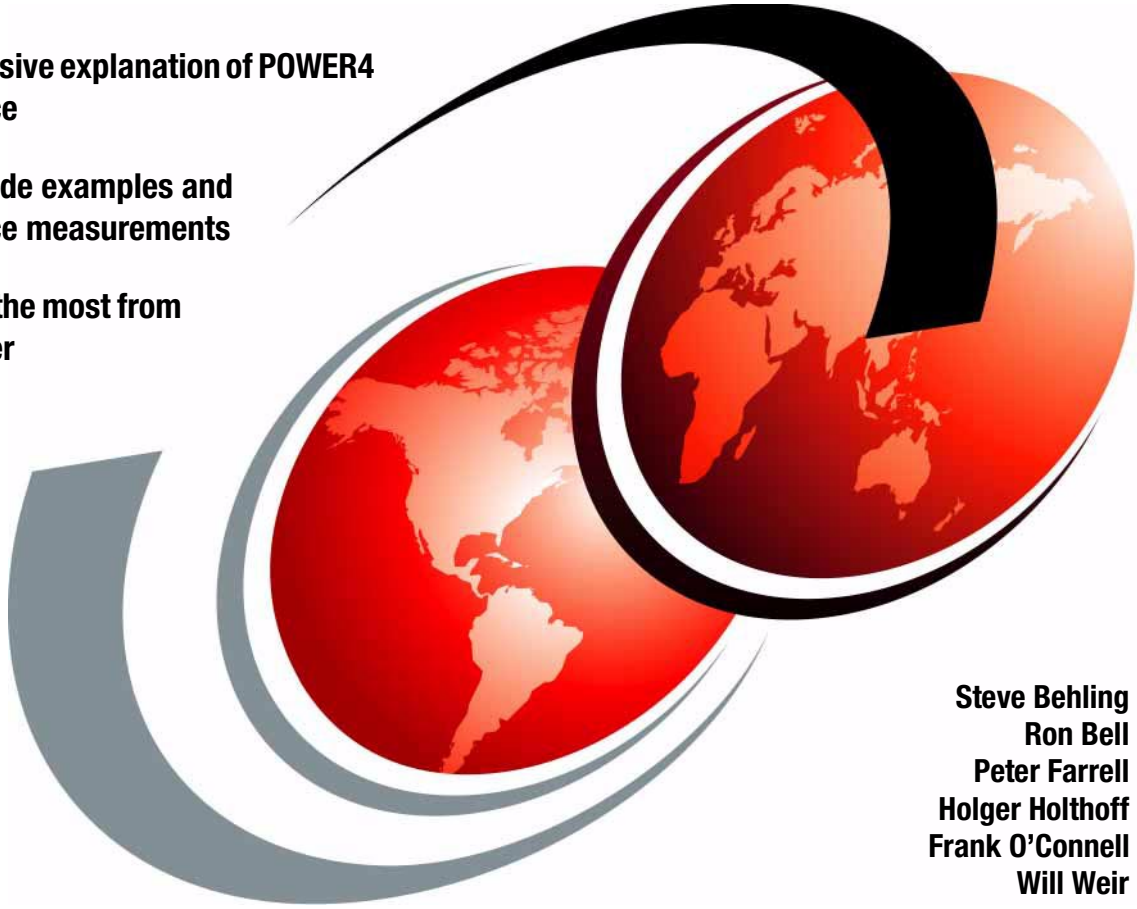
IBM

The POWER4 Processor Introduction and Tuning Guide

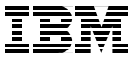
- Comprehensive explanation of POWER4 performance
- Includes code examples and performance measurements
- How to get the most from the compiler



Steve Behling
Ron Bell
Peter Farrell
Holger Holthoff
Frank O'Connell
Will Weir

Redbooks

ibm.com/redbooks



International Technical Support Organization

The POWER4 Processor Introduction and Tuning Guide

November 2001

SG24-7041-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 175.

First Edition (November 2001)

This edition applies to AIX 5L for POWER Version 5.1 (program number 5765-E61), XL Fortran Version 7.1.1 (5765-C10 and 5765-C11) and subsequent releases running on an IBM @server pSeries POWER4-based server. Unless otherwise noted, all performance values mentioned in this document were measured on a 1.1 GHz machine, then normalized to 1.3 GHz.

Note: This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Building 003 Internal Zip 2834
11400 Burnet Road
Austin, Texas 78758-3493

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This redbook is designed to familiarize you with the IBM @server pSeries POWER4 microarchitecture and to provide you with the information necessary to exploit the new high-end servers based on this architecture.

The eight to 32-way symmetric multiprocessing (SMP) pSeries 690 Model 681 will be the first POWER4 system to be available. Thus, most analysis presented in this publication refers to this system.

Specifically, this publication will address the following issues:

- POWER4 features and capabilities
- Processor and memory optimization techniques, especially for Fortran programming
- AIX XL Fortran Version 7.1.1 compiler capabilities and which options to use
- Parallel processing techniques and performance
- Available libraries and programming interfaces
- Performance examples of commonly used kernels

The anticipated audience for this redbook is as follows:

- Application developers
- Technical managers responsible for equipment purchase decisions
- Managers responsible for project planning
- Researchers involved in numerical algorithm development
- End users with an interest in understanding the performance of their applications

While this publication is decidedly technical in nature, the fundamental concepts are presented from a user point of view and numerous examples are provided to reinforce these concepts.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Stephen Behling is an Application Specialist based in Minneapolis, Minnesota, USA. He has 27 years of experience in the computer aided engineering field. He has specialized in high performance computing for the past 15 years with Cray Research Inc., Silicon Graphics, Inc., and since 1999, IBM. His areas of expertise include computational fluid dynamics, parallel programming, benchmarking and performance tuning.

Ron Bell is an IBM IT Consultant in the UK. He has an MA in Physics and a DPhil in Nuclear Physics from the University of Oxford. He has 30 years of experience with IBM High Performance Computing. His areas of expertise include the Fortran language, performance tuning for POWER architecture, and MPI parallel coding and tuning for the RS/6000 SP. He has for many years collaborated with HKS Inc. to optimize their ABAQUS product for IBM platforms.

