464
DCU x 4	1117	16000	12.7x12.7	366
SCU	349	——	9.4x9.4	276
MCM	3597	19440	1217 mm sq.	512

Table 1 Physical Attributes of the POWER2 Processor Chip Set

Chip and Packaging Technology

The physical attributes of the POWER2 CPU chip set are shown in Table 1. The POWER2 processor chip set contains over 23 million transistors on a silicon area of 1217mm^2. The chips are implemented in CMOS technology with an effective channel length of 0.45 micron. The chips contain one level of polysilicon and four levels of metal wiring.

The chips are mounted on a ceramic Multi–Chip Module (MCM). The MCM provides most of the chip–to–chip wiring. While the total chip I/O is 3181, all but 512 connections are internal to the MCM. The MCM includes a 64mm X 64mm pin grid array with 512 signal I/Os. The total power dissipated by the MCM at 66.5 MHz is about 65 watts.

Performance

The performance gains associated with the improvements described above can be separated into two categories. The first category is based on the code which the compiler generated prior to POWER2–based systems. Without recompiling, many programs obtain benefits from the larger caches, larger TLB structures, and the enhanced address translation process. Calls to library routines, such as ESSL, can obtain POWER2–specific tuning benefits when the new libraries are linked. Furthermore, the additional functional units exploit the instruction–level parallelism which has been exposed in the compiled code.

The second category is based on enhanced exploitation of the functional units through recompiling for a POWER2 target. Additional instruction–level parallelism can be exposed by the compiler after more aggressive loop unrolling. POWER2 scheduling, which takes into account POWER2 latencies and interlocks, can improve performance. Recompiling also allows applications to benefit from the new instructions.

In the previous three sections, we describe architectural, implementation, and technological details which allow the RS/6000 to increase its clock rate while significantly increasing its superscalar abilities. This article opened with the debate topic: ”Which delivers more performance, instruction–level parallelism or clock rate?”. The complexity of the trade–offs and their interactions makes this a difficult question. The best proof of a good answer is overall performance. Table 2 compares a POWER2–based system with a POWER–based system and two competitive workstations and servers [5,11,12,13,14]. While the HP and DEC systems have substantially higher clock rates, the POWER2–based system uses instruction–level parallelism to move ahead on all of these industry standard benchmarks.

The SPEC CINT92 suite consists of six integer codes representing compilers, spreadsheets, etc. The SPEC CFP92 suite consists of fourteen floating–point benchmarks from the workstation and server market. In both SPEC suites, no hand optimization of the codes is permitted. The overall measure in each suite is the geometric mean for the benchmarks within the suite. The Model 590 wins the SPECint92 race. A win by the POWER2 implementation confirms that a 3X clock rate advantage can be compensated with superscalar ability, even on SPECint92. Due to the characteristics of SPECfp92, one expects even greater opportunity to exploit a superscalar approach. This is demonstrated by the 25% win of the POWER2 system over competitors with up to a factor of three clock rate advantage, as well as by the large gain over POWER–based system.

The Linpack benchmark represents one problem area: solving dense systems of equations. The benchmark rules allow no hand optimization while solving a 100x100 matrix problem. For POWER, the optimized inner loop asymptotically approach a load and a store per element update, resulting in an FXU limit of two cycles. HP documentation describes that the PA7100 requires two cycles for a store. The resulting three–cycle limit per update almost exactly counteracts the rough 3:2 clock rate advantage of PA7100 over POWER (Model 580). As a result, the performance of these two systems is very similar on this benchmark. As discussed earlier, POWER2 benefits substantially on Linpack from quad–word storage references. It receives almost another factor of two improvement as the number of functional units doubles. While the POWER Linpack performance almost matches the competitive systems, the POWER2 boasts more than a factor of three advantage.

The TPP (Toward Peak Performance) benchmark solves the same type of problem as Linpack, on a 1000x1000 size problem, but allows hand optimization of the code. Since this is a numerically intensive benchmark which can be solved using hand–coded library routines on many systems, this benchmark often approaches the peak MFLOPS rate of a system. The TPP value for POWER2 surpasses the theoretical peak rates of the HP and DEC Alpha systems which are 198 and 200 MFLOPS, respectively.

TPC–C is a prominent commercial benchmark. Unlike the other benchmarks in this comparison which consist of running a controlled source version of a program, TPC–C is a specification of a workload that ”is a mixture of read–only and update intensive transactions that simulate the activities found in complex OLTP [on–line transaction processing] application environments” [15]. Detailed TPC–C characteristics, and its POWER2 performance, are described in [16].

Summary

A popular debate topic concerns the design philosophy dichotomy evident in the workstation and server markets. RS/6000 POWER–based systems are often characterized as the aggressive superscalar performance. DEC Alpha and HP PA7100 illustrate the aggressive clock rate (moderate instruction–level parallelism) approach. The quad–word storage references and the (moderate clock rate) approach to additional functional units in POWER2–based systems makes the divergence more pronounced. While the leadership position in the workstation/server performance race changes frequently, industry standard benchmarks demonstrate POWER2–based systems allow the RS/6000 family to move ahead of other implemented design points. This result confirms that instruction–level parallelism is a justifiable alternative to clock rate in the performance race.

Acknowledgements

The POWER2 extension of the RS/6000 product family is the result of the efforts of the entire Austin team as well as many people at the Yorktown, Burlington, and Toronto sites. Many of the key technical individuals are the authors of the companion papers and cited references. Management leadership for the chips, systems, and compilers include Phil Hester, George Lerom, Rudy Pirovitz, Dwayne Crider, Don Johnson, Bob Gerber, and Karen Bennet. We thank Ron Seaman for an outstanding job of managing the POWER2 system development schedule and for making early systems available. Finally, we thank John Reysa and Kate Stewart for reviewing this manuscript and providing many useful suggestions.

System	RS/6000	RS/6000	HP PA7100	DEC 10000
	Model 580	Model 590	Models 735/755	Model 610 AXP
Clock rate	62.5 MHz	66.5 MHz	99 MHz	200 MHz
SPECint92	73.3	117.0	80.6	116.5
SPECfp92	134.6	242.4	149.8	193.6
Linpack	38	130	41	43
TPP	104	236	107	155
TPC–C (tpmC)	N/A	726.13	613.80 **	N/A
$/tpm–C	N/A	$1603	$2488 **	N/A

Table 2 Industry Standard Benchmark Comparison

** The TPC values for HP are not available for the Models 735/755. The TPC–C values shown in Table 2 for HP are for the 96 MHz HP 9000 series 800 Model H50 which is based on the HP PA7100 processor.

References

1. H.B. Bakoglu, G. F. Grohoski, R. K. Montoye, ”The IBM RISC System/6000 processor: Hardware overview,” *IBM J. Res. Develop.*, Vol.34, No. 1, Jan. 1990, pp. 12–22.
2. John Cocke and V. Markstein, ”The evolution of RISC technology at IBM,” *IBM J. Res. Develop.*, Vol.34, No. 1, Jan. 1990, p.9.

3. Linley Gwennap, "Speed Kills? Not for RISC Processors," Microprocessor Report, March 8, 1993, p.3.
4. Steven W. White, Phil D. Hester, Jack W. Kemp, and G. Jeanette McWilliams, "How Does Processor MHz Relate to End–User Performance?," *IEEE Micro*, August and October 1993.
5. Jack Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," CS–89–85, April 15, 1993.
6. D. J. Shippy and T.W. Griffith, "POWER2 Fixed–Point, Data Cache, and Storage Control Units," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.
7. E.H. Welbon, C.C. Chan–Nui, D. J. Shippy, and D.A. Hicks, "The POWER2 Performance Monitor," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.
8. Sohel Saiyed, J. Michael O'Connor, and Maurice Franklin, "POWER2 CPU–Intensive Workload Performance," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.
9. Baba Arimilli, Jama Barreh, Robert Golla, and Paul Jordan, "POWER2 Instruction Cache Unit," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ?? .
10. Troy N. Hicks, Richard E. Fry, and Paul E. Harvey, "POWER2 Floating–Point Unit: Architecture and Implementation," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ? .
11. Kaivalya Dixit and Jeff Reilly, "SPEC Developing New Component Benchmark Suites," *SPEC Newsletter*, Vol.3, issue 4, Dec. 1991, pp. 14–17.
12. *SPEC Newsletter*, Vol. 5, issues 1&2, March 93 (HP) and June 1993 (DEC on p.28 and p.54).
13. IBM Announcement, Sept. 21, 1993, New York.
14. HP TPC–C Full Disclosure Report, Sept. 7, 1993.
15. "TPC Benchmark C," Standard Specification Revision 1.1, Transaction Processing Performance Council (TPC), June 1, 1993, p. 1.
16. Maurice Franklin, William Alexander, Rajiv Jauhari, Ann Marie Grizzaffi Maynard, and Bret Olszewski, "POWER2 Commercial Workload Performance," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ?? .

The POWER2 Performance Monitor

E.H. Welbon, C.C. Chan–Nui, D.J. Shippy, and D.A. Hicks

Introduction

The POWER2 Performance Monitor ("monitor"), currently an internal tool, provides detailed hardware measurements necessary to study the hardware/software interaction of workloads executed by the POWER2 processor. This monitor is integrated into the POWER2 processor and is fully software accessible. This monitor can selectively measure specific software processes with minimal disruption.

Performance measures provided include instructions executed, elapsed cycles, counts and delays associated with cache and TLB misses, and utilization of the various execution elements. Such measures can help to efficiently locate and eliminate performance bottlenecks.

The Need For Hardware Measurement

Regardless of the advances in CPU design, improving system performance increasingly depends on a detailed understanding of hardware and software interaction [1]. An effective measurement facility can provide such understanding. An effective measurement facility can also provide the basis for evaluating design decisions of future systems by identifying existing bottlenecks.

For example, it is useful to identify opportunities for better CPU execution utilization by improving compiler instruction scheduling. Instances of such opportunities are described in [2, 3]. Development of the compiler can be hastened if information identifying scheduling opportunities is readily available.

Of course, hardware measurement is not the only method of understanding system behavior. Simulation is an effective way to study hardware / software interaction in the earliest phases of the design and implementation of a system [4]. However, if measurable hardware is available, it is clearly preferable to measure rather than simulate.

Drawbacks Inherent to Previous RISC System/6000 Measurement Facilities

In previous RISC System/6000 systems, IBM development teams collected hardware performance information using special external instrumentation, which was adapted to the processor model under consideration.

The external instrumentation approach was possible since the RISC System/6000 multiple chip design provided some access to signals typically internal to a single chip [5]. The implementation also included special signals solely for performance monitoring. These special signals were provided either multiplexed functional pins or by pins dedicated to measurement. Additional logic was typically required to convert the externally available signals into useful performance information.

Subtle differences between the various CPU models required each CPU to have different instrumentation. As a result, instrumentation was not available on all CPU models. Additionally, the difficulty and cost of instrumenting any particular CPU model has further constrained the number of systems instrumented.

Desirable Monitor Characteristics

The availability of a hardware measurement facility can be an asset to understanding hardware / software interaction and thus aid in improving the performance of current and future systems.

A desirable characteristic is the ability to select specific times and specific threads of execution to measure. An analyst studying a

This paper has been submitted to the IBM Journal of Research and Development.

numerical analysis application may be interested in particular subroutines or even particular loops [1]. On the other hand, an analyst studying a commercial application may want to be able to exclude I/O wait intervals [6].

A measurement facility is much easier to use if the software can efficiently access and control the measurements of its own execution. This is true whether the instrumentation is internal or external.

Ideally, the monitoring instrumentation should be directly incorporated into the CPU chips and be accessible by software in a simple yet effective manner[1]. An example of such capability is the Cray Y–MP [7] which has software accessible counters. It was desired that POWER2 implementation provide similar capabilities.

Implementation of Monitor

Systems are becoming more complex and highly integrated causing less data to be available for external hardware to intercept and record. One example in particular is the POWER2 processor which is packaged on a Multiple Chip Module (MCM).

The MCM package includes five basic units, each being a distinct chip. These units are: the Fixed–Point Unit (FXU), the Floating–Point Unit (FPU), the Branch Unit (or Instruction Cache Unit, ICU), the Storage Control Unit (SCU), and the Data Cache Units (DCU). Additional details are provided in [8, 9,10,11].

1. Generally, the signals necessary to detect the required event will be available within the chip. Signal availability caused by pin restrictions tend to prohibit this on external instrumentation.

Most of the chip inputs are interconnected on the MCM substrate and not externally accessible. The memory and I/O buses are the only external interfaces to the MCM [11,12]. Consequently, there are few external POWER2 CPU signals suitable for deriving performance data.

Providing the measurement data via software accessible registers can reduce the pin I/O requirements. To this end, the ICU, FPU, SCU, and FXU are each allocated five counters. Similarly, each of the four basic units is provided with a four–bit control field in the Monitor Mode Control Register (MMCR) that selects the set of events to be counted. Thus for each unit, it is possible to choose any one of sixteen groups of five events for monitoring and to require only nine pins per chip as shown in Figure 1.

As illustrated by Figure 1, the monitor contains twenty–two 32–bit counters which are used to count CPU/Storage performance related events. The MMCR provides monitor control functions. The MMCR, along with two status bits in the POWER Machine State Register (MSR), allows selective measurement of specific threads of execution. The counters and the MMCR are addressable for read and write operations via Programmed I/O (PIO).

With one special exception discussed later, the selection of the events to count for any unit is independent of the selection of the other three units.

An Instruction Match Register (IMR) provided in the FXU counts the occurrence of specific instructions. By repetitively cycling through all desired instruction codes, a sampling of the instruction execution frequencies can be obtained.

Finally, a bit referred to as the Software Programmable Event (SPE) allows software to create “software events” as desired. The FXU counters can count the number of cycles that the SPE bit is “1” (set by software).

Monitoring States

The POWER architecture defines the system wide Machine State Register (MSR) as part of process state. MSR bit number 29, the The Process Mark (PM) and Problem (PR) bits of the MSR are used to control the state of the monitor.

Figure 2 shows the Monitor Mode Control Register (MMCR) which provides control modes that take advantage of the MSR PM and PR bits. Since the MSR is part of process state and is saved and restored when processes pause and resume execution, the MSR PM bit can be used to selectively qualify processes for monitoring with low overhead. Thus the MMCR and MSR together can efficiently control the state of the Monitor.

Selection of Observable Events

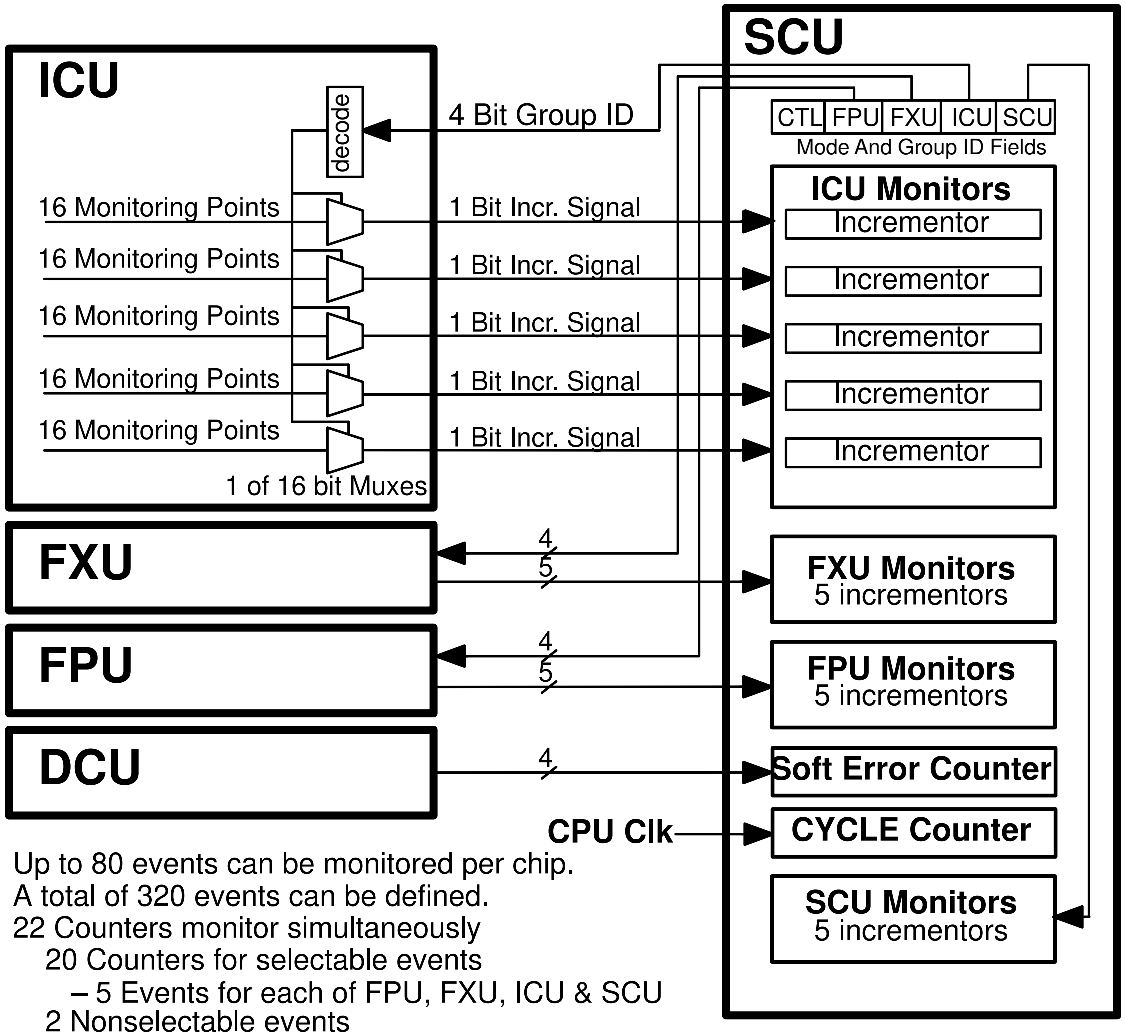
The previous RISC System/6000 measurement facilities rely on externally accessible signals that external hardware converts into performance information. The POWER2 monitor includes hardware in each unit that directly detects the performance information.

Therefore, the selection of the POWER2 performance events is based on what the users of the performance community wanted as opposed to what could be observed given the available signals. As a consequence of the monitor's organization, the chips provide a wealth of event information. Although this paper is not intended to serve as an implementation guide, the following four sections provide a summary of these events.

<u>FXU Observable Events</u>

The FXU provides cycle counts for categories such as branch delays, data cache misses, floating–point interactions, TLB reloads[2], and the timing of multiple cycle instructions. The FXU can count the number of instructions that it executes. Other counters allow monitoring the utilization of the various fixed–point execution elements.

A special set of counts identify cases where opportunity exists for performance improvement via alterations of the scheduling of the instruction streams being executed. A closely related set of counts identify the *kind* of scheduling opportunities; these counts categorize the causes of FXU idle cycles.



2 Nonselectable events
– 1 cycle counter
– 1 soft error counter

Figure 1 Organization Of Monitor Hardware

---

2. In the POWER and POWER2 implementations, storage accesses that miss in either the data or instruction TLB are satisfied by the FXU in hardware and, barring page faults, are transparent to software.

Thus, the type of opportunity, as well as an upper bound on its magnitude, can be identified.

ICU Observable Events

The ICU provides counts of instruction cache and TLB misses, the number of instructions fetched, dispatched, and executed, and the number of interrupts acknowledged. Also provided is a breakdown of the classes of instructions executed (principally branch instructions), and a set of counts identifying the sources of dispatch and branch delay.

FPU Observable Events

The FPU provides counts that give many details on conditions causing floating–point hold off cycles, execution element utilizations, the number of instructions executed, and IEEE special number (e.g. infinity, NaN, etc.) processing delays. In addition, the FPU provides counts of the classes of instructions executed, counts of cycles incurring register renaming stalls, and counts of various queue related stall cases.

SCU Observable Events

The SCU counts the number and type of storage operation requests that are presented to it, measures of latency for such requests, and measures of DMA, memory bus, and SIO bus activity. In addition to housing the selectable counters, the SCU contains two dedicated counters: a CPU cycle counter and a correctable memory error counter.

Storage performance is critical in RISC machines [13]. To most effectively measure storage–caused delays, a special mode was

MONITOR MODE CONTROL REGISTER (MMCR)

DIS	DP	DU	DMS	DMR	SGA	//	FPU	FXU	ICU	SCU
0	1	2	3	4	5	6	16	20	24	28

This register is set to zero on power up. Reading this register does not change its content. Bits 6 though 15 are reserved, and always read as zeroes. To maintain compatibility with future implementations, software must not write nonzero values into bits 6 through 15. The named fields have the following definitions.

DIS: Disable counting unconditionally: This bit, when a one, causes the counters to stop counting unconditionally. The counter values are not changed, only suspended.

DP: Disable counting when in privileged state: This bit, when a one, causes the counters to stop counting when MSR(PR) is a zero. The counter values are not changed, only suspended.

DU: Disable counting when in problem state: This bit, when a one causes the counters to stop counting when MSR(PR) is a one. The counter values are not changed, only suspended.

DMS: Disable counting when PM is set: This bit, when a one, causes the counters to stop counting when MSR(PM) is a one. The counter values are not changed, only suspended.

DMR: Disable counting when MSR(PM) is reset: This bit, when a one, causes the counters to stop counting when MSR(PM) is a zero. The counter values are not changed, only suspended.

SGA: SCU Gets All: This bit, when set to a one allows the SCU to control all twenty two counters i.e., the SCU gets all of the counters. This bit over rides the FPU, FXU, ICU and SCU source event set selection.

FPU: MMCR[16:19], 4–bit code selecting FPU source event set.

FXU: MMCR[20:23], 4–bit code selecting FXU source event set.

ICU: MMCR[24:27], 4–bit code selecting ICU source event set.

SCU: MMCR[28:31], 4–bit Code selecting SCU source event set.

Figure 2 Monitor Mode Control Register Format defined for the SCU. Ordinarily, the FXU, ICU, FPU, and SCU can operate only five counters each. However, Figure 2 describes a special case involving the SGA bit of the MMCR that allows the SCU to operate twenty–one of the twenty–two counters with strictly storage control related counts. This mode provides a detailed view of the storage and I/O activity.

Basic Monitoring Considerations

In general the monitor is a serially reusable resource and must be managed to ensure that it is utilized in a reliable manner. If the counters are managed by the operating system at a user's request, multiplexing the monitor function across multiple processes is possible. In this case, the counters are treated as part of process state and are saved and restored to allow each process to use the monitor as though it is private to each process.

Because the monitor cannot count every event simultaneously, alternating the events to be counted can provide more data. This technique is valid when monitoring a steady state or cyclic environment. On a periodic basis, one could cycle the values of FPU/FXU/ICU/SCU counter event selection fields through the full range of values. This technique provides a sample of all measurable events. Such a procedure was used to obtain the data for the results of [6] and is discussed in the "Monitor Data Reduction Tools" section of this paper.

Selection Of Code to be Monitored

A key feature of the monitor is the ability to automatically focus on supervisory activities or particular processes with no overhead. Conversely, particular processes or supervisor activity can be excluded from the measurement process. Some of the possibilities will be described in the following four sections.

Monitoring Specific Blocks Of Code Within Processes and Programs

An analyst studying a numerical analysis application might be interested in particular subroutines or even particular loops [1]. Such measurements may be made by placing a call to enable monitoring at the desired entry point and corresponding call to extract values at the appropriate exit point. Although the calls to enable and disable monitoring will incur some overhead, this overhead will not markedly affect the measurements made on the code segment being analyzed. This technique is further discussed in the "Monitor Command Line Tools" section of this paper.

Monitoring One Process

Many performance studies focus on workloads consisting of a single process. In such a case, the MMCR and the MSR control functions described in Figure 2 can be used to focus performance monitoring to just a single process with no software overhead beyond setup. This is discussed further in the "Monitor Command Line Tools" section of this paper.

The benchmark in question will have both MSR(PM) and MMCR(DMR) set to one. All other processes executing would have their respective MSR(PM) bits set to zero. As a consequence, only the benchmark process with its MSR(PM) bit set will be able to alter the performance counters. Thus, the performance counters would not have to be saved or restored, eliminating most of the control overhead.

Since the counters will "roll over" roughly every 65 seconds (depending on the processor clock speed), it will be necessary for the operating system to schedule an event that will save the performance counters. Also, the monitor control software must supply certain managerial functions for the monitor, since reading and writing the MMCR and the MSR are privileged operations.

Monitoring A Set Of Processes

It is possible to monitor the cumulative effects of any set[3] of the executing processes using the above technique. The control software must be able to recognize what subset is to be monitored. When the subset has been determined, each process of the subset will have its MSR PM bit set appropriately.

In this technique a fixed set of parameters is measured for the processes of the set. Each

3. A simple example of such a set would be an AIX "process group". In this case, each process will inherit the process state of its parent process; if the parent process has the requisite MMCR and MSR bit settings, then these settings will be automatically propagated to the parent process's children.

time a process from the set to be monitored commences execution, that process's MSR will be loaded with a PM bit value which will enable monitoring. Note that there will be no need to save or restore the monitor counters or MMCR in this case. The data values produced by the counters is treated in the same way as the data values produced by a single process.

Monitoring All Processes Using A Log

To obtain a trace of counts for all process and operating system activity, an algorithm similar to that used by the *aixtrace* [14] has been applied. In this algorithm a log file is constructed for a performance tracing session. This log file is processed after the tracing session by the postprocessor *rmap* as described by [14].

Because all processes are monitored, the MSR is not needed to select specific processes. Note that the monitor counters in this case are considered part of a process's state just as its general purpose registers are.

At desired intervals, the counter values are written to the *aixtrace* trace log. Each entry consists of the process id, the MMCR, the counter values, and the real time clock at the time that the counters were read.

The implementation of this procedure is discussed in the "System Level Software" section of this paper.

Monitor Support Software

In the balance of the paper we discuss the monitor support software. This software provides performance information to the analyst for both system level and process specific measurement strategies. The next section, "System Level Monitor Software", describes a measurement strategy which provides system level data suitable for overall system analysis and visualization software [15].

The section entitled "Monitor Command Line Tools" describes software that efficiently provides process level measurement. Finally, the "Monitor Data Reduction Tools" section discusses the software used to convert the raw monitor data into useful performance information. This software has been successfully used to analyze several benchmarks including TPC as described in [6].

In all of the following methods, the monitor support software is implemented as an AIX kernel extension.

System Level Monitor Software

This section describes the manner in which the monitor interfaces with the *aixtrace* facility [14]. This facility provides a detailed view of all processes that execute and, in particular, captures the interaction between system and application software. This feature is constructed around a special trace subroutine call provided by the AIX kernel.

Trace calls which request the capture of information, are embedded at desired locations in the software to be monitored. These calls are referred to as "hooks". Each distinct "hook" has a unique identifier, the "hook" ID, which distinguishes between the various "hooks".

These "hooks" may be system or user defined and may be disabled and enabled as desired. Each "hook" is designed to pass "hook" specific data. In particular, the monitor counter data can be inserted into the *aixtrace* log as an *aixtrace* record every *n* software clock ticks (or when the process dispatched is different from the current process).

The monitor kernel extension intercepts trace calls. The monitor examines the arguments of the "hook" and determines if it is an "appropriate" time to record the monitor data. If so, the kernel extension calls the trace routine on its own behalf, inserting the counter data into the trace.

This approach is flexible since the kernel extension allows the user to specify a list of which "hooks" will cause the monitor counters to be recorded. This allows report generators that postprocess the trace to produce per process performance statistics. Also, as is characteristic of kernel extensions, enhancements to the monitor facility may be added without rebuilding the kernel.

The main drawback of this method is that the *aixtrace* facility and the appropriate kernel extension must be enabled. This means that depending on the number of "hooks" enabled, data gathering is slightly intrusive (in the neighborhood of 10%).

Monitor data for visualization software such as described in [15] is provided through the *aixtrace* facility. This allows the visualization software to correlate the data provided by the performance monitor with the normal trace data to provide more informative displays.

A more efficient way of implementing this functionality involves calling the monitor kernel extension directly rather than via “hook” invocations. The Command Line Tools discussed in the next section address this feature.

<u>Monitor Command Line Tools</u>
Monitor command line tools provide a simple efficient way to acquire the performance monitor counters. These command line programs and system calls allow the user to manipulate the Performance Monitor Control registers (the MMCR and IMR) and to periodically write the values of the counters to the operating system *standard out* stream or a *file*.

If a program runs in a short time (under 65 seconds with a 66 MHz processor), a user can reset the 32–bit counters, run the program, and acquire the counter values, without the possibility of overflow. Thus, the program can be monitored without intrusion.

For longer programs it is necessary to periodically save the counters to prevent them from overflowing. One of the tools operates somewhat like *iostat* [14] and writes the counters at a frequency specified by the user. Using this program or an analogous one, the amount of intrusion is roughly equivalent to a context switch each time the counters are recorded.

To measure specific sequences of code within a program (e.g. loops or subroutine calls), the user can enable and access the monitor with calls as shown in Figure 3.

```
main() {

   /* code NOT monitored */

   ph_init(MMCR,IMR);
   ph_start();

   /* code to be monitored */

   ph_stop();
   ph_read(COUNTERS);
   print(COUNTERS);

   /* code NOT monitored */
}
```

Figure 3 Coding Example

<u>Monitor Data Reduction Tools</u>
The large amount of data collected by the monitor introduces a need for data reduction tools to help the user interpret the data more efficiently. To reduce the data generated by the monitor into useful information, two methods have been implemented. In both cases the monitor data is not dynamically analyzed; it is captured for subsequent processing.

The first method reduces the captured monitor data into sets of measures useful to analysts. Examples of these measures include total instructions executed, cache and TLB miss rates and delays, and utilizations of the various CPU internal elements. To easily generate these performance measures, a set of post processors was developed. Computing a wide range of interesting derived values is possible. Rather than enumerate the complete set of such values, the post processor accepts user–specified algorithms to determine how to arrive at a desired set of results. Based on the algorithms, the post processor reduces the input data into performance information, complete with statistics.

Based on the recorded MMCR values, the post processor automatically selects the algorithm appropriate for each given measurement. Since the command line tools can cycle the MMCR event selection fields though the range of permitted values, the full set of events provided in hardware can be obtained easily.

The second data reduction method relies on the IBM Research ”PV” Program Visualization Prototype [15]. This visualization tool processes the *aixtrace* log file with a filter that converts the log file data into a format suitable for visualization.

Because the *aixtrace* log file contains significant amounts of operating system data in addition to the monitor data, the visualization tool is able to give a much richer display than would otherwise be possible.

Summary

The POWER2 Performance Monitor provides hardware measures that are important to software developers tuning the operating system and application software as well as system designers responsible for the development of new systems. The measures characterize storage system performance and compiler instruction scheduling opportunity, both critical to good system performance [6].

Acknowledgements

We would like to thank the members of the AIX and System Performance groups, and the Engineering Center for their help in developing the monitor. A partial list includes Jama Barreh, Michael Fortin, Maurice Franklin, Richard Fry, Robert Golla, Randy Heisch, John Iacoletti, Doug Kimelman, Peter Markstein, John O'Quin, Bryan Rosenburg, Sohel Saiyed, Ed Silha, and Larry Thatcher. This paper was considerably improved by the comments of John Reysa, Donald Tsang, Charlie Ward, and Steve White. Any inadequacies are the responsibility of the authors.

References

1. Sohel Saiyed, J. Michael O'Connor, and Maurice Franklin, "POWER2 CPU–Intensive Workload Performance," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
2. H.S. Warren Jr.,"Instruction Scheduling for the IBM RISC System/6000 processor," *IBM J. Res. Develop.,* Vol. 34, No. 1, 1990, pp.85–92.
3. M.C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM J. Res. Develop.*, Vol.34, No.1, 1990, pp. 93–97.
4. Syed Z. Pasha and E.H. Welbon, "Performance–Directed Design Guidance Using Simulation," *IBM RISC System/6000 Technology*, Order Number SA23–2619, IBM Corporation, 1990, pp. 78–85.
5. G.F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM J. Res. Develop.*, Vol. 34, No. 1, 1990, pp.37–58.
6. Maurice Franklin, WWilliam Alexander, Rajiv Jauhari, Ann Marie Grizzafi Maynard, and Bret Olszewski, "POWER2 Commercial Workload Performance," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
7. *Cray X–MP Computer Systems 4–Processor Mainframe Reference Manual*, Order Number HR–0097, pp. C1–C4.
8. L.E. Thatcher, T.W. Griffith Jr. and D.J. Shippy, "POWER2 Fixed–Point, Data Cache, and Storage Control Units," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
9. Baba Arimilli, Jama Barreh, Robert Golla, and Paul Jordan, "POWER2 Instruction Cache Unit," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
10. Troy N. Hicks, Richard E. Fry, and Paul E. Harvey, "POWER2 Floating–Point Unit: Architecture and Implementation", *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
11. Steven W. White and Sudhir Dhawan, "POWER2: Next Generation of the RISC System/6000", *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
12. Rathna Reddy, J.L. Huerta, and H.C. Nguyen, "POWER2 Packaging Technology", *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
13. John Cocke and V. Markstein, "The Evolution of RISC Technology at IBM", *IBM J. Res. Develop.*, Vol.34, No. 1, 1990, pp.4–11.
14. *IBM AIX Version 3.2 for RISC System/6000, Performance Monitoring and Tuning Guide*, Order Number SC23–2365, IBM Corporation.
15. D.N. Kimelman and T.A. Ngo, "The RP3 Program Visualization Environment", *IBM Journal of Research and Develop.*, Vol.35, 1991, pp.635–651.

The PowerPC Alliance

Charles R. Moore and Robert L. Mansfield

Introduction

The computer industry is widely viewed as one of the most rapidly changing industries in the world today. New technologies from a vast array of disciplines help to fuel this growth. Competition in the industry is fierce. To stay competitive, the players must embrace the changes and exploit technological advancements.

In February 1990, IBM introduced the RISC System/6000™ and AIX Version 3. These products offered unparalleled price–performance characteristics and were well received by the industry. Over the past several years, IBM has continued to improve the product line by introducing faster systems, lower–cost systems, and more advanced operating system features. During this same period, IBM began investing in several exciting new technologies and began exploring possible alliance partners to exploit these technologies.

In September 1991, IBM, Apple and Motorola announced a broad alliance to jointly pursue the development of some of these emerging technologies. The alliance included five principle initiatives:

- ***Object–oriented technology***
  Apple and IBM agreed to form Taligent, an independent company, to develop and license a future operating system based on object–oriented design principles.
- ***Multimedia technology***
  Apple and IBM agreed to form Kaleida, a joint venture company, to create and license multimedia technologies.
- ***Interconnectivity and networking***
  IBM and Apple agreed to develop hardware and software solutions that allowed their systems to interact more effectively.
- ***Open Systems Environment***
  Apple, IBM and Motorola agreed to jointly define the PowerOpen environment. PowerOpen supports both AIX and Macintosh applications, and will be licensed to other vendors.
- ***Microprocessor technology***
  IBM and Motorola agreed to jointly develop a broad family of microprocessors based on a derivative of IBM's POWER Architecture called the PowerPC Architecture.

This type of alliance allows the partners to share in the expense and the risk of exploiting new technology. At the same time, it is a clear demonstration of collective commitment to openness. The alliance benefits from the particular strengths that each partner brings to the venture. In this case, Apple brings an innovative consumer–oriented system design style, the industry's best personal computer software technology, and a sophisticated user interface paradigm. Motorola is one of the industry–leading merchant market semiconductor manufacturers, with a worldwide distribution network. In addition, Motorola offers outstanding microprocessor design skills, especially in the area of low cost and compact design styles. IBM also brings outstanding microprocessor design skills to the alliance, with emphasis on high performance and proven design methods. In addition, IBM brings years of experience and research in the areas of computer architecture, multimedia, and object–oriented technology.

Central to this alliance is the commitment to the PowerPC Architecture and a family of microprocessors. Essentially, the companies have agreed to a vision that brings the architecture to the forefront of the computing world, and enables a broad range of PowerPC–based systems. This paper will primarily focus on the PowerPC aspects of the alliance.

PowerPC Architecture

The PowerPC Architecture is a third–generation RISC architecture that has been optimized for the diverse computing requirements of the future [1]. It has been jointly adopted by IBM, Motorola and Apple. The collective experience of these companies provided a firm foundation for making appropriate architectural trade–offs. The powerful new architecture embraces fundamental concepts of simplicity and general applicability while also extending itself to advanced techniques that will carry it into the next decade and beyond.