Peter Farrell is a Consulting IT Specialist in Australia. He has worked with computer systems for over 30 years and has extensive experience in operating systems and networking software development. He has over 15 years experience in UNIX technical support with a special interest in system performance. He has worked at IBM for two years and before that was with Sequent Computer Systems. His current areas of responsibility include benchmarking and performance tuning.

Holger Holthoff is an IBM IT Consultant in Germany. He has been involved in parallel computing on RS/6000 SP since he joined the IBM Scientific Center, Heidelberg in 1994. Currently, he is a member of the pSeries Technical Support group focusing on high-performance computing projects and CAE applications in manufacturing industries. His areas of expertise include performance tuning for the POWER architecture and message passing programming for the RS/6000 SP.

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

Will Weir is an IBM IT Specialist in the UK. He has worked on scientific applications and systems on IBM RS/6000 and RS/6000 SP for 11 years. Currently he is a member of the High Performance Computing team based in IBM Bedfont. His areas of expertise include application porting and benchmarking, and RS/6000 SP systems.

The project that produced this Redbook was managed by:

Scott Vetter from IBM Austin

A special thanks to Steve White from IBM Austin, for his determination and pursuit of excellence.

Thanks to the following people for their contributions to this project:

Arthur Ban, Bill Hay, Harry Mathis, John McCalpin, Alex Mericas, William Starke, Steve Stevens, Joel Tendler from IBM Austin

Joan McComb and Frank Johnston from IBM Poughkeepsie

Richard Eickemeyer from IBM Rochester

Bob Blainey, Ian McIntosh, and Kelvin Li from IBM Toronto

Notice

This publication is intended for developers of numerically intensive code for the IBM @server pSeries POWER4, for business partners and sales specialists wanting supporting metrics for pSeries 690 Model 681, and for technical specialists who require detailed product information to help demonstrate IBM's industry-leading technology. See the PUBLICATIONS section of the IBM Programming Announcement for Fortran Version 7.1.1 for more information about what publications are considered to be product documentation.

IBM trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX®
AIX 5L™
e (logo)® 
IBM ®
iSeries™
LoadLeveler®
PowerPC®
PowerPC Architecture™
PowerPC 601®
PowerPC 603™
PowerPC 604™
pSeries™
Redbooks™
Redbooks Logo 
RISC System/6000®
RS/6000®
Sequent®
SP™

Comments welcome

Your comments are important to us!

We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to the address on page ii.

1

Processor evolution

In this section, the stages of RS/6000 and pSeries processor development are discussed, starting with the POWER1 architecture through to the latest POWER4.

1.1 POWER1

The first RS/6000 products were announced by IBM in February of 1990, and were based on a multiple chip implementation of the POWER architecture, described in *IBM RISC System/6000 Technology*, SA23-2619. This technology is now commonly referred to as POWER1, in the light of more recent developments. The models introduced included an 8 KB instruction cache (I-cache) and either a 32 KB or 64 KB data cache (D-cache). They had a single floating-point unit capable of issuing one compound floating-point multiply/add (FMA) operation each cycle, with a latency of only two cycles. Therefore, the peak MFLOPS rate was equal to twice the MHz rate. For example, the Model 530 was a desk-side workstation operating at 25 MHz, with a peak performance of 50 MFLOPS. Commonly occurring numerical kernels were able to achieve performance levels very close to this theoretical peak.

In January of 1992, the Model 220 was announced, based on a single chip implementation of the POWER architecture, usually referred to as RISC Single Chip (RSC). It was designed as a low-cost, entry-level desktop workstation, and contained a single 8 KB combined instruction and data cache.

The last POWER1 machine, announced in September of 1993, was the Model 580. It ran at 62.5 MHz and had a 32 KB I-cache and a 64 KB D-cache.

1.2 POWER2

Announced in September 1993, the first POWER2 machines included the 55 MHz Model 58H, the 66.5 MHz Model 590, and the 71.5 MHz 990. The most significant improvement introduced with the POWER2 architecture for scientific and technical applications was the floating-point unit (FPU) that was enhanced to contain two 64-bit execution units. Thus, two floating-point multiply/add instructions could be executed each cycle. A second fixed-point execution unit was also provided. In addition, several new hardware instructions were introduced with POWER2:

- Quad-word storage instructions. The quad-word load instruction moves two adjacent double-precision values into two adjacent floating-point registers.
- Hardware square root instruction.
- Floating-point to integer conversion instructions.

Although the Model 590 ran with only a marginally faster clock than the POWER1-based Model 580, the architectural improvements listed above, combined with a larger 256 KB D-cache size, enabled it to achieve far greater levels of performance.

In October 1996, IBM announced the RS/6000 Model 595. This was the first machine to be based on the P2SC (POWER2 Super Chip) processor. As its name suggests, this was a single chip implementation of the POWER2 architecture, enabling the clock speed to be increased further. The Model 595 ran at 135 MHz, and the fastest P2SC processors, found in the Model 397 workstation and RS/6000 SP Thin4 nodes, ran at 160 MHz, with a theoretical peak speed of 640 MFLOPS.

1.3 PowerPC

The RS/6000 Model 250 workstation, the first to be based on the PowerPC 601 processor running at 66 MHz, was introduced in September, 1993. The 601 was the first processor arising out of the partnership between IBM, Motorola, and Apple. The PowerPC Architecture includes most of the POWER instructions. However, some instructions that were executed infrequently in practice were excluded from the architecture, and some new instructions and features were added, such as support for symmetric multiprocessor (SMP) systems. In fact, the 601 did not implement the full PowerPC instruction set, and was a *bridge* from

POWER to the full PowerPC Architecture implemented in more recent processors, such as the 603, 604, and 604e. Currently, the fastest PowerPC-based machines from IBM for technical purposes, the four-way SMP system RS/6000 7025 Model F50 and the uniprocessor system RS/6000 43P 7043 Model 150, use the 604e processor running at 332 MHz and 375 MHz, respectively. The POWER3 and POWER4 processors are also based on the PowerPC Architecture, but discussed in the following sections.

1.4 RS64

The first RS64 processor was introduced in September of 1997 and was the first step into 64-bit computing for RS/6000. While the POWER2 product had strong floating-point performance, this series of products emphasized strong commercial server performance. It ran at 125 MHz with a 2-way associative, 4 MB L2 cache and had a 64 KB L1 instruction cache, a 64 KB L1 data cache, one floating-point unit, one load-store unit, and one integer unit. Systems were designed to use up to 12 processors. pSeries products using the RS64 were the first pSeries products to have the same processor and memory system as iSeries products.

In September 1998, the RS64-II was introduced. It was a different design from the RS64 and increased the clock frequency to 262 MHz. The L2 cache became 4-way set associative with an increase in size to 8 MB. It had a 64 KB L1 instruction cache, a 64 KB L1 data cache, one floating-point unit, one load-store unit, two integer units, and a short in-order pipeline optimized for conditional branches.

With the introduction of the RS64-III in the fall of 1999, this design was modified to use copper technology, achieving a clock frequency of 450 MHz, with a L1 instruction and data cache increased to 128 KB each. This product also introduced hardware multithreading for use by AIX. Systems were designed to use up to 24 processors.

In the fall of 2000, this design was enhanced to use silicon on insulator (SOI) technology, enabling the clock frequency to be increased to 600 MHz. The L2 cache size was increased to 16 MB on some models. Continued development of this design provided processors running at 750 MHz. The most recent version of this microprocessor was called the RS64-IV.

During the history of this family of products, top performance publications have been made for a large variety of benchmarks, including TPC-C (online transaction processing), SAP (enterprise resource planning - ERP), Baan (ERP), PeopleSoft (ERP), SPECweb (web serving), and SPECjbb (Java).

1.5 POWER3

The POWER3 processor brought together the fundamental design of the POWER2 microarchitecture, as currently implemented in the P2SC processor, with the PowerPC Architecture. It combined the excellent floating-point performance delivered by P2SC's two floating-point execution units, while being a 64-bit, SMP-enabled processor ultimately capable of running at much higher clock speeds than current P2SC processors. Initially introduced in the fall of 1998 at a processor clock frequency of 200 MHz, most recent versions of this microprocessor incorporate copper technology and operate at 450 MHz.

1.6 POWER4

The new POWER4 processor, described in detail in Chapter 2, "The POWER4 system" on page 5, continues the evolution. The POWER4 processor chip contains two microprocessor cores, chip and system pervasive functions, core interface logic, a 1.41 MB level-2 (L2) cache and controls, the level-3 (L3) cache directory and controls, and the fabric controller that controls the flow of information and control data between the L2 and L3 and between chips.

Each microprocessor contains a 64 KB level-1 instruction cache, a 32 KB level-1 data cache, two fixed-point execution units, two floating-point execution units, two load/store execution units, one branch execution unit, and one execution unit to perform logical operations on the condition. Instructions dispatched in program order in groups are issued out of program order to the execution units, with a bias towards oldest operations first. Groups can consist of up to five instructions, and are always terminated by a branch instruction. The processors on the first IBM POWER4-equipped servers, the IBM @server pSeries 690 Model 681 servers, operate at either 1100 MHz or 1300 MHz.

A quick look at comparative metrics may help you put the capacity of the latest POWER-based processors in perspective, as provided in Table 1-1.

Table 1-1 Comparative POWER3-II, RS64-III, and POWER4 processor metrics

Metric	POWER3-II 450 MHz	RS64-III 450 MHz	POWER4 1300 MHz
SPECint2000	335.0	234.0	814.0
SPECfp2000	433.0	210.0	1169.0

2

The POWER4 system

The POWER4 system is a new generation of high-performance 64-bit microprocessors and associated subsystems especially designed for server and supercomputing applications. POWER4 systems power the next generation of servers that will be the replacements for the POWER3 and RS64-series high-end RS/6000 and pSeries technical servers. This chapter provides details of the POWER4 system that are significant to application programmers concerned with understanding or improving application performance.

2.1 POWER4 system overview

The POWER4 system is a high-performance microprocessor and storage subsystem utilizing IBM's most advanced semiconductor and packaging technology. It is the building block for the next-generation pSeries and iSeries SMP servers. The POWER4 system implements the PowerPC AS Processor Architecture, which specifies the instruction set, register set, and storage model, to name a few, in other words, all functions that are visible to the programmer.

A POWER4 system logically consists of multiple POWER4 microprocessors and a POWER4 storage subsystem, interconnected together to form an SMP system. Physically, there are three key components: the POWER4 processor chip, the L3 Merged Logic DRAM (MLD) chip, and the memory controller chip.

- The POWER4 processor chip contains two 64-bit microprocessors, a microprocessor interface controller unit, a 1.41 MB (1440 KB) level-2 (L2)

cache, a level-3 (L3) cache directory, a fabric controller responsible for controlling the flow of data and controls on and off the chip, and chip/system pervasive functions.

- The L3 merged logic DRAM (MLD) chip, which contains 32 MB of L3 cache. An eight-way POWER4 SMP module will share 128 MB of L3 cache consisting of four modules each of which contains two 16 MB merged logic DRAM chips.
- The memory controller chip features one or two memory data ports, each 16 bytes wide, and connects to the L3 MLD chip on one side and to the Synchronous Memory Interface (SMI) chips on the other.

The pSeries 690 Model 681 is built around the POWER4 Multi-chip Module (MCM) which contains four POWER4 chips. A 32-way SMP system contains four MCMs. POWER4 MCMs are mounted on system boards along with the L3, memory cards including the memory controllers, and support chips to form the heart of the pSeries 690 Model 681.

2.2 The POWER4 chip

The main components of the POWER4 chip are shown in Figure 2-1 on page 7. The POWER4 chip has a maximum of two microprocessors, each of which is a fully functional 64-bit implementation of the PowerPC AS Architecture specification. Also on the chip is a unified second-level cache, shared by both microprocessors through a core interface unit (CIU). The L2 cache is physically divided into three equal-sized parts, each having an L2 cache controller. The CIU connects each of the three L2 controllers to each processor though separate 32-byte wide data reload and instruction reload ports. Each microprocessor also has an 8-byte wide store port to the CIU that in turn is used to store data through the appropriate L2 controller.

Each processor also has associated non-cacheable unit (NCU), shown in Figure 2-1 on page 7, responsible for handling instruction-serializing functions and performing any non-cacheable operations in the storage hierarchy. Logically, these are part of the L2 cache.