There were several goals for the PowerPC Architecture. First, it was important that the architecture maintain an application binary interface (ABI) compatible with IBM's POWER Architecture. This allows PowerPC–based machines to leverage the application base that exists for IBM's RISC System/6000 machines. To this end, the user–level instruction set and programming model of POWER was selected as a starting point for the PowerPC Architecture. Although some instructions were ultimately added and others deleted, these changes can effectively be managed by compilers and operating systems.

The second goal was to simplify the architecture and ease unnecessary implementation requirements. This flexibility allows implementors to make optimizations that are appropriate for specific market targets. In addition, the simplifications allow for smaller chip sizes, faster cycle times, and more aggressive superscalar implementations.

A third objective of the PowerPC Architecture was to provide support for a wide range of uniprocessor and multiprocessor system configurations. This objective was achieved by recognizing key abstractions of the storage hierarchy and defining the storage control architecture to allow effective management of these abstractions. Furthermore, the architecture allows storage references to follow either a big–endian or a little–endian byte ordering convention.

Finally, the PowerPC Architecture was defined with a set of 64–bit extensions that allow for upward compatibility of 32–bit applications. This was achieved by defining the 64–bit instruction operation as a logical extension to the 32–bit execution model. The memory management architecture was also extended to allow address translation of 64–bit addresses. To allow flexibility, each implementation can be compliant with either the base 32–bit PowerPC Architecture or the extended 64–bit architecture.

PowerPC Microprocessors

In addition to a commitment to the PowerPC Architecture, the alliance also calls for the development of a family of single chip PowerPC microprocessor implementations. This initial family of devices is charged with providing superior solutions for system designers across a very wide range of system design points. As a result, the microprocessors must be general purpose and offer versatile interfaces. The economics of the computer industry today demand that these designs be relatively inexpensive, and allow for inexpensive system level solutions. In addition, although the various members of the family may require specific optimization trade–offs for particular market segments, they must all exploit the inherent advantages of the PowerPC Architecture to achieve high performance.

To develop these devices, IBM and Motorola have jointly formed a PowerPC design center in Austin, Texas called Somerset. This jointly–managed center is staffed with approximately 300 experienced microprocessor designers. A design methodology has been established that allows the team to achieve high quality, aggressive designs on relatively short development cycles. This is achieved by blending tools that enhance designer productivity with tools that allow for full custom design and analysis. The tool set allows the designs to be fabricated by either IBM or Motorola manufacturing facilities using a common half–micron CMOS process.

The alliance partners agreed to an initial microprocessor roadmap (Figure 1) that specifies four independent design points. Each of these designs are challenged with bringing industry leadership solutions to their specific segments of the computing market.

The initial four design points include:

- ***PowerPC 601 Microprocessor***
  The PowerPC 601 microprocessor is the first member of the family and is responsible for bringing PowerPC to the market as early as possible [2]. It is targeted for early PowerPC adopters and is suited for use in desktop computers, portable systems, and low–end multiprocessor systems. The design implements the 32–bit PowerPC Architecture and achieves competitive performance at a relatively low cost. Processor chips have been available since early 1993, and first 601–based systems should be available by late 1993.

- ***PowerPC 603 Microprocessor***
  The PowerPC 603 processor is defined to address the low–end and low cost range of the 32–bit PowerPC family [3]. It is organized to fit onto a very small chip and to dissipate very little power. The chip is intended for use in low–end desktop

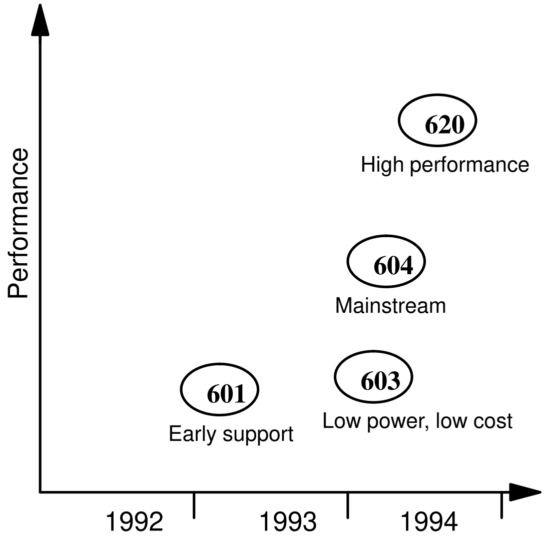


Figure 1 PowerPC Roadmap

systems and portable systems. It will offer competitive performance and will be available in early 1994.

- ***PowerPC 604 Microprocessor***
  The PowerPC 604 microprocessor is designed for mainstream computing environments including personal computers, midrange workstations, and multiprocessor systems. It is organized to offer superior performance over the competition, and outstanding price/performance. The chip is a 32–bit implementation of the PowerPC Architecture, and uses advanced superscalar design techniques to achieve high performance. It will be available in the middle of 1994.

- ***PowerPC 620 Microprocessor***
  The PowerPC 620 processor is designed to deliver the maximum performance achievable with the currently available half–micron CMOS process technology. This superscalar design implements the full 64–bit PowerPC Architecture and includes an embedded L2 cache controller that interfaces to standard static random access memory (SRAM) chips. The design is targeted at high end desktop computers, work–group servers and transaction processing based systems. The chip will be available in the second half of 1994.

These four microprocessors will be heavily used in future IBM products, and they will also be independently marketed and sold by both IBM and Motorola for use by other system developers. The extension of the PowerPC family to the open market is an important and strategic part of the alliance's plan to make PowerPC successful. This will, in turn, entice more independent software developers to write applications for PowerPC–based systems.

PowerPC–Based Systems

Although it is not appropriate to discuss product plans of external PowerPC adopters, clearly the development of the PowerPC microprocessors will have a broad effect on IBM's internal product line. The inherent cost effectiveness of microprocessors, combined with high performance and binary compatibility with POWER, open up many new opportunities to take advantage of the PowerPC technology. During the next several years, it is quite likely that IBM will offer PowerPC–based systems ranging from palmtop personal organizers to massively parallel teraFLOPS machines. Already, several product lines have committed to PowerPC, and other important statements of direction have been publicized.

IBM's Advanced Workstations and Systems division will offer compatible systems based on both the POWER and the PowerPC Architectures. This approach will allow the mainstream workstations to leverage the inherent cost advantages of the PowerPC, while still allowing market distinction by offering even higher performance with their investments in POWER based systems.

In addition, a separate alliance between IBM and Groupe Bull has committed to developing symmetric multiprocessor based systems using PowerPC microprocessors. The approach allows this team to leverage the high performance and low cost of several processors to offer superior solutions in the transaction processing and server environments.

Evidence of IBM's strong commitment to the PowerPC Architecture is the creation of a new division, Power Personal Systems, to develop a line of personal computers that will use the PowerPC microprocessors. These new systems will offer an alternative to traditional personal computers, and will be designed to be open. These systems will be capable of supporting a wide range of industry standard operating systems.

Future Directions

The four microprocessor designs identified on the current roadmap represent only the start of the PowerPC processor development. As technology continues to advance, new opportunities for higher performance, greater levels of integration, lower power consumption and lower cost will arise. IBM and Motorola are committed to driving these technology advancements into their processor development efforts, and to extend the product roadmap over time to cover an even greater range of computing requirements.

In addition, over time, the existing design points will become usable as "cores" for more fully integrated and optimized solutions for particular target applications or markets. These cores will also be available for customers to use in development of their own application–specific VLSI solutions.

Finally, relevant aspects of the PowerPC Architecture will be driven into other market segment opportunities. For example, in a separate effort, IBM is developing a line of embedded controllers based on portions of the PowerPC architecture. These controllers will be suitable for environments that are less demanding than the general purpose computing environment, yet still require high performance at a very low cost.

Conclusion

The PowerPC alliance represents a significant commitment by IBM and its alliance partners to make the PowerPC Architecture and the PowerPC microprocessors successful. The architecture has been optimized for the diverse computing requirements of the future and will allow for the development of a very wide range of compatible systems. The initial microprocessor roadmap identifies a broad range of designs and the alliance is committed to continued investment in future implementations.

References

1. Ed Silha, ”PowerPC Architecture: A High–Performance Architecture with a History”, *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??
2. Charles R. Moore, ”The PowerPC 601 Microprocessor”, I*BM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??
3. J. Kahle and D. Ogden, ”The PowerPC 603 Microprocessor”, *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??

PowerPC Architecture: A High–Performance Architecture with a History

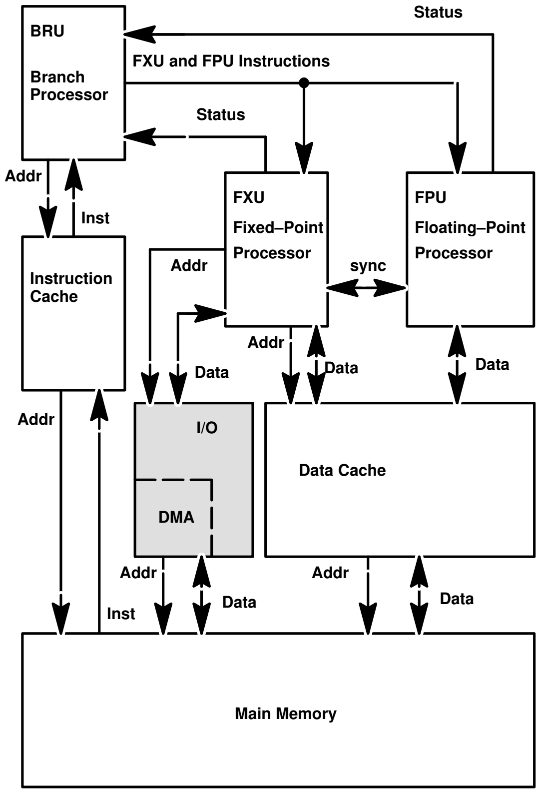
Ed Silha

Introduction

IBM began delivery of RISC System/6000 products in February of 1990 [1,2]. These products were developed in response to customer needs for workstations and midrange systems with UNIX operating systems. The processors used in these products were implementations of the POWER architecture, a second generation Reduced Instruction Set Computer (RISC) architecture.

The POWER architecture incorporated characteristics common to most other RISC architectures. Instructions were a fixed length (four bytes) with consistent formats, permitting simple instruction decoding mechanism. All accesses to memory were performed using load and store instructions to copy data between memory and registers. A set of general purpose registers (GPR's) was provided for fixed–point computation, including the computation of memory addresses, and a separate set of floating–point registers (FPR's) was provided for floating–point computation. All computations retrieved source operands from one register set and placed results in the same register set. Most instructions performed one simple operation.

The POWER architecture was unique among the existing RISC architectures in that it was functionally partitioned, separating the functions of program flow control, fixed–point computation, and floating–point computation. The architecture was partitioned to facilitate the implementation of super–scalar designs, in which multiple functional units concurrently executed independent instructions (see Figure 1).



BRU – Fetch instructions, execute branch and CR instructions, and dispatch FXU and FPU instructions.

FXU – Execute fixed–point instructions and compute addresses for floating–point loads and stores.

FPU – Execute floating–point instructions and manage data for floating–point loads and stores.

I/O – Definition is outside the scope of the processor architecture.

Figure 1 POWER and PowerPC Architectures

The POWER architecture diverged somewhat from the path taken by most other RISC architectures. The primary objective of those architectures was to be sufficiently simple that implementations could have a very short cycle time, resulting in processors that would be capable of executing instructions at the fastest possible clock rate. The designers of the POWER architecture chose to minimize the total time required to complete a task. The total time is the product of three components: path length, number of cycles needed to complete an instruction, and cycle time.

Loads and stores account for twenty to thirty percent of the instructions executed by most programs. In addition, many applications manipulate arrays, for which the pattern of memory accesses is often regular (e.g., every "nth" element). Based on these observations, the designers included "update" forms of most load and store instructions (the memory access is performed and the updated address used for the access is placed in the base address register). The use of these instructions avoids the need for a separate address computation after each access.

In addition, a common operation in many floating–point computational algorithms is adding a value to the product of two other values. This observation prompted the

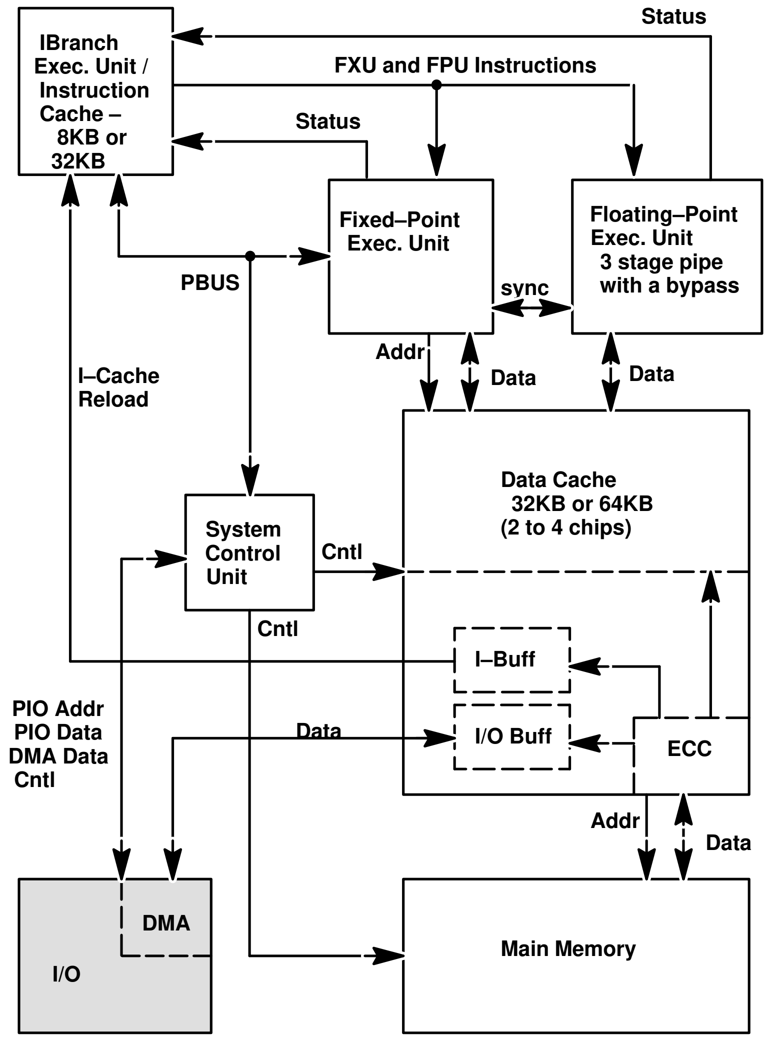


Figure 2 Initial POWER System

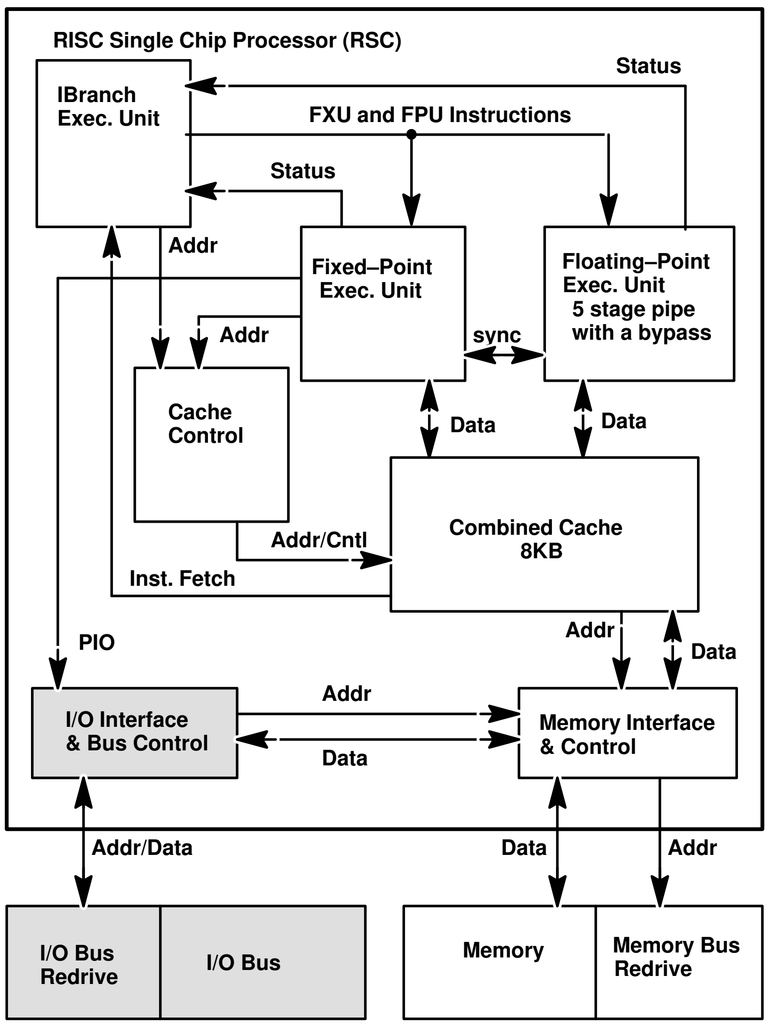


Figure 3 Low–cost POWER System – RSC Base

inclusion of a floating–point multiply–add instruction, which performs this operation.

The first POWER systems provided exceptional performance. The processor complex (including a memory controller, instruction cache, and data cache) was composed of seven or nine chips depending on the system model. The seven–chip complex provided a 32K–byte data cache and the nine–chip complex provided a 64K–byte data cache. These systems contained an 8K–byte instruction cache which was expanded to 32K bytes in later systems.

Figure 2 shows the system organization of these products.

A second POWER processor design [3] was made possible by silicon technology advances which enabled designers to put more than a million transistors on a single chip. This RISC Single Chip (RSC) design integrated a simple branch unit, a fixed–point unit, a floating–point unit, a unified cache, a memory controller, and an I/O controller on a single silicon chip for use in low–cost desktop systems. Delivery of products using this processor began in April of 1992. Figure 3 shows the system organization of these products.

The POWER2 design, described in another article [4], delivers more performance than the earlier POWER designs by using nearly twice as many execution units.

Broadening the Application Market

Performance is not the only characteristic customers consider when purchasing computer systems. They purchase these systems to improve the productivity of their employees and reduce the cost of running their business. A key to achieving these objectives is the number of different tasks the systems can perform. Customers prefer systems for which a broad range of applications is available. With these systems, customers can choose applications based on performance, cost, employee skills, application integration requirements, and other individual needs.

In 1990 five RISC architectures were competing for market share. It was unlikely that all five would succeed in the workstation market. The application marketplace would be less attractive to application developers if it were fragmented by five architectures, because each version of an application would attract fewer customers.

In addition, customers began to feel that multiprocessor systems provided extra value because they offered a better price to performance ratio and higher ultimate performance than uniprocessor systems. Due to these market forces, Apple, Motorola, and IBM formed a partnership whose foundation is the use of a common architecture derived from the POWER architecture.

PowerPC Architecture

Early in 1991, Apple, Motorola, and IBM formed an alliance based on the use of a common processor architecture. Processor architects, compiler experts, operating system developers, processor designers, system architects, and system designers from the three companies worked together to develop an architecture that would meet the needs of the alliance. Because it would have been impossible to develop a completely new architecture in time to satisfy the needs of the their customers, the companies decided to use the POWER architecture as the starting point. Changes were made to achieve a number of goals. The architecture had to:

1. permit a broad range of implementations, from low–cost controllers to high–performance processors
2. be sufficiently simple to permit the design of processors that have a very short cycle time
3. minimize effects that hinder the design of aggressive superscalar implementations
4. include multiprocessor features
5. define a 64–bit architecture that is a superset of the 32–bit architecture, providing application binary compatibility for 32–bit applications

By analyzing the needs of applications and operating systems, considering typical instruction mixes, and inspecting application and operating system traces, the architecture group reached a consensus on the definition of the PowerPC architecture [5]. This architecture achieves the goals the listed above, yet permits POWER customers to run their existing applications on new systems and to run new applications on their existing systems.

Most of the POWER instructions were included in the PowerPC architecture. Nearly all the POWER instructions that were not included are instructions that were used infrequently and can be replaced by several other instructions that are in both architectures. The excluded instructions will cause an Illegal Instruction type Program Interrupt on PowerPC processors and will be emulated by the AIX operating system. Most POWER applications will benefit from the improved performance of new PowerPC processors. Others that frequently perform the operations listed below will produce correct results on PowerPC systems but may run slowly unless they are recompiled, because these operations used excluded instructions.

1. extended–precision bit string computation
2. extended–precision multiplication
3. integer division
4. generating or modifying instructions about to be executed

The first PowerPC processor, the 601, implements all but two of the non–privileged POWER instructions. It was designed as a "bridge" processor to allow application vendors additional time to recompile their products for PowerPC systems. Most existing POWER applications will run well on 601–based systems. Applications, that generate or modify code, use the POWER cache flush instruction which is not implemented in the 601. Such programs will produce correct results (the operation is emulated by AIX), but they may run slowly unless they are recompiled. As noted above, the 601 does implement nearly all the excluded non–privileged POWER instructions. However, these excluded instructions should not be used in new applications as they will not be implemented in other PowerPC processors.

PowerPC architecture is a 64–bit architecture. This architecture extends addressing and fixed–point computation to 64 bits, and supports dynamic switching between the 64–bit mode and the 32–bit mode. In 32–bit mode, a 64–bit PowerPC processor will execute application binaries compiled for the 32–bit subset architecture. Because a description of the entire architecture is too large to be addressed here, this paper concentrates on the descriptions of the changes that affect the user–mode 32–bit subset architecture.

The Slimming of POWER Architecture

As noted earlier, some POWER instructions were excluded from the PowerPC architecture. Some non–critical requirements in the POWER architecture were relaxed or removed. New instructions and features were added to PowerPC to complete the architecture and to achieve the goals set by the architecture group.

The first task undertaken by this group was to identify the features of the POWER architecture that were too restrictive, burdensome, or not cost–effective. This resulted in the exclusion of thirty–two problem–state instructions for reasons described in the remainder of this section.

The POWER architecture contained an MQ register which had two characteristics that would have complicated a super–scalar design: the use of the register was implicit, and it was a single resource. The instruction dispatching function in an aggressive design would have been required to identify all instructions that used the MQ and to implement a special renaming mechanism for it. The added complexity could not be justified by the functions the MQ provided. The exclusion of the MQ resulted in the exclusion of fifteen extended–precision shift instructions, two divide instructions, and one extended–precision multiply instruction.

Three of the POWER fixed–point rotate–insert instructions required three source operands. Two instructions that computed absolute values and two others that performed a subtract with a lower bound of zero required selection logic after the arithmetic computation and before the write–back to the register file. Including these instructions in PowerPC architecture would have made processor designs significantly more complex. The need for an extra source operand could have required an additional register file port and would have drastically increased the complexity of the renaming mechanism in super–scalar design. The need for selection logic after the arithmetic computation could have resulted in a longer minimum cycle time for some implementations.

The specification of the PowerPC memory management mechanism and other related changes resulted in the exclusion of the POWER cache management instructions. The database locking and tracking mechanism, another feature of the POWER memory model, was excluded because it would have made memory management significantly more complex but did not provide the granularity needed by most applications.

Five other seldom used problem–state instructions were excluded. The most complex instruction in the POWER architecture, Load String and Compare Byte, was one of these.

In some areas, PowerPC requirements are more flexible than POWER requirements. In the POWER architecture, an update form load could specify the same register for both target (RT) and base address (RA). If this occurred, the load was performed but the update was suppressed. The PowerPC architecture relaxed this requirement, recognizing that there is no reason to use the update form instruction in this case because the updated address will be discarded. In the PowerPC architecture, this use of the update form is not a valid operation and the result is not defined. A similar problem exists with an update form load and store instructions for which RA is GPR 0. If RA is GPR 0 the effective address computation uses the value 0 rather than the content of GPR 0. Saving updated effective address is senseless when the next use also ignores the value in RA (GPR 0); this operation is not valid in the PowerPC architecture and the result is not defined.

Another example of relaxed requirements deals with fields in instruction formats which are not necessary for a particular instruction. For example, many POWER instruction formats have an ”Rc” bit, which controls whether status information describing the result of the operation is recorded. In the POWER architecture, if the Rc bit is set to 1 in an instruction for which this status information is irrelevant (e.g., loads and stores), the instruction is executed correctly, but undefined values were used for the status information. In the PowerPC architecture, such instructions are considered ”invalid forms”. If an invalid form instruction is executed, the result is not defined, except that the requirements of memory protection and privilege are not violated.

The definition of the timing facility in the POWER architecture would have been very difficult to implement in an environment of power management and of processors clocked at rates that might be non–integral multiples of the system clock. The specification of the timing facility in the PowerPC architecture allows more flexibility to permit more cost–effective designs. A consequence of this approach is that PowerPC applications will normally have to use a system service to obtain the correct time, because direct use of the timing facility will not be possible without knowledge of environmental variables, available only to the system.

Because of the rapid progress of software and hardware technology, the cost to value relationship of some POWER architecture features changed. Relaxing some of the requirements and excluding some of the elements of the POWER architecture eliminated aspects of the POWER architecture that added significant complexity, that limited the speed of implementations, and that restricted the number of execution units that could be implemented in a superscalar design.

These exclusions created deficiencies in the resultant architecture. Also, the POWER architecture lacked other features necessary to compete in the fast moving marketplace. Low cost processors with good single–precision floating–point performance, and processors for use in symmetric multiprocessor systems were not possible using the POWER architecture.

Completing PowerPC Architecture

Features correcting both sets of deficiencies were added to the PowerPC architecture. In defining the additions, the architects defined features in a manner to avoid hardware complexity for handling software errors where it was easy for software to avoid the errors. An example is that software must put zeroes in the fields of instruction.

A full set of single–precision floating–point arithmetic instructions was added, as well as two instructions to convert floating–point values to integers. Because most of these instructions provide the single–precision version of existing double–precision instructions, the complexity added by the new instructions is minimal.

Fixed–point divide and multiply instructions were added to provide both normal and extended–precision computation without the complexities associated with the MQ register. Two other instructions were added to complete the fixed–point computation set.

Apart from the changes to the instruction set, the most significant changes made were in the memory model and the memory management definition. In the POWER architecture, the processor did not maintain data memory consistent with either I/O accesses or instruction fetches. Software had to manage memory consistency for both these areas. Before copying an area of memory to disk, software had to ensure that any copies of the memory area that were in the data cache and had been modified had been written to main memory. Before starting a read from disk, software had to ensure that the data cache did not contain a copy of any part of the memory area and had to invalidate any copy of the memory area in the instruction cache before restarting the program that requested the operation. POWER processors always accessed main memory through the caches. The only memory accesses that were performed without using the caches were accesses to a separate address space referred to as ”T=1 space”. This address space was used to perform programmed I/O (PIO) operations.

The PowerPC memory model was defined to provide greater flexibility. Attributes associated with each page of virtual memory allow software to control how accesses are performed. The model allows speculative access to any page unless it has an attribute indicating that it contains I/O or exhibits other volatile characteristics. Other attributes control whether a page may be cached, must not be cached, or may be cached with stores being written through to main memory. This definition makes it possible to map I/O into the main memory space. Processor–enforced data memory consistency was also included in the definition, relieving software of the responsibility for the consistency of memory with respect to I/O operations. As in the POWER memory model, the PowerPC memory model requires software to maintain instruction memory consistent with data memory. Programs that modify or generate instructions must ensure that the memory containing the new instructions is consistent before attempting to execute those instructions.

Because software is required to manage the consistency of instruction memory and the POWER cache management instructions were not carried over to PowerPC architecture, five new cache management instructions were defined.

The PowerPC memory model was defined to be weakly consistent, allowing memory accesses to complete out of order. The Enforce In–order Execution of I/O (eieio) instruction was defined to give software an efficient means of controlling the order in which accesses to I/O devices are completed.

Because POWER architecture lacked an atomic memory operation, two instructions were added to provide a means of performing a memory operation that appears to read and then modify a memory location as one atomic operation. These instruction are similar to those proposed in a report from the Lawrence Livermore National Laboratory [6]. Load Word and Reserve (lwarx) copies the content of the target memory location, which can be regarded as a lock variable, into a register and creates a reservation on that location. The program can use the loaded value to compute the new value to store in the lock variable. When a Store Word Conditional (stwcx) is executed, it attempts to store the new value. If the lock variable has not been modified since the lwarx completed, the store is performed, otherwise the store is not performed. In both cases status is recorded to indicate the result. If the stwcx fails to perform the store, the sequence can be repeated. Because these are new instructions that do not affect the binary compatibility of POWER applications, they were defined to access only word–aligned memory locations. If these instructions are used to access a location that is not word–aligned, the result of the access is not defined. This definition permits simpler implementations, avoiding the need for hardware checking for a programming error that is easily avoided.

The POWER memory model was big–endian. The final addition to the PowerPC memory model was to support both big–endian and little–endian memory models. A mode, controlled by software, specifies whether the current memory semantics are big–endian or little–endian. This permits the use of PowerPC processors in systems designed to run big–endian applications, little–endian applications, or both.

Summary

In early 1991 a group made up of technical leaders from Apple, Motorola, and IBM began work to develop a sleeker and cleaner architecture with expanded function, using the POWER architecture as a base. The PowerPC architecture, the result of this effort, permits a range of implementations from low–cost controllers through high–performance processors. It allows the implementation of processors targeted for desktop and notebook systems, yet it contains features to support the efficient implementation of processors for use in a range of multiprocessor systems.

References

1. *IBM RISC System/6000 Technology*, SA23–2619, IBM Corporation, 1990.

2. *IBM Journal of Research and Development*, Volume 34, 1990.
3. Moore, Balser, Muhich, and East, "IBM Single Chip RISC Processor (RSC)," *Proceedings of the International Conference on Computer Design*, 1992.
4. "POWER2: Next Generation of the RISC System/6000 Family," Steven W. White and Sudhir Dhawan, *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, p. ???
5. Paap and Silha, "PowerPC: A Performance Architecture," *Proceedings of the IEEE Computer Society International Computer Conference*, 1993.
6. "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors," Lawrence Livermore National Laboratory Report UCRL–97663, November 1987.

The PowerPC 601 Microprocessor

Charles R. Moore

The PowerPC 601 microprocessor is a highly integrated single chip processor that combines a powerful RISC architecture with a superscalar machine organization and a versatile, high performance bus interface. The processor contains a 32K–byte cache and is capable of dispatching, executing and completing up to three instructions per cycle. The bus interface can be configured for a wide range of system bus interfaces including pipelined, non–pipelined, and split transactions. The result is a cost effective, general purpose microprocessor solution that offers very competitive performance.

Introduction

The PowerPC 601 Microprocessor is the first member of a family of devices developed by IBM and Motorola in Austin, Texas. IBM and Motorola are committed to providing a wide range of processor designs that are compliant with the PowerPC Architecture [1].

The 601* project had several key goals. First, it was important that the 601 establish the PowerPC Architecture into the marketplace as early as possible. Second, the processor had to offer competitive performance at a low cost. Finally, the 601 had to be suitable for a wide range of system design points including those requiring support for symmetric multiprocessing.

To achieve these goals, the 601 project was put on an aggressive 12 month development schedule. An evolutionary path was taken from technology that existed at both IBM and Motorola. From IBM, the RISC Single Chip (RSC) Microprocessor [2] was taken as a base. The superscalar machine organization was enhanced to achieve greater performance, and additional custom circuit design was applied to reduce the die size and to allow higher frequency operation. From Motorola, substantial aspects of the 88110 Microprocessor bus interface were used as a base in the development of the 601 bus interface [3].

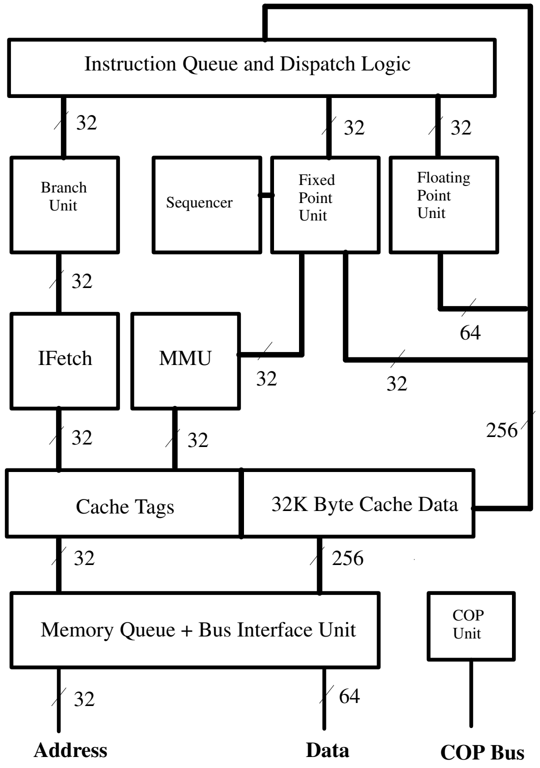


Figure 1 The PowerPC 601 Microprocessor

Block diagram

As shown in the high level block diagram (Figure 1), the 601 is a superscalar design with three pipelined execution units. Up to three 32–bit instructions can be dispatched each cycle – one each to the fixed–point unit (FXU), one to the floating–point unit (FPU) and one to the branch unit (BPU). The 32K–byte unified cache provides a 32–bit interface to the FXU, a 64–bit interface to the FPU, and a 256–bit interface to both the instruction queue and the memory queue. The chip I/O's include a 32–bit address bus and a 64–bit data bus. In addition, the chip supports a special asynchronous serial port, the *COP Bus*, to provide advanced debug and test features.

Pipeline description

The 601 pipeline structure has been optimized for high performance and concurrent instruction processing in each of the execution units (Figure 2). In general, the fixed–point unit serves as the master pipeline managing the synchronization control required to achieve precise exceptions. The fixed–point pipeline is used for all integer arithmetic logic unit (ALU) operations and all processor load and store instructions (including the floating–point loads and stores, although in some cases, they also progress through the floating–point pipeline). All hazards and dependencies are handled by the hardware, and forwarding logic is provided so that the integer ALU operations can operate in a fully pipelined manner. There is a one cycle penalty for a dependent operation following a load instruction.

The branch instruction pipeline has only two stages. The first stage can dispatch, decode, evaluate, and, if necessary predict the direction of a branch instruction in one cycle. On the next cycle, the resulting fetch can be accessing new instructions from the cache. This allows the processor to react quickly to branches detected in the instruction stream and to reduce the latency of subsequent instructions.

The floating–point instruction pipeline contains six stages and has been optimized for fully pipelined execution of single–precision operations. Although all double–precision operations are supported in hardware, and most are fully pipelined, those that involve multiplication are *double pumped* through the Execute1 and Execute2 stages of the pipe.

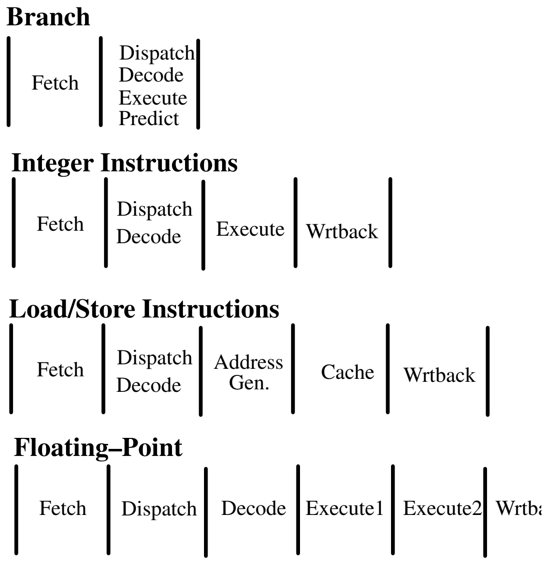


Figure 2 601 Pipeline Structure

Processor functional units

The 601 microprocessor comprises of several functional units as shown in the block diagram in Figure 1. Each of these functional units is described in the following sections.

<u>Instruction queue and dispatch unit</u>

The 601 contains an eight–entry instruction queue for holding prefetched instructions. The queue is fed by an eight–word bus from the cache. During each cycle, the dispatch logic considers the bottom four entries of the instruction queue and dispatches up to three instructions (Figure 3). There are no instruction alignment restrictions, and the queue supports all possible shift amounts.

Branch instructions and most floating–point instructions can be dispatched from any of the bottom four entries of the instruction queue. Fixed–point instructions are always dispatched from the bottom queue entry. Floating–point stores are dispatched to both the fixed–point unit (for address generation) and the floating–point unit (for data sourcing). Floating–point loads are dispatched only to the fixed–point unit, but the floating–point unit is made aware of any dependencies and data is forwarded directly to the floating–point unit.

The 601 supports *out of order* dispatch; Branch instructions and most floating–point instructions can be dispatched and removed from the queue even if a preceding fixed–point instruction is interlocked. These *folded* instructions are executed concurrently and do not occupy a position in the fixed–point pipeline. A unique tagging and counting mechanism is used to preserve the program order completion of these instructions. Although out of order dispatch is more complex to implement, it also allows the 601 to expose subsequent branches earlier which can reduce the potential dispatch stalls that may otherwise result.

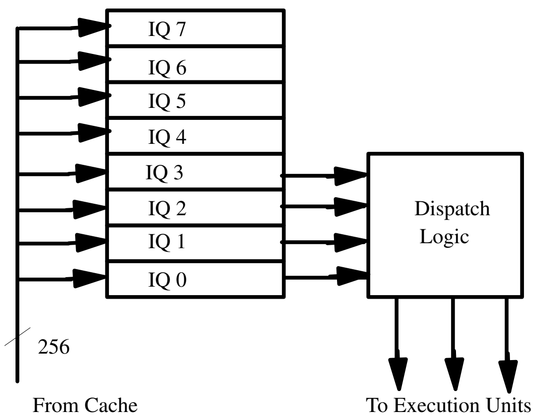


Figure 3 Instruction Queue and Dispatch

<u>Instruction fetch unit</u>

The instruction fetch unit coordinates instruction fetching from the cache. Instruction fetch addresses are generated by several different sources in the 601. The branch processor provides the address that results from branch instructions. The sequencer unit provides addresses associated with interrupts and other synchronizing events. The instruction fetcher itself generates the next sequential address if no branch or interrupt has occurred. Each cycle, the appropriate address is selected, translated and forwarded to the cache arbitration logic for consideration to access the cache. Instructions fetched from the cache are accepted and processed by the instruction queue and dispatch logic.

The instruction fetcher also provides means for address translation of instruction fetch addresses. The *translation shadow array* (TSA) automatically keeps track of the four most recently used instruction address translations, and provides associative comparison in parallel with the address generation of any instruction fetch. The TSA provides support for both page and block oriented address translations. In the event of a miss in the TSA, the instruction fetcher arbitrates for access to the 601's primary memory management unit for translation, and then updates the TSA based on a least recently used replacement algorithm.

Branch processing unit

The branch processing unit (BPU) executes all branch instructions. In the PowerPC Architecture, branch instructions can be conditional on a bit in the Condition Register, conditional on the state of the Count Register (useful in iterating loops), conditional on both registers, or simply unconditional. In addition, the branch target address can be absolute, program counter relative, or indirect from either the Link Register or the Count Register. Also, the execution of certain branch instructions require that the next sequential address to be automatically saved in the Link Register (this is useful for subroutine linkages). The branch processor is optimized to sort through the various interlocks and provide fast response to all types of branch instructions.

The branch processor can execute branches out of order with respect to the fixed–point unit. As a result, it is possible for the BPU to alter the Link and/or the Count Registers before all preceding fixed–point instructions have completed. To guarantee correct program operation when a preceding fixed–point instruction is dependent upon either the Link or Count Registers, a technique called *register renaming* is used to keep the architected values of these registers synchronized with the fixed–point unit. This technique is also used to restore the correct value of the Link and Count registers in the event that a preceding instructions takes an exception.

Unconditional branches and branches that are conditional only on the Count Register are completely executed during the same cycle that they are dispatched. As a result, these types of branch instructions are generally handled as *zero cycle branches*; they present no delay in the instruction dispatch stream to the fixed–point or the floating–point units.

Conditional branches that are dependent on the Condition Register can either be *immediately resolved* or *unresolved* at the time of dispatch. Each of the eight fields of the Condition Register has a set of associated interlocks which are activated by instructions that will eventually update that field. If a conditional branch is dependent upon a non–interlocked Condition Register field, then the branch can be immediately resolved. In this case, the condition is evaluated based on the contents of the Condition Register, and the branch is handled with performance characteristics similar to the unconditional branches described above.

On the other hand, branches conditional on an interlocked Condition Register field are considered to be unresolved. In these cases, the 601 employs a static branch prediction algorithm to predict the direction of the unresolved branch. The 601 algorithm essentially predicts that the branch will be ”taken” if the displacement of the target address is negative, and predicts ”not taken” if it is positive. As an aid for compiler directed prediction, a bit in the opcode of the branch instruction allows this prediction scheme to be reversed. Instructions fetched on behalf of a predicted conditional branch can be conditionally dispatched, but they are never executed until the prediction has been validated as correct.

Eventually, Condition Register interlocks are released and the validity of the prediction can be checked. If the prediction was correct, then the branch is finished and all prefetched instructions are marked valid and eligible for execution. If the prediction was incorrect, then any prefetched instructions are discarded and fetching resumes down the correct path. To ease the burden of mis–predicted conditional branches, the 601 employs several features to speed up recovery. First, a fast alternate address restore mechanism saves the alternate path address and quickly redirects the Instruction Fetcher. Second, the fixed–point unit is specially optimized to produce and forward the results of Compare instructions directly to the branch processor as well as the Condition Register. This permits earlier resolution of unresolved conditions. Finally, the instruction queue implements a delayed purge mechanism which retains any prefetched sequential instructions beyond the conditional branch until the prefetched predicted instructions are made available for loading into the instruction queue. If the condition is resolved before the predicted instructions are available (the likelihood of which is increased by the optimizations for the compare instructions in the fixed–point unit), and the prefetch was incorrectly predicted down the ”taken” path, then dispatch simply continues down the sequential stream using the instructions that are remaining in the instruction queue.

Fixed–point execution unit

The fixed–point execution unit (FXU) receives instructions from the instruction dispatch logic. The instructions are executed through a three stage pipeline (*decode, execute, writeback*) and all hazards are automatically interlocked by the hardware. All PowerPC Architecture instructions are handled in hardware, and most execute in fully pipelined manner. Some of the arithmetic (multiply and divide) and multiple–word storage operations require several cycles in the execute stage of the pipeline. Full data forwarding is provided between pipeline stages (for register dependencies) and from the cache unit (for load operations). In addition, although the differences between IBM's POWER Architecture [4] and the PowerPC Architecture at the user level are small, the 601 also provides hardware support for all POWER Architecture user mode instructions.

The address generation portion of all load and store instructions is handled by the FXU. In addition, it sequences the execution of the load/store multiple and the load/store string instructions. Most forms of unaligned addresses are automatically handled by the hardware; however, in some cases multiple accesses from the cache are required.

The fixed–point unit also features a fast execution of compare instructions and results forwarding to the branch processing unit. As discussed previously, this allows efficient handling of subsequent branches that are conditional on the result of the compare instruction.

Floating–point execution unit

The floating–point execution unit (FPU) is compliant with the IEEE–754 standard for both single–precision and double–precision floating–point arithmetic operations. In addition, the FPU supports the compound *multiply–add* operations that are defined in the PowerPC Architecture. Furthermore, all floating–point data types including denormalized numbers, NaNs, QNaNs and infinities are automatically handled by the hardware.

The FPU pipeline is divided into four stages. The *decode* stage contains 32 double–word registers, the instruction decode logic and the main pipeline control for the FPU. The *execute1 or multiply* stage contains the booth encoder and CSA (carry save adder) tree as well as an alignment shifter for the second operand. The *execute2 or add* stage accepts the sum and carry values from the previous stage, and produces a single intermediate result. Finally, the *writeback* stage performs result rounding, normalization, and the register update.

The floating–point unit is optimized so that most single–precision operations (all except divide) and most double–precision operations (those not involving multiplication or division) which involve operands with normal data types are fully pipelined to achieve a four cycle latency and a one cycle throughput. Double–precision operations involving multiplication are obtained by *double pumping* the execute1 and the execute2 stages of the pipeline and achieve a five cycle latency and two cycle throughput. Division is performed using a 2 bit non–restoring division algorithm which takes 17 cycles for single–precision and 31 cycles for double–precision. Due to silicon area constraints, no register renaming of floating–point registers is implemented in the 601 [5].

The floating–point unit receives instructions from the instruction dispatch logic. In many floating–point intensive applications, it is possible that the FPU will lag behind the full dispatch rate of the dispatch unit. To alleviate the adverse effects of this, the FPU maintains an additional two entry instruction queue. This allows the dispatch logic to remove floating–point instructions from the primary instruction queue and expose subsequent fixed–point and branch instructions earlier.

In general, the FPU operates independently of the FXU and can concurrently execute floating–point instructions. A carefully tuned synchronization scheme allows these execution units to progress relatively independent from one another, and still achieve the effect of precise interrupts. Although these units cooperate in the execution of floating–point storage access instructions, they are not required to operate on these instructions in lock step. For example, the FXU can process the addressing portion of a floating–point store operation and pass it onto the cache unit without waiting for the FPU to produce the store data. The cache unit will then the store operation once the data has been provided by the FPU.

The 601 also provides a mode that forces enabled floating–point exceptions to be reported in a precise manner. This mode is useful as a software debugging aid for floating–point programs producing degenerate results.

Memory management unit

The memory management unit (MMU) performs the virtual–to–real address translation for load and store instructions. It also acts as a backup for instruction fetch address translations that miss in the instruction fetcher's translation shadow array (TSA). The MMU is tightly coupled to the FXU and address translation occurs during the execute phase of the FXU pipeline. The cache access occurs the following cycle using the translated address for tag comparison. Arbitration logic determines whether the FXU or the instruction fetcher has use of the MMU on a particular cycle. It is important to note that the instruction fetcher's use of the MMU should be fairly low since the hit rate of the TSA should be fairly high for most applications.

The MMU provides support for segment oriented, page oriented and block oriented translations. In brief, address translation starts with indexing into one of sixteen Segment Registers. Control bits in the Segment Register allow the access to map immediately to a real address, map to I/O space, or to extend the translation process onto the paging and block translation mechanism.

The PowerPC Architecture specifies a *hashed page table* structure and a fixed page size of 4K bytes. Portions of the virtual address are used to hash into the page table entries which are subsequently examined for matching virtual page numbers. A matching entry produces a real page number which can then be used to access real memory. If no match is found after 16 attempts, a page–fault exception is taken. To speed up the page–based address translation mechanism, the 601 implements a 256–entry, 2–way set–associative *translation look–aside buffer* (TLB) cache that is used for both instructions and data. Updates to the TLB and maintenance of the storage access recording bits (that is, the *reference* and *change* bits) are automatically handled in hardware.

The block address translation mechanism allows up to four block mappings that can range in size from 128K bytes to 8M bytes. This is useful for handling large sections of memory that are not subject to demand paging but still require address translation and protection. Proper usage of the block translation registers can reduce TLB thrashing and increase overall processor performance.

Cache structure

The 601 provides a 32K–byte, eight–way set–associative, unified cache. The line size is 64 bytes, and each line is split into two 32–byte sectors. The cache is indexed with a real (or physical) address and the tags are associated with the real address as well. In general, a store–in (or copy–back) policy is followed with respect to store data; however, the cache allows for page and block level control of cacheability, write through, and coherency (via the MMU). The cache replacement policy is LRU (least recently used), and coherency is maintained on a sector basis using the 4 state MESI (modified, exclusive, shared, invalid) cache coherency protocol.

In any unified cache implementation, it is important to make the most effective use of the available cache bandwidth. The 601 uses several techniques to achieve this. First, the cache itself presents an eight–word read interface, which at 50MHz allows an effective 1.6G byte/second data rate. Although the entire eight words are not used every cycle, this bandwidth is especially important for sector cast–outs, snoop pushes and instruction fetching. Second, the cache is capable of performing a complete read–modify–write every cycle which allows the 601 to complete store operations using a single cache cycle. Third, a balanced arbitration scheme has been developed to prioritize the various cache access requests that can occur each cycle. To minimize the effects of cache arbitration stalls from holding up the execution units, operation queuing is provided above the cache for fixed–point and floating–point storage operations. Finally, the cache is designed to be non–blocking, and a five–entry memory queue sits below the cache to hold pending storage operations.

The 601 cache supports a full range of cache control operations including ”flush”, ”clean”, ”kill”, ”touch”, and ”allocate with zeros”. These are useful for applications that understand the performance aspects of the memory hierarchy, and for systems that perform software assisted coherency schemes.

The cache is a full custom design that exploits leading edge circuit design techniques to achieve the required density and performance characteristics. The cache uses a proven six transistor static RAM cell, and the entire cache is also protected with byte parity for enhanced error detection and reliability. In addition, the cache itself is subject to a deterministic array self–test as part of the power–on reset process.

Memory queue

The memory queue cooperates with the cache and the bus interface unit to effectively handle operations that require access to or from the 601 bus interface. It allows the processor to decouple the cache from the bus interface, which improves the overall effectiveness of the cache.

The memory queue is made up of a two–entry read queue and a three–entry write queue. The write queue holds up to three 32–byte sectors of dirty data. The read queue, the write queue, and in some cases, the cache itself arbitrate for access to the 601 bus. Strict program order is not a consideration in the arbitration logic, which allows dependent read operations to proceed ahead of pending write operations. This can substantially reduce the latency of storage operations that miss in the level 1 (L1) cache, increasing the overall processor performance. The memory queue is maintained fully coherent with the processor, the cache, and memory automatically by the hardware.

The memory queue also supports a feature called *dynamic reload* which permits background fetching of the ”other” sector of a cache line after the critical sector has been successfully loaded. When enabled, this feature provides an automated cache prefetching capability for either instructions or data.

Bus interface unit

The bus interface unit (BIU) converts operations in the memory queue into transactions on the 601 bus. The bus protocols used by the 601 are based on the interface defined by the Motorola 88110 microprocessor chip [3]. This interface has been chosen for several reasons. First, it provides a general purpose framework for system level design alternatives. Second, the framework is easily extendable, so that several 601 specific features could be added without disrupting the overall footprint of the protocols. Finally, the base technology is readily available, meets the objectives for the 601 and has been proven in the 88110. This was critically important for achieving the aggressive development cycle, and it also allowed an easy migration path for some of the early users of the part.

The interface defines a 32–bit address bus and a 64–bit data bus. These buses are logically and physically decoupled from one another so that the protocol can support system bus organizations that use pipelined, non–pipelined or even split bus transactions. To reduce latency and increase performance, the 601 itself is capable of pipelining up to two outstanding operations onto the bus.

A variety of address coincident signals are defined to qualify the type of bus transaction in progress. These signals identify the specific transaction type, the size of the transaction, and the storage control attributes (cacheable, write through, and coherent). An *address retry* mechanism is defined to allow cancellation of bus transactions. This is used to enforce and adhere to the MESI cache coherency protocol. The interface presents sufficient information about the state of the internal L1 cache to allow an external L2 cache to maintain inclusion and perform effective snoop filtering. Optional byte parity is supported on both the address and the data bus.

In addition, the bus interface can be configured to operate at integer multiples of the processor cycle. As a result, simple bus structures can be constructed using minimal external control logic. More sophisticated and higher performance structures can be created using more advanced external control. At 50 MHz, a full speed, pipelined system bus can achieve and sustain a 320M byte/second data rate.

<u>Sequencer unit</u>

The sequencer unit is essentially an embedded support processor that assists the core CPU in handling many of the algorithmic and area intensive functions of the PowerPC Architecture. The sequencer contains a 1K entry micro–code ROM, 8 single–word general purpose registers, a 32–word private RAM, and the control logic required to execute the robust 18–bit instruction set.

Although the scope of functions that the sequencer unit performs on the 601 is greatly reduced from those that were handled on the RSC chip, the sequencer unit was retained to minimize the redesign of these required functions. For example, it sequences the power on reset functions which include array self–test for the cache, and array initialization for all other storage cells on the chip. It maintains an on–chip real time clock (RTC) and the system decrementor function. In addition, it handles the recording and sequencing of interrupts, context synchronizing events and errors.

Relative to the processor core, the sequencer performs the table–walk through the hashed page table for address translations that miss in the TLB, and it performs the update of the storage access recording bits as appropriate. In addition, to minimize the chip area overhead, several of the less frequently used system control registers are physically implemented in the RAM structure within the sequencer. As a result, the instructions that operate with these registers are passed from the FXU to the sequencer for execution assist.

<u>COP unit</u>

The Common on–chip processor (COP) is the master control logic for the built–in self–test (BIST), debug and test features of the 601 chip. A simple serial command interface allows external devices to communicate with the COP and initiate various actions. This command interface can be configured to be compatible with the IEEE standard for JTAG access to boundary scan.

The COP contains a *linear feedback shift register* (LFSR), a *multiple input signature register* (MISR) and the control logic required to sequence such BIST operations as DC logic self test, AC logic self–test, array self test, array initiation and chip to chip wiring tests. In addition, the COP provides the capability to stop and start the internal clocks and to dump the state of all registers, RAMs and register files (via the scan strings) on the chip. These features combine to create a powerful debug environment that provides excellent visibility into the chip and its specific capabilities.

Multiprocessor features

One of the key goals of the PowerPC 601 Microprocessor was to integrate features to facilitate symmetric multiprocessor based systems. In general, these systems require an ability to efficiently maintain memory coherency, an ability to perform atomic memory references, and an ability to control the order in which storage operations are presented onto the interface [6]. These features are described in the following sections.

<u>Memory coherency</u>

The 601 participates in maintaining memory coherency in several ways. First, the processor performs bus snooping and adheres to a four state MESI cache coherency protocol. To increase performance, the 601 provides a second port into the cache directory that is dedicated for snooping purposes. This serves to isolate the normal processor operation during non–offending bus transactions.

Second, the processor provides support for a full range of cache control operations. These instructions are useful for some operating systems (and even some applications) that require an ability to perform software oriented memory coherency. In addition, the 601 *broadcasts* these operations onto the bus interface. This allows the operations to be exposed to subsequent levels of the cache hierarchy as well as to other processors on the bus. On the snooping side, the 601 will detect these bus transactions and will perform the required function with respect to the cache internal to that processor.

Finally, the 601 allows for page and/or block level control of cacheability, write through and coherency. These can be effectively used to manage special sections of the memory space. For example, the cacheability control is useful for memory mapped I/O regions, and the write through control is useful for simple graphics frame buffers. Selective coherency control can be used to increase overall system performance by enforcing coherency only on the sections of memory that require it.

Atomic storage access and synchronization

For atomic storage access (that is, an undisturbed read, modify, write operation), the PowerPC Architecture defines a *storage reservation mechanism* that is independent of the caching mechanism. The *load and reserve* instruction performs a normal load operation, but also makes a reservation for the address specified by the load address. The reservation is maintained valid until the processor detects that the address has been written to. The *store conditional* instruction performs a store operation that is conditional on the validity of the reservation. If the reservation is not valid (that is, another device has already written to the reserved address), then the store is not performed and a bit is reset in the Condition Register. If the reservation is valid, then the store is performed and the bit in the Condition Register is set. These operations can be combined with a conditional branch instruction to create sequences of code which perform traditional synchronization primitives such as compare and swap, exchange memory, and fetch and add.

In general, the 601 follows a weakly ordered storage model. This allows the processor to more effectively utilize the available bus bandwidth, reduce the latency of storage operations that miss in the cache, and, in general, increase performance. However, in some cases, software requires the ability to control storage ordering. To achieve the effect of strong ordering, the 601 provides support for an instruction which acts as a barrier to prevent reordering of storage operations across it. In addition, a stronger storage synchronization instruction is also provided which serves to force all preceding storage operations to complete before any proceeding ones can be processed.

TLB coherency

As previously mentioned, the 601's MMU automatically maintains a TLB cache for recently used page table based address translations. In any system (uniprocessor or multiprocessor), it is important that the capability to maintain coherency between the TLB and the page tables in memory is provided. In the 601 processor, the *TLB Invalidate (TLBI)* instruction allows software to selectively invalidate entries from the TLB. As changes are made to the page table, the TLBI instruction can be used to enforce coherency.

In multiprocessor systems, the TLB coherency problem is intensified since several processors' TLBs could contain information that is being modified in storage by another processor. In an effort to assist in handling these situations efficiently, the 601 broadcasts the essence of the TLBI instruction onto the bus interface. On the snooping side, the 601 will detect the TLBI bus operation and will perform the required function with respect to its own internal TLB. This can reduce the number of inter–processor interrupts in a symmetric multiprocessor systems and increase the overall performance.

Chip and packaging technology

The 601 microprocessor is implemented in an IBM CMOS technology with .6 micron minimum feature size and four levels of metal wiring. The design is fully static and LSSD compliant. The 601 is packaged in a 304 pin ceramic quad flat pack. Table 1 summarizes other key physical characteristics of the 601.

Performance

This paper has described key features of the 601 microprocessor. These features can be combined in many different ways to achieve a range of performance characteristics. Other factors, such as the operating system, memory characteristics, system support and operating frequency also play an important role in the overall performance of the system.

To establish a rough basis for comparison, Table 2 provides estimated SPEC™ performance for the 601–based IBM RISC

System/6000 Model 250 [7] which does not contain a second–level cache. Systems which support a second–level cache may achieve higher performance levels.

Conclusion

The PowerPC 601 Microprocessor is the first member of a family of PowerPC processors. The implementation uses state of the art CMOS technology to achieve an extremely compact single chip design. The superscalar machine organization, large internal cache and versatile bus interface provide competitive performance at a low cost. As a result, the 601 achieves an outstanding price/performance ratio and is well suited for a wide range of system design points.

Technology	.6 um CMOS, 4 levels of metal
Die Size	10.95mm x 10.95mm
Transistors	2.8 Million
Voltage	3.6v
Power	6.5 Watts @ 50MHz (typical)
Package	304 pin Quad Flat Pack
Frequency	50 Mhz and 66 Mhz initially
I/Os	184 Signals CMOS/TTL levels compatible

Table 1 601 Physical Characteristics

Frequency	SPECint92	SPECfp92
66 MHz	>60*	>70*

*Estimated based on simulated results

Table 2 601 Performance Estimates

Acknowledgements

This was a special project. It had aggressive functionality, performance and schedule goals. It required the cooperation of many people from several companies and several divisions within IBM. In addition, the project was viewed as a barometer for the health of the overall relationship of the alliance. In the end, the team stepped up to these expectations and met all of the project goals.

The author would like to acknowledge all of the logical design, physical design, methodology support and verification engineers that worked on the 601 processor. This project would not have been possible without the hard work and personal sacrifices made by these people. In particular, John Muhich provided much of the architectural and organizational framework for the chip. Ray East, Doug Balser, Mark Papermaster, Susan Tiner, Michael Becker and David Tuttle provided exceptional management focus and vision. Chris Olson provided key expertise in the implementation of the floating–point unit. Brian Vicknair, Tim Elliott and Ed O'Neil designed the cache unit. Jack Wilson and Mike Vaden were the designers of the fixed–point unit and the memory management unit. Terence Potter and Tan Chu designed the branch processor and instruction fetcher. John Zimmerman and Tony Penaloza were responsible for the sequencer unit and the microcode. Rob Reese, Michael Garcia, Michael Allen and Larry Merkel did an outstanding job on the bus interface unit. Richard Billings, Eddie Rodriguez and Casey Nguyen provided expertise in the design of the COP and in the test features of the chip. Tim Brodnax and Charlie Roth drove the logic tuning and timing analysis effort. Mike Schiffli, Joe Werner, Frank Palermo, Scott Glenn, Chris Byant, Brian Reynolds, Larry Powell, Tom Sartorius, John Meyer, Peter Voldstadt, Bob Young, Hai Vu, and Robert Keist were key contributors to the design verification effort. Gavin Meil and Jeff Brooks took care of all aspects of the hardware verification. Finally, Floyd Watson and PT Patel led the chip integration and physical design efforts.

References

1. Silha, E. and Paap, G., "PowerPC: A Performance Architecture", *Proceedings of COMPCON 1993, February 1993.*
2. Moore, C.R., Balser, D.M., Muhich, J.S., and East R.E., "IBM Single Chip RISC Processor (RSC)", *Proceedings of the 1992 International Conference on Computer Design*, 1992
3. Diefendorff, K. and Allen, M., "Organization of the Motorola 88110 Superscalar RISC Microprocessor", *IEEE Micro*, Vol. 12, No. 2, April 1992, pp. 40–63.
4. Oehler, R. R., and Groves, R. D., "IBM RISC System / 6000 Processor Architecture", *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 23–36, January 1990.
5. Grohoski, G.F., "Machine Organization of the IBM RISC System / 6000 Processor" *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 37–58, January 1990.
6. Allen, M., and Becker, M., "Multiprocessing Aspects of the PowerPC 601 Microprocessor", *Proceedings of COMPCON 1993*, February 1993.
7. Hanna, J., "RISC System/6000 Model 250: The First PowerPC System," IBM RISC System/6000 Technology: Volume II, IBM Corporation,1993, pp. ??–??.

* PowerPC 601 Microprocessors are available from IBM as part number PPC601 (hereinafter "601").

RISC System/6000 Model 250: The First PowerPC System

Jim Hanna

Introduction

The introduction of the IBM RISC System/6000 Model 250 extends IBM's existing Model 220/230 products into the PowerPC arena. The system is designed to achieve excellent performance at an attractive system cost with the first PowerPC processor chip, enhancing the low end of the RISC System/6000 family. This paper describes some of the key features of the Model 250 system design.

The RISC System/6000 Model 250 design addresses a number of product objectives associated with the desire to exploit the new PowerPC processor in a RISC System/6000 product. The key objectives include software compatibility, performance, and cost.

Minimizing the change to existing RISC System/6000 system software is an important consideration. The PowerPC 601 processor chip supports this objective by implementing most POWER Architecture instructions in addition to the 32–bit PowerPC instruction set. System level design largely preserves the existing RISC System/6000 I/O model. Even though simplification of the I/O structure requires changes to AIX Kernel I/O Services, compatibility is maintained for existing Device Drivers, other AIX subsystems, and, most importantly, applications.

Performance is enhanced over previous low–end RISC System/6000 products. The use of the PowerPC 601 processor, with its larger on–chip cache and higher operating frequency, is the major contributor to the performance improvement. System level design provides a higher performance memory subsystem, increased bandwidth to graphics, and a higher performance I/O controller.

A final objective is to deliver this performance at a competitive price. The 601 processor, with its small die size, is very competitively priced. The reuse of I/O components, the power supply, and the mechanical packaging developed originally for the Model 220 all allow for the upgrade of existing 220/230 systems to PowerPC and provide an attractive system level cost for new systems.

System Overview

The Model 250 system structure, shown in Figure 1, is driven by the desire for good performance at a reasonable cost. The processor and the optional graphics subsystem are directly attached to system memory with a multi–master System Bus. A high performance Micro Channel Bridge chip attaches the I/O subsystem to the System Bus. Each of the components shown in Figure 1 are described in more detail below.

PowerPC 601 Processor

The PowerPC 601 processor chip is described more fully in a companion article [1], but a few of the features influencing the Model 250 system design are mentioned here. The Motorola 88110 based processor interface used on the 601 is robust enough both functionally and electrically to be extended into the low cost, high performance Model 250 System Bus. The ability of the 601 to operate its interface more slowly than the base processor frequency is utilized in the system design; the Model 250 runs the 601 at 66 MHz and the System Bus at 33 MHz. The inclusion of a large, 32KB, 8–way set–associative cache enables a system to achieve competitive performance, even without an expensive second level cache. The low power consumption of the 601 allows the Model 250 to use the power supply and mechanical packaging previously designed for the Model 220/230.

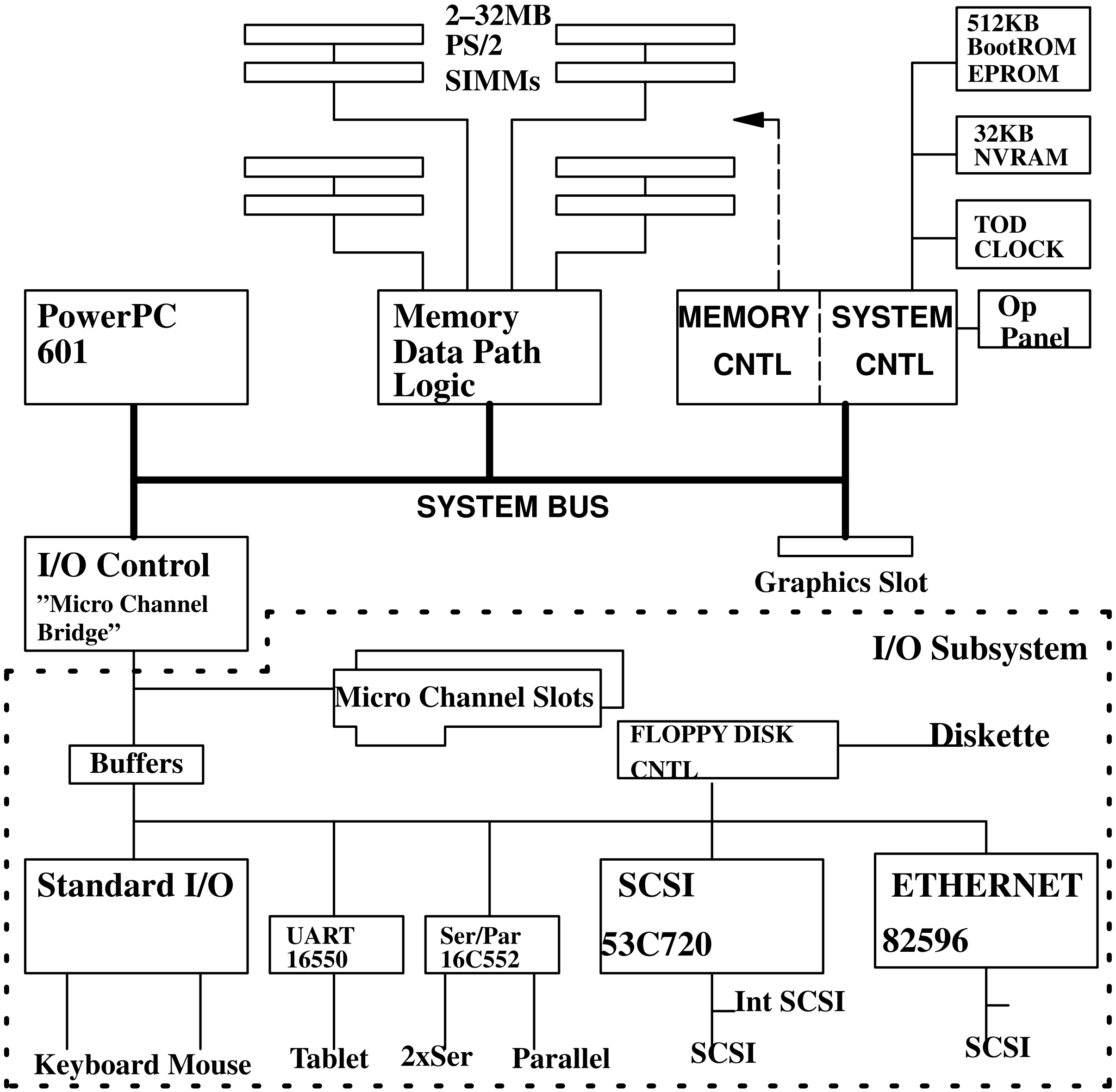


Figure 1 Model 250 Block Diagram

System Bus

The System Bus used in the Model 250 is an extension of the PowerPC Processor Interface, which in turn is a derivative of the Motorola 88110 interface. The System Bus is synchronous with an 8–byte bidirectional data bus and an independent 32–bit address bus. In addition to the basic processor functions of memory Load/Store and Programmed I/O Load/Store, the Model 250 System Bus supports interrupt routing, byte steering for 4–byte devices, and protocols for system configuration. Memory coherency is maintained by a 4–state snooping protocol.

The System Bus operates at 33 MHz in the Model 250 implementation. Single level pipelining supports four 8–byte data transfers every five bus clocks, for a maximum System Bus bandwidth of 213 MB/S.

The address and data buses have byte parity to ensure data integrity. A 'data retry' function is provided to allow forwarding of memory data in parallel with Error Checking and Correction (ECC) checking.

Electrically, the System Bus is driven directly by standard CMOS drivers to TTL levels. AC characteristics at these frequencies are dominated by transmission line effects, and the physical topology of the bus is managed very closely to guarantee that signal timings are met. A separate bus clock signal with controlled skew is provided to each device on the System Bus from the same phase–locked–loop clock generator that provides the processor clock.

Memory

The RISC System/6000 Model 250 can have from 16MB to 256MB of system memory installed. As in previous 2xx models, standard 36–bit DRAM single in–line memory modules (SIMMs) from the Personal Computer industry are used to minimize cost. 2MB PS/2 style SIMMs can be used, as well as the 4MB and 8MB PS/2 SIMMs that were used in previous models. In addition, new JEDEC standard 16MB and 32MB SIMMs can be used. Up to eight SIMMs can be installed for maximum capacity.

Interleaving provides sufficient memory subsystem performance from standard 70 nanosecond DRAMs. Two banks of two SIMMs each are implemented, requiring the SIMMs to be installed in matched sets of four. The memory controller is optimized to support the cache sector fill/flush characteristics of the 601 processor using a 32–byte Load/Store buffer. These features yield a memory bandwidth of 142MB/S for Loads and 213MB/S for Stores.

The memory subsystem implements ECC. Single–bit errors are corrected "on the fly" for loads from memory, using the data retry capability of the System Bus. Double–bit errors cause a Machine Check to preserve data integrity. ECC codes are maintained on 8–byte double–words, so that no additional DRAM cost penalty is incurred compared to byte parity. Read–modify–write operations are performed in the memory subsystem to enable stores of less than 8 bytes.

The memory controller is implemented in two Application Specific Integrated Circuits (ASICs). One of these ASICs contains all of the data path logic, including the Load/Store buffer and ECC check/generate logic. The second ASIC contains address and control functions of the memory controller in addition to other system control functions described below.

System Control

The Model 250 implements system control functions, including the System Bus arbiter and the system interrupt controller, in the same ASIC that contains the memory control logic.

The System Bus requires a centralized arbiter that grants ownership of bus resources to contending devices and controls pipelining. It manages the various System Bus functions such as interrupt transactions, extended (PIO) cycles, and snooping in a way that promotes efficient use of the System Bus while providing fair access for all devices.

The system interrupt controller design is based on routing interrupt requests over the System Bus. The controller fields interrupt request transactions from devices attached to the System Bus. Then, based on the current operating priority of the processor, it either activates the interrupt signal to the processor or generates an interrupt reset transaction back to the interrupting device, instructing it to retry the request later. The interrupt controller also sends a reset transaction to the interrupting device when the interrupt has been successfully serviced by the processor.

Other functions implemented in the system control ASIC include support for system configuration and the interface to the Operator Panel. In addition, this chip provides the interface that allows processor access to standard system resources as the Time of Day Clock, Non–Volatile RAM (NVRAM), and BootROM.

BootROM

The BootROM on Model 250 is based on the Model 220/230 design. The power on self test (POST) is modified to accommodate the 601 processor and new system ASICs. The POST tests all system elements which are required to load the operating system boot image. These elements include the processor, system memory, I/O Controller, and any I/O Device that may be required for the boot process, including network devices. The results of this testing and configuration activity are either passed to the operating system through the enhanced IPL Control Block in main memory, or sent to the Operator Panel LED Display to assist system diagnosis.

Following system verification and configuration, the operating system boot image is loaded into system memory. Device Driver capability sufficient to transfer the boot image from media to system memory is included either in the BootROM firmware itself, or in ROM located on I/O Devices which can be accessed with a ROM scan technique. The location of the boot image is chosen by prioritized boot lists maintained in NVRAM and selected by the position of the Operator Panel key switch. Menus are provided that allow the user to input network parameters necessary for booting over certain types of networks.

Graphics

The 2D Graphics Adapter developed for the Model 250 is more fully covered in a companion article [2]. It provides 8–bit color graphics with both 1024x768 and 1280x1024 resolutions. The Model 250 continues the Model 220/230 strategy of attaching low cost graphics to a high–performance, low–latency local bus. The Model 250 improves on the Model 220/230 design by attaching graphics directly to the System Bus, which provides higher performance levels than the local I/O buses used previously, and by incorporating a more powerful, single chip rasterizer. This combination provides excellent X–windows performance, with about 1.6 million X11 2D vectors/second.