To improve performance by reducing the latency to memory, the directory for the level 3 cache (L3 cache) and its controller are also located on the POWER4 chip (while the actual L3 arrays are located on the L3 MLD module). Additionally, for I/O device communication, the GX bus controller and the two associated four-byte wide GX bus, one on chip and one off chip, are on the chip as well.

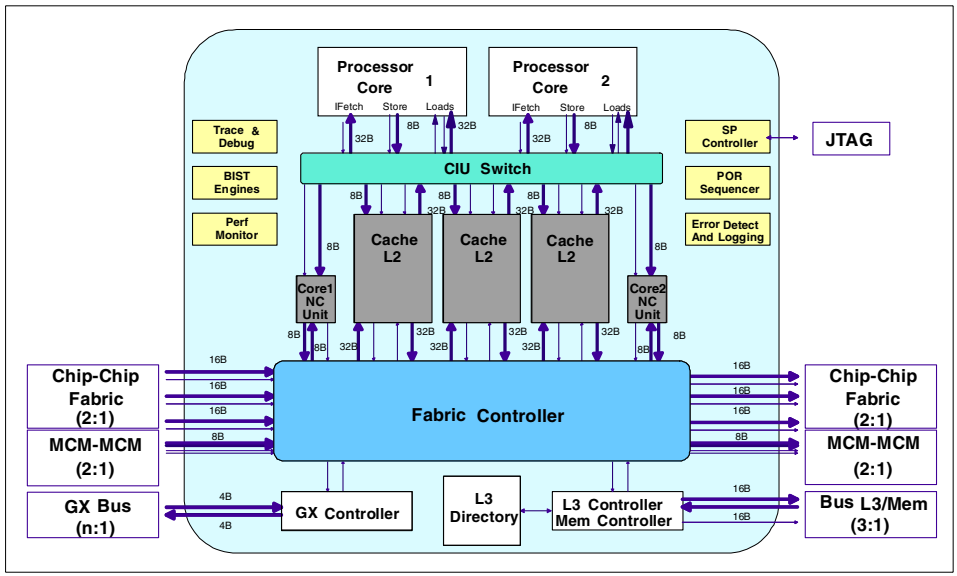


Figure 2-1 The POWER4 chip

Each POWER4 chip contains a fabric controller that provides master control of the network of buses. These buses connect together the on-chip L2 controllers, L3, other POWER4 chips, and other POWER4 modules, and also perform snooping and coherency duties. The Fabric Controller directs a point-to-point network between each of the four chips on the MCM made up of unidirectional 16-byte wide buses running at half the processor frequency, the 8-byte buses also operating at half the processor speed connecting each chip to a corresponding chip on a neighboring MCM, and also controls the unidirectional 16-byte wide buses (running at 3:1 in the pSeries 690 Model 681) between the POWER4 chip and the L3 cache, as well as the buses to the NCU and GX controller.

Although not related to performance, it is worth mentioning that the chip also includes an important set of *pervasive* functions. These include trace and debug facilities used for First Failure Data Capture, built-in self-test (BIST) facilities, performance monitoring unit (PMU), an interface to the service processor (SP) used to control the overall system, power-on reset (POR) sequencing logic, and error detection and logging circuitry.

2.3 Processor overview

Figure 2-2 shows a high-level block diagram of a POWER4 microprocessor. The POWER4 microprocessor is a high-frequency, speculative superscalar machine with out-of-order instruction execution capabilities. Eight independent execution units are capable of executing instructions in parallel providing a significant performance attribute known as *superscalar* execution. These include two identical floating-point execution units, each capable of completing a multiply/add instruction each cycle (for a total of four floating-point operations per cycle), two load-store execution units, two fixed-point execution units, a branch execution unit, and a conditional register unit used to perform logical operations on the condition register.

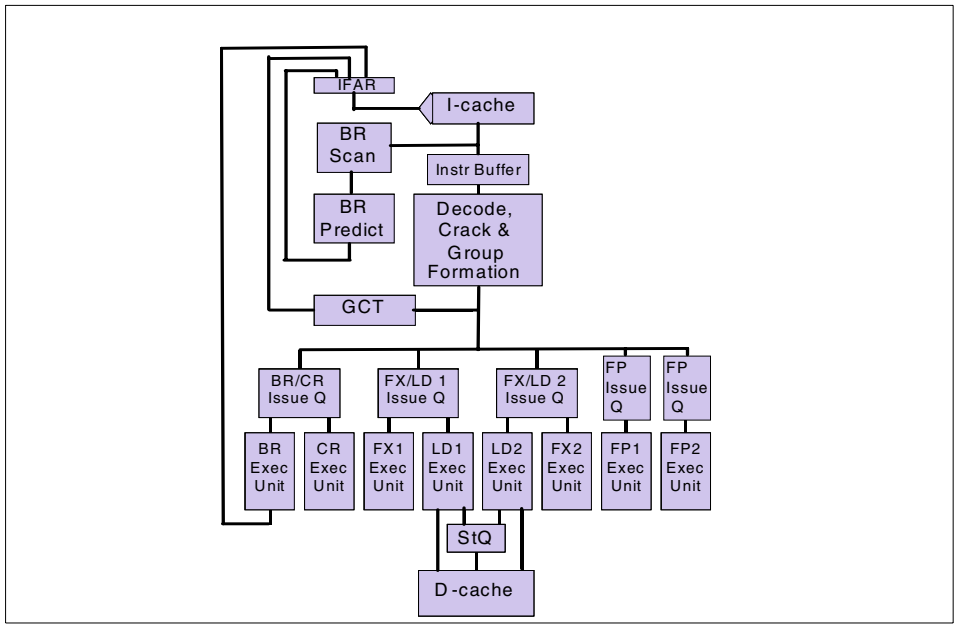


Figure 2-2 The POWER4 processor

To keep these execution units supplied with work, each processor can fetch up to eight instructions per cycle and can dispatch and complete instructions at a rate of up to five per cycle. A processor is capable of tracking over 200 instructions in-flight at any point in time. Instructions may issue and execute out-of-order with respect to the initial instruction stream, but are carefully tracked so as to complete in program order. In addition, instructions may execute speculatively to improve performance when accurate predictions can be made about conditional scenarios.