I/O Subsystem

As with previous RISC System/6000 products, the I/O Subsystem is based on the Micro Channel I/O Bus. The RISC System/6000 I/O Architecture is simplified for PowerPC systems to allow reduced hardware costs and to optimize software path lengths with hardware enforced coherency. The performance and error recovery capabilities of RISC System/6000 I/O Architecture are retained.

A new I/O Control ASIC bridges between the Micro Channel Bus and the System Bus. It contains the Micro Channel Bus arbiter, DMA controller, interrupt controller and configuration support. Programmed I/O is supported to the Micro Channel through extended protocol transactions on the System Bus. Programmed I/O performance is aided by a 64–byte PIO buffer. Micro Channel Bus Masters access system memory through the I/O control ASIC acting as a memory slave. High performance access to System Bus attached memory is assisted by a 64–byte bus master cache and a 32–byte DMA slave buffer. 8–byte 10MHz Streaming Data protocol is supported for a peak data rate of 80 MB/S.

The standard complement of RISC System/6000 integrated I/O devices is provided on the Model 250. A keyboard, mouse, tablet, parallel port and two serial ports are provided. The integrated Ethernet function is improved with the addition of standard Twisted Pair media (10BaseT) attach capability. Fast capability is added to the integrated SCSI (small computer systems interface), improving maximum SCSI bandwidth to 10MB/S.

Technology/Packaging

Four system level ASICs were designed for the Model 250 in addition to the PowerPC 601 processor chip [1] and the 2D Rasterizer [2]. These chips use IBM's 0.8um, 3 level metal CMOS technology. Logic functions are partitioned to fit in cost effective Quad Flat Packs. Three of the four new ASICs have 304 pins and one has 240 pins. Each chip dissipates between 1 and 2 watts at 3.6 volts.

The base system, including the processor, processor support, memory, and standard I/O, is packaged on a single 8.5" x 11" planar, using 5 signal planes and 3 ground planes. High frequency System Bus components are separated from the rear I/O connector strip as much as possible to limit electromagnetic emissions. The planar layout is shown in Figure 2.

The Model 250 planar is designed to fit in the existing compact Model 220/230 cover set shown in Figure 3.

Summary

The RISC System/6000 Model 250 is the first system based on IBM's new PowerPC technology. It maintains application binary compatibility with previous RISC

System/6000 systems. The performance levels of the new processor and system design, along with the cost effective system design based on Model 220/230, provide excellent price/performance in a UNIX workstation.

References

1. Charles R. Moore, "The PowerPC 601 Microprocessor," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??

2. Charles R. Johns and Taggart Robertson, "The PAX 2D Rendering Engine," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??–??.

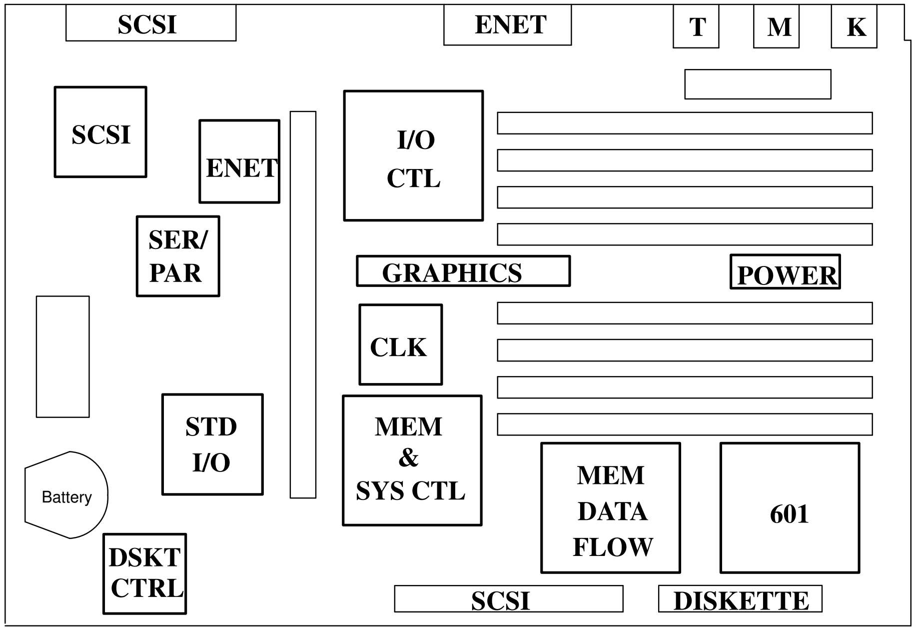


Figure 2 RISC System/6000 Model 250 Planar Layout

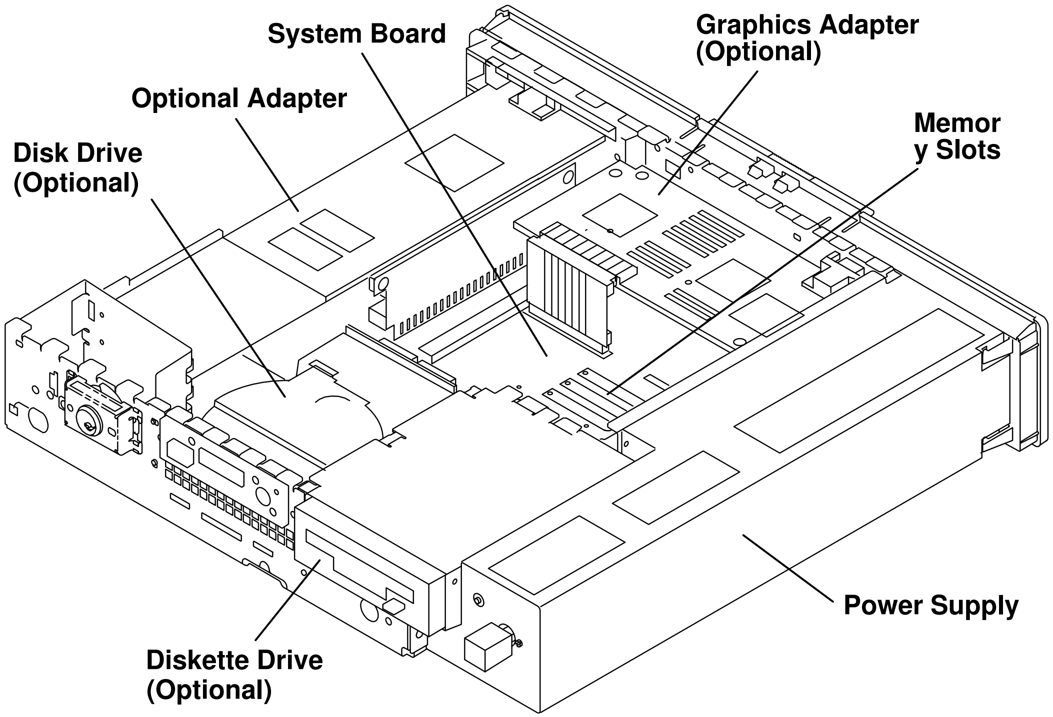


Figure 3 RISC System/6000 Model 250 Mechanical Packaging

Migration and Compatibility

John Reysa, Da–Gung Lu, Kate Stewart, Ahmed Chibib, and Sohel Saiyed

Introduction

The introduction of the RISC System/6000 (RS/6000) Models 250, 58H, 590, and 990 marks a significant milestone in the successful POWER architecture history. The Models 58H, 590, and 990 establish new workstation performance levels using the new POWER2 implementation of the POWER architecture. The high–performance, low–cost Model 250 introduces the first PowerPC 601 implementation of the PowerPC architecture.

The new models are upward compatible; existing POWER applications will run on all RS/6000 models with impressive performance. Furthermore, compiler enhancements can preserve binary compatibility across all platforms. For applications where optimal performance is desired and binary compatibility is not essential, the new release of the compilers allow the user to ”target” specific implementations.

This article has four major components. The ”Implementation Goals” section discusses the new systems in relation to the existing RS/6000 family. The ”Architectural Relationship” section describes the differences in instruction sets for the chip implementations. The ”Existing Application Binary” section shows that users can obtain performance improvements on existing applications without recompilation. Finally, the ”Considerations for Recompilation” section discusses the performance / compatibility tradeoffs when recompiling.

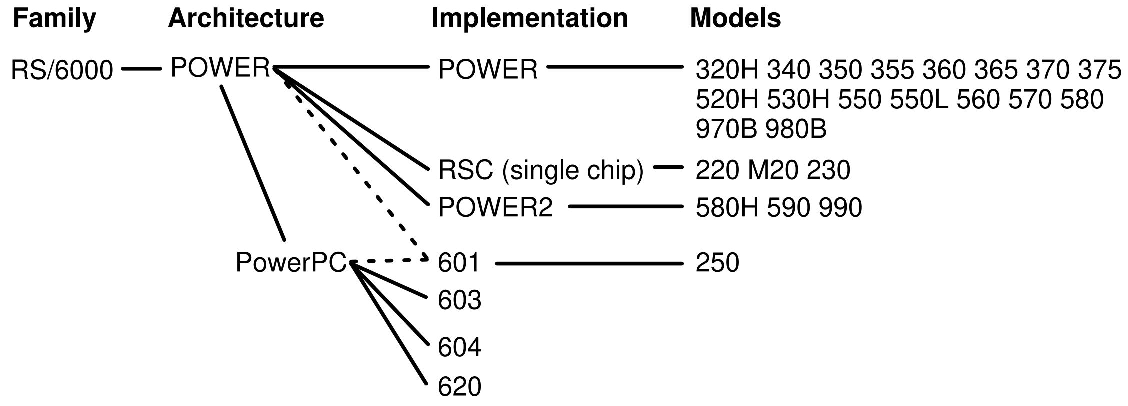


Figure 1 RISC System/6000 Family

Implementation Goals

Before describing compatibility issues, the relationship of the POWER2 and PowerPC601 implementations in the RS/6000 family should be understood. The entire RS/6000 family originates from the POWER architecture base as shown in Figure 1. The current implementation of the POWER architecture includes models such as the 375, 580, and 980B. The RISC Single Chip (RSC) implementation of the POWER architecture is found in the Models 220 and 230.

To provide industry leading performance for both scientific and commercial applications, POWER2 enhancements include increased data cache capacity, additional functional units, and improved branching characteristics. The POWER2–based systems, Models 58H, 590, and 990, improve both scientific and commercial performance on existing applications. Additional compute–intensive performance can be obtained by using POWER2’s instructions extensions [1].

The PowerPC architecture also leverages the POWER architecture base and the existing

RS/6000 applications. Motivations for the PowerPC architecture changes included improved single–precision performance and multiprocessor support. Several papers provide more information on the changes and motivation behind the PowerPC architecture [2,3].

The dotted lines in Figure 1 indicate that PowerPC 601 implementation used in the Model 250 ”bridges” between the POWER and PowerPC architectures. PowerPC 601's support of both architectures is described more in the following section. The future 603, 604, and 620 implementations of the PowerPC architecture will support the PowerPC instructions set and span a broader range of price / performance targets [3].

Architectural Relationship

Figure 2 provides a pictorial representation of the instructions implemented by the POWER, the POWER2, the PowerPC 601, and the other PowerPC implementations. The figure also indicates that the compiler's new **–qarch** option can select between the four sets of instructions.

The original POWER instruction set is shown at the top. The entire set of POWER instructions will execute on any RS/6000 platform, although some PowerPC platforms will emulate the instructions, resulting in lower performance.

The POWER2 implementation (second row) provides hardware Square Root, convert to integer, and quad–word load/store instructions. None of the additional instructions will run on the POWER implementation and most will not run on the PowerPC implementations.

Consistent with the Reduced Instruction Set Computer (RISC) philosophy, the PowerPC architecture (third row in Figure 2) removed several of the seldom used POWER instructions. To maintain compatibility, the future PowerPC implementations (603, 604, and 620) will execute the removed POWER instructions by emulation within the operating system. The additional PowerPC architecture 32–bit instructions include those which enable better single–precision floating–point performance and multiprocessing support. The PowerPC architecture also defines a set of 64–bit instructions for future implementations. PowerPC's new 32–bit and 64–bit instructions will not execute on the POWER and POWER2 implementations.

The PowerPC 601 processor provides a ”bridge” between the POWER and PowerPC architectures by supporting most of the

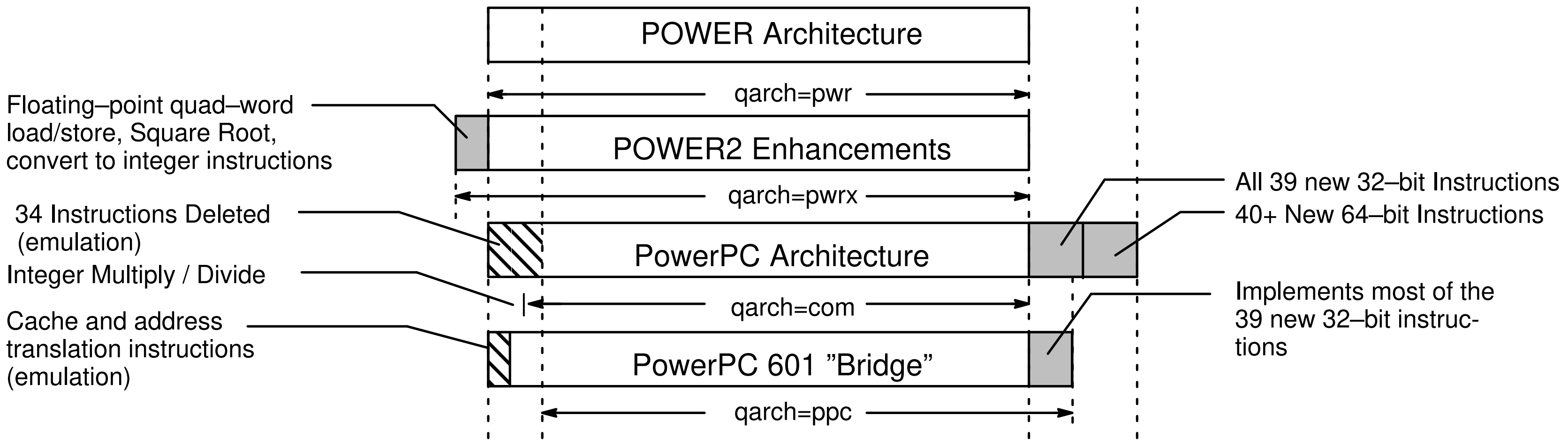


Figure 2 Architecture Relationships

PowerPC and POWER instructions (fourth row in Figure 2). The PowerPC 601 executes all compiler generated user–level POWER instructions. The implementation also supports all but a few of the 32–bit PowerPC instructions. Existing binaries, which perform well on the PowerPC 601, may show a degradation on future PowerPC implementations using emulation. However, the PowerPC 601 ”bridge” allows time for software developers to recompile to a more ”common” set of instructions.

Existing Application Binaries

Binary compatibility allows existing applications to obtain performance and cost benefits of the new hardware without recompilation. Preserving binary compatibility is essential to allow complex software application vendors flexibility in deciding when to recompile. This section describes the performance implications of running an existing application binary on the Model 250 and Model 590.

The performance analysis comparing ”existing binary” performance improvements in Table 1 uses the SPEC CFP92 suite of 14 floating–point benchmarks and SPEC CINT92 suite of 6 integer benchmarks [4]. For more information on the characteristics of the benchmark suites, see [5]. As the baseline case for ”existing application binaries”, the SPEC benchmarks are compiled using the **–O** option with the previous XLC v1.2.1 and XLF v2.3 compilers. The **–O** option enables the level of optimization that the compiler designers believe represents a good tradeoff between compilation speed and run–time performance.

The Model 590 provides significant gains on both integer and floating–point benchmarks. Table 1 shows that the SPEC CINT92 integer suite runs 47% faster on the Model 590 than the Model 580 without recompilation. The POWER2 implementation's dual fixed–point units exploit the inherent parallelism in the program. The POWER2's larger data cache, faster clock rate, and improved branch characteristics also contribute to the performance improvement. The gains on applications will vary considerably depending on the amount of inherent parallelism and the data access characteristics. The SPEC CFP92 floating–point suite runs 58% faster on the Model 590. The floating–point codes' performance increase is due to the POWER2's dual floating–point units, increased data cache, increased data and instruction cache line size, and larger Translation Lookaside Buffers (TLBs).

Bench-mark	Option	Model 580	Model 250	Model 590
CINT92	–O	100%	82.6%	147.8%
CFP92	–O	100%	53.3%	158.0%

Table 1 Performance relative to a Model 580 using binaries compiled with XLC v1.2.1 and XLF v2.3

The Model 250 performs well compared to the Model 580 on both integer and floating–point applications. Table 1 shows the SPEC CINT92 integer performance is 82% of a Model 580. Although the Model 250 runs at a slightly higher frequency, the Model 580's larger cache is the primary cause of the performance difference. On the SPEC CFP92 floating–point suite, the Model 250 achieves 53% of the Model 580 performance. The Model 580's larger cache and shorter floating–point pipeline are better suited to these floating–point codes whereas the PowerPC 601 achieves a good cost/performance balance. Without recompiling, the PowerPC 601 cannot use the single–precision instructions to achieve its full SPECfp92 potential. The performance of customer applications will vary considerably depending on many factors such as cache miss characteristics.

Binary compatibility provides performance and cost benefits to existing applications without requiring recompilation.

Considerations for Recompilation

Given that existing POWER application binaries run on the entire RS/6000 family, recompilation is recommended only where the new instructions and new compilers are likely to benefit the application. Two key considerations are compatibility and performance.

The new compiler enhancements, found in the XL C (XLC) v1.3, XL Fortran (XLF) v3.1, and C Set ++ v2.1 compilers, are designed to preserve binary compatibility across the entire RS/6000 family yet allow exploitation of specific new implementations. The new XL compilers are capable of generating binaries which are ”common” to all implementations or binaries which are ”targeted” to a particular implementation (see Figure 2). The compiler provides a compile time option, **–qarch**, to select the working set of instructions to be used when creating the binary.

This section describes the compatibility / performance tradeoffs between recompiling using ”common” or recompiling using the **–qarch** option to target a specific implementation. In addition, the section includes a brief discussion on incompatibility scenarios and application tuning.

Recompiling to ”Common”

If binary compatibility is the primary recompilation issue, the compiler's ”common” mode is recommended. ”Common” mode is the default option but may be explicitly identified by using the **–qarch=com** option. The current implementation of **–qarch=com** produces a binary which contains the intersection set of ”common” instructions which will run on all POWER, POWER2, and PowerPC 601 hardware platforms.

This option removes most of the POWER architecture–specific instructions which are not native in the future PowerPC architecture implementations. The POWER instructions which remain are the integer multiply and divide which will be removed when full operating system support is available. The objects produced are binary compatible across all existing platforms today, and minimize the emulation required on future PowerPC implementations (603, 604, and 620). This version is the first step towards generating code that will not require emulation on any implementation of the POWER or PowerPC architectures.

The performance improvements of compiling to ”common” mode are shown in Tables 2 and 3. The data in Tables 2 and 3 include three sets of measurements. One set consists of the ”existing binaries”. The second set uses the new C Set ++ v2.1 and XLF v3.1 compilers with the **–O** option to compile a run for ”common” mode and a run ”targeted” for each specific implementation. The third set uses the compilers' new **–O3** option to compile for the ”common” and ”targeted” runs. The **–O3** option performs additional optimizations which are memory resource intensive, compile–time intensive, and/or may change the semantics of the program slightly. These optimizations are recommended when the desire for run–time performance improvements outweighs the limitations for compile–time resources [6].

Recompiling using ”common” mode currently provides similar performance to an existing application binary. As an example, Table 2 shows that, using **–qarch=com** and **–O3** to compile the SPEC CINT92 integer suite, a 3.93% improvement is seen for running the new binary on a Model 580 compared to running the ”existing binaries” (CINT92 compiled using **–O** with XLC v1.2.1) on the same Model 580. Similarly, a 3.92% improvement is seen for running the new ”common” SPEC CINT92 suite on a Model 590 compared to running the ”existing binaries” on the same Model 590. Table 3 shows that, using **–qarch=com** and **–O3** on the new C Set ++ v2.1 and XLF v3.1 compilers to compile the SPEC CFP92 floating–point suite, a 0.04% to 6.31% improvement is seen over the ”existing binaries” on the Models 250, 580, and 590. Because the 6.31% figure is a geometric mean, individual applications could see a greater improvement.

	C Set ++ v2.1			
	–O		–O3	
	Com	Target	Com	Target
250	1.50%	1.44%	1.86%	2.12%
580	1.30%	1.26%	3.93%	4.00%
590	0.80%	0.66%	3.92%	4.28%

Table 2 SPECint92 recompilation performance improvements over previous XLC v1.2.1 using –O. Percentages are with respect to each platform.

	C Set ++ v2.1 and XLF v3.1			
	–O		–O3	
	Com	Target	Com	Target
250	0.36%	10.90%	0.04%	11.67%
580	–1.61%	–1.61%	2.05%	2.08%
590	–0.67%	5.49%	6.31%	15.78%

Table 3 SPECfp92 recompilation performance improvements over previous XLC v1.2.1 and XLF v2.3 using –O. Percentages are with respect to each platform.

Recompiling to a Target Implementation

In cases where performance is critical and binary compatibility across all implementations is not necessary, targeting a specific implementation may improve performance over ”common” mode. For example, an engineering/scientific code which is load/store bound may nearly double in performance using the POWER2 quad–word load/store instructions [1]. The application binary, containing POWER2 instructions, is limited to executing on models using the POWER2 implementation (e.g. Models 58H, 590, and 990).

The new **–qarch** compiler option selects which instructions to use when generating the executable binary. In addition to the **com** setting, **–qarch** can be set to **pwr**, **pwrx**, and **ppc** to ”target” the POWER, POWER2, and the PowerPC 601 implementations, respectively.

The following performance discussions for each ”target” implementation will focus on floating–point applications. Integer applications typically show minimal performance benefit by ”targeting” a specific implementation compared to compiling for ”common” (see Table 2). This observation is expected because most of the new instruction enhancements relate to floating–point computations.

Target for POWER

–qarch=pwr produces code which runs on all the POWER hardware platforms. In prior versions of the compilers, such as XLC v1.2.1 and XLF v2.3, all applications were created in this mode by default.

This option should be avoided if the binaries will eventually be run on the future PowerPC implementations. These implementations will emulate the removed POWER instructions, resulting in a potential decrease in performance. Because the PowerPC 601 bridges the two architectures and implements most of the removed POWER instructions, binaries created with this option may not show a performance degradation when running on the Model 250.

The Model 580 row of Table 3 show that recompiling the CFP92 suites with **–qarch=pwr** and **–O3** provides no significant performance advantage compared to using **–qarch=com** and **–O3**.

Target for POWER2

Recompiling to POWER2 is a good choice for engineering / scientific codes which both require the computing power of the high–end systems and exploit POWER2’s new instructions. Using **–qarch=pwrx** allows the compiler to use the new Quad Load and Quad Store instructions, the Convert to Integer instructions, and the Square Root instruction. In addition, the compiler will schedule the instructions to more fully exploit the dual floating–point and fixed–point units. Many compute intensive floating–point codes will benefit significantly by targeting to the POWER2 instruction set [7].

The Model 590 row in Table 3 indicates that targeting the POWER2 provides a substantial 9.5% (15.78% – 6.31%) performance improvement over a ”common” binary (using the **–O3** option for both). The result reflects the performance benefits of the Load/Store Quad, Square Root, and Convert to Integer instructions as well as compiler scheduling.

The results in Table 3 compare the geometric mean of performance improvements and hide some of the underlying trends for individual applications. Table 4, which separates the single–precision and double–precision codes, shows that both categories achieve significant performance gains by targeting for POWER2 and using **–O3** compared to running the ”common” binary on the same platform. Double–precision applications show a larger gain due to the quad–word storage reference instructions.

CFP92	–O3	
	com	pwrx
single	5.65%	10.59%
double	6.56%	18.14%
combined	6.31%	15.78%

Table 4 POWER2 recompilation performance improvements for double and single precision CFP92 using new compilers.

Target for PowerPC 601

Applications dominated by single–precision floating–point computations will benefit from targeting the PowerPC 601. Using **–qarch=ppc** allows the compiler to use PowerPC’s single–precision floating–point instructions. In addition, the compiler will schedule the instructions to recognize the longer floating–point pipeline in the PowerPC 601 implementation. For many applications which do not require double–precision accuracy, such as graphics and seismic processing, single–precision arithmetic provides good performance at a low cost.

CFP92	–O3	
	com	ppc
single	–1.21%	31.12%
double	0.41%	4.54%
combined	0.04%	11.67%

Table 5 PowerPC 601 recompilation performance for double and single precision CFP92 using new compilers.

The Model 250 row of Table 3 indicates that recompilation which targets the PowerPC instruction set provides a substantial 11% gain over SPEC CFP92 binaries compiled to ”common” for both **–O** and **–O3**.

Targeting the PowerPC 601 on single–precision CFP92 codes provides outstanding performance improvements over the ”common” binary running on a Model 250 (see Table 5). Recompilation for single–precision applications on the PowerPC 601 is recommended to exploit the single–precision instruction support.

<u>Incompatibility Scenarios</u>

”Targeting” a specific implementation to achieve optimal performance can sacrifice binary compatibility across the RS/6000 family. Figure 3 provides a summary of which ”targeted” binaries execute on a given platform. Compiling for ”common” provides full binary compatibility across the RS/6000 family. The second column shows that existing POWER applications and binaries ”targeted” to POWER are also compatible across all current implementations.

Although both ”common” and POWER ”targeted” binaries are binary compatible across the entire family, ”common” is recommended. On future implementations of the PowerPC which will emulate the removed POWER instructions, the POWER ”targeted” binary may run slower than the ”common” binary (see the PowerPC row in Figure 3).

A binary ”targeted” for the PowerPC 601 implementation may contain instructions that will run on the PowerPC 601 hardware only; if so, the binary created will not run on POWER or POWER2 implementations. Similarly, a binary ”targeted” for the POWER2 may run only on POWER2 implementations, not on POWER or PowerPC system.

Compiled for / Can run on	Common (–qarch=com)	POWER (–qarch=pwr)	POWER2 (–qarch=pwrx)	PowerPC 601 (–qarch=ppc)
POWER	OK	OK	—	—
POWER2	OK	OK	OK	—
PowerPC 601	OK	OK	—	OK
PowerPC	OK	OK (emulation)	—	OK

Figure 3 Compatibility Matrix

For example, on AIX 3.2.5, a POWER2 ”targeted” application will begin execution on a POWER or PowerPC platform; however, if a POWER2–specific instruction occurs, the program will halt execution with a core dump and will display an error message stating that an illegal instruction was encountered. An example of the error message which may occur is:

```
7776 Illegal instruction – core dumped
```

The compatibility restriction requires application developers to maintain more than one binary if the ”targeted” application needs to run on several implementations.

<u>Application Tuning</u>

In addition to using the **–qarch**, most applications can benefit from tuning with additional compiler optimizations [6]. The **–qarch** option implies a default setting for **–qtune**. When ”common” mode is used, it may be beneficial to use **–qtune** to override this setting and to bias code for a particular machine. The **–qtune** option affects scheduling, but does not affect the instruction selection so binary compatibility is preserved across the entire line. With optimizing compilers, such as those in the XL compiler family, run–time performance can be improved significantly with appropriate tuning and analysis.

Summary

The POWER2 enhancements to the POWER architecture propel the new Models 58H, 590, and 990 to industry leading performance. The emergence of PowerPC and the single–chip PowerPC 601 implementation gives the Model 250 outstanding performance at a low cost. Binary compatibility allows the new hardware to provide performance and cost benefits while leveraging the existing applications. The strategic definition of PowerPC resulted in a large intersection of instructions which are common to all implementations. When compatibility is the primary concern, recompilation to ”common” is the best choice.

The new XL compilers provide the user with a range of options to balance recompilation compatibility/performance tradeoffs. Integer applications show negligible gains when compiling to target a specific implementation compared to compiling for ”common”. Double–precision applications benefit greatly by targeting POWER2 while singe–precision applications show a significant improvement. Targeting PowerPC produces outstanding gains on single–precision codes and modest gains on double–precision codes.

Acknowledgements

The new architectures and compiler technology are the result of contributions from the Austin and Toronto team as well as many people at the Haifa Scientific Center and T.J. Watson Research Lab. Many of the key technical individuals are the authors of the other articles in this issue. Management leadership for the architecture, implementations, and compilers include Phil Hester, Jack Kemp, Rich Oehler, George Lerom, Bob Gerber, and Karen Bennet. Finally, we thank Steve White, Bill Hay, and Kate Stewart for providing many useful suggestions about this article.

References

1. Steven W. White and Sudhir Dhawan, ”POWER2: Next Generation of the RISC System/6000 Family,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.
2. Ed Silha, ”PowerPC: A High Performance Architecture with a History,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.
3. Charles R. Moore and Robert L. Mansfield, ”The PowerPC Alliance,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp??.
4. Jeff Reilly, ”CINT92 & CFP92 Benchmark Descriptions,” *SPEC Newsletter*, Volume 4, Issue 4, December 1992, Standard Performance Evaluation Corporation, p. 9.
5. Sohel Saiyed, Michael O’Connor, and Maurice Franklin, ”POWER2 CPU–Intensive Workload Performance,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.
6. Kate Stewart, ”Using the XL Compiler Options to Improve Application Performance,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??.

POWER2 Commercial Workload Performance

Maurice Franklin, William Alexander, Rajiv Jauhari, Ann Marie Grizzaffi Maynard, and Bret Olszewski

Introduction

Data–intensive applications like transaction processing and file servers form a major market segment for computer systems. These applications are collectively called ”commercial” applications as most were initially used primarily by commercial enterprises such as banks, airlines, and insurance companies. Early microprocessor–based systems simply could not handle such data–intensive applications because they did not have enough processing power and I/O connectivity to handle many data–processing users concurrently. In recent years, the dramatic increase in the processing power available from microprocessors has made it possible for systems such as RISC–based workstations to compete in markets that were once exclusively the domain of mainframes. In fact, commercial applications represent one of the most rapidly growing market segments for RISC–based Unix workstations today [1].

In this article, we examine some of the features of the POWER2–based RISC System/6000 that make it well suited for the commercial workload arena. We begin by describing the performance characteristics of commercial workloads. We then discuss our analysis methodology and some of the common benchmarks used by the industry to compare commercial performance. This is followed by a discussion of the POWER2 hardware features that contribute to the enhanced commercial performance of the RISC System/6000.

Characteristics of Commercial Workloads

Commercial workloads include a wide variety of applications; some of the more prominent include on–line transaction processing, other database management services such as batch transaction processing and decision support, and file servers.

While different commercial applications stress the system in different ways, most share some common characteristics. These include:

- Many (up to hundreds or thousands of) concurrent users;
- Long path lengths over a large set of instructions, with a substantial part of the path length in the operating system code;
- Fewer loop iterations and more (non–loop) branches than in scientific applications (see [2] for an analysis of the branching characteristics of some scientific benchmarks);
- Extensive manipulations of data structures via pointers, requiring integer arithmetic for address resolution;
- Relatively little floating–point arithmetic; data manipulation consists primarily of string or integer comparison, updates, or insertions;
- High random I/O rates, with data spread over many megabytes or gigabytes of disk; disk I/Os are primarily short (4 or 8 KBytes) and successive disk I/Os are often randomly distributed over the total disk space used.

Note that the characteristics of commercial applications differ dramatically from those of scientific or numerically intensive programs, which often have small instruction working sets with tight loops, use floating–point arithmetic extensively, and often do sequential rather than random I/O.

Table 1 shows some characteristics of certain commercial benchmarks. The benchmarks themselves, as well as the methodology used to determine these characteristics, will be discussed in the next section.

The first two numeric columns show the percentage of instructions executed that are branches, and of these the percent that are taken. An analysis of branching behavior helps to explain one of the reasons why the I–cache miss rates for commercial workloads are higher than the I–cache miss rates for scientific applications. Consider a typical scientific

workload that is dominated by short to medium length loops. For such a workload, where most branch instructions are executed to return to the head of a loop, the percentage of branches that are taken would be much higher than the percentages shown in Table 1. Also, if the largest instruction loop for this workload fits in the I–cache, the I–cache miss rate for the scientific application would be very low. In contrast, the branching data in Table 1 shows that, for commercial applications, the percentage of taken branches is relatively low, indicating that there are relatively few loops executed in these applications. This scarcity of loops implies that the application has a lower probability of re–executing recent instructions, leading to a higher I–cache miss rate. Other characteristics of commercial applications that lead to high I–cache miss rates include many processes and a high context switch rate. The number of processes is often proportional to the number of concurrent users, and frequent context switches are the result of frequent short I/Os and inter–process synchronization. Commercial applications tend to have higher data cache (D–cache) miss rates as well, because their data exhibits less locality and sequentiality than scientific applications.

The column labeled ”Average Sequential Block Size” in Table 1 shows the branchy nature of commercial workloads in a different way. In this paper we define the term ”sequential block” to be the sequence of instructions executed between two taken branches (including the second branch instruction). (This is a dynamic measure, not something obtainable from a static analysis of the code.) These sequential block sizes are much shorter than in the scientific and engineering applications we have analyzed.

Benchmark	%Branches	%Branches Taken	Avg. Seq. Block Size (Instrs)	Percentage of Instrs in OS
TPC–A da	18.8%	66.4%	8.0	67%
TPC–A cs	16.9%	67.1%	8.8	40%
TPC–C	18.5%	65.1%	8.1	43%
Laddis	18.9%	68.7%	7.7	100%
Kenbus	16.3%	64.9%	9.5	23%
Sdet	17.8%	66.5%	8.4	50%

Table 1 Characteristics of Commercial Benchmarks

The last column in Table 1 shows the fraction of the total number of instructions executed in the AIX operating system code, as opposed to the instructions executed by the user application and in shared library code. These numbers indicate that much of the work in commercial applications is actually done by the operating system. One reason for this relatively high usage of the operating system code is that in these applications, there is frequent movement of small amounts of data between different levels of the system, and little arithmetic computation on the data. (For example, queries have to be sent from an on–line user's terminal to a database server, and responses have to be sent back; this involves the use of a lot of operating system communication code, and very little arithmetic.) This is in contrast to many scientific applications, where once data is brought to the application space by the operating system, extensive arithmetic manipulation is performed on it before it is handed back to the operating system for storage. Operating system code is known to have a high incidence of branches and few loops.

Clearly, architectural features such as the number of integer arithmetic units, the sizes, organization and the number of levels of instruction and data caches, and the latency to caches and to memory, all have a strong bearing on the degree to which commercial applications perform well on a system. The I/O subsystem will have an important effect on the performance of most applications, but in this paper we focus on processors and memory subsystems, and I/O will not be discussed further.

Until recently, many RISC–based Unix workstations focused on the scientific application environment, but as this article and others in this publication [3,4] show, the POWER2–based workstation is designed to provide superior performance in both scientific and commercial environments.

Performance Methodology

The results presented in this paper come from one of three sources: direct measurement on POWER2–based systems, analysis of traces taken on POWER–based systems, or simulations using these traces.

<u>POWER2 Performance Measurement</u>
The POWER2 processor has a built–in hardware performance monitoring capability that collects measurements which allow not only detailed performance analysis, but also detailed workload characterization and analysis of system behavior [5].

<u>Traces and Simulation</u>
The AIX Performance group has a tool that collects instruction traces. These traces record the sequence of instructions executed (for both system code and application code) along with the virtual address of each instruction and data reference. The traces can be post–processed to reveal workload characteristics such as those shown in Table 1. They can also drive simulators of the processor and memory subsystem. From the simulations we can determine many of the same system performance characteristics as from hardware measurements, including cache miss rates.

Determining these quantities from a software trace plus simulations is less accurate than direct hardware measurements, but it has the advantage of applying to system configurations that do not actually exist; for example, we can create a trace once and use it to investigate many different cache configurations. It also allowed us to begin analyzing POWER2 performance before the processor was actually available, and to analyze workload characteristics on systems that do not have hardware counters.

We have validated our trace–based simulation methodology against direct hardware measurements on both POWER–based and POWER2–based systems, and the results almost always agree to within 10%.