2.3.1 The POWER4 processor execution pipeline

Figure 2-3 depicts the POWER4 processor execution pipeline. The deeply pipelined structure of the machine's design is shown. Each small box represents a stage of the pipeline (a stage is the logic which is performed in a single processor cycle). Note that there is a common pipeline which first handles instruction fetching and group formation, and this then divides into four different pipelines corresponding to four of the five types of execution units in the machine (the CR execution unit is not shown, which is similar to the fixed-point execution unit). All pipelines have a common termination stage, which is the group completion (CP) stage.

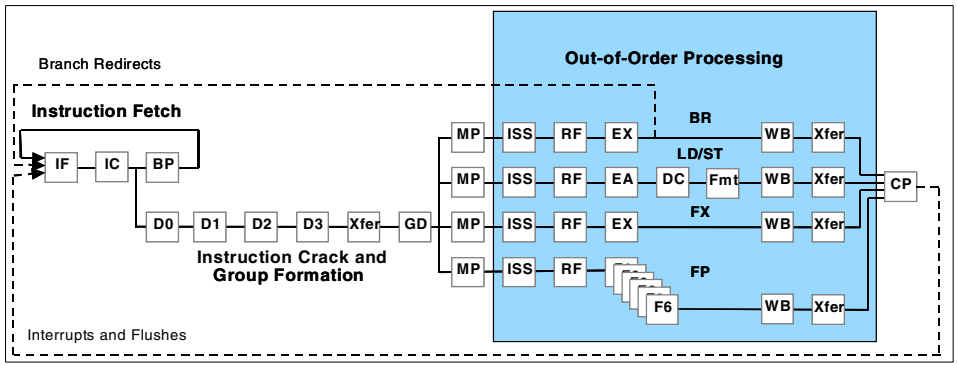


Figure 2-3 The execution pipeline

2.3.2 Instruction fetch, group formation, and dispatch

The instructions that make up a program are read in from storage and are executed by the processor. During each cycle, up to eight instructions may be fetched from cache according to the address in the instruction fetch address register (IFAR) and the fetched instructions are scanned for branches (corresponding to the IF, IC, and BP stages in Figure 2-3).

Since instructions may be executed out of order, it is necessary to keep track of the program order of all instructions in-flight. In the POWER4 microprocessor, instructions are tracked in groups of one to five instructions rather than as individual instructions. Groups are formed in the pipeline stages D0, D1, D2, and D3. This requires breaking some of the more complex PowerPC instructions down into two or more simpler instructions.

Instructions that are broken into two internal instructions are named *cracked* instructions. Instructions that are broken into three or more internal instructions are named *millicoded* instructions. Common instructions that are cracked are:

- All load/store-update forms (cracked into load/store + addi for gpr update)

- X-form fixed-point stores (cracked into an add + store)
- Load algebraic (cracked into load + sign extend)
- CR-logicals, except for *destructive* forms (RT=RB)

Common instructions that are millicoded are:

- lmw, lswi (all multiples and string load instructions)
- mtcrf (move to condition register fields, more than one target field)
- mtxer and mfxer

Groups are formed that contain up to five internal instructions, each occupying an internal instruction slot (numbered 0 through 4) of a dispatch group. After a group is assembled, it is readied for *dispatch,* which is the process of sending the instructions as a group to the issue queues. As part of the dispatching operation, internal group instruction dependencies are determined and internal resources such as issue queue slots, rename registers, reorder queues, and mappers are assigned (GD and MP stages). Groups are dispatched and tracked using the 20-entry global completion table (GCT) in program order at a rate of up to one per cycle. (See discussion on completion in Section 2.3.10, “Group completion” on page 16).

Each internal instruction slot in a group feeds separate issue queues for the floating-point units, the branch execution unit, the CR execution unit, the logical CR execution unit, the fixed-point execution units and the load/store execution units. The fixed point and load/store execution units share common issue queues. Table 2-1 summarizes the depth of each issue queue and the number of queues available for each type of queue. For the floating-point issue queues and the common issue queues for the fixed point and load/store units, the issue queues fed from slots 0 and 3 of the instruction group hold instructions to be executed in one of the execution units, while the issue queues fed from slots 1 and 2 of the group feed the other execution unit. The CR execution unit draws its instructions from the CR logical issue queue fed from instruction slots 0 and 1.

Table 2-1 Issue queues

Queue type	Entries per queue	Number of queues
Fixed point and load-store units	9	4
Floating point	5	4
Branch execution	12	1
CR logical	5	2

During the issue stage (ISS), instructions that are ready to execute are pulled out of the issue queues and enter the register file access stage, where they access their source operands from registers. If more than two instructions from a particular queue are ready to execute, the issue logic attempts to issue the oldest instruction.

2.3.3 Instruction execution, speculation, rename resources

The speculative-execution design of the POWER4 microprocessor can execute instructions before it is certain that those instructions will be required to be executed. Speculative execution can significantly enhance performance by potentially eliminating stalls associated with waiting for a condition associated with a branch to be resolved.

It is important to understand the distinction between instructions that *finish execution* and instructions that *complete*. The term *completion* carries an important architectural meaning: completion makes results available to a program. An instruction or group of instructions may have been speculatively executed by the hardware, but unless they complete, their results are not visible to the program. (This, however, does not hold up speculative execution of instructions dependent on these results.)

A superscalar speculative-execution design requires an orderly way to manage machine resources and to flush instructions, along with affected registers, when predictions are found to be incorrect. The POWER4 microprocessor uses physical resources called rename registers throughout its design that are critical to this capability. Rename registers are assigned to instructions during the mapping (MP) stage and are typically released when the next instruction writing to the same logical resource (for example, the same architected general purpose register) is completed. At any point in time, a rename register may represent an architected register or a target buffer register. In the latter case, the register will be reclassified as an architected register upon successful completion of the instruction or released for reuse if the instruction is flushed. For each type of PowerPC register group, Table 2-2 lists the number of architected registers in the PowerPC specification and the corresponding number of physical (rename) registers in the POWER4 microprocessor.