Commercial Benchmarks

In the intensely competitive marketplace of open systems, performance comparisons provide a key differentiator in making purchasing decisions.

While some customers have developed their own, often proprietary, benchmarks to help them choose between different systems, several benchmarks have been accepted by the industry as good indicators of system performance capabilities for specific parts of the commercial market. These include the set of TPC benchmarks for database applications, the LADDIS benchmark for file server applications, and SPEC SDM benchmarks like SDET and KENBUS for multi–user software development applications. Traces of these benchmark programs are used in this paper.

<u>The TPC Benchmarks</u>
The Transaction Processing Performance Council (TPC) benchmarks are probably the best known and most frequently cited measures of commercial performance. The TPC is a collection of representatives from many of the computer systems vendors and several of the database vendors competing in the commercial arena throughout the world. At the time of writing, the TPC has released three benchmarks, called TPC–A, TPC–B, and TPC–C, along with rules for running them and reporting results that are intended to produce fair comparisons among possibly dissimilar commercial computing systems.

TPC–A simulates a banking application where users submit simple debit or credit transactions to the system from automated teller machines. (TPC–B is logically very similar to TPC–A; it differs mainly in that it is easier and cheaper to configure a system to run TPC–B than to configure it for TPC–A. We do not discuss TPC–B further in this paper.)

The trace called ”TPC–A da” in this paper was collected while the TPC–A benchmark was running in ”direct–attach” mode using a commercially available relational database management system (RDBMS) on a RISC System/6000 Model 530. In the direct–attach mode, the workstation executes the entire TPC–A benchmark code, including the user interface, application code, and the database server.

The trace called ”TPC–A cs” was collected while the TPC–A benchmark was running in ”client–server” mode using a different commercially available RDBMS on a RISC System/6000 Model 530 as a server. In the client–server mode, the user interface part of the TPC–A benchmark runs on one or more client machines, while the rest runs on a server. Since ”TPC–A cs” captures only server code, it provides a different view of the workload than the ”TPC–A da” trace, and the two traces are not directly comparable; we include both in our analysis because both modes of operation are fairly common in the marketplace.

TPC–A transactions are relatively short and uniform. In fact, TPC–A transactions are so simple and so short that this benchmark is becoming obsolete as a measure of high–end commercial performance; the demand from industry for more realistic and more complex benchmarks has led to the development of TPC–C. TPC–C is intended to simulate an order–entry environment, with a mix of read–only and update–intensive transactions.

The ”TPC–C” trace was collected on the server portion of the TPC–C benchmark in a client–server configuration running on yet another flavor of a commercially available RDBMS on a RISC System/6000 Model 550.

The LADDIS Benchmark

The LADDIS benchmark measures a system's file serving capability [6]. One or more client workstations submit a mix of Network File System (NFS) requests to the server. LADDIS is the basis of a new system–level file server (SFS) benchmark suite recently announced by the Standard Performance Evaluation Corporation (SPEC) [7]. The ”Laddis” trace was collected on a RISC System/6000 Model 530H.

The KENBUS and SDET Benchmarks

KENBUS and SDET are two of the benchmarks that make up the SPEC SDM (System Development Multi–Tasking) suite [8]. Both KENBUS and SDET represent multi–user software development environments with many concurrent users executing common Unix commands such as edits, compiles, and binds. KENBUS is designed to represent a Unix/C research and development environment, while SDET is intended to represent a C–based commercial software development environment. The ”Kenbus” and ”Sdet” traces were collected on a RISC System/6000 Model 530.

System Performance

The performance metric that is published for each of the benchmarks described in the previous section is a measure of throughput: transactions per second or minute for the TPC benchmarks and LADDIS, and scripts per hour for the KENBUS and SDET benchmarks. Systems being tested against these benchmarks are usually configured so that the processor is the bottleneck by, for example, adding enough disks to assure that this is so. In these configurations, the latencies for I/O are entirely overlapped by the processing of instructions, and the processor is fully utilized, while I/O devices are not. I/O performance and system response time are important considerations, but are beyond the scope of this paper.

When the processor is the bottleneck, system throughput is determined by three factors: the average number of instructions (NI) in the path length for the transaction or operation, the average number of cycles it takes to execute an instruction (CPI or ”cycles per instruction”) for that benchmark, and the number of processor cycles per second (usually expressed in MegaHertz or MHz). Throughput (TP) in transactions or operations per second is expressible as a function of these three variables as:

$$TP = \frac{MHz \times 1,000,000(cycles/sec)}{NI(instr/trans) \times CPI(cycles/instr)}$$

Operating system and application developers can improve throughput by reducing the average path length. MHz is influenced by processor size, design, implementation technology and cost. In the next section, we describe features of the POWER2 that are aimed at raising throughput by reducing CPI.

POWER2 Features For High Performance

There are three features of the POWER2 design that improve its performance on commercial workloads compared to the POWER design:

- Second integer instruction unit
- Larger caches
- Longer cache lines

We will discuss the effect of each of these features in more detail.

Second Integer Instruction Unit

The POWER2 processor is 'superscalar', meaning that it is capable of executing more than one instruction per processor cycle. As described in [3], the POWER2 has two floating–point instruction execution units (FPUs), two fixed–point (integer) instruction execution units (FXUs), and an instruction cache unit (ICU) that handles branch instructions. The POWER2 processor is capable of executing up to six instructions simultaneously in one cycle. In practice, the number of instructions that can be executed in any given cycle is constrained by which instructions are actually present in the section of code being executed, and by data dependencies among the instructions.

Multiple floating–point instruction units, which allow multiple floating–point instructions to be executed simultaneously, are fairly common in processors aimed at scientific/engineering workloads. However, commercial workloads generally have relatively few floating–point instructions; Table 2 shows POWER2 hardware counts of the fractions of all instructions executed by the ICU, the FXUs and the FPUs for the TPC–C benchmark, run under the same RDBMS flavor used to produce our TPC–C trace.

Our instrumentation also indicates that almost all of the ICU instructions are branch instructions. Taken together, the data in Tables 1 and 2 shows that most instructions

Execution unit	Percentage of instruction mix
FXU	78.0%
ICU	21.8%
FPU	0.2%

Table 2 TPC–C Instructions by Unit

in commercial applications are executed by integer units; about 15–20% of instructions executed are branches, and only a small fraction are floating–point instructions.

Measurements taken on the POWER2 running TPC–C show that of all the FXU instructions executed, one FXU executed 68%, while the second FXU executed the remaining 32%. Generally, an instruction is dispatched to the second FXU only if there is also an instruction to dispatch to the first FXU; this implies that the presence of the second FXU resulted in a reduction of approximately 32% in the execution time for integer instructions. Analysis of other hardware measurements confirms that, in fact, the second FXU was busy 48% of the time that the first FXU was busy. Put another way, the presence of the second FXU allows the POWER2 to increase integer instruction execution throughput for TPC–C by a factor of about one–half. Note that the theoretical maximum performance improvement (a doubling of the integer instruction execution throughput by two equally utilized FXUs) is difficult to attain due to dependencies between instructions, which require one unit to hold off execution until the other completes the preceding operation. (The exact speed–up provided by the second FXU also depends on how many of the instructions executed by it could have been scheduled in cycles when the first FXU would otherwise have been idle, and possibly on some second–order memory subsystem effects.) As integer instructions account for about three quarters of the total instructions, the presence of the second FXU leads to a significant performance gain for the POWER2.

Configuration		POWER2		Hypothetical A		Hypothetical B	
Workload	size type	32K I	256K D	16K I	16K D	8K I	8K D
	assoc/ln sz	2/128	4/256	4/64	4/64	2/32	2/32
TPC–A da		2.4	0.5	4.5	2.1	9.0	3.5
TPC–A cs		4.0	1.2	6.9	4.1	12.7	6.7
TPC–C		2.7	0.4	5.2	2.6	10.3	4.6
Laddis		2.5	0.4	5.0	3.5	9.9	6.6
Kenbus		0.9	0.4	1.7	1.6	4.4	3.1
Sdet		1.5	0.5	2.7	1.8	6.2	3.3

Table 3 L1 Cache Miss Rates for Three Cache Configurations

Larger Caches

The POWER2 has separate level 1 (L1) instruction and data caches. The I–cache holds 32K bytes and is 2–way set associative with a line size of 128 bytes. The D–cache is 256K–byte, 4–way and has a 256–byte line. The original POWER implementation had an 8K–byte I–cache and a 64K–byte D–cache.

The larger caches mean fewer cache misses. They especially help the execution speed of commercial workloads, which tend to have larger cache footprints than scientific/engineering workloads.

To show the effect of cache size and geometry, we will compare the POWER2 L1 cache configuration with two smaller, hypothetical configurations as detailed in Table 3. The cache size, set associativity, and line size (in bytes) of the Instruction and Data caches are given for each of these three configurations.

For each configuration, Table 3 shows the cache miss rates as predicted by our model using the traces of the benchmarks described earlier. Miss rates are expressed as percentages, normalized to total instructions issued:

$$miss\ rate = 100 \times \frac{\#of\ misses}{total\ instructions}$$

As Table 3 shows, the larger caches reduce misses significantly, thereby speeding execution of these benchmarks. Note that even small changes in miss rates are significant since each miss may contribute a delay of over ten cycles.

Longer Cache Lines

The POWER2 also has longer L1 cache lines than POWER systems. (A "line" is the unit of transfer between memory and caches.) The I–cache has 128–byte lines, and the D–cache has 256–byte lines. Longer I–Cache lines help performance in two related ways.

First, longer lines increase the probability that sequential instructions (instructions in a sequential block as defined in the section on characteristics of commercial workloads) and sequential data references will fall within the same line rather than crossing a line boundary into a different line. When two consecutive references lie in the same cache line, the second reference will almost always be found in the cache; the probability of generating a cache miss increases when consecutive references lie in different lines.

We used our traces to drive simulations of two I–cache geometries: both I–caches held 32K total bytes, but one had a 32–byte line and the other a 128–byte line (with, of course, one–fourth as many lines). We counted the fraction of I–cache misses that were references to the first byte of the new line; many of these

misses are caused by sequential blocks that happened to cross line boundaries. Note that I–cache misses caused by references to all bytes of a line other than the first are caused by taken branches, and cannot be attributed to the fact that sequential blocks sometimes cross line boundaries.

Table 4 shows that longer cache lines reduce I–cache misses to the first byte of a line, resulting in fewer total misses; not all misses are due to sequential blocks crossing line boundaries, of course.

Workload	Average Sequential Block Size (Instrns)	Total I–Cache Miss Rate		Fraction of Misses to First Byte of Line		Branches to Same Line	
		Lines size (bytes)		Lines size (bytes)		Lines size (bytes)	
		32	128	32	128	32	128
TPC–A da	8.0	5.0%	2.4%	65.3%	38.9%	13.8%	34.5%
TPC–A cs	8.8	9.5%	4.0%	67.1%	40.6%	15.1%	38.3%
TPC–C	8.1	6.0%	2.7%	67.0%	41.9%	15.1%	36.3%
Laddis	7.7	5.6%	2.5%	67.4%	42.4%	13.7%	38.0%
Kenbus	9.5	1.8%	0.9%	69.9%	41.9%	14.4%	38.5%
Sdet	8.4	3.1%	1.5%	68.3%	41.0%	13.8%	36.5%

Table 4 I–Cache Miss/Branch Characteristics with Different Line Sizes

Table 4 shows the average sequential block size for each workload repeated from Table 1 and the total I–cache miss rate expressed as a percentage of instructions executed. Table 4 also records, for I–cache misses, the percentage that were references to the first byte of a line and the percentage of branches taken that were to a location within the same cache line.

The second benefit of longer I–cache lines shown in Table 4 is that they increase the probability that a branch will be to an instruction that is present in the same cache line, again reducing cache misses. Recall that the two I–cache configurations were the same size, 32K bytes; only the line size was changed.

As can be seen, longer I–cache lines significantly reduce the fraction of branches that land outside the current line, thereby helping to reduce total cache misses as shown in Table 4. This effect should especially benefit commercial workloads that, like these benchmarks, have a high incidence of branches.

While our simulations show that they reduce cache misses, longer cache lines have the disadvantage of taking longer to fill than shorter lines. To help alleviate this effect, the POWER2 has an 8–word (256–bit) wide memory bus.

Summary

Commercial applications form an increasingly larger segment of the market for microprocessor–based systems like the RS/6000. Commercial workloads differ substantially from scientific and engineering workloads in their demands on the system. In this paper we describe these differences and, using data obtained from widely accepted commercial benchmarks, explain how some of the features of the POWER2 processor and memory subsystem make it well–suited for the commercial arena.

Acknowledgments

We would like to thank several members of the AIX and System Performance groups for their help in putting this paper together. We appreciate the assistance of Bob Urquhart in analyzing cache performance and Marc Stephenson in helping to get database systems running. We would like to thank Chris Chan–Nui for the invaluable post–processing tools he developed for reducing hardware monitor data and Ed Welbon for helping us properly interpret the POWER2 hardware measurements.

References

1. ”A Major Step for Commercial UNIX,” *AIXpert*, November 1992, IBM Corporation.
2. Chriss Stephens, Bryce Cogswell, John Heinlein, Gregory Palmer, and John P. Shen, ”Instruction Level Profiling and Evaluation of the IBM RS/6000,” *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.
3. Steve W. White and Sudhir Dhawan, ”POWER2: The Next Generation of the RISC System/6000 Family,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??–??.
4. Sohel R. Saiyed, J. Michael O’Connor, and Maurice Franklin, ”POWER2 CPU–Intensive Workload Performance,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??–??.
5. E.H. Welbon, C.C. Chan–Nui, D.J. Shippy, and D.A. Hicks, ”The POWER2 Performance Monitor,” *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??–??.
6. ”LADDIS File Server Benchmark,” *SPEC Newsletter*, Volume 4, Issue 1, March 1992; Standard Performance Evaluation Corporation.
7. ”SPEC Announces System File Server Benchmark Suite – SFS Release 1.0,” *SPEC Newsletter*, Volume 5, Issue 1, March 1993, Standard Performance Evaluation Corporation.
8. ”SPEC SDM Technical Fact Sheet,” *SPEC Newsletter*, Volume 3, Issue 2, Spring 1991, Standard Performance Evaluation Corporation.

POWER2 Performance on Engineering/Scientific Applications

E. L. Hannon, F. P. O'Connell, and L. J. Shieh

Introduction

The new POWER2–based Model 590 brings more good news to technical professionals who require high computing performance at workstation prices. By adding new architectural and hardware features to the proven RISC System/6000 base, POWER2 improves the performance of many types of engineering/scientific applications. In this paper, application performance of the new Model 590 (POWER2) is compared to that of the Model 580 (POWER), and the salient features of the POWER2 processor are discussed. The comparison is introduced within the context of fundamental loops which highlight the various improvements. Next, performance on a wide variety of frequently encountered constructs is examined. Finally, the system performance of the Model 590 is demonstrated through measurements on a collection of full scientific/engineering application programs. Related papers [1,2] discuss the performance of the SPEC benchmarks and commercial applications.

Some key architectural extensions and POWER2 hardware features are listed here and described in more depth in a related paper [3]:

- Quad–word floating–point load and store instructions
- Square Root instruction
- Dual floating–point execution units (FPU)
- Dual fixed–point execution units (FXU)
- Improved divide performance
- Additional physical floating–point registers for improved register renaming
- Decoupled floating–point arithmetic, floating–point load, and floating–point store pipelines
- Improved instruction fetch capabilities
- Larger, dual–ported data cache
- Wider bus interfaces
- Larger translation look–aside buffers
- Faster cycle time
- Improved branch processing

The intent of this paper is to relate these capabilities to measurable improvements exhibited in engineering and scientific application programs.

Loop Kernel Performance

Table 1 lists the performance for loops selected to demonstrate the effects of certain POWER2 enhancements. The data working sets for these loops fit within the cache so the measurements reflect the performance of the processor and not the memory subsystem. The loops consist of double–precision calculations which, except where noted, are coded in a straightforward manner such as:

```
 do j = 1,M
   do i = 1,N
     x(i) = sqrt(y(i))
    enddo
    call dummy(x,N)
end do
```

Variables N and M are set to 256 and 15360, respectively. The outer *j*–loop is required to obtain accurate inner loop timings. The call to subroutine *dummy* prevents the compiler from simply eliminating the outer loop.

The first two loops (rows 1 and 2 of Table 1) show the effect of the new divide and square root capabilities. The columns in the table are, left to right, the kernel description, the number of floating–point operations per loop iteration, the number of floating point operands loaded per iteration, the number of floating point operands stored per iteration, the average number of cycles per inner loop iteration, and the corresponding MFLOPS rate. The compiler will unroll such loops where possible to expose two independent Square Root instructions to fully utilize both FPU pipelines. As a result, the average number of cycles for the Square Root instruction is 15. Similarly, the average divide time is 10 cycles.

The **daxpy** (double precision *a* times *x* plus *y*) loop, in row 3 of Table 1, demonstrates the effects of the quad–word storage references and dual floating–point pipes. The compiler unrolls the loop to a depth of four and thus generates eleven instructions (four Load

Quads, four floating–point multiply–adds (FMAs), two Store Quads, and a branch). Each iteration of the loop executes in three cycles since the FXU can perform two load/store operations per cycle. The FMAs and the branch execute in the same three cycles. The measured performance confirms this, exhibiting 168.5 million floating–point operations per second (MFLOPS), which is roughly eight floating–point operations per three cycles at 66.5 MHz. This is greater than four times the performance of the Model 580, demonstrating the effectiveness of the 32–byte bandwidth of the Model 590 on storage reference limited loops.

The dot product (row 4) is unaffected by unrolling because the FMA in each iteration depends on completion of the FMA in the previous iteration. Since the FPUs have latencies of two cycles, only one FMA is performed per every two cycles; this loop will operate at one–fourth the peak performance of the machine.

The performance of the dot product improves greatly by using partial sums (see row 5 of Table 1). A code example for a dot product 4–way sum reduction is:

```
do i=1,n,4
  s1=s1+x(i )*y(i )
  s2=s2+x(i+1)*y(i+1)
  s3=s3+x(i+2)*y(i+2)
  s4=s4+x(i+3)*y(i+3)
enddo
s=s1+s2+s3+s4
```

The compiler generates nine instructions for the four–way sum reduction dot product (four Load Quads, four FMAs, and one branch). The instructions execute in three cycles (0.75 cycles per original iteration). As expected, two–thirds of peak performance is measured. In this case, instruction fetch capability limits the performance. Since the branch is the ninth instruction, the branch is not fetched and processed until the second fetch cycle. By the time the branch is decoded, a third sequential fetch cycle has begun.

If, in this example, the reference to array y is changed to another reference to x, the sum of the squares for elements of the x array is calculated. A code example for a four–way sum of squares is:

```
do i=1,n,4
  s1=s1+x(i )*x(i )
  s2=s2+x(i+1)*x(i+1)
  s3=s3+x(i+2)*x(i+2)
  s4=s4+x(i+3)*x(i+3)
enddo
s=s1+s2+s3+s4
```

In this case, two load instructions are eliminated and data reuse within the loop is increased. The loop instruction count for this loop becomes seven: two Load Quads, four FMAs and one branch. Since the number of instructions fetched per cycle is eight, the branch is processed in the first cycle of the loop, permitting the loop to execute in two cycles and yielding nearly peak performance (see row 6 in Table 1).

In addition to increasing data bandwidth, Load Quads have minimized the instruction fetch requirements for the loop by combining four double–word references into two quad–word references.

PER ITERATION MEASUREMENTS					
LOOP KERNEL	**F–P OPS**	**LOADS**	**STORES**	**AVE. CYCLES**	**MFLOPS**
X(I)=SQRT(Y(I))	**1**	**1**	**1**	**14.5**	**4.6**
X(I)=Y(I)/U	**1**	**1**	**1**	**10.0**	**6.7**
Y(I)=Y(I)+A*X(I) (DAXPY)	**2**	**2**	**1**	**0.79**	**168.5**
S=S+X(I)*Y(I) (Dot Product)	**2**	**2**	**0**	**2.0**	**66.6**
S=S+X(I)*Y(I) (4–Way Unrolled Dot Product)	**2**	**2**	**0**	**0.77**	**174.2**
S=S+X(I)*X(I) (4–Way Unrolled Sum of Squares)	**2**	**1**	**0**	**0.52**	**255.4**

Table 1 Performance of selected loops

The use of partial sums affect the order the computations in the previous loops. Because floating–point arithmetic is non–associative, this may produce a different floating–point result than the original version. The result of the unrolled loop tends to be more accurate because there is a greater chance that smaller product terms are reflected in the sum.

Frequent Loop Kernels

Having examined selected loops which highlight the POWER2 capabilities, the POWER2 performance gains on an arbitrarily selected set of loops will now be presented. Table 2 compares measurements on a 62.5MHz Model 580 with those from a 66.5MHz POWER2 system. These loops are selected from an ensemble of over 200 loops used to measure compiler and hardware performance. They represent typical loops found in engineering and scientific applications. Unless specifically stated, all unrolling is accomplished automatically by the compiler (and preprocessor). Measurements are stated in cycles per original loop iteration and the resulting MFLOPS ratings.

Most kernels show roughly a factor of two or more improvement over the Model 580. Loops which are storage reference limited tend to show performance ratios exceeding two, as the **daxpy** loop illustrated. Loops which show a speedup less than two include the accumulation and recursive loops (rows 26–35) and the Fortran intrinsic routines (rows 36–47). Unrolling using partial sums, as in the dot product example, will improve the performance of many of the accumulation loops (rows 26 to 33).

The recursive loop in row 34, which shows a slowdown relative to the Model 580, is the result of less than optimal instruction scheduling by the version of the compiler used in this study. Disabling the *–qarch=pwrx* flag for that loop results in Model 590 performance of 33 MFLOPS, which is a speedup of 1.3 over the Model 580. Note that the recursive loops of rows 34 and 35 cannot use more than one floating–point pipe since each iteration of the loop requires the result from the previous iteration. For these loops, the performance is limited by the pipeline depth of two.

Currently, the performance of the Fortran intrinsic routines are limited by library routines optimized for POWER implementations, with the notable exception of **MAX** (row 45).

Application Performance

The first two sections of this paper discussed kernels or loops. This section will focus on the results for several classes of applications, shown in Tables 3 and 4 as well as Figure 1. Measurements are stated in CPU seconds. Speedups over the Model 580 are shown in the last row or column. In all cases, the Model 580 timings are for executables produced using the XLF Version 2.3 compiler. When recompilation was performed for the Model 590 timings, an early version of the new XLF Version 3.1 compiler was used.

Computational Structural Mechanics

Computational Structural Mechanics (CSM) is used across many industries to solve problems of classical mechanics involving mechanical structures subject to prescribed forces and boundary conditions. Applications span a broad spectrum from the analysis of offshore oil drilling platforms and ophthalmic lenses to the simulation of car crashes and metal forming processes. CSM codes are also used to solve problems in acoustics, electromagnetics, and heat transfer.

Two types of computational structural analysis programs were included in this study. Both use finite element discretization techniques for the spatial dimensions [4]. For the temporal dimension, one program uses an implicit integrator; the other uses an explicit integrator. The techniques affect the localization of the computation within the programs as well as the hardware capabilities stressed by the computation.

Implicit CSM

Application of finite element techniques to the equations of motion of a structure produces a system of ordinary differential equations. In general, if an implicit scheme is chosen to integrate the differential equations (or, if the dynamic terms are dropped when only a static solution is desired), a nonlinear system of equations results. This nonlinear system of equations is commonly solved using Newton's method which has a linear system of equations solver as its computational core. Factorization of the tangent stiffness matrix associated with the linear equation solver often dominates the entire computational task of the structural analysis, especially for moderate to large sized, two–dimensional and three–dimensional models.

ABAQUS/Standard is one of a number of commercial, general purpose, finite element programs which employ implicit time

PER ITERATION MEASUREMENTS						
Row Reference Number	LOOP KERNEL DESCRIPTIONS	Model 580 (62.5 MHz)		Model 590 (66.5 MHz)		Speedup
		Average cycles	MFLOPS	Average cycles	MFLOPS	
1	X(I)=SQRT(Y(I))	55.32	1.1	14.5	4.6	4.0
2	X(I)=Y(I)/U	20.10	3.1	10.0	6.7	2.2
3	Y(I)=Y(I)+A*X(I) (DAXPY)	3.03	41.3	0.79	168.5	4.1
4	S=S+X(I)*Y(I) (Dot Product)	2.02	61.9	2.00	66.6	1.1
5	X(I)=A(I)+B(I)	3.02	20.7	0.91	72.9	3.5
6	X(I)=A(I)*B(I)	3.02	20.7	0.91	72.9	3.5
7	X(I)=U/A(I)	20.00	3.1	9.99	6.7	2.2
8	X(I)=U*X(I)	2.51	24.9	0.79	84.3	3.4
9	X(I)=A(I)+X(I)	3.02	20.7	0.78	83.6	4.0
10	X(I)=U+A(I)+B(I)+C(I)	5.99	31.3	3.25	61.3	2.0
11	X(I)=A(I)+B(I)+C(I)+D(I)	6.02	31.2	2.78	71.7	2.3
12	Y(I)=U+A(I);X(I)=A(I)+B(I)+C(I)+D(I)	8.01	31.2	3.30	80.7	2.6
13	X(I)=A(I)+B(I)+C(I)+D(I); Y(I)=U+D(I)	7.02	35.6	3.06	86.8	2.4
14	X(I)=A(I)+B(I);Y(I)=A(I)–B(I)	5.01	24.9	2.54	52.4	2.1
15	X(I)=C(I)+A(I)*B(I)	4.02	31.1	1.30	102.5	3.3
16	X(I)=A(I)+B(I)+C(I);Y(I)=X(I)+C(I)+U	8.02	39.0	4.02	82.7	2.1
17	X(I)=(A(I)+B(I))*(C(I)+D(I))	6.02	31.2	3.28	60.7	2.0
18	X(I)=(U+A(I))*(V+B(I))	6.01	31.2	2.52	79.3	2.5
19	X(I)=U*A(I);Y(I)=V*B(I)	4.02	31.1	2.04	65.2	2.1
20	X(I)=A(I)+B(I);Y(I)=A(I)*B(I)	5.01	24.9	2.54	52.4	2.1
21	X(I)=A(I)*B(I)+C(I)*D(I)	5.03	37.2	1.54	129.5	3.5
22	X(I)=X(I)+A(I)*B(I)+C(I)*D(I)	6.03	41.5	2.67	99.6	2.4
23	X(I)=A(I)*B(I)+C(I)*D(I);Y(I)=B(I)+D(I)	6.03	41.4	2.49	106.8	2.6
24	X(I)=A(I)*C(I)–B(I)*D(I);Y(I)=A(I)*D(I)+B(I)*C(I)	6.04	62.1	2.80	142.3	2.3
Compilation options: –O –qarch=pwrx –Pv –Wp,–unroll=4,–O=5,–SO=3						

Row Reference Number	LOOP KERNEL DESCRIPTIONS	Model 580 (62.5 MHz)		Model 590 (66.5 MHz)		Speedup
		Average cycles	MFLOPS	Average cycles	MFLOPS	
25	X(I)=U*B(I);Y(I)=V*B(I)+W*A(I)+U*C(I)	6.03	62.2	2.88	138.4	2.2
26	S=S+A(I)	2.98	21.0	2.24	29.7	1.4
27	S=S+A(I)+B(I)	5.00	25.0	4.24	31.3	1.3
28	S=S+A(I); T=T+B(I)+C(I)	5.02	37.3	4.25	46.9	1.3
29	S=S+A(I)*B(I)+C(I)*D(I)	4.04	61.9	4.02	66.1	1.1
30	S=S+(A(I)+B(I))*(C(I)+D(I))	4.05	61.7	2.04	130.1	2.1
31	S=S+A(I); X(I)=A(I)+B(I)	3.03	41.3	3.15	42.2	1.0
32	S=S*A(I) (Product Reduction)	2.03	30.8	1.99	33.5	1.1
33	S=A(I)+S*U (Sum of Polynomial)	2.98	41.9	2.24	59.4	1.4
34	X(I)=A(I)+X(I–1)	2.50	25.0	2.97	22.4	0.9
35	X(I)=A(I)+B(I)*X(I–1)	3.02	41.4	2.97	44.7	1.1
36	X(I)=EXP(E(I))	61.56	8.1	52.43	10.1	1.3
37	X(I)=LOG(C(I))	73.37	6.8	55.63	9.6	1.4
38	X(I)=SIN(D(I))	46.91	10.7	42.12	12.6	1.2
39	X(I)=ASIN(A(I))	62.11	8.1	50.04	10.6	1.3
40	X(I)=SINH(E(I))	193.99	2.6	175.07	3.0	1.2
41	X(I)=TANH(E(I))	116.02	4.3	103.48	5.1	1.2
42	X(I)=TAN(D(I))	100.50	5.0	86.47	6.2	1.2
43	X(I)=ATAN(F(I))	71.17	7.0	61.97	8.6	1.2
44	X(I)=ATAN2(A(I),B(I))	459.21	1.1	394.76	1.3	1.2
45	X(I)=MAX(A(I))	9.49	6.6	4.04	16.5	2.5
46	X(I)=ERF(E(I))	63.70	7.8	57.60	9.2	1.2
47	X(I)=GAMMA(C(I))	271.87	1.8	231.38	2.3	1.3
Compilation options: –O –qarch=pwrx –Pv –Wp,–unroll=4,–O=5,–SO=3						

Table 2 Loop Kernels

integration. Table 3 shows the performance of ABAQUS Version 5.2 on the standard ABAQUS timing runs, which are described in detail in [5]. The timing runs consist of six ABAQUS analysis problems ranging from linear statics to nonlinear dynamics. The timings shown are produced using the standard binary executables from the ABAQUS distribution tape; no recompilaton is performed. Figure 1 shows the speedup relative to a Model 580 plotted against the root mean squared (RMS) wavefront for analyses T1–5–2 through T5–5–2. In general, a larger RMS wavefront corresponds to a greater speedup over the Model 580. The RMS wavefront is proportional to the square of the work required in the solver, and tends to grow with the spatial dimensionality of the problem. Problem T6–5–2 uses the unsymmetric solver and achieves a better speedup than the other problems; no diagonal element or block processing is required for the unsymmetric case.

For efficiency, a blocked algorithm was used in the ABAQUS equation solver. A blocked algorithm organizes many of the required operations into a **rank–n update**, which closely resembles a matrix–matrix multiply algorithm [6]. This algorithm was streamlined for the POWER architecture using strip–mining and unrolling [7]. Strip–mining is a technique which exploits the processor cache by partitioning the computations into rectangular regions which fit within the cache. Currently, ABAQUS Version 5.2 is optimized for 64K–byte data caches. As shown in the first section of this paper, unrolling increases the work in the inner loop, allowing maximum overlap of load/store instructions with FMA instructions. Where possible, the product matrix is formed in blocks of three by three sub–matrices by unrolling two levels of loops. Forming each sub–matrix product requires a total of six floating–point loads and nine FMAs. The POWER2 hardware can perform the nine FMAs in 4.5 cycles, on the average, using both floating–point units, and the six loads can be overlapped with the FMAs.

Preliminary measurements with ANSYS, another widely used implicit commercial finite element code, shows similar speedups. Other commercial implicit codes such as MSC/NASTRAN, SPECTRUM, and MECHANICA are also expected to demonstrate comparable speedups on solutions involving their direct solvers.

Explicit CSM

Another class of computational mechanics codes applies explicit integration to the temporal variable and has the computational advantage of avoiding the solution of simultaneous equations. Unlike the implicit case, the computational work is spread across more of the code; there is no distinct computational hot spot. Still, many key kernels contain the computational richness to exploit the multiple functional units of the POWER2 implementation. In addition, hardware supported Square Root and divide, increased memory bandwidth, data cache capacity, improved store performance, and the faster cycle time all contribute to the speedup over the Model 580.

Table 4 contains two sample problems run with Version 910 of LS–DYNA3D. First, a standard bar impact problem with 972 brick elements is carried out for 2,233 problem cycles; second, a metal rail crushing problem containing 1632 shell elements is calculated to 707 time steps. The POWER2 system execution times are given for both the original Model 580 executable and the recompiled POWER2 executable (rows 1 to 4). The recompiled version probably benefits most from the inclusion of the Square Root instruction.

Global Weather

Weather prediction codes are used continuously on both global and local levels to provide weather information. High computational performance is paramount to weather prediction calculations to provide timely weather information (and to contribute to the employment stability of your local meteorologist). The global weather prediction model in this study [8] contains eighteen vertical layers with the prognostic variables represented in the horizontal direction by a truncated series of spherical harmonics (truncated at 126). For each time step, the base algorithm includes a forward and inverse Legendre transformation and an FFT in the horizontal direction, as well as discretized grid computations in the vertical direction. Performance tuning involved replacing the FFT code with the corresponding function in the IBM Engineering and Scientific Subroutine Library (ESSL) and using standard unrolling techniques in the Legendre transformations. Quad–word storage references, Square Root, and divide performance are the primary contributors in the speedup over the Model 580. The CPU times reported in row 5 of Table 4 are for a three–hour real time simulation excluding model initialization and post–processing of the output data set.

ABAQUS Version 5.2 Standard Timing Runs (CPU Times in Seconds)						
	T1–5–2	T2–5–2	T3–5–2	T4–5–2	T5–5–2	T6–5–2
Model 580 512MB	1371	1709	1215	585	2298	1019
Model 590 512MB	712	1018	666	299	1445	473
RMS Wavefront	992	230	509	1106	79	418
Speedup	1.93	1.68	1.82	1.96	1.59	2.15

Table 3 ABAQUS (no recompile) Performance results.

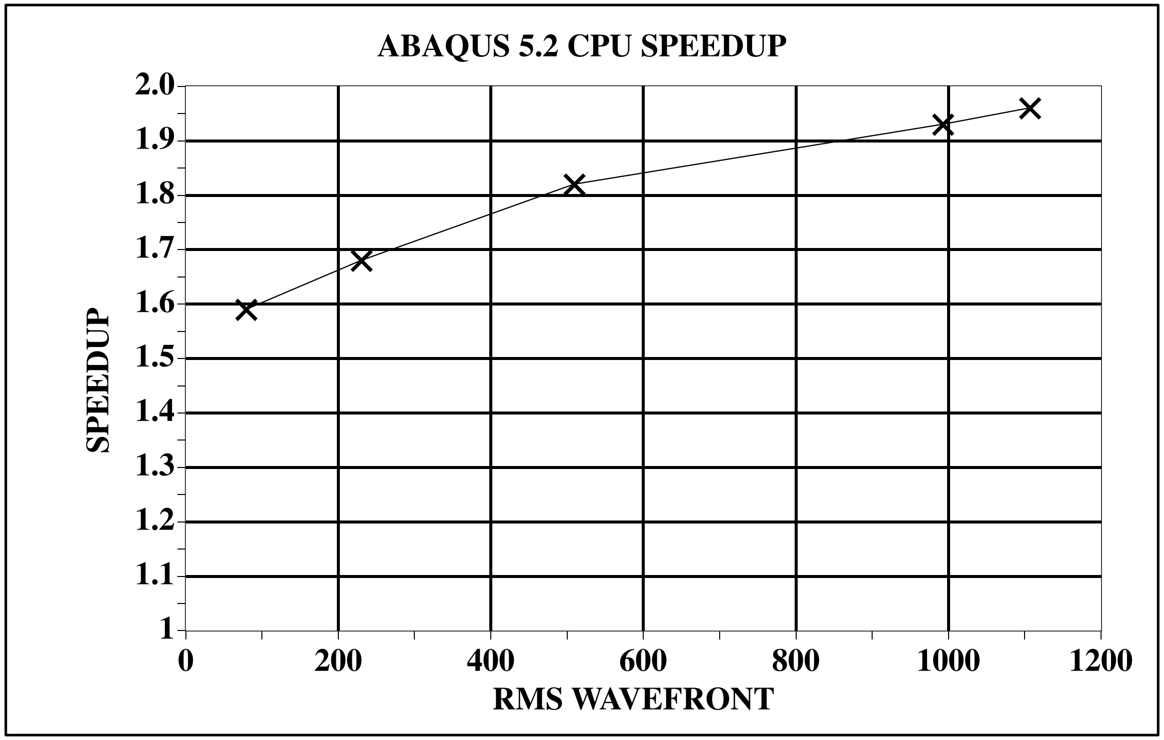


Figure 1 ABAQUS 5–2 speedup for ABAQUS/Standard benchmark problems T1–5–2 through T5–5–2.