Table 2-2 Rename resources

Resource type	Architected (PowerPC)	Physical
General-Purpose Register (GPR)	32	80
Floating-Point Register (FPR)	32	72
Condition Register (CR)	eight 4-bit fields	32

Resource type	Architected (PowerPC)	Physical
Link/Count Register (LCR)	2	16
Floating-Point Status and Control Register (FPSCR)	1	20
Fixed-Point Exception Register (XER)	four fields	24

2.3.4 Branch prediction

If an instruction sequence contains a conditional branch instruction, the conditional test associated with that branch directs the flow of execution, either to take the branch or to continue execution at the next sequential instruction. In the POWER4 microprocessor, all such conditional branches are predicted, and instructions are fetched and executed speculatively based upon that prediction. Instruction streams are scanned for branch instructions, and upon encountering a conditional branch, a prediction is made as to the outcome of its conditional test. This prediction is used to direct the fetching of instructions beyond the branch. If the prediction is correct, processing simply continues and the branch instruction completes normally. If, however, the prediction is incorrect, the instructions corresponding to the incorrect prediction are flushed and instruction fetching is redirected down the correct path, incurring a performance penalty of at least 12 cycles.

To make accurate predictions about the outcome of a conditional branch instruction, the POWER4 microprocessor tracks two different prediction methodologies simultaneously, and also tracks which method is predicting a particular branch more effectively, so that it may use the more successful prediction method for a given branch. The first employs a traditional branch history table, each entry of which corresponds to whether a given branch was taken or not taken. The second method attempts to predict the direction of a branch by using information about the path of execution that was taken to get to that branch. Both methods use a 16 KB entry table to hold their 1-bit prediction per branch, and a third 16 KB table holds the 1-bit selector indicating the preferred predictor for that branch. This combination of branch prediction methods produces very accurate predictions across a wide range of workload types. As branch instructions are executed and resolved, the branch history tables and the other predictors are updated to reflect the latest and most accurate information.

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

Dynamic branch prediction can be overridden by hint bits in the branch instructions. This is useful in cases where knowledge at the application level exists that can result in better predictions than the execution-time hardware prediction methods. It is accomplished by setting two previously reserved bits in conditional branch instructions, one to indicate a software override and the other to predict the direction. When these two bits are zero, the hardware branch prediction previously described is used. Since only reserved bits are used for this purpose, 100 percent binary compatibility with earlier software is maintained.

The POWER4 processor also has *target address prediction* logic for predicting the target of *branch to link* and *branch to count* instructions, which often have repeating and therefore predictable targets.

2.3.5 Translation buffers (TLB, SLB, I- and D-ERAT)

The PowerPC Architecture specifies a virtual storage model for applications, in which each program's effective address (EA) space is a subset of a larger virtual address (VA) space that is managed by the operating system (see Section 3.3.1, "POWER4 virtual memory architecture overview" on page 54). Virtual addresses are, in turn, translated into real (physical) storage locations. Each POWER4 processor has three types of buffer caches to speed this process of translation: a translation look-aside buffer (TLB), a segment look-aside buffer (SLB), and an effective-to-real address table (ERAT). The SLB is a 64-entry, fully associative buffer for caching the most recent segment table entries (STEs). The TLB is a 1024-entry, four-way set-associative buffer for caching the most recent page table entries (PTEs). These page table entries may represent either the standard 4 KB page or a 16 MB large page. The POWER4 microprocessor also has separate ERATs for instructions (I-ERAT) and for data (D-ERAT), both of which are 128-entry, two-way set-associative arrays. The ERATs hold the most recent {EA,RA} pairs to facilitate the high-frequency, high-bandwidth design of the POWER4 microprocessor. Both ERATs are indexed using the effective address and require 10 cycles to reload from the TLB, assuming that pages EA to RA translation exists in the TLB. ERAT entries are always maintained on a 4 KB page basis.

2.3.6 Load instruction processing

Load instructions execute in the LD/ST pipeline shown in Figure 2-3 on page 9. After a load instruction issues, it must generate the effective address of the operand being loaded or stored using the contents of the general-purpose registers specified along with the instruction. The RA stage is the cycle in which the registers are accessed and the EA cycle is the address generation stage, also called AGEN.

To keep track of hazards associated with loads and stores executing out of order with respect to each other, two 32-entry queues exist: the load reorder queue (LRQ) and the store reorder queue (SRQ). All loads and stores are allocated an entry in these queues at dispatch, respectively. Loads and stores are checked against the entries in these tables to ensure that program correctness is maintained.

The cycle following the AGEN cycle is the DC cycle, in which the real address from the D-ERAT is obtained and the data cache is accessed for the appropriate cache line. If the DC cycle is successful, the data is formatted and written into a register, and it is ready for use by a dependent instruction. If a D-ERAT miss occurs, the instruction is *rejected*, but it is kept in the issue queue. Meanwhile a request is made to the TLB to reload the D-ERAT with the address translation information. The rejected instruction is then re-issued a minimum of 7 cycles after it was first issued. If the D-ERAT still does not contain the translation information, the instruction is again rejected. This process continues until the D-ERAT is reloaded.

In the case of loads, hits in the L1 data cache result in the requested bytes being formatted and written into the appropriate register. In the event of a cache miss, a request is initiated to the L2 cache to retrieve the line. Requests to the L2 cache are stored in the load miss queue (LMQ), which acts as a repository for all outstanding L1 cache line misses. The LMQ can hold up to eight requests to the L2 cache; hence each POWER4 microprocessor is capable of managing up to eight data cache line requests to the L2 cache (and beyond) at any given time, providing an effective mechanism for reducing the average latency of cache line reloads. If the LMQ is full, the load instruction that missed in the data cache is rejected and is re-issued again in a minimum of 7 cycles. If there is already a request to the L2 cache for the same line from another load instruction, the second request is merged into the same LMQ entry. If a third request to the same line occurs, the load instruction is rejected and processing continues as above. All reloads from the L2 cache check the LMQ to see if there is an outstanding request yet to be honored against a just-returned line. If there is, the requested bytes are forwarded to the register to complete the execution of the load instruction. After the line has been reloaded, the LMQ entry is released for reuse.

2.3.7 Store instruction processing

Store instructions are assigned an entry in the SRQ during the issue stage for tracking by the real address of the stored data. The store data queue (SDQ) has 32 double-word entries and receives the data being stored in an entry corresponding to the address entry in the SRQ. Stores are removed from the SRQ and SDQ and the data is written to the L2 once it has been completed and all older stores have been successfully sent to the L2.