Computational Fluid Dynamics

The main purpose of Computational Fluid Dynamics (CFD) is to simulate the flow field around or through an object. During the past several decades, CFD applications have been developed and intensively applied within the aerospace industry to design aircraft structures and engines with higher performance and fuel efficiency. Application of CFD has expanded in recent years to include other major users such as the automotive and process industries.

CFD – 3D Pipe Flow

The first computational fluid dynamics application is a three–dimensional pipe flow simulation using an implicit finite difference discretization with the SIMPLE method. The dominant kernel is line successive over relaxation (SOR) with a tri–diagonal solver. Grid resolution is 67 by 35 by 90, with each grid point having 6 variables (3 velocity components, temperature, pressure, and turbulent energy). The test data is for 11 time steps. The computations contain a high frequency of divide instructions. The sixth row of Table 4 shows the statistics for a version tuned originally for the Model 580. The tuning consisted primarily of using loop interchanges to produce a more favorable storage reference pattern (stride–one) for a cache–based machine. Also shown (row 7) are the statistics for the original version of the program. Note that although the absolute performance is less, the relative speedup is greater than that of the tuned code. This demonstrates the advantage of the larger cache for programs which are not written to take full advantage of a cache–based implementation.

CFD – ARC3D

ARC3D solves the three–dimensional Euler equations using an implicit central differences scheme. The version used in this study is set up for a body with a plane of symmetry. Grid resolution for the first case is 71 by 35 by 35 and for the second case is 101 by 51 by 51. The test data is for 20 time steps for both grid resolutions. The alternating–direction implicit (ADI) method used in ARC3D sweeps the data space in the x, y and z directions. The x direction sweep contains a large stride memory reference pattern (i.e. memory references with large storage gap between them). Larger computational grids result in larger strides. The Model 590 results reflect both hardware and software advances. Here, the XLF 3.1 compiler automatically detects this unfavorable stride loop construct and changes the nesting structure of the loops for the x–direction calculations to improve the stride. This, in combination with the dual execution units, combines to give excellent speedups of 2.57 and 3.16 for the two data cases, respectively (rows 8 and 9).

Computational Chemistry

Computational chemistry applications play a major role in modern research and development efforts to provide information about molecular structures and interactions. In particular, they have wide use in the pharmaceutical and chemical industries for assisting in areas such as rational drug design and new material development.

Computational Chemistry – Ab Initio

Ab Initio calculations are used to predict molecular structures and electron distributions for systems of molecules. The example used in this study computes an extended basis set for an ethylene singlet. This program is characterized by bit manipulations, integer calculations, and an extensive branch structure. A profile of the code shows a majority of time spent in computed **GO TO** structures where there are two or less instructions between branches. The speedup over the Model 580 is in large part attributable to the additional integer capabilities due to the second fixed–point execution unit, the larger data cache, improved instruction fetching and branch processing, and faster clock speed (see row 10).

Computational Chemistry – Molecular Dynamics

The second example from the field of computational chemistry is a molecular dynamics simulation of 13,500 particles where atoms interact by a Lennard–Jones potential. This numerical experiment is carried out to five time steps. Shown are successive refinements of the particle interaction algorithm (rows 11 to 13). The Model 590 attains a speedup of 1.43 over the Model 580 on the original version of the code after recompiling. Examination of the run–time profile shows a significant percentage of time in the Fortran intrinsic DINT. A more efficient version of the code eliminates the need for the DINT intrinsic by using two unrolled IF constructs along with scaled molecular coordinates. This modification increases the utilization of the multiple functional units, resulting in a speedup of 1.91. In a far more extensive refinement of the same code, neighborhood lists are employed along with a section sorting algorithm [9]. The constructs are unrolled, but the improved algorithm reduces

Row Reference Number	Application Description	CPU Seconds		Speedup
		Model 580	Model 590	
1	CSM – LSDYNA3D BAR IMPACT	120.4	71.1	1.69
2	CSM – LSDYNA3D BAR IMPACT (Recompiled)	120.4	62.3	1.93
3	CSM – LSDYNA3D RAIL CRUSH	114.5	71.7	1.60
4	CSM – LSDYNA3D RAIL CRUSH (Recompiled)	114.5	61.9	1.85
5	GLOBAL WEATHER MODEL (Recompiled)	2085.5	1064.5	1.96
6	CFD – 3D PIPE FLOW (Tuned for Model 580)	226.5	133.3	1.70
7	CFD – 3D PIPE FLOW (Original version)	402.1	204.31	1.96
8	CFD – ARC3D 71x35x35 (Recompiled)	339.0	131.8	2.57
9	CFD – ARC3D 101x51x51 (Recompiled)	1381.9	437.0	3.16
10	CHEMISTRY –Ab Initio SCF (Recompiled)	643.1	562.0	1.62
11	CHEMISTRY – MD (Original version, recompiled)	920.0	641.1	1.43
12	CHEMISTRY – MD (Tuned—no DINT'S)	300.0	156.9	1.91
13	CHEMISTRY – MD (Section sort with lists)	14.9	9.8	1.52
14	SEISMIC – 3D DEPTH MIGRATION (Recompiled)	133.1	69.2	1.92
15	SEISMIC – MG3D (Recompiled)	595.0	296.3	2.01
16	PHYSICS – MCNP (Recompiled)	382.1	244.9	1.56

Table 4 Application Performance

the computation involved. This refinement reduces the overall CPU time, and, in this case, the speedup relative to the Model 580 as well.

Seismic

Seismic applications are used extensively in oil exploration to assist in the search for oil reservoirs. Data is collected by exploding charges on land or ocean surfaces and then recording the resulting seismic waves over a large area. This data is processed by a series of computer programs to effectively reconstruct the earth's subsurface features. Experts examine the reconstructed images to locate high potential sites for oil drilling.

Seismic – 3–D Depth Migration

The first seismic application examined is one of the compute stages used in a process called 3–D depth migration. Due to the accuracy of the data collection equipment, seismic codes such as this are generally single–precision codes. The key kernel is a complex 2D convolution, which has roughly the same number of arithmetic instructions as load/store instructions. The kernel obtains a speedup of approximately 2 by utilizing the dual fixed–point and floating–point units. Overall, the speedup is 1.9 (see row 14).

Seismic – MG3D

MG3D is another 3–D depth migration code and is a member of the Perfect Club Benchmark Suite 1. The benchmark uses an X–Z grid of 125 x 120 points; FFT computations consume the bulk of the CPU time [10]. The FFT calculations are dominated by arithmetic operations, not load/store operations. However, many of the load/store operations involve large strides, a characteristic of the routines initially developed for Cray supercomputing systems. The Model 590 combination of the large data cache and dual floating–point units provides twice the Model 580 performance (see row 15).

Physics – Monte Carlo Neutron/Proton Transport

Monte Carlo methods are very general and are used in many branches of technical computing from solid state physics to finance. Monte Carlo calculations fundamentally generate a large number of random events (using a random number generator) and subsequently categorize each event.

MCNP (Monte Carlo Neutron/Proton) is a Monte Carlo based radiation transport model. The model requires specification of a radiation source and a geometry through which the radiation is transported. Random numbers are used to generate events or particle tracks. MCNP is used in many fields including reactor design and radiology.

The results presented here (row 16) are the total run–times of the twenty–five example problems provided with the code. This code is characterized by extensive branch structures (finding the correct event bucket in which to count the event). The larger caches, improved instruction fetch and branch processing capabilities, faster cycle time, and dual fixed–point execution units all contribute to the observed speedup over the Model 580.

Conclusion

The POWER2 architectural and hardware extensions bring productive new capabilities to the RISC System/6000 family of processors, as demonstrated by the Model 590 performance measurements documented in this paper. The increased number of concurrent operations at the processor level, and the widening of the data paths, substantially improve engineering/scientific application performance. Some applications, especially those which are tuned to take advantage of the original POWER architecture, show performance gains of up to two times the Model 580 with the same executable. For many other applications, recompilation provides a significant improvement in performance. The larger data cache, quad–word storage references, faster divide hardware, Square Root, improved instruction fetch, improved branch processing, and dual units all contribute to the outstanding POWER2 engineering and scientific application performance. For practicing engineers and scientists needing supercomputer class performance at workstation budgets, IBM POWER2 systems bring yet another exceptional leap in productivity.

Acknowledgements

The authors would like to acknowledge Jacob Thomas for his assistance in obtaining the performance data, Steve White for his valuable editorial and technical advise, and Linton Ward for sharing his insight and opinions on the subject matter.

References

1. Sohel Saiyed, J. Michael O'Connor, and Maurice Franklin, "POWER2 CPU–Intensive Workload Performance," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??
2. Maurice Franklin, William Alexander, Rajiv Jauhari, Ann Marie Grizzaffi Maynard, and Bret Olszewski, "POWER2 Commercial Workload Performance," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??
3. Steven W. White and Sudhir Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??
4. Hughes, Thomas J.R., T*he Finite Element Method,* Englewood Cliffs, NJ: Prentice–Hall, 1987.
5. *ABAQUS/Standard Benchmark Times*, Hibbitt, Karlsson, and Sorensen, Inc., 1080 Main Street, Pawtucket, RI 02860.
6. Dongarra, Jack J., Duff, Iain S., Sorensen, Danny C., van der Vorst, Henk A., *Solving Linear Systems on Vector and Shared Memory Computers,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.
7. Bell, R., *IBM RISC System/6000 NIC Tuning Guide for FORTRAN and C*. Poughkeepsie, NY, GG24–3611, IBM Corporation, 1991.
8. Clive Whittaker of IBM Federal Systems Company was responsible for porting this code to the RISC System/6000.
9. The neighborhood list program was originally written for this purpose by Rad Olson of the IBM Almaden Research Center
10. Kipp, L., *Perfect Club Benchmark Suite 1 Documentation*, Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, 1989.

ABAQUS/Standard is a proprietary software product developed and maintained by Hibbitt, Karlsson, & Sorensen, Inc. LS–DYNA3D is a proprietary software product developed and maintained by Livermore Software Technology Corporation. MSC/NASTRAN is an enhanced, proprietary version developed and maintained by The MacNeal–Schwendler Corporation.

Using the XL Compiler Options to Improve Application Performance

Kate Stewart

Introduction

The interaction between compiler options and preprocessors and their effects on an application's overall performance is generally not well understood. This paper examines the problem in the context of the new POWER2 and PowerPC 601 implementations in the RISC System/6000 family of machines, using the new XL compilers which exploit them. Experimentation shows that most applications can achieve a significant portion of their potential performance improvement using just basic optimization options. The performance improvements from additional architecture and application specific options, provided by the compilers and preprocessors, are described as well. This discussion illustrates the effect that tuning can have, and provides a framework for the user to judge the amount of tuning effort likely to be worthwhile.

Methodology

In this paper, the CINT92 and CFP92 benchmark suites from SPEC [1] have been selected as a set of sample applications for illustrating the improvements possible by the addition of various compiler and preprocessor options. The results are expressed SPECratios which are calculated by dividing a benchmark's standard reference time by its actual execution time. The SPEC benchmark aggregates are defined as the geometric mean of the SPECratios of the integer (SPECint92) and floating–point (SPECfp92) application sets. These aggregates should only be used as a general guide. If one of the benchmarks is similar to your own application, it should be used in preference to the aggregates.

All measurements were made with early versions of the compilers (IBM C Set ++ for AIX/6000 V. 2.1 and XL Fortran V. 3.1) on a RISC System/6000 Model 250 (PowerPC 601 processor), a RISC System/6000 Model 590 (POWER2 processor), and for comparison purposes, a RISC System/6000 Model 580 (POWER processor). The KAP for IBM C 1.3 preprocessor was used with the C applications. The preprocessors used for Fortran applications were early versions of the new release of KAP for IBM Fortran from Kuck and Associates, Inc. and VAST–2 for XL Fortran from Pacific–Sierra Research. Although the experiments were done with early versions of the compilers, the general conclusions are expected to hold when the production level versions of the compilers become generally available. Figures 1 through 3 show the SPECratios of the benchmarks compiled for the POWER processor. Figures 4 through 6 show the SPECratios of the benchmarks compiled for the POWER2 processor. Figures 7 through 9 show the SPECratios of the benchmarks compiled for the PowerPC 601 processor. Figures 1 through 9 are expressed in SPECratios to provide a normalized basis for performance comparison. These figures can be found at the end of the paper.

Using Basic Optimization

Adding the basic optimization flags substantially improves the performance of most applications. The primary optimization flag **–O** is equivalent to **–O2**. It performs only safe transformations on the code with very little context knowledge of the code. As a rough guide, applications compiled with **–O** will run between two and three times as fast as those with no optimization. Table 1 gives the increase in performance for the SPEC aggregates when the **–O** option is used. This is also illustrated in Figures 1,4, and 7.

Implementation	SPECint92	SPECfp92
POWER	116%	213%
POWER2	120%	206%
PowerPC 601	82%	123%

Table 1 Improvements due to adding **–O**

The amount of possible improvement will vary, based on the architecture's implementation and the specific application. Overall, the compiler produces more dramatic increases on POWER and POWER2 systems than on PowerPC 601 systems when the **–O** option is added. Individually, on the 6 CINT92 and 14 CFP92 benchmarks, adding **–O** yields the range of improvements shown in Table 2.

Implementation	minimum	maximum
POWER	26%	485%
POWER2	31%	508%
PowerPC 601	26%	265%

Table 2 Improvement ranges due to **–O**

When benchmarks are reported, the best known set of options for the compiler and preprocessors is used. Controversy over which flags are acceptable when measuring benchmarks has developed, as vendors become more aggressive in their search for higher SPEC numbers [2]. It should be noted that the **–O** option on its own can achieve a significant portion of this final performance. This is quantified in Table 3, and illustrated in Figures 1 through 9 at the end of the paper.

Implementation	SPECint92	SPECfp92
POWER	74%	74%
POWER2	69%	67%
PowerPC 601	73%	76%

Table 3 **–O** compared to best known options

In this release, the **–O3** level of optimization has been significantly enhanced. This option now allows the compiler to use more time, more memory intensive transformations, and more aggressive optimization assumptions. In many cases, an improvement in the performance of an application results.

Floating–point applications which should benefit the most from this more aggressive optimization level include those designed not to raise exceptions, not to care about the sign of zero, and not to be sensitive to side effects in precision from reassociating floating–point expressions. Applications dominated by memory accesses may also see some improvement from the more aggressive optimization assumptions.

The **–O3** optimization level does have the potential to change the semantics of a program unless the **–qstrict** option is specified. When **–O3** is used on its own, code that may trigger an exception is allowed to be moved by the compiler. Specifically, loads and floating–point calculations can be moved by the compiler into areas where they are determined to be profitable, even though the operations may not have been executed in a stricter interpretation of the original program. The sign of zero is ignored by the compiler when this option is enabled. Floating–point expressions are also allowed to be reassociated, providing more opportunities for the compiler to detect common subexpressions in the calculations, and enhance floating multiply–add instruction generation. This may affect the rounding of results: equivalent mathematical results are delivered, but, the answers may not be exactly identical since the calculations are performed with finite precision. If your code is sensitive in any of these areas, you can still get some of the benefit by using **–O3** with the **–qstrict** option. The **–qstrict** option directs the compiler not to change the semantics of the program in the manner outlined above.

The **–O** option always has the **–qstrict** option implied.

When **–O3** is specified, the default settings of some options change, specifically **–qfloat=fltint:rsqrt** and **–qmaxmem** is set to use unlimited memory during compilation. The **–qfloat=fltint** option removes range checking on conversion of floating–point values to integer values. The **–qfloat=rsqrt** option means that when a square root operation appears as the divisor in a calculation, the optimizer can substitute a multiply by the result from a reciprocal square root library call. This saves an expensive divide operation from being required. The **–qmaxmem** option allows the user to specify the amount of memory to be used during compilation. When **–O3** is specified, this option's default is set to allow the compiler to use as much memory as it needs during the optimization of an application. Longer compile times can generally be expected for complex applications when this option is used. The **–qmaxmem** option defaults to a lower setting for the **–O** option.

Implementation	SPECint92	SPECfp92
POWER	3%	4%
POWER2	4%	7%
PowerPC 601	0%	0%

Table 4 Improvement of **–O3** from **–O**

Table 4 quantifies the improvements in the SPEC aggregate's performance realized by increasing the optimization from **–O** to **–O3.** The aggregate numbers are slightly misleading for the PowerPC 601 processor:

certain individual benchmarks are improved by adding **–O3**, but, the performance of others becomes worse. A user should verify that **–O3** does improve the performance of an application before using it by default. By providing more information about the application and the target machine on which the application will run, further performance improvements can be achieved.

Using architecture information

The new POWER2 floating–point instructions present opportunities recognized by the compiler that are not present on the POWER machines [3]. Similarly, the PowerPC 601 has single–precision floating–point instructions which are not present in the POWER architecture [4]. To provide the ability to customize an application to exploit specific architectures, two new options have been introduced: **–qarch** and **–qtune.** Binaries created using the **–qarch** option may have instructions that will not run on architectures other than the one to which they are explicitly targeted.

Using the architecture specific option **–qarch** and **–O3** option together, a larger portion of the possible performance is obtained than with just **–O**. Table 5 illustrates this by comparing the aggregates for the SPEC benchmarks obtained with just the **–O3** and **–qarch** options applied compared to the aggregates obtained with the best known set of options. A comparison of Table 5 with Tables 3 and 4, illustrates the major improvement potential for floating–point applications on POWER2 and PowerPC 601 based systems when the **–qarch** option is used.

Implementation	SPECint92	SPECfp92
POWER	76%	77%
POWER2	72%	78%
PowerPC 601	74%	85%

Table 5 –O3 –qarch compared to best

The **–qarch** option tells the compiler which instruction set to use to generate code. Each machine architecture has its own set of valid instructions. Valid **–qarch** values are **PWR**, **PWRX**, **PPC,** and **COM**. Using **–qarch=PWR** instructs the compiler to generate code for the classic POWER processors. The POWER, POWER2, and PowerPC 601 implementations all support the POWER user instruction sets generated by the compiler.

The **–qarch=PWRX** option instructs the compiler to use the instruction set of the POWER2 processor, and to tune the application to this platform. If an application is dominated by double–precision floating–point calculations, or floating–point loads, stores, or square roots [5], it may be beneficial to specify **–qarch=PWRX** when compiling the application to run on the POWER2 machines.

The **–qarch=PPC** option tells the compiler to generate code for the PowerPC architecture and tune it for the 601 implementation of PowerPC architecture. Applications containing single–precision floating–point code should benefit from this.

If binary compatibility on all the POWER, POWER2, and PowerPC 601 machines is required, the **–qarch=COM** option should be used. This option instructs the compiler to generate only instructions which are common to all three systems. This option does not generate a true intersection of the POWER and PowerPC architectures. It may generate a couple of seldom used instructions that exist on PowerPC 601 but will need to be emulated on other PowerPC implementations. This setting is the default assumed by the compiler if no **–qarch** option is explicitly specified.

If binary compatibility is required but the application is expected to run predominantly on a specific set of systems, the **–qtune** option may improve performance. Valid machines to tune for are **PWR, PWRX,** and **601**. If the **–qarch** option is set to **PWR**, PWRX, or **PPC**, the **–qtune** option will default to a setting of **PWR**, **PWRX**, or **601** respectively. When **–qarch=COM** is active, the default **–qtune** option is set to **PWR**. If an application is expected to run primarily on a POWER2 processor, it may be desirable to tune the generated code to favor that processor. This can be accomplished by specifying **–qtune=PWRX**, which will schedule for the multiple execution units of the POWER2 and perform some optimizations specific for that processor [6]. Similarly, if an application is expected to run predominantly on a PowerPC 601 machine, experimentation with the **–qtune=601** option should be performed. Since the PowerPC 601 processor is a bridge between the POWER and PowerPC architectures, some applications may perform better if **–qtune=PWR** is specified. If **–qtune** is specified without **–qarch**, the default is to assume **–qarch=COM**. Table 6 indicates the valid combinations of the **–qarch** and **–qtune** options.

	–qtune=		
–qarch=	PWR	PWRX	601
COM	X	X	X
PWR	X	X	X
PWRX	X	X	
PPC	X		X

Table 6 Valid **–qarch**, **–qtune** combinations

In this release, control over the memory model used by the compiler is given to the user through the **–qcache** option in the Fortran compiler (XLF 3.1). Customizing the size of and penalties for accessing the memory hierarchy allow the compiler to more accurately transform storage references. This is only effective when used with the **–qhot** option, which allows the compiler to manipulate the order of references to storage, and minimize the total penalty for accessing memory. Default compiler settings for the memory hierarchy configuration correspond to the **–qtune** options; if your machine does not match these configurations, adjusting the **–qcache** option can improve your performance [7]. This option is particularly important for Fortran applications that are dominated by patterned array accesses (i.e., stride–1, stride–n, triangular, etc.).

C applications compiled for the POWER2 may further benefit from the **–qunroll** option introduced in the C Set ++ 2.1 compiler. Some unrolling of inner loops will occur by default when **–O3** is specified. This option allows the user further control of the number of iterations that will be unrolled. On POWER2 machines, this may benefit applications whose inner loops are dominated by memory references.

Using application specific information

Without information about an application, the compiler must make pessimistic assumptions. Compiler options are one mechanism to communicate information, which lets the compiler relax assumptions and produce faster correct code. Options aimed at minimizing call penalties, minimizing storage reference penalties, and enabling exploitation of native floating–point capabilities can be used to significantly enhance the performance of an application beyond the basic optimization levels. These options will not affect the binary compatibility of applications across the POWER, POWER2 and PowerPC 601 implementations.

The results of applying these additional options to the SPEC benchmarks are shown in Figures 1 through 9. In these figures, the shading that indicates the best compiler options, refers to the best set that was known at the time the runs were made. Better combinations of options may exist. Table 7 shows the improvement of the SPEC aggregates from applying the additional compiler options over that of –**O3** and **–qarch** only.

Implementation	SPECint92	SPECfp92
POWER	21%	10%
POWER2	22%	11%
PowerPC 601	23%	3%

Table 7 Improvements from more options

In well–structured modular programs, the cost of calling and returning from routines can be minimized using a variety of options. Using the inlining and interprocedural analysis options while minimizing aliases can aid the compiler in improving the overall application performance.

<u>Inlining and Interprocedural Analysis</u>

In all of the XL compilers, the **–Q** option tells the compiler to automatically inline (copy the body of a called routine to the call site) the routines it believes are suitable, based on size [8,9,10]. This is only valid when one of the optimization options (**–O**/**–O3**) is applied as well. Manual control over which routines are inlined is also possible through **–Q+name1:name2** to inline named routines, and **–Q–name1:name2** to exclude named routines from being inlined. If a routine is called from within a loop, inlining may be particularly beneficial, as certain optimizations can be enabled that were blocked by the presence of the call. Excluding specific calls from being inlined when they are not in the main path of execution (e.g., debug code and error code) may also improve the performance if automatic inlining is enabled.

One side effect of inlining code is that the overall code size usually increases. The size of the application, the order instructions are accessed, and the instruction cache configuration on the system, may result in a decrease in the overall performance of an application when inlining is specified.

In this release, a new option is introduced into the XLF and C Set ++ compilers: **–qipa**. This option enables interprocedural analysis of the application, and based on this information, the compiler may optimize the code further to improve the overall application performance [11]. It can optionally make use of the profile data provided by the **prof** and **gprof** commands, if they have been run for the application. This allows the optimizer to concentrate its efforts on the routines which are believed to dominate the application's running time. A significant increase in the compile time may result when this option is used, as the application must be compiled twice to allow the information to be collected, then applied to the entire application. The **–qipa** option should be particularly beneficial for large applications with minimal aliasing side effects in the calls between routines.

Aliasing

Aliasing is a compiler term for a storage location having multiple variables which reference it. When potential aliases occur, they inhibit the assumptions a compiler can make when optimizing a program. Some examples of aliased references in Fortran are: variables in a COMMON block associated with other variables via an EQUIVALENCE statement, and variables used for multiple parameters in a procedure call whose variables are passed by reference. In C and C++ applications, pointers may result in pessimistic compiler aliasing assumptions. If the pointers in an application are known only to point to objects of the same type, then the **–qansialias** flag can be used in the C compilers to let the compiler know that ANSI compliant pointers can be assumed.

In the C Set ++ 2.1 compiler, the options **–qdisjoint** and **–qassert** allow the user to provide further aliasing information to the compiler [7,10]. The **–qdisjoint** option allows the user to specify variables in a function which are explicitly not aliased to each other. The **–qassert** option can be set to the following assertions: **typeptr** indicates that no two pointers to different types will point to the same storage location, **allptrs** indicates that no two pointers will point to the same storage location, and **addrtaken** indicates that pointers never interact with variables unless one variable of a class had its address is explicitly taken. In code dominated by well behaved pointers, the appropriate assertions allow the compiler more opportunities to improve the performance of an application.

By default, the Fortran compiler does not check for aliasing side effects. If there is the possibility of aliasing that violates the ANSI standard existing in an application, it may be necessary to specify **–qxflag=xalias**. This option may slow down the application. Aliasing should be avoided when possible.

Floating–Point Precision

Options that exploit the double–precision floating–point instructions native to the POWER2 processor can significantly improve the performance of an application. On the POWER and POWER2 processors, the greatest boost in performance is usually seen if a single–precision application can be transformed into a double–precision application (space limitations permitting). This can happen automatically in Fortran applications by using the **–qautodbl=dblpad** option. This option promotes REAL*4 and REAL*8 variables to REAL*8 and REAL*16 respectively and pads appropriately. In XLF 3.1, a new subset of these options has been introduced to give the user a greater degree of control over the data types promoted and padded. Specifically, **–qautodbl=dblpad4** will pad and promote REAL*4 to REAL*8, but will not promote REAL*8 variables. In C and C++, there is no equivalent of this, but using the substitution **–Dfloat=double** achieves the same effect.

If promoting the application to double precision is not possible, the XL compilers provide the **–qfloat=hssngl,** the **–qfloat=norndsngl,** and the **–qfloat=hsflt** options which may be used to minimize rounding to single–precision values. Since the POWER and POWER2 floating–point hardware runs in double–precision mode, the results must be rounded to single–precision values after each calculation unless the compiler is told it is safe to keep the intermediate results in double–precision format. The **–qfloat=norndsngl** tells the compiler to only round after the full expression has been evaluated. This is the default for the compiler; otherwise, if **–qfloat=rndsngl** is in effect, rounding occurs after every floating–point instruction. The **–qfloat=hssngl** tells the compiler to keep all intermediate results in double–precision format and only convert to single–precision format when a store occurs. The **–qfloat=hsflt** option is even more aggressive; there is no explicit conversion to single–precision values prior to the results being stored and no checking for overflows on floating–point to integer conversion. This option is not recommended unless rounding error is not an issue and the numbers are guaranteed to be in range.

The PowerPC 601 processor differs from the POWER and POWER2 processors in that it performs single–precision computations more efficiently than double–precision ones. When the compiler is enabled to generate the single–precision floating–point instructions through use of the **–qarch=PPC** option, performance of an application may be improved if the application is dominated by single–precision computations.

If a floating–point application can accept rounding error and can tolerate a looser implementation of the fringes of the IEEE 754 standard, then some of the **–qfloat** suboptions may be of benefit. Some of these are assumed by default for **–O3** and have been discussed already. The **–qfloat=fold** option is enabled by default, and allows the compiler to evaluate constant floating–point expressions at compile time rather than at run time. The **–qfloat=nonans** option instructs the compiler not to worry about generating code to detect signaling NaNs (Not a Number) on conversion from single–precision format to double–precision format at run time. The default settings for these options and others can be found under the description of **–O** and **–O3** options in [8,9,10].

Using source preprocessors

Some applications may see further improvements in performance from the use of source preprocessors. These preprocessors read in a source file (currently only C and Fortran preprocessors are available), parse it, apply optimizations, and produce a modified source file for the compiler to optimize. There are currently three preprocessors available that work with the XL compilers. KAP for IBM C will work with the XLC and C Set ++ compilers. KAP for IBM Fortran from Kuck and Associates, Inc. and VAST–2 for XL Fortran from Pacific–Sierra Research will work with the XLF compiler. Guidelines on how and when to use them can be found in [7].

Implementation	SPECint92	SPECfp92
POWER	8%	18%
POWER2	14%	14%
PowerPC 601	10%	15%

Table 8 Improvements from preprocessors

These preprocessors contribute significantly to the final performance of some applications. Figures 1 through 9 illustrate this at the benchmark level. Table 8 shows the improvement in the SPEC aggregates that has been obtained through use of the preprocessors.

One set of floating–point applications which benefit from the use of the preprocessors are those containing algebraic algorithms that exit in specially tuned libraries. In code that contains algebraic algorithms that are equivalent to **BLAS** (Basic Linear Algebra Subroutines) or **ESSL** (Engineering and Scientific Subroutine Library) routines, the preprocessors will attempt to recognize the code and transform it into an appropriate library call. There is a version of the **ESSL** library tuned to the POWER2 processors, in addition to the current version tuned for POWER processors. The **BLAS** routines are tuned to the POWER processors. Of the two Fortran preprocessors, our experiments show that VAST–2 for XL Fortran recognizes more of these special library idioms.

Another group of floating–point applications that benefits from the preprocessors are those dominated by loop based accesses to large memory structures. The preprocessors allow the user a greater degree of control over the source transformations to be applied to exploit the memory organization of the machine. Of the two Fortran preprocessors, our experiments show that KAP for IBM Fortran performs more sophisticated memory management.

Applications written in C and dominated by I/O may benefit from the KAP for IBM C preprocessor’s ability to reduce the overhead of reading and writing variables using the **–stdio** option. In particular, those applications dominated by *printf* statements should benefit from this optimization [12].

Historically, integer based applications have benefited from a preprocessor's ability to do inter–file inlining. This ability extended the XL compilers' native ability to inline code and allowed for the penalties of calls to be minimized for applications spread across several source files. This function may now be achieved through the **–qipa** and **–Q** options in the XL compilers as well. Experimenting with the preprocessor's inter–file inlining and the compiler's native ability to find the best one for an application is recommended.

Both integer and floating–point applications can also benefit from a preprocessor's ability to transform the source code to minimize the delays in a loop and maximize the parallelism available to the compiler. Transformations such as unrolling are under explicit user control in the preprocessor. This extends the capability provided by the XL compilers natively. For a further description of how unrolling benefits an application on the POWER2 environment, refer to [5]. KAP for IBM C also provides an additional tuning parameter through the **–small_loops** option, which benefits applications whose dominant loops contain just a couple of statements.

Conclusion

The use of basic optimization (**–O**) with code compiled to the intersection of the POWER and PowerPC architectures achieves a significant portion of an application's potential performance. Table 3 shows that between 67% and 76% of the potential performance of the SPECint92 and SPECfp92 aggregates is achieved. Increasing the optimization level and adding architectural information further improves the performance to between 72% and 85% as Table 5 indicates, but may jeopardize portability. Application specific information provided through options that do not jeopardize portability, may further improve the performance. For some applications, the XL compilers on their own, achieve the best possible performance. Other applications benefit further from the use of additional preprocessors such as KAP for IBM C, KAP for IBM Fortran, and VAST–2 for XL Fortran. The possible improvement that an application can realize using these tools ultimately depends on a sound knowledge of the application and the amount of time a user is willing to invest in experimenting with the different options.

Acknowledgments

The information in this paper has been drawn from the experience of many users and developers of the XL compilers. The TOBEY development group (Toronto and Hawthorne), and compiler performance group (Toronto) have provided much of the information presented here. The author would like to thank Joyce Janczyn, Karen Bennet, Bill Hay, Mike O'Connor, Sohel Saiyed, Da–Gung Lu, Muthukumar, Donald Tsang and Steve White for their comments and ongoing discussions.

References

1. Jeff Reilly, "CINT92 & CFP92 Benchmark Descriptions," *SPEC Newsletter*, Volume 4, Issue 4, Standard Performance Evaluation Corporation, December 1992, p. 9.
2. Kaivalya M. Dixit, "Flags–the Good, the Bad and the Ugly," *SPEC Newsletter*, Volume 5, Issue 3, Standard Performance Evaluation Corporation, September 1993, pp. 3–5.
3. Steven W. White and Sudhir Dhawan, "POWER2: The Next Generation of the RISC System/6000 Family," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp. ??–??.
4 Charles R. Moore, "The PowerPC 601 Microprocessor," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp. ??–??.
5. Sohel Saiyed, J. Michael O'Connor, and Maurice Franklin, "POWER2 CPU–Intensive Workload Performance," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp. ??–??.
6 Robert Blainey, "Instruction Scheduling in the TOBEY Compiler," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp. ??–??.
7. *AIX Version 3 for RISC System/6000: Optimization and Tuning Guide for Fortran, C and C++,* SC09–1705, IBM Corporation.
8. *IBM AIX XL Fortran Compiler/6000 User's Guide Version 3.1,* SC09–1354, IBM Corporation.
9. *AIX Version 3.2 for RISC System/6000 XL C User's Guide Version 1.3,* SC09–1259–03, IBM Corporation.
10. *IBM C Set ++ for AIX/6000 User's Guide Version 2.1,* SC09–1605, IBM Corporation.
11 David M. Gillies and Allan H.Kielstra, "Interprocedural Optimization in the XL Compilers", *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993, pp. ??–??.
12. *KAP for IBM C Version 1.0*, Kuck and Associates, 1992.

POWER PERFORMANCE

Best using preprocessors

Best using compiler options

–03 –qarch=pwr

–0

No optimization used

Figure 1

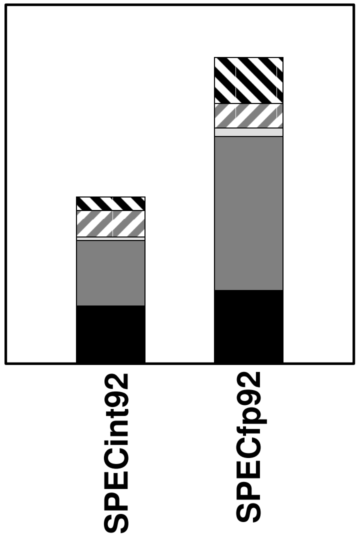
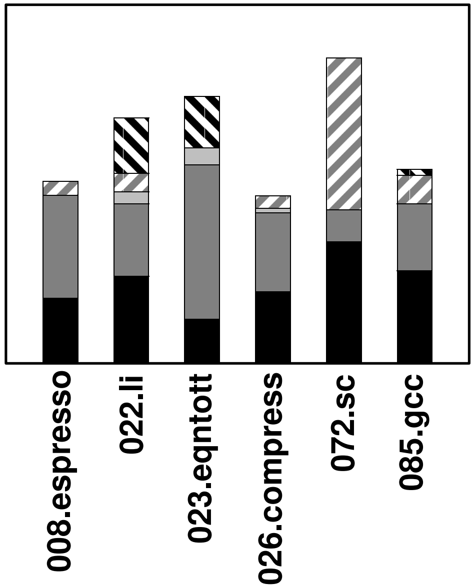


Figure 2: CINT92



ENVIRONMENT:

compilers:
C Set ++ 2.1 alpha
XLF 3.1 beta

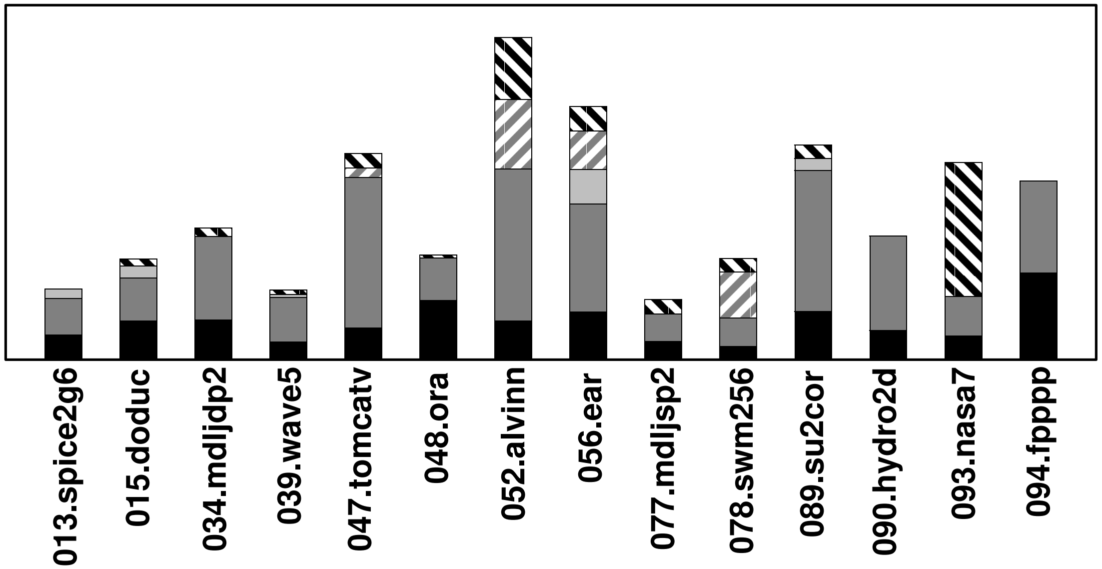
preprocessors:
KAP/C 1.3
KAP/FORTRAN beta
PSR/VAST beta

operating system:
AIX 3.2.5 beta

hardware:
RISC System/6000 Model 580

Details on the options used in the SPEC runs are available from the author on request.