Loads that are to the same address as a previous store may be forwarded directly from the SDQ to the target register of the load, provided the data for the load is completely contained within the store operand and the data has not yet been written to the cache.

The L1 data cache is a store-through design: all data stored to cache lines that exist in the L1 data cache are also sent to the L2 cache to ensure that modifications to lines in the L1 cache are always reflected in corresponding lines in the L2 cache. If data is stored to a cache line that is not found in the L1 data cache, the data is simply transferred straight through to the L2 cache without establishing the cache line in the L1 data cache. All data contained in the L1 data cache is guaranteed to be in the L2 cache. If the L2 needs to cast out data that is contained in the L1 data cache, that line is invalidated in the L1 data cache.

Stores can be sent to the L2 cache at a maximum rate of one store per cycle. Store data is directed to the proper L2 controller (through a hashing function) by way of the storage slice queue (SSQ) and the L2 store queue (STQ). Steady-state store performance is described in detail in Section 3.1.7, “Selected fundamental kernel performance within on-chip cache” on page 49.

2.3.8 Fixed-point execution pipeline

The pipeline for the two fixed-point execution units (FXUs) is shown as the FX pipe in Figure 2-3 on page 9. Both units are capable of basic arithmetic, logical, and shifting operations, and both units are capable of fixed-point multiplies (non-pipelined). One of the FXUs is capable of fixed-point divides, and the other can handle special-purpose register (SPR) operations.

2.3.9 Floating-point execution pipeline

The POWER4 microprocessor contains two symmetrical floating-point execution units each of which implement a fused multiply/add pipeline with single cycle throughput conforming to the PowerPC microarchitecture. All floating-point instructions pass though both the multiply stage and the add stage. For floating-point multiplies, 0 is used as the add operand, and for floating-point adds, 1 is used as the multiplicand. Each floating-point execution unit supports single-cycle throughput and six-cycle data forwarding for dependent instructions.

The floating-point operations square root (*fsqrt* and *fsqrts*) and divide (*fdiv* and *fdivs*) are not pipelined. Each pipeline can execute the operations with the assistance of additional logic to handle their numerical algorithms. The performance of these and other floating-point operations is highlighted in Section 3.1.7, “Selected fundamental kernel performance within on-chip cache” on page 49.

The POWER4 microprocessor also implements the optional PowerPC instructions *fres* (floating-point reciprocal estimate) and *frsqrte* (floating-point reciprocal square-root estimate), as well as *fsel* (floating-point select). The last instruction provides for a conditional floating-point assignment operation without branching, which eliminates the chance of incurring a performance penalty for a mispredicted branch.

2.3.10 Group completion

Results are written to registers or cache/memory when the group completes. Completion carries an important architectural meaning: completion makes results available to a program through architected resources (such as floating-point registers). An instruction or group of instructions may have been executed speculatively by the hardware, but do not complete unless all conditions associated with their execution have been successfully resolved. A group can complete when all older groups have completed and when all instructions in the group have finished execution free of exceptions. One group can complete in a cycle, which matches the rate at which groups can be dispatched.

2.4 Storage hierarchy

The POWER4 system storage hierarchy consists of three levels of cache and the memory subsystem. The L1 caches and L2 cache is physically on the POWER4 chip. The directory for the L3 cache is also on the chip, but the actual cache itself is on a separate. Table 2-3 summarizes the capacities and organization of the various levels of cache.

Table 2-3 Storage hierarchy organization and size

Component	Organization	Capacity
L1 instruction cache	Direct map, 128-byte line	128 KB per chip (64 KB per processor)
L1 data cache	Two-way, 128-byte line	64 KB per chip (32 KB per processor)
L2 cache	Four-way to eight-way, 128-byte line	1440 KB per chip (1.41 MB)
L3 cache	Eight-way, 512-byte lines, managed as four 128-byte sectors	128 MB per MCM

2.4.1 L1 instruction cache

Each POWER4 microprocessor has an L1 instruction cache that is a 64 KB direct mapped cache and capable of either one 32-byte read or write each cycle. It is indexed by the effective address of the instruction cache line.

2.4.2 L1 data cache

Each POWER4 microprocessor contains an L1 data cache that is 32 KB in size, two-way set associative, and has a replacement policy of first-in-first-out (FIFO). It is capable of two eight-byte reads and one eight-byte write per cycle (it is effectively triple ported).

When the cache line containing the operand of a load instruction is not in the L1 data cache, the processor requests a cache line *reload*, which retrieves the line from the memory subsystem and places it in the L1 data cache across a reload interface to the CIU that is 32 bytes wide. Since the maximum the processor can demand per cycle from the register file is two doubleword loads (for example, 16 bytes/cycle) this reload rate is twice the rate that the processor itself can demand data.

The L1 data cache implements a *store-through* design, which means that any updates to data in the L1 data cache are immediately stored through to the L2 cache to keep it synchronized with the L1 data cache. If the operand of a store instruction is not found in any of the cache lines currently resident in the L1 data cache (such as when there is an L1 store miss), the data that is in the source register of the store instruction is stored through to the L2 cache, and the cache line is not established or reloaded into the L1. The data to be stored passes through various queues in the processor (the Store Data Queue), the CIU (the Slice Store Queue), and the L2 cache (the L2 Store Queue) before it actually gets stored into the L2 cache. These queues act as buffers for stored data, which allows the store instruction itself to complete and facilitates the optimization of store performance through a technique named *store gathering*.

2.4.3 L2 cache

Each POWER4 chip has an L2 cache that is supervised by three L2 *controllers*, each of which manages 480 KB, for a total L2 size of 1440 KB. Cache lines are hashed across the three controllers. Cache line replacement is implemented as a binary-tree pseudo-LRU algorithm. The L2 cache is a unified cache: it caches instructions, data, and page table entries. The L2 cache is also shared by the processors on the chip. For HPC features of the pSeries Model 690, there is only one processor per chip, and thus the L2 cache is entirely owned by that processor.

Memory coherency in the system is enforced primarily at the L2 cache level by L2 cache controllers. Each L2 has associated command queues, known as coherency processors. Snoop processors within each controller observe all transactions in the system and respond accordingly, providing responses or delivering cache lines if the situation merits.