Figure 3: CFP92



POWER2 PERFORMANCE

Best using preprocessors

Best using compiler options

–03 –qarch=pwrx

–0

No optimization used

Figure 4

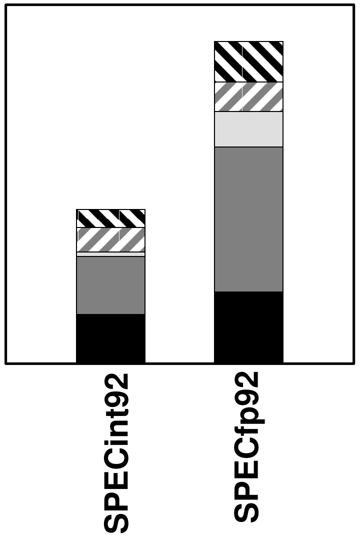
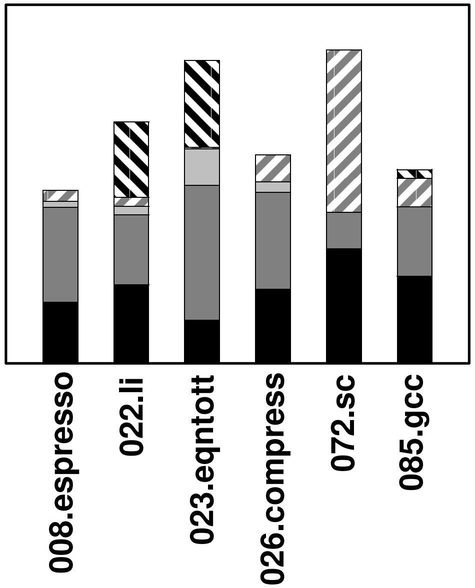
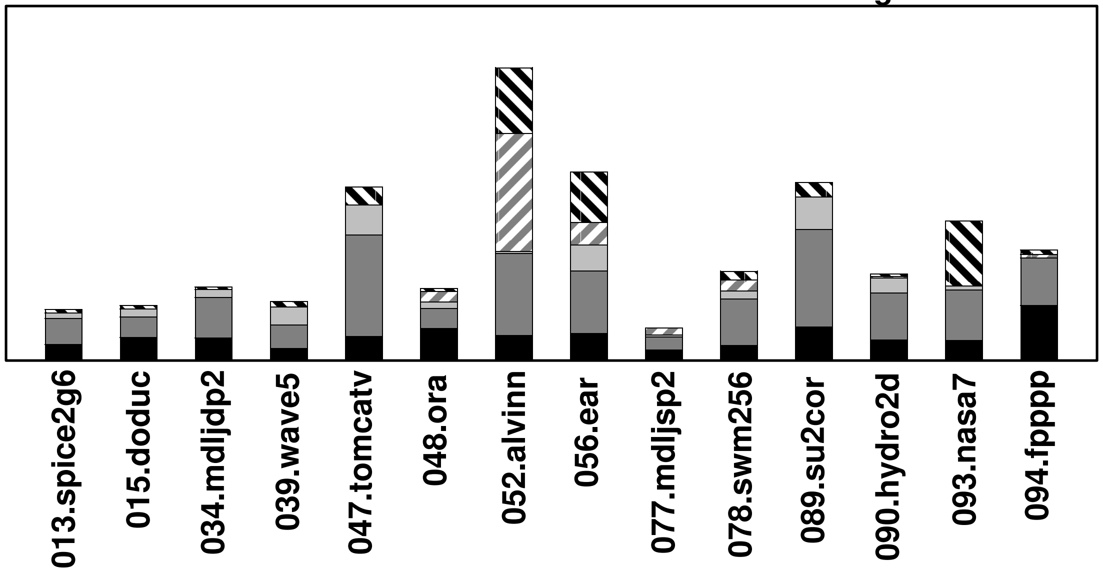


Figure 5: CINT92



ENVIRONMENT:

Figure 6: CFP92



compilers:
C Set ++ 2.1 alpha
XLF 3.1 beta

preprocessors:
KAP/C 1.3
KAP/FORTRAN beta
PSR/VAST beta

operating system:
AIX 3.2.5 beta

hardware:
RISC System/6000 Model 590

Details on the options used in the SPEC runs are available from the author on request.

PowerPC (601) PERFORMANCE

- Best using preprocessors
- Best using compiler options
- –03 –qarch=ppc
- –0
- No optimization used

Figure 7

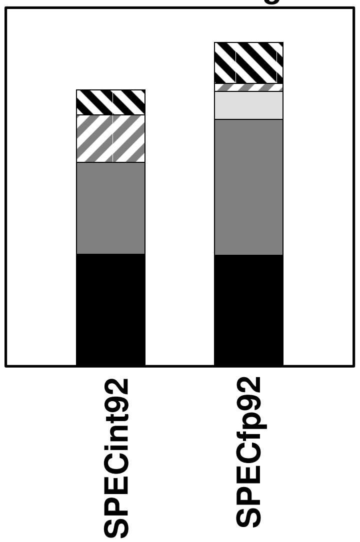
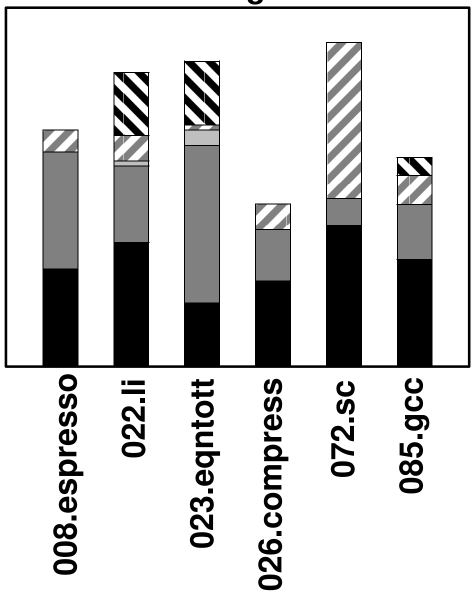


Figure 8: CINT92



ENVIRONMENT:

compilers:
C Set ++ 2.1 alpha
XLF 3.1 beta

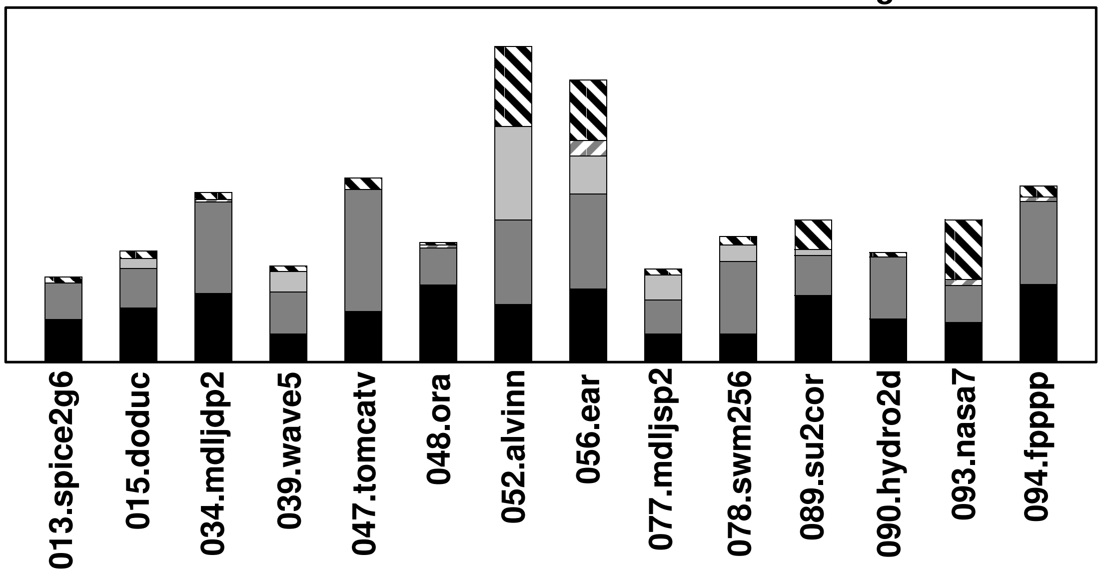
preprocessors:
KAP/C 1.3
KAP/FORTRAN beta
PSR/VAST beta

operating system:
AIX 3.2.5 beta

hardware:
RISC System/6000 Model 250 beta

Details on the options used in the SPEC runs are available from the author on request.

Figure 9: CFP92



Micro Channel Features

Dan M. Neal and James O. Nicholson

Introduction

The IBM Micro Channel interface is a general purpose computer interface intended to support the use of I/0 and feature cards in electronic data processing equipment. It was originally introduced with the IBM Personal System/2 series of products and is also utilized in the IBM RISC System/6000 (RS/6000), other IBM products, and other manufacturers' product lines.

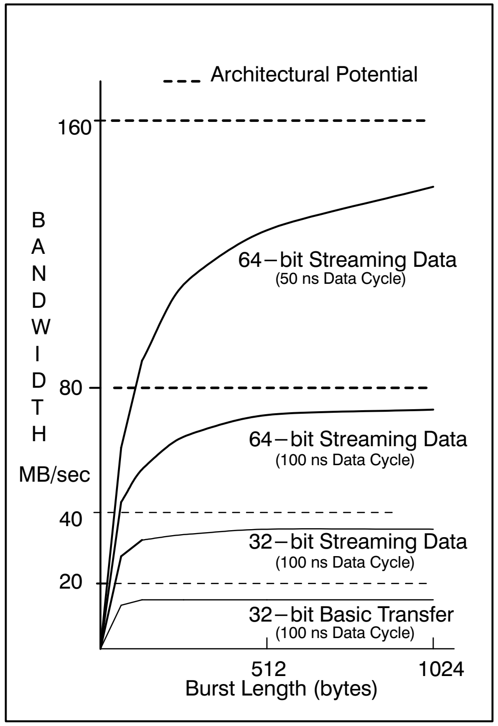


Figure 1 Streaming Data Performance Improvements

The Micro Channel bus has been designed to allow use across a broad product spectrum. Enhanced features of the Micro Channel architecture provide higher levels of performance and error reporting and are fully compatible with the base Micro Channel architecture. This article describes those features supported on the RISC System/6000 family of products. These include:

- Streaming Data procedure (32–bit and 64–bit at 10 MHz)
- Address and data parity
- Synchronous exception signalling

Streaming Data

Streaming Data is a procedure that provides the ability to transfer multiple data cycles within one bus envelope. The Streaming Data procedure amortizes device selection overhead across the total packet, significantly increasing the performance capability of the Micro Channel bus. Sustainable performance is a function of the total packet length and is illustrated in Figure 1.

Figure 2 illustrates the procedures used in basic transfer and 10 MHz streaming data cycles. A basic transfer cycle is initiated by a bus master placing an address on the bus,

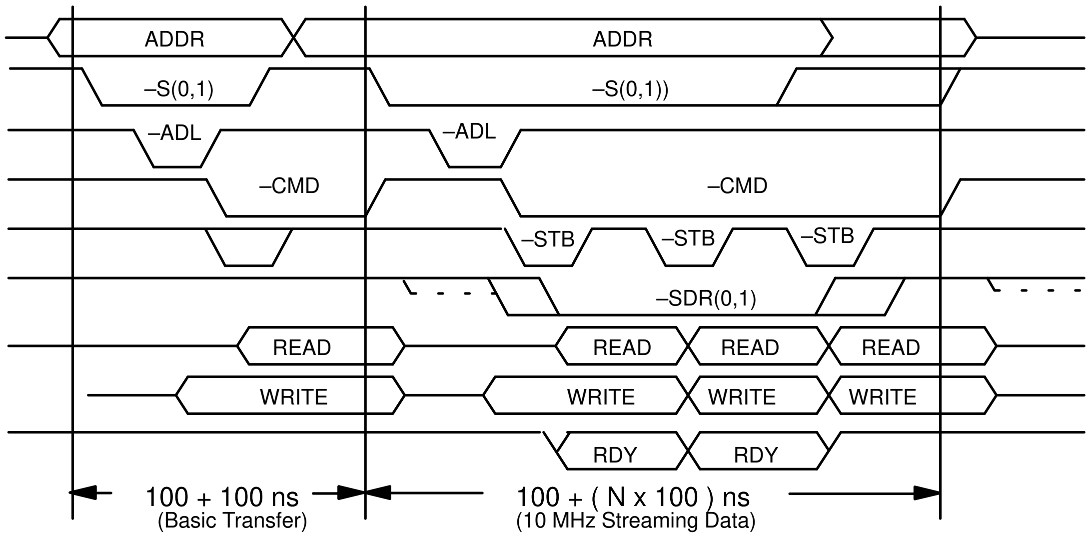


Figure 2 16/32–bit, Basic Transfer and 10 MHz Streaming Data

and then activating one of the status signals, –S(0,1). A typical bus master will allow 100 ns for device selection, then activate the command (–CMD) signal for 100 ns to control the data transfer. A streaming data cycle is similar, but instead of a word of data, a block of data is transferred while the –CMD signal is active. The address specifies only the starting address of the block.

Three signals, streaming data request 0 and 1 (–SDR (0,1)) and streaming data strobe –SD_STROBE (–STB), mediate 16–bit and 32–bit streaming data cycles. The –SDR (0,1) signals are driven by the bus slave following activation of the address data latch (–ADL) signal to indicate that it supports streaming data. The –STB signal is driven by the bus master and serves as a bus clock delimiting words in the data transfer. Each negative transition of the –STB signal defines a new word. The –SDR(0,1) and –STB signals use tri–state drivers and require resistive pullups on card inputs to hold the signals high when not being driven. At the end of a streaming data cycle, the deactivation of the –CMD signal serves as the last bus clock.

SDR 0 1	Resulting Cycle
1 1	**Basic Transfer Cycle**
0 1	**10 MHz Max (100 ns Min Cycle)**
1 0	**20 MHz Max (50 ns Min Cycle)**
0 0	**Reserved**

Table 1 Code Points for –SDR(0,1) Signals

The –SDR(0,1) signals are coded to indicate the performance capabilities of the slave. This information is used by the bus master for speed matching to ensure that it does not exceed the maximum clocking rates of the slave. A master can perform streaming data transfers at 10 MHz when –SDR(0,1) is a binary 00, 01, or 10. A master can perform streaming data transfers at 20 MHz when –SDR(0,1) is a binary 00 or 10. The valid streaming data signal combinations from the selected slave are shown in Table 1. At 10 MHz, the period of the –STB clock signal is 100 ns minimum. At 20 MHz, the period of the –STB clock signal is 50 ns minimum. The streaming data procedures are fully downward compatible. Masters and slaves that support 20 MHz streaming data must also support 10 MHz streaming data and Basic transfers. Likewise, Masters and slaves that support 10 MHz streaming data must also support Basic Transfers. The –SDR(0,1) signals do not have to be valid for a bus master to indicate streaming data support. Instead, a bus master normally activates the –STB signal concurrently with the –CMD signal and samples the –SDR(0,1) signals just before the deactivation of the –STB signal. If the –SDR(0,1) signals are not active at that time, the bus master deactivates the –S(0,1) signals and completes the cycle as a basic transfer. This provides relaxed timing requirements for the –SDR(0,1) signals.

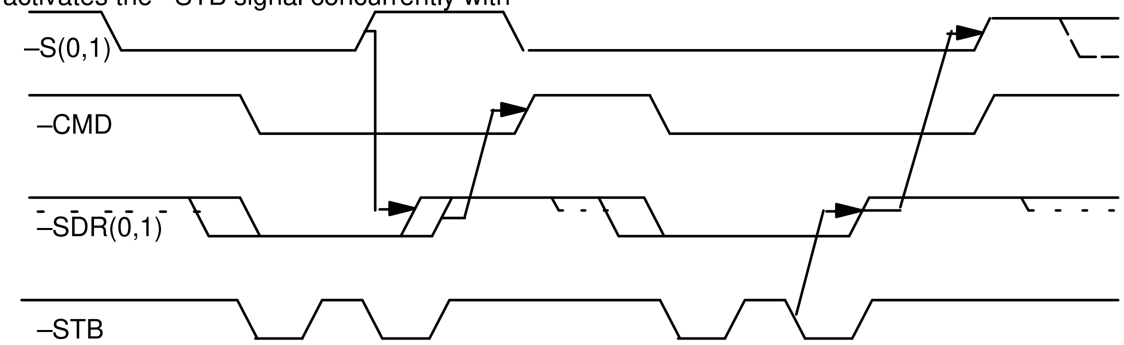


Figure 3 Streaming Data Termination (10 MHz)

The Streaming Data procedure for 16–bit and 32–bit transfers includes synchronous data pacing using the card channel ready (CD_CHRDY) and channel ready return (CHRDYRTN) signals. The slave indicates that it is not ready to complete a particular data cycle by activating the CD_CHRDY signal (RDY) after the –STB signal is activated. This is propagated to the CHRDYRTN signal by a logical AND operation on the system planar for sampling by the bus master. The bus master is required to repeat the data cycle if the CHRDYRTN signal is inactive (low) on the next transition of the –STB signal. The –STB signal must continue cycling until the slave activates the CD_CHRDY signal.

Termination controls are symmetric so that either the bus master or slave can end a streaming data cycle. The procedure is synchronous and is illustrated in Figure 3 for 10 MHz streaming data transfers. Bus masters request termination by synchronously deactivating the –S(0,1) signal coincident with activation of the –STB signal, while slaves request termination by synchronously deactivating the –SDR(0,1) signals following the activation of the –STB signal. Bus masters should not terminate the cycle until the –SDR(0,1) signals are deactivated. If a slave is not ready when a master deactivates the –S(0, 1) signal, the slave should set the CD_CHRDY signal low and hold the –SDR(0,1) signals active until the cycle in which the CD_CHRDY signal will be set high. Although not illustrated, the architecture also supports the ability to defer initiation of streaming data cycles using the CD_CHRDY signal.

The termination procedures for 10 MHz and 20 MHz streaming data transfers are similar, but have differences. The primary difference is that the request for termination occurs on the data transfer before the desired stopping point for 10 MHz streaming data, while the termination request occurs on the next–to–last negative transition of –STB for 20 MHz streaming data transfers. For more detail on streaming data termination, the reader should refer to the Micro Channel architecture specification [1].

For compatibility reasons, all 16–bit and 32–bit streaming data cycles must be initiated on 4–byte address boundaries. This is because the Micro Channel architecture includes a byte–steering function to facilitate transfers between 16–bit bus masters and 32–bit bus slaves. This steering function is latched at the time the –CMD signal is activated and remains static throughout the data transfer envelope. As such, it would not properly steer the data when operating with 16–bit streaming masters. Specifying streaming data to begin on 4–byte address boundaries disables the steering controls, but results in the requirement that 32–bit slaves provide their own byte steering when engaged in data streaming operations with 16–bit masters.

Another streaming data capability includes use of the address bus to transfer data. Since only a single address cycle is provided by the master during a streaming data operation, the address bus can be made available (multiplexed) during the data portion of the cycle to transfer data. This allows a ”doubling” of the available bandwidth by transferring 64–bits of data each data cycle instead of only 32–bits. An additional Streaming Data signal is used to indicate that the operation will include 64–bit data transfers, ”Multiplexed Streaming Data Request” (–MSDR). This signal is driven by the slave to indicate to the controlling master that it supports 64–bit streaming data transfers.

The 64–bit streaming data transfers can only be performed between 64–bit streaming masters and 64–bit streaming data slaves. A 64–bit streaming data cycle is initiated as a 32–bit basic transfer cycle, and all the rules associated with this procedure applies. The selected slave activates –MSDR to indicate that it supports 64–bit streaming data operations. The data rate requested by the slave is indicated by –SDR(0,1). The –BE(0–3) signals are driven inactive by the master to indicate a 64–bit streaming data procedure will be used. During the transfer to the slave, the master starts –STB and gates the data onto the data and address buses. During a transfer to the master, the master tri–states the address bus after activating –CMD, and, after the trailing

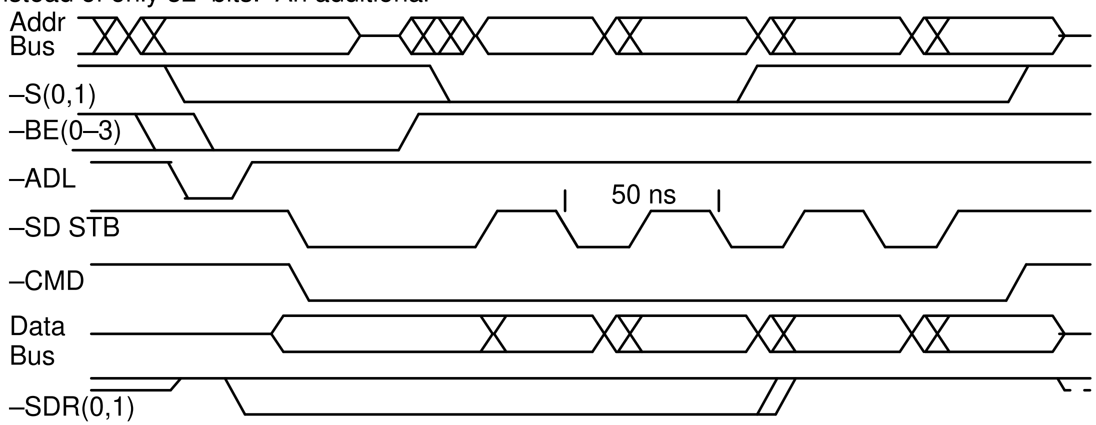


Figure 4 64–bit 20 MHz Streaming Data Overview

edge of –BE(0–3), the slave gates the data onto the data and address buses.

If 20 MHz streaming data is supported by both the master and the slave, both must execute at least 4 streaming data ”data cycles”. The master indicates 20 MHz streaming data transfers by driving the second Status line active. Figure 4 illustrates the 20 MHz streaming data procedure with 64–bit transfers. 20 MHz streaming data is currently not supported in the RS/6000 Systems.

Though a slave may delay the start of streaming data transfers using the CD_CHRDY signal, data pacing is not supported during 64–bit streaming data operations, and is not supported during 20 MHz streaming data operations.

Address and Data Parity

Address and Data Parity are provided to improve error detection characteristics of the Micro Channel architecture. These mechanisms are particularly well suited to detecting typical errors such as card–seating problems and electromagnetic interference. Address parity and data parity support are optional and independent, and mixtures of parity and non–parity devices are permitted.

Address parity is controlled by a signal driven by the bus master called address parity enable (–APAREN). Normally, a slave device responds to a valid address by activating the card selected feedback (–CD_SFDBK) signal. If the –APAREN signal is active, all slave devices supporting address parity check address parity and, if incorrect, activate the channel check (–CHCK) signal as described in ”Synchronous Exception Signaling.” A slave should not become selected or activate the –CD SFDBK signal if address parity is incorrect and should not set any internal error status as the error may not have affected it. The bus master is then expected to suspend operations, set internal error status indicating that a –CHCK occurred with no –SFDRKRTN signal, and interrupt the system.

Data parity checking is controlled by the data parity enable (–DPAREN) signal. If a parity error is detected during a write operation, and the –DPAREN signal is active, slave devices supporting data parity activate the –CHCK signal as described in ”Synchronous Exception Signaling”. If the bus master detects a read data parity error and the –DPAREN signal is active, it is expected to suspend operations, set internal error status indicating that a read parity error occurred, and interrupt the system.

Synchronous Exception Signaling

The Micro Channel architecture provides for exception signaling with the –CHCK signal. Initial use of the –CHCK signal was asynchronous in the Personal System/2 products. The use of the –CHCK signal was enhanced to support the signaling of synchronous events such as page faults, write parity errors, and command queue overflows. Synchronous exception signaling simply allows the pulsing of the –CHCK signal in the same bus cycle that caused the exception. This feature is key to meeting system data integrity and error recovery requirements of the RS/6000 system.

Conclusion

Three features of the Micro Channel architecture have been discussed. These features: increasing the ”peak” bandwidth capabilities first from 20M bytes/sec to 40M bytes/sec, then to 80M bytes/sec, and most recently to 160M bytes/sec; addition of parity checking as an optional feature; and support of synchronous exception signaling, broaden the characteristics of that bus to support advanced I/0 devices or systems. These features retain compatibility with existing card and planar board designs. These features are implemented in the RS/6000 family of products and are key factors in meeting the performance, data integrity, and error recovery goals of that system.

Acknowledgments

The Micro Channel architecture was jointly developed with Boca Raton architecture and various product development groups worldwide. B.M. Bass, D.M. Desai, J.K. Langgood, S. Dhawan, R.A. Kelley, L.F. McDermott, D.M. Neal, J.O. Nicholson, J.D. Reid, F.E. Strietelmeier, J.D. Touchton, R.K. Arimilli, D.W. Siegel, and L.P. Vergari provided substantial contribution to the Micro Channel features described in this article and their contributions are appreciated.

References

1. IBM Personal System/2 Hardware Interface Technical Reference – Architectures Manual, IBM Corporation.

The PAX 2D Rendering Engine

Charles R. Johns and Taggart Robertson

Introduction

Powerful processors and window managers have enabled application writers to exploit the power of graphics. This power has created increased pressure on workstation suppliers to provide high–performance graphics solutions to the entry level market segment. The POWERstation/POWERserver 250 series workstations along with the POWER GXT100 and GXT150 adapters provide such a solution.

The POWER GXT100 and GXT150 are 2D graphics adapters based on the Pixel Accelerator for X (PAX) 2D rendering engine and the IBM RGB 530 Palette Digital to Analog Converter (Palette–DAC). The PAX engine draws or renders the graphical data sent from the 601 processor complex and the RGB 530 displays the data on the screen. The GXT100 is an entry–level adapter with memory to support a 1024 x 768 screen. The GXT150 has added memory to support up to a 1280 x 1024 screen and to utilize the multiple color palettes in the IBM RGB 530 Palette–DAC. Both the GXT100 and the GXT150 are 8–bit adapters which require AIXwindows Environment/6000 Version 1.2.5 2D and AIX Version 3.2.5.

This article presents the design goals, a brief overview of the GXT100 and GXT150 graphics subsystem organization, a detailed look at the programming models, and the architecture of the PAX 2D rendering engine. This article describes the features and functions of the PAX rendering engine. All of these features and functions may not be exploited by the graphics subsystem.

Design Goals and Considerations

The major goal of the PAX 2D rendering engine is to address the cost–performance requirements of the entry workstation market segment. Since X Window System is the predominant Graphical User Interface (GUI) on IBM workstations in this market segment, the architecture is dedicated to accelerating X Window applications. This goal, along with the considerations of time to market and cost, drove the architecture and design of the PAX 2D rendering engine.

The architecture of a graphics subsystem can influence X Window performance and time to market in several areas. The PAX architecture targeted three areas: efficient interfaces, simple programming model, and rendering speed. Details of how each area is addressed appear in the following sections.

Graphics Subsystem Organization

The POWER GXT100 and GXT150 consists of the PAX 2D rendering engine, the IBM RGB 530 Palette–DAC, 1M to 3M bytes of frame buffer memory, and the initialization Read Only Memory (ROM). Figure 1 shows the complete block diagram and the major interfaces of the graphics subsystem. The PAX chip serves as the graphics accelerator as well as the system interface for the adapter. It contains the control for accessing the initialization ROM, the Palette–DAC control port, and the video RAM (VRAM) parallel port.

PAX attaches directly to the PowerPC 601 Microprocessor bus. Attaching directly to the PowerPC 601 bus improves performance by avoiding the latency and synchronization overhead created when converting from one bus to another. The 601 bus is a split address and data bus with 32 bits of address and 64 bits of data [1]. The processor uses byte, half–word, or word operations to access the graphics subsystem. Based on the address of the operation, PAX directs the access to one of four locations: the internal registers, the frame buffer (i.e. the VRAM), the RGB 530 Palette–DAC, or the initialization ROM.

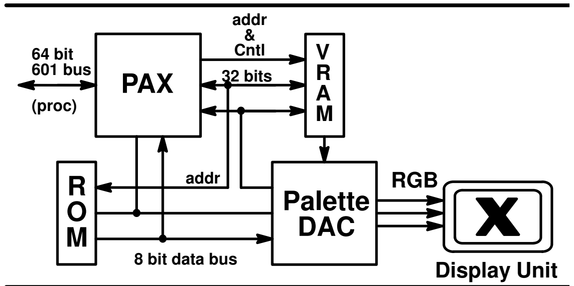


Figure 1 Graphics subsystem block diagram

The frame buffer is constructed of specialized memory called video RAM (VRAM). These memory devices contain key features for increasing graphics performance including:

- Serial output port
- Block Write
- Write per bit

The serial output port is a dedicated high speed interface used by the Palette–DAC for scanning pixels out of the frame buffer VRAM. This interface eliminates the overhead of refreshing the screen from the parallel port which is often required with more conventional DRAM interfaces.

Block Write is a unique feature of VRAM which allows a constant color to be written to multiple locations within a single write cycle. This feature allows PAX to render up to 32 pixels in a single write cycle.

The Write per bit feature provides a write mask which the rendering engine uses to select which bits of the pixel are to be updated. This feature eliminates costly read / modify / write operations.

PAX's 32–bit frame buffer architecture exploits these VRAM features, as well as employs advanced interleaving techniques referred to as Pixel Interleaving [2] and Load Clock Interleaving to enhance performance. The architecture supports 1M to 5M bytes of memory which provides the capability to support screen resolutions ranging from 1024 x 768 to 1280 x 1024. With 2M bytes or more of memory, PAX can support a double buffer display.

The IBM RGB 530 Palette–DAC serves as the video controller. It provides four color palettes, video output (red, green, blue), display timings, VRAM serial port control, and two hardware cursors. This device also provides an on–chip programmable Phase Locked Loop (PLL) which allows the graphics subsystem to support a wide range of monitors with varying timing requirements.

As the block diagram and descriptions illustrate, most of the logic required for the graphics subsystem is integrated into the PAX and the IBM RGB 530 custom chips. This level of integration is key to keeping the cost of the subsystem to a minimum while enhancing the performance and function.

Programming Models

The PAX chip supports two different programming models: Direct Frame Buffer Access (DFA) and "Poly" commands.

Direct Frame Buffer Access allows the Frame Buffer to be accessed by the PowerPC 601 processor as if it were part of system memory. This simplified interface is very effective in reducing the X Window software development time which directly correlates to time to market. X Window software development uses DFA to directly map the X Window System code received from Massachusetts Institure of Technology (MIT) into the initial device driver for the adapter. Then, X Window software development optimizes the device driver to exploit the hardware accelerated functions. DFA is also key to the performance of certain X Window commands such as points, circles, and complex area fills because there is no additional hardware acceleration provided for these primitives.

The "Poly" command interface provides a rich set of rendering instructions which complement the DFA programming model. The word "Poly" refers to the ability to render multiple primitives of the same type with only one command. The ability to render multiple primitives with only one command eliminates the overhead of sending a new command with each primitive. The command set is designed to map closely to the X Window protocol. The performance of accelerated functions using the "Poly" command interface is significantly faster than DFA. Some of these accelerated functions include line draw, area fill, and bit block transfer.

Rendering Engine Architecture

The PAX rendering engine architecture includes rendering functions, rendering attributes, and 3D Application Programming Interface (API) assist functions.

<u>Rendering Functions</u>

The PAX architecture consists of several processing units for accelerating X Window System rendering commands. These processing units are used to accelerate:

- Line Draw
- Points
- Area Fill
- Bit Block Transfer

See Figure 2 for a block diagram of the internal architecture. These processing units provide the X Window server (X server) with dedicated hardware to accelerate the rendering of lines, points, and area fills. In addition to rendering, PAX also provides assistance for moving pixels to and from system memory or to and from another screen location. This function is referred to as Bit Block Transfer (Blits).

Vertices define the boundary of the region to be rendered. These vertices are included in "Poly" commands. All vertices for these drawing commands can be sent as 16–bit, two's complement, window relative coordinates. This 16–bit vertex provides the X server with a 64 K x 64 K virtual screen.

The origin of this virtual screen space is at the center, thus allowing for both positive and negative X and Y addresses.

Line Draw
The line draw engine provides the rasterization of a line between two points provided by an application. A technique known as Bresenham's line algorithm [3] is used to render the line. PAX supports two commands for rendering lines: Poly Line and Poly Segment.

Every vertex sent with the Poly Line command defines a new line. This is useful for drawing connected lines. The Poly Segment command requires two vertices to define a line. This command is used to render multiple non–connecting lines.

The X Window protocol supports styled lines. These are referred to as OnOffDashed and DoubleDashed lines [4]. An OnOffDashed line appears as a line with sections, or dashes, rendered in the foreground color and sections not rendered. DoubleDashed lines appear as a line with sections rendered in the foreground color and sections rendered in the background color.

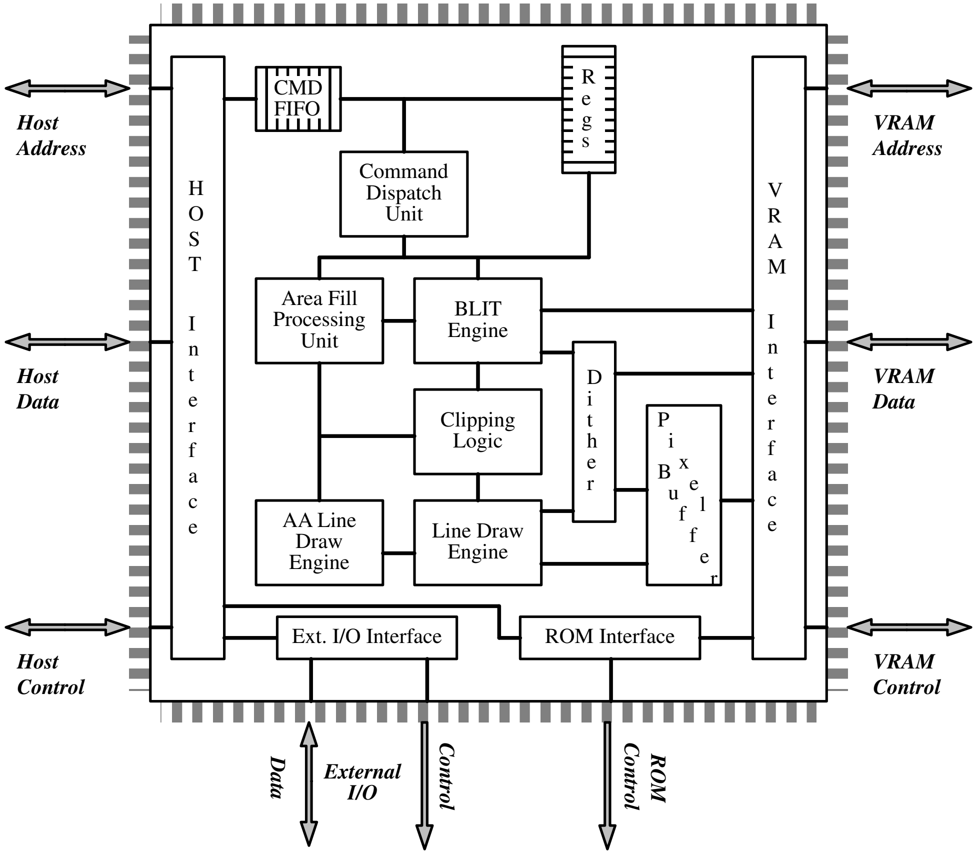


Figure 2 Rendering Engine Block Diagram.

PAX supports a set of dash counters which allow the server to define a line style with up to 8 unique segments or dashes. The X server selects between Dashed and Double Dashed by enabling transparent rendering.

The major performance bottleneck for the line generation logic is the frame buffer. As mentioned earlier, PAX employs Pixel Interleaving and Load Clock Interleaving to reduce this bottleneck.

Points
Points are rendered using either DFA or "Poly" point commands. Points are not accelerated since there is no significant processing required to generate them. Instead, points are provided a fast path through the hardware so that they are rendered into the frame buffer at the maximum frame buffer bandwidth.

Area Fill
PAX supports four types of area fills: Spans, Triangles, Rectangles, and Quadrilaterals. A unique command is dedicated to each type of area fill operation to reduce the number of stores required. The Block Write VRAM feature is used to increase the fill rate for these objects. The Block Write feature allows PAX to write up to 32 pixels in one memory cycle – four times faster than normal writes.

Spans are a continuous row of pixels. These are the basic area fill primitives. All other types of area fills are broken down into spans by the internal area fill logic.

The triangle and quadrilateral fill logic uses the line logic and another simpler line generator to find the edges of the area. Since only two line generators (edge walkers) are available, PAX can only support quadrilaterals which are convex in the Y direction. See Figure 3 for examples of the supported and unsupported areas.

Rectangles are a special case of a quadrilateral and are handled separately. Special casing rectangles provides additional performance since the overhead of walking the edges is eliminated. This extra performance is beneficial to window management and clearing areas on the screen.

Quadrilaterals, rectangles, and triangles can be drawn in one of two modes: X Window compliant or Full Fill. The X Window mode draws the fill area such that when an object is connected to other objects along an edge, no pixel is written twice. This is accomplished by rendering pixels whose center lies inside the area, on the left or top edge, but not pixels on the right and bottom edges. Full Fill mode draws all pixels including the edges.

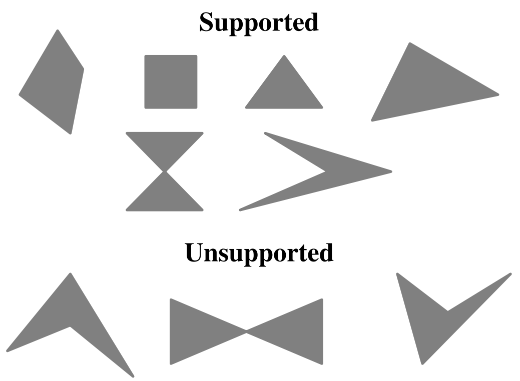


Figure 3 Area Fill Examples

Bit Block Transfer

Bit Block Transfers (Blits) are a hardware assist for the movement of blocks of pixel data. PAX supports three types of Blits: Screen to Screen, System to Screen, and Screen to System. They are used extensively by the X server to accelerate the movement of windows.

Screen to Screen Blits are used to copy a block of pixels from one location on the screen to another. PAX automatically handles overlapping source and destination blocks so that the source block appears correctly at the destination.

Screen to Screen Blits continuously switch between frame buffer reads and writes. These transitions drastically reduce the usable frame buffer bandwidth. PAX has an internal buffer to reduce the number of transitions which increases the utilization of the frame buffer's bandwidth.

System to Screen and Screen to System Blits are used to copy pixels between system memory and the frame buffer. There are two modes of operation for these commands: Direct and Indirect. In Direct mode, system software controls the pixel transfer from system memory and PAX only controls the frame buffer address. In Indirect mode, the PAX chip becomes a master of the PowerPC bus and completes the transfer of data with no processor intervention. The software in this case only supplies the source and destination addresses.

PAX supports a special System to Screen Blit mode which accelerates character performance. When operating in this mode, each bit of the data sent is interpreted as a pixel. The hardware renders the foreground color for all the **1**s in the data word. The background color is rendered for all the **0**s in the data word if transparency is disabled and nothing is rendered if transparency is enabled. With transparency enabled, this function uses the Block Write VRAM feature.

<u>Rendering Attributes</u>

PAX also supports a variety of rendering attributes. These attributes can be applied to both the DFA and "Poly" command programming models. Some of these attributes are listed below.

- Boolean operations
- Window Management
- Stipple

These attributes modify and control the rendering functions by modifying vertices and pixel generation. Different attributes apply at different stages in the rendering process.

Boolean Operations

The X Window protocol specifies the rendered pixel to be a combination of the pixel's current color (destination) and the source color. One of 16 logical functions can be selected, all of which are supported by the PAX architecture. Below is a list of the logical functions.

- Clear (0)
- Destination (D) (NoOp)
- !S
- S & D
- S & !D
- !S & D
- !S & !D
- S^D
- Set (1)
- Source (S) (Copy)
- !D
- S | D
- S | !D
- !S | D
- !S | !D
- !(S^D)

By implementing Boolean operations in hardware, the X server is relieved of the slow task of reading the frame buffer, modifying the color, and writing the new color to the frame buffer. This function is extremely important to applications which require Boolean operations.

Window Management

Applications running under the X Window System request a window from the X server. The user is free to resize and move these windows at any time. Since multiple windows may be open on the screen, there is a possibility a window may be partially obscured by others. It is the X server's responsibility to manage these windows so that pixels rendered to the obscured sections are not visible. PAX supports the following functions which assist the X server:

- Rectangular Clippers
- Clipping Planes
- Window Origin Offset
- Window ID planes

The Rectangular Clipping logic consists of four pairs of extent registers. Each pair defines a rectangular region with either an inclusive (all pixels inside the region are rendered) or exclusive (all pixels inside the region are NOT rendered) attribute. The X server uses these registers to define the window's geometry to the PAX chip so that pixels in the obscured sections of the window are clipped.

Four regions are not always enough to define a window's geometry. Such is the case when the window is obscured in four or more unique areas or when shaped (i.e. non rectangular) windows are used. For these cases, the X server may render the window's geometry to off screen memory, referred to as clipping planes, with a unique ID. PAX then reads the pixel's corresponding clip plane value and compares it with the clipping ID to determine if the pixel should be written to the frame buffer.

Applications draw using a window relative coordinate system. Vertices in this system must be converted to screen coordinates before they can be used to render an object. This conversion is accomplished, in hardware, by adding the Window Origin Offset to every pixel before it is rendered. Providing this capability in the PAX chip eliminates the task of converting the coordinates in software which increases the rate at which the vertices are sent to PAX. Overall, adding the window origin offset increases the performance of processor bound primitives such as points.

Some applications require a different color palette than the default. If only one palette is available, the colors of other windows will change when focused on these types of applications. The PAX chip supports an additional four planes of memory, referred to as Window ID planes, which allow the X server to select a unique palette on a per pixel basis. These planes also identify other attributes of the pixels such as: frame buffer select, pixel type, and overlay plane enable.

Stipple

The X Window protocol allows a fill pattern, such as a checkerboard, to be applied to the objects rendered [4]. This pattern is referred to as the Stipple pattern. The pattern can be transparent or opaque. Transparent patterns result in the foreground color being rendered where there are **1**s in the pattern and nothing rendered where there are **0**s in the pattern. The only difference for an opaque pattern is that the background color is rendered where there are **0**s in the pattern.

PAX supports a fixed 16 x 16 Stipple pattern. This pattern is addressed by the four least significant bits of the window coordinate of the pixel to be rendered. The value at that location is the stipple value for that pixel. PAX supports stippling for all rendering operations.

Applying a transparent stipple pattern to an area fill operation does not effect the ability to use the Block Write function. However, applying an opaque stipple prevents the use of Block Write since two colors must be rendered (foreground and background). This normally reduces the performance of large opaque stippled objects by a factor of approximately four. To prevent such a drastic drop in performance, the PAX architecture employs a unique feature called Stipple Invert. This feature allows an opaque stippled object to be rendered twice using Block Write. The second time it is rendered, the stipple pattern is inverted and the foreground and background colors are swapped. This simple feature almost doubles the performance for opaque stippled objects.

<u>3D API Assist Functions</u>

PAX supports a few functions which are intended to enhance the performance of 3D Application Programming Interfaces (APIs). These additional functions are listed below:

- Anti–Aliased Lines
- Sub pixel positioning of lines
- 24–bit RGB to 8–bit RGB Dither

The Anti–Aliased line draw function provides the system software with the ability to render lines without the familiar problem of ”stair steps” or ”jaggies” (i.e. aliasing). This function uses a proprietary two pixel approximation technique to visually remove the aliasing caused by the discrete pixels on a raster display.

Sub pixel positioning of lines allows the system software to more precisely place the anti–aliased lines on the display.

Dithering is a technique which trades spatial resolution for more color resolution. Essentially a 24–bit color value is converted to an 8–bit value and then slightly modified based on the pixel's position in the window. The overall appearance is that the graphics subsystem has more than 256 colors.

Conclusion

The PAX 2D rendering engine provides the POWER GXT100 and GXT150 graphics subsystem with an efficient, simple, high speed rendering engine to accelerate the X Window System. Attaching PAX directly to the PowerPC processor bus provides a low latency, high performance interface for the efficient transfer of data. The DFA and "Poly" command programming models are key to reducing the time to market for the POWER GXT100 GXT150 graphics adapters and X Window system software. Optimization of the programming models for the X Window protocol and dedicated hardware for rendering graphics primitives provides the foundation for a implementation of a high speed X server. The advanced frame buffer design, exploitation of the advanced VRAM features, and the proprietary rendering algorithms are instrumental in achieving the rendering speeds. The high performance rendering, the optimized programming interfaces, and direct attachment to the PowerPC 601 bus allow the POWER GXT100 and GXT150 graphics adapters and the AIXwindows software to provide a high performance, low cost 2D graphics subsystem for the entry–level POWERstation/POWERserver 250 workstations.

Acknowledgments

Teamwork and dedication were required of many individuals to create the graphics subsystem and rendering engine described in this paper. Credit is given to Bill Lawless for driving the development of the architecture. Below is a list of the individuals and one facet of their contribution. Thanks is given to: Mark Baur, Mike Brady, Linh Bui, Daryl Kokoszka, Chris LeBlanc, Chi Lee, Mickey McInnis, Richard Mitchell, Thomas Powell, Avijit Saha, and Jason Young for their contributions to the development of the PAX 2D rendering engine; Jim Daniels, David Dick, David Jeske, Kaveh Massoudian, and John Spannaus for their contributions to the design of the POWER GXT100 and GXT150 graphics adapters; Gary Anderson, Po Cheung, Debi Kipping, Sid Manning, Dac Nguyen, Mark Nutter and Tim White for their contributions to the development of AIXwindows. Thanks is given to all the above, Larry Henson, Chris Nelson, and John Roberson for their contributions to the architecture of the PAX 2D rendering engine.

The authors would like to thank Susan White–Robertson, John Reysa, Steve White, and the many others who reviewed this paper.

References

1. *PowerPC 601 RISC Microprocessor User's Manual*, Motorola Inc., 1993.
2. *Brooktree Product Databook*, Brooktree Corporation, 1991.
3. Foley, vanDam, Feiner, and Hughes, *Computer Graphics Principles and Practice*, second edition.
4. X Protocol Reference Manual, O'Reilly & Associates Inc. May 1990.

Software OpenGL: Architecture and Implementation

Chandrasekhar Narayanaswami, Daniel Brokenshire, Bruce D'Amora, Suzy Deffeyes, Gordon Fossum, James Miller, Barry Minor, Christopher Mussack, John Spitzer, and LInas Vepstas

Due to improved processor architectures and higher clock rates, recent workstation performance has doubled approximately every 18 months. Meanwhile, the price of these machines has continued to decrease. The decreasing cost of computational power has opened doors to many applications which had been considered too computationally intensive. One such case is entry three–dimensional (3D) graphics – an area previously dominated by hardware accelerated graphics adapters. Graphics algorithms, which were previously enabled in dedicated hardware, can now be implemented in a high–level language on the host processor, with an attractive price–performance ratio. This enables 3D graphics application programming interfaces (APIs) such as OpenGL [1], PEX [2], and graPHIGS [3] to perform well on low–cost machines without the addition of costly 3D graphics adapters.

The software approach has many short–term and long–term benefits:

- Complete graphics functionality is immediately available over a wide range of products, from the low end to the high end on the POWER, POWER2 and PowerPC implementations.
- Graphics performance is directly proportional to processor performance and will continue to increase as processors improve.
- Implementation in high–level programming languages on the host processor enables quick response to changes and enhancements in graphics APIs. For example, software implementation, testing, and maintenance are simpler for general purpose processors than for custom processors (ASICs). In addition, development on general purpose processors generally includes use of better debuggers and several other productivity enhancement tools.
- The software solution also inherits powerful features of a computer system such as virtual memory, pointers, fast context switching, and direct access to main memory. As a result, faster responses to graphics queries are possible with this approach than with heavily pipelined outboard graphics accelerators. This advantage is significant; users are becoming increasingly aware that fast graphical rendering is not the only important aspect of graphics applications.

The following sections detail the design and implementation points which were used to fully optimize OpenGL. The rendering software (renderer) described in this article was implemented within AIXwindows 1.2.5 (X Window System [4] under AIX). Software renderers for graPHIGS and PEX also exist in AIXwindows but are beyond the scope of this article.

OpenGL: Introduction

OpenGL is a network transparent 3D graphics rendering interface. Users can generate high quality pictures from user defined graphical objects through a simple low–level modular interface.

The OpenGL API is licensed from Silicon Graphics, Inc. and is governed by the OpenGL Architecture Review Board. Current members of the board include IBM, Intel, Microsoft, Digital Equipment Corporation, and Silicon Graphics. The similarity between OpenGL and IRIS Graphics Library, a proprietary programming standard, bodes well for the success of OpenGL as an open 3D programming standard.

OpenGL supports 3D transformation, clipping and culling, Phong lighting, as well as Gouraud shaded, texture–mapped, anti–aliased point, line, and surface primitives. Utilization of stencil and accumulation buffers enables the efficient implementation of constructive solid geometry, shadows, full scene anti–aliasing, and motion blur effects [1].

In a typical scenario (see Figure 1), the OpenGL application must first create a **drawable** (window or off–screen image) and an OpenGL rendering context. This context must then be associated with the drawable. The required graphics buffers (depth buffer, alpha buffer, stencil buffer, and accumulation buffer) are created when a context is first associated with a drawable. The application may then issue OpenGL commands to be interpreted and drawn into the drawable. A detailed specification of OpenGL exists [1] and will not be discussed here.

Architecture Overview

To facilitate development work and to allow future optimization and expansion, the complete system was segmented into three well–defined components, the window system component, the geometry pipeline, and the rasterizer, as shown in Figure 1.

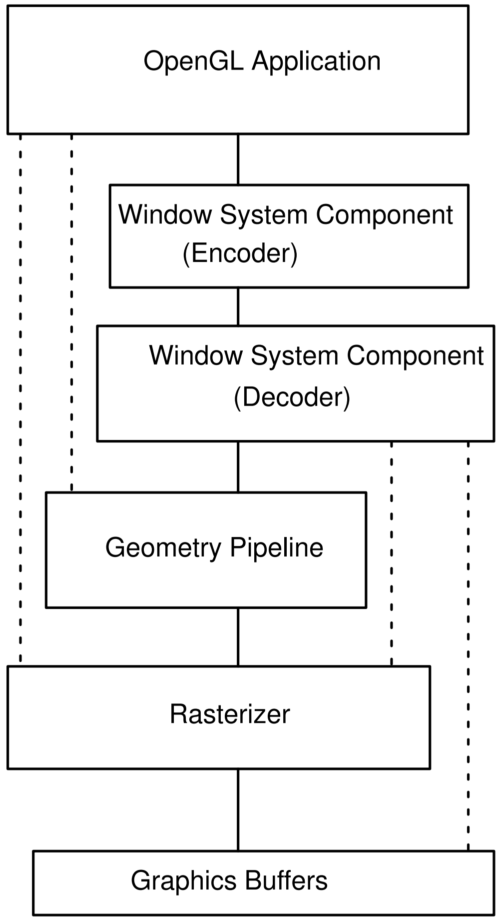


Figure 1: Components of OpenGL. Includes an encoder/decoder, a geometry pipeline, and a rasterizer. Lines connecting components indicate bidirectional communication links. Dashed lines indicate communication links that may not be currently enabled.

Window system dependent component

This component encodes commands from the application and sends them (across the network if necessary) to the renderer. It also performs operations upon windows that will host OpenGL rendering (creation, movement, resizing, clipping and destruction). Management of rendering contexts is performed in this component as well. OpenGL does not explicitly handle user input and interaction, but relies upon the window system to do so.

Geometry pipeline

This component accepts primitives such as vertex coordinates, colors, normal vectors, and texture coordinates. It then performs a subset of the following operations depending upon the current OpenGL state: transformation, clipping, culling, lighting, fog calculation, and texture coordinate generation. The processed primitives are then passed to the rasterizer. In addition, this component manages state variables such as light and material properties.

Rasterizer

This component accepts primitives (from the geometry pipeline) such as points, lines, and triangles and computes which pixels they cover on the drawable. The generated pixels, called fragments in OpenGL, are then processed according to the current OpenGL state. These operations may include stippling, scissoring, texture mapping, fogging, anti–aliasing, alpha testing, stencil testing and updating, depth testing and updating, blending, logical operation, masking, and frame–buffer writing. The output of the rasterizer goes to the graphics buffers shown in Figure 1. In addition, this component manages all state variables pertaining to rasterization.

The partitioning described above makes it easier to migrate from the X Window System to other window systems (such as Presentation Manager), to build a wide range of graphics accelerators which may implement a subset of the above three components, and to port to multiprocessor systems. The partitioning also allows concurrent software development to proceed in groups across multiple sites. Minimizing component interaction ensures low overhead, resulting in higher performance. Each of the components will now be discussed in greater detail.

Window System Dependent Component

This component has two subcomponents as shown in Figure 1: one in the OpenGL application side (Encoder), the other in the OpenGL extension or library (Decoder). Note that in the current implementation (OpenGL 1.0.0 on AIX 3.2.5) the case where the application communicates directly with the geometry pipeline is not enabled. Several steps were taken to ensure reasonable performance while retaining the advantages of network independence. These steps are described below. The Encoder accepts each application command and creates a command packet which includes an opcode for the command followed by the data. The fast and extensive string manipulation features [5] of the POWER, POWER2, and PowerPC implementations were exploited for encoding and moving data. The command packets are stored in command buffers. These buffers are then sent to the Decoder whenever the user explicitly makes a request or when the buffer is full.

Minimizing communication overhead between the Encoder and Decoder is crucial. To address this issue, shared memory transport, which is faster than the standard socket transport, is used when the Encoder and the Decoder are on the same machine. The Decoder accepts the command packets and stores them into command buffers. These command buffers are then processed in a round–robin fashion. The Decoder is capable of ensuring fairness across clients, so that no client dominates the processor with long sequences of time–consuming atomic commands. The Decoder also manages the graphics buffers, contexts, and shared display lists (a set of OpenGL commands which are stored and executed as a single unit). Other commands, such as those received from the mouse and keyboard input, are handled by the core window system mechanisms such as those in X [4]. In the current implementation, all graphics buffers are in system memory. Thus, the color buffer in system memory is copied to the frame buffer whenever necessary. Keeping the color buffers in system memory allows us to support OpenGL on most AIX–based POWER, POWER2, and PowerPC workstations.

Geometry Pipeline

The geometry pipeline accepts vertex data from the application (only via the Encoder and Decoder in the current implementation) and stores it in local buffers. The vertices in the buffer are then transformed and processed before being sent to the rasterizer.

The following requirements influenced the overall design and organization of this component:

- Fast switching among various rendering contexts in the client/server model.
- Window system independence for maximum component reuse.
- Geometry pipeline access from either the Decoder or the application directly; allows the application to execute in a client/server or a direct rendering mode [1].
- Optimization based upon the current rendering state.
- Connection of the geometry pipeline component to rasterizers with varying degrees of hardware support. In some situations, the hardware renderer may have insufficient functionality and may require a switch to a conformant software renderer.
- Ability to switch quickly between rendering modes: immediate mode, feedback mode, selection mode, and display list compilation and execution mode.
- Flexibility to accommodate future OpenGL enhancements.

These criteria led to a rendering function hierarchy consisting of procedure tables for internal and external interfaces which satisfied the requirements without sacrificing performance.

At the highest level, the external interface to the pipeline is a procedure table attached to the graphics context. Besides context management functions, the only interface into the pipeline is through this procedure table. There is a mapping between OpenGL rendering calls and entries in the procedure table. The Decoder dispatches rendering calls through this table. Since the table is anchored to the rendering context, it is automatically swapped when a context switch occurs. This table is explicitly changed when the rendering mode changes. The ability to change tables enables trace facilities, error checking, and rendering mode checking with little or no cost to the user. Because the state is implicit in the procedure table used, no extra branches are required in each call.

Internal function calls are handled similarly. A **methods** table consists of pointers to internal functions. These pointers are switched when state changes are made. An example of a pointer would be a **methods** table element which is responsible for lighting. Often optimizations can be made for lighting calculations — this makes some calculations far less computationally intensive than the general calculations. If the current state dictates the use of the cheaper function, the lighting entry will be set appropriately instead of pointing to the general function.

Three critical considerations in our design from the standpoint of the processor were:

1) In several situations, the POWER architecture can achieve zero–cycle branch penalties for floating–point compares. Graphics algorithms have several branches and the above property is very useful. However, some sequences of code such as a compare followed by a conditional branch which is further followed by a unconditional branch could cause as much as a 10–cycle delay in the floating–point unit (FPU) and a 7–cycle delay in the fixed–point unit (FXU) [6]. Reordering code allows the compiler to schedule compares well in advance of the branch, which can reduce this penalty.

2) Our target system has a fast on–chip cache. To make optimal use of the cache, the processing–intensive areas or **hot spots** of the geometry pipeline code should be small enough to fit inside the cache. To minimize instruction cache misses, each of the pipeline operations is performed upon a buffered set of vertices before advancing to the next operation in the geometry pipeline. In a processor with a unified data/instruction cache, this solution offers more room for vertex data at the cost of a small amount of loop overhead.

3) Deviating from the logical sequence of instructions required to implement a given algorithm can greatly increase performance by removing pipeline bubbles so as to increase the utilization of the processor's multiple computation units. Instruction scheduling within a single computation unit is also critical [6] since the FXU [7,8] and FPU are each pipelined. These factors altered the code sequences in several situations (e.g., the transformation phase).

Now we discuss the buffering, transformation, and validation phases of the geometry pipeline.

Vertex Buffering

This phase accepts vertices received from the Decoder and stores them into the vertex buffer along with the current color, normal vector, and texture coordinates.

Such buffering allows amortization of function call and other overheads over several vertices, thereby improving overall performance. Buffering the vertex and color data in contiguous memory locations reduces data stride and therefore improves data cache utilization. The buffer is passed to the transformation stage when the primitive is finished or the buffer is full.

An important consideration is how many vertices should be buffered before they are actually processed. Rendering could be performed as soon as enough vertices to define the geometric primitive are collected (e.g., after every vertex for points, after two vertices for lines, after three for triangles). Instead, vertices for an entire OpenGL primitive are buffered (up to a maximum of 256). The maximum vertex buffer size was chosen to be 256 vertices since many IRIS Graphics Library applications use that as a limit. (This limit for each primitive aggregate originates from early Silicon Graphics hardware.)

Transformation

This phase transforms the coordinates of the vertices in the vertex buffer from the application–defined coordinate system to the normalized–device coordinate system.

The transformation portion of the pipeline demands a large percentage of processor cycles when processing high–bandwidth primitives such as lines and points. This code is a prime candidate for optimization. This section will describe the different parts of the transformation process and show how it has been specifically tuned for the POWER architecture.

The transformation code performs three major functions:

- Modeling/projection matrix transformation of each vertex.
- Clip code generation and storage for each vertex.
- Normalized device coordinate (NDC) calculation and storage for each vertex.

These three functions exploit superscalar implementations effectively since they have both integer and floating–point computations. First, all vertices are multiplied by the current modeling/projection matrix to transform them into eye space (also called clipping space). **Clip codes** are then computed using vertex values held in registers. Clip codes determine whether a vertex is inside the viewing region. Such codes are helpful in determining whether primitives are trivially accepted (all vertices of the primitive are in the viewing region), trivially rejected (the primitive is not in the viewing region), or need to be clipped (some part of the primitive is in the viewing region). If the vertex's clip code determines that it is within the window, the NDC values are then computed. This three stage process has been tuned to take advantage of the POWER implementation's superscalar features, single–cycle throughput for floating–point operations, and large register files.

The POWER FPU's ability to execute pipelined floating–point multiply–add (FMA) instructions each cycle and to concurrently perform operand loads is used extensively in the transformation code. The entire 16–element transformation matrix is moved into the FPRs (FPU's register file) and then a tight loop performs the transformation, clip code generation, NDC calculation, and storage for the entire primitive buffer. Many of the cycles which calculate clip codes are overlapped on the FXU during the transformation, resulting in nearly free clip code generation. This is followed by calculation of the NDC coordinates from the intermediate results stored in the FPRs. Lastly, the resulting NDC values and clip codes are stored if needed. Branch logic throughout the entire process is overlapped on the POWER implementation's Instruction Cache Unit (ICU), resulting in zero or single cycle conditional branches and loop control. The zero or single cycle delay is made possible by reordering instructions so that the condition code field of a conditional branch is set far enough before the branch that the BPU can determine the validity of the branch before it is executed, thereby effectively executing it in zero cycles.

The above tuning results in the following approximate cycle counts, for the above three step process, determined from assembler listings for the POWER architecture (thus ignoring cache miss penalties):

Trivially rejected points:
32 cycles/vertex.

Trivially accepted parallel projected points:
38 cycles/vertex.

Trivially accepted perspective projected points:
60 cycles/vertex.

Lines and triangles differ from points in that the concept of trivial rejection is extended to include a deferred clip of the primitive in the next stage of the pipeline. The following are the approximate cycle counts, determined from assembler listings on the POWER architecture, for lines and triangles for this phase of the pipeline:

Primitives that require clipping:
37 cycles/vertex. Trivially accepted parallel projected:
39 cycles/vertex. Trivially accepted perspective projected:
59 cycles/vertex.

Because these performance numbers are for one stage of the complete rendering process, the data should not be extrapolated for rendering performance numbers.

Vertex Buffer Validation

This phase examines the vertex buffer created at the end of the transformation process and determines the set of operations that need to be performed based on the currently specified options. Validation routines contain sections for transforming normal vectors, calculating lighting, calculating fog factors, and generating texture coordinates. The vertices output from the validation routine are then sent to the rasterizer.

Since the validation phase operates on several vertices and makes several passes, the size of the vertex buffer should be such that the buffer stays in cache between the passes. Smaller buffers would flush too frequently and result in high function–pointer and loop–overhead costs. The 256–element buffer satisfied the above requirements.

The first stage of validation determines whether all the vertices in the buffer are trivially rejected. If so, the buffer is discarded. Thus no intensive computations are be performed in this case. If all the vertices are trivially accepted, none of the primitives are clipped and the buffer is passed to the culling stage. If some primitives are clipped, then the buffer is passed to the clipping stage, where new vertices are generated as needed, and then passed to the culling stage. After culling, the vertices are sent to the final stage of validation. The final stage of validation is performed by using a function table that is indexed through the current state. This reduces the number of branches required per vertex.

Lighting is a typical example of the final stage of validation. This phases evaluates the light intensity at the vertices of the objects. Since it is an important and time consuming part of validation, we explain it in greater detail. OpenGL supports up to eight lights with specular, diffuse, and ambient lighting components. Each light can be local or infinite. Local lights can also be specified as spotlights with cones of a specified width. The user can also select two–sided lighting and can independently define front and back material properties. Additionally, attenuation coefficients (constant, linear and quadratic) are defined on a per–light basis. The **methods** table holds the lighting function for the current lighting state. When the vertex buffer is ready to be lit, it is passed to the correct function through one of three lighting function pointers: front–faced lighting, back–faced lighting, or two–sided lighting.

Certain commonly used lighting conditions have fast paths, where redundant or unnecessary calculations are avoided. For example, if an enabled light is located at infinity, then the light's direction is constant across all vertices. Moreover, most lights are not spotlights and that simplifies many of the lighting equations.

For specular lighting and spotlights, a power function is needed. To increase speed, tables of values based on the exponent are used to linearly interpolate between evenly spaced values. These tables represent the slope and intercept of the line that approximates the function between two points. The appropriate tables are calculated when the user sets the specular exponent or the spotlight exponent. An input threshold is first computed. Any input below the threshold will produce a zero output. The range from the threshold to 1.0 is divided into a constant number of segments, so no resolution is lost if large exponents are used. A larger exponent will have a higher threshold, therefore a smaller range and closer interpolation points.

These techniques have reduced the performance penalty usually associated with lighting, especially adding multiple lights.

Similar techniques have been applied to the other stages such as fog factor computation and texture coordinate generation.

Rasterizer

The rasterizer accepts the transformed and validated vertex buffers and determines which pixels in the frame–buffer need to be updated and performs all pixel processing operations. The rasterizer also accepts pertinent state transition commands, but we will not discuss it here, since it is rather straightforward.

The rasterizer has a broad primitive level interface. It has native entry points for points, independent lines, connected lines, line loops, independent triangles, triangle strips, independent quadrilaterals, quadrilateral strips, polygons, bit maps and pixel maps. Three processing steps may be applied to the incoming data:

1) Edge generation. For line and surface primitives, sub–pixel accurate slopes and offsets are generated for color, depth, alpha, fog and texture coordinates.
2) Fragment generation. Edge walking and span interpolation is performed for surface primitives; interpolation along a line is performed directly. The resulting interpolated data is referred to as "fragments" to distinguishing them from pixels, as they are not yet ready to be placed in the frame buffer.
3) Fragment processing. Fragments are converted into pixels by applying stippling, scissor test, texture mapping, fog, anti–aliasing coverage, alpha test, stencil test and update, depth test and update, alpha blending, dithering, logical operations, and masking before the frame buffer contents are updated. The fragment processing also supports rendering into multiple color buffers.

Our implementation of the rasterizer is designed to maximize code reuse, utilize common algorithms, and provide optimum performance. It currently supports three APIs: PEX, graPHIGS, and OpenGL. Common rasterization algorithms are used for all three APIs whenever possible to maintain rendering consistency across all APIs and to reduce development cycles.

The biggest challenge to the rasterizer development is finding an architecture that is compact, yet optimizes performance. The broad primitive interface to the rasterizer and the immense number of possible fragment processor paths (there are at least 31 bits of state, leading to at least 2^{31} possible paths), it is impractical to write a tuned renderer for each path. Writing a single renderer with a conditional branch for each bit of state would have been simple, but would have resulted in poor performance.

A two–pronged approach is used. A renderer was written for each basic primitive, for each combination of interpolant. There are roughly 200 renderers, of which the depth–buffered, true–color, texture–mapped aliased polygon renderer is a typical example. The multiple execution units of the RISC System/6000 implementation [5] were used to maximize interpolation performance by performing depth and texture coordinate interpolation in floating point, and other data in fixed point.

Each renderer comes in one of two varieties. In one case, the fragment processing path deemed most likely to occur was coded and "in–lined" with the renderer. This allows the compiler to schedule instructions in an optimal fashion. For all other fragment processing paths, the renderers place fragments into a fragment buffer, which is handed over to a fragment processing specialist.

As stated earlier, it is impractical to write code for each of the fragment processors. So code for a particular fragment processor is assembled dynamically, when needed.

Assembly code snippets are written for each stage of the fragment processing pipeline. As an OpenGL application changes the state of the rasterizer, appropriate snippets are chained together to create a custom fragment processor. The resulting fragment processor is flushed from the data cache (where it was built) into system memory, where it is invoked as a call through a function pointer. Thus, any previous fragment processor is flushed from the instruction cache automatically. To avoid repeated regeneration of fragment processing specialists, the cache of the most recently used specialists is maintained.

Wherever possible, the fragment processing pipeline stages are reordered so that expensive stages occur after all testing stages. For example, texture mapping and fogging can generally be performed after the testing stages. By doing so, fragments that are not visible do not get texture mapped or fogged. Commonly executed fragment processor code is designed so that variables which are constant for all fragments (for example, buffer pointers and reference values) are loaded into registers once for an entire fragment buffer. The POWER architecture's reasonably large register file helps avoid register spillage in the fragment specialists in all but some cases where sophisticated texture filters are used.

Buffer clearing (including color, depth, stencil, and accumulation buffer) is significantly accelerated by utilizing the cache instructions provided by the POWER architecture. When clearing a buffer to zero, an entire cache line/block is cleared in a single instruction. When clearing a buffer to a nonzero value, the cache line is cleared to zero using these same instructions, then the nonzero values are written into the cache line. Clearing the cache line before writing eliminates the cache loading overhead. The writing of nonzero values is further improved by using 64–bit double–precision floating–point stores.

Improving Application Performance

To obtain better performance, the application developer can adopt several techniques, including:

1. Judicious use of display lists so that the communication overhead between the Encoder and Decoder is minimized.

2. Judicious choice of rendering primitives based on the underlying architecture. The cost of communication overhead and computation speed must be considered. If communication is fast and the architecture has significant parallelism, using primitives with few vertices will increase overall efficiency. On the other hand, if communication overhead is large and rendering is slow, using primitives with many vertices is more effective.

   Similarly, using the appropriate option for rendering primitives improves performance. For example, if the set of polygons to be drawn are all triangles, rendering them using the specific OpenGL disjoint triangle option (treats several disjoint triangles as one primitive) provides better performance than rendering them each as an independent polygon.

3. Avoiding redundant state transitions. For example, modification of rasterizer state in feedback and selection mode may not be necessary. Also, inappropriate application of object–oriented concepts to graphics rendering can result in redundant state changing commands. For example, it is common to define a ”rendering method” for an object that sends down several state changing commands for each object. Many of these state change requests will be redundant when rendering several objects that possess similar attributes.

4. Positioning lights at infinity if that satisfies the application's requirements.

Several others tips are provided in [1].

Conclusions

This article described some of the salient features of the current software implementation of OpenGL on IBM's workstations based on the POWER, POWER2 and PowerPC family of processors. Several optimizations have been made to improve speed while retaining sufficient flexibility to provide even faster solutions in the future.

As parallel systems, such as symmetric multiprocessor systems and workstation clusters, become increasingly powerful and available [9], general software solutions will be even more interesting.

Acknowledgments

We would like to thank a few people who have significantly contributed to the development of OpenGL for the POWER, POWER2, and PowerPC systems. They are: Abraham Mammen, for high level design of the rasterizer, Brian Horton and John Readey for their contribution to the rasterizer; and Bill Luken and John Gerth of IBM Research for their valuable input. Several others have also contributed their time and ideas to the effort.

References

1. OpenGL Architecture Review Board, ”OpenGL Reference Manual,” Addison–Wesley, 19928.

2. R. J. Rost, J. D. Friedberg and P. L. Nishimoto, ”PEX: A Network–Transparent 3D Graphics System,” *IEEE Computer Graphics and Applications*, Vol. 9., No. 4, July 1989, pp. 14–26

3. graPHIGS Programming Interface, Subroutine Reference Manual, Document No. SC33–8194, IBM Corporation.

4. R. W. Scheifler and J. Gettys, ”The X Window System,” *ACM Transactions on Graphics*, Vol. 5, No. 10, pp. 79–109, 1986.

5. R. Groves and R. Oehler, ”RISC System/6000 Processor Architecture,” *IBM RISC System/6000 Technology*, SA23–2619, IBM Corporation, 1990, pp. 16–23.

6. Predicting Execution Time on the IBM RISC System/6000, Document No. GC24–3711–00, IBM Corporation, pp. 14.

7. B. Olson, R. Montoye, P. Markstein, and M. NguyenPhu, ”RISC System/6000 Floating–Point Unit,” *IBM RISC System/6000 Technology*, SA23–2619, IBM Corporation, 1990, pp. 34–43.

8. R. K. Montoye, E. Hokenek and S. L. Runyon, ”Design of the IBM RISC System floating–point execution unit,” *IBM Journal of Research and Development,* Vol. 34, No. 1, January 1990, pp. 59–70.

9. IBM Shared Memory System Power/4 User's Guide and Technical Reference, IBM Corporation.

International Business
Machines Corporation
11400 Burnet Road
Austin, Texas 78758–3493