2.4.4 L3 cache

The L3 cache is eight-way set-associative organized in 512-byte blocks, but with coherence still maintained in the system cache line size of 128 bytes. POWER4 chips are connected to memory through an L3 cache (see Figure 2-4). Generally, it caches data that comes from the memory port to which it is attached. An exception to this is when the cache line has been sent from a remote MCM, in which case an attempt is made to cache the line in an L3 cache on the requesting module.

The L3 cache is designed to be combined with other L3 caches on the same processor module in pairs or quadruplets to create a larger, address-interleaved L3 cache of 64 MB or 128 MB. Combining L3 caches into groups not only increases the L3 cache size, but also increases the L3 bandwidth available to any processor. When combined into groups, L3 caches and the memory behind them are interleaved on 512-byte granularity.

2.4.5 Interconnecting chips to form larger SMPs

The basic building block for a pSeries is a multi-chip module (MCM) with four POWER4 chips forming an 8-way SMP, as shown in Figure 2-4. Multiple MCMs can then be interconnected to form 16-, 24-, and 32-way SMPs.

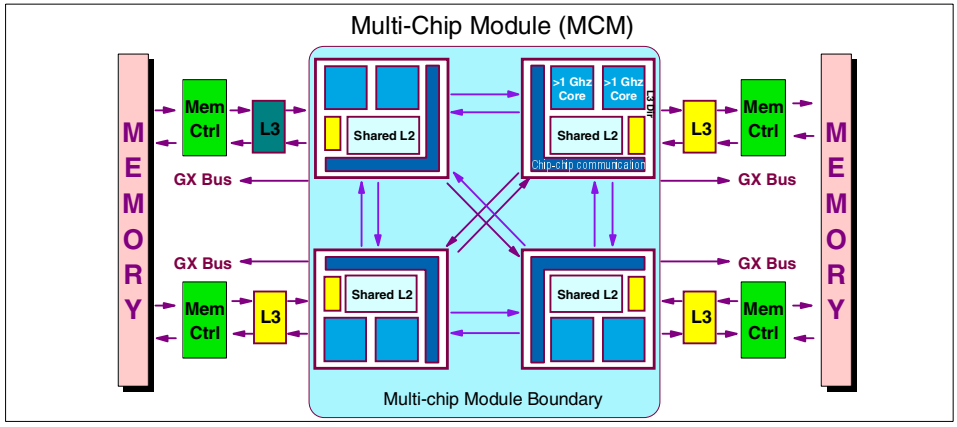


Figure 2-4 A logical view of the interconnection buses within an MCM

The logical interconnection of four POWER4 chips is point-to-point, with uni-directional buses connecting each pair of chips to form an 8-way SMP with an all-to-all interconnection topology. The fabric controller on each chip monitors (for example snoops) all buses and writes to its own bus, arbitrating between the L2 cache, I/O controller, and the L3 controller for the bus. Requests for data from an L3 cache are snooped by each fabric controller to determine if it has the data being requested in its L2 cache (in a suitable state), or in its L3 cache, or in the memory attached to its L3 cache. If any one of these is true, then that controller returns the requested data to the requesting chip on its bus. The fabric controller that generated the request then sees the response on that bus and accepts the data.

2.4.6 Multiple module interconnect

Figure 2-5 shows the interconnection of four MCMs to form a 32-way SMP. Up to four MCMs can be interconnected by extending each bus from each module to its neighboring module in one direction. Inter-module buses run at half the processor frequency and are 8-bytes wide. The inter-MCM topology is that of a ring in which requests and data move from one module to another module in one direction. As with the single MCM configuration, each chip always sends requests, commands and data on its own bus but snoops all buses for requests or commands from other MCMs.

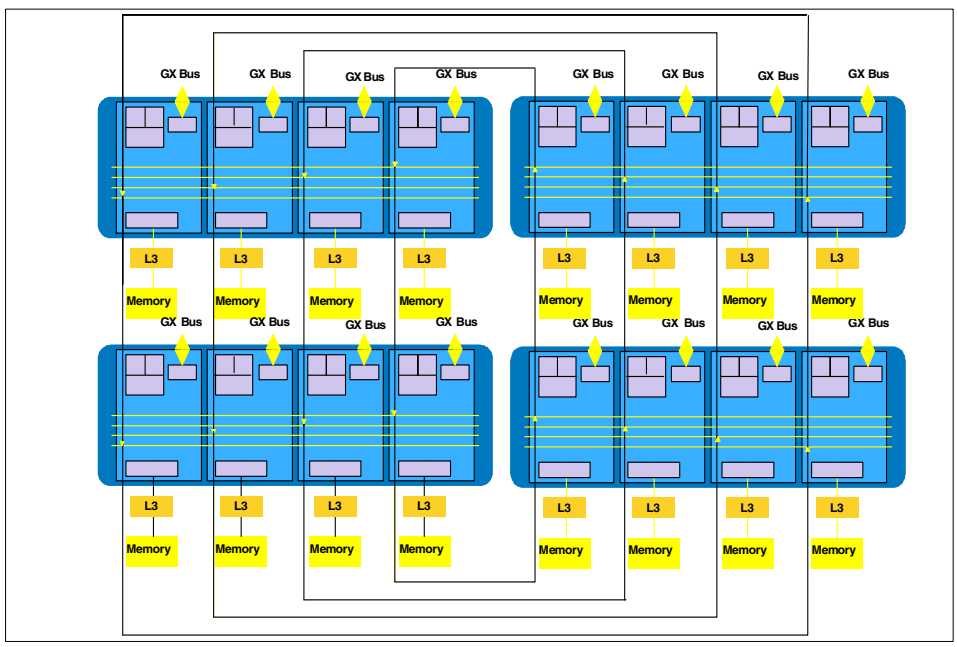


Figure 2-5 Logical view of MCM-to-MCM interconnections

2.4.7 Memory subsystem

Each POWER4 chip can optionally have a memory controller attached behind the L3 cache. Memory controllers are packaged two to a memory card and support two of the four POWER4 chips on a module (as shown by the placement of memory slots in Figure 2-6). A pSeries 690 Model 681 has two memory slots associated with each module. No memory cards, one, or two memory cards can be installed per module. Memory controllers can each have either one or two ports to memory.

The memory controller is attached to the L3 MLD chips, with each memory controller having two 16-byte buses to the L3, one in each direction. These buses operate at one-third of the processor speed.

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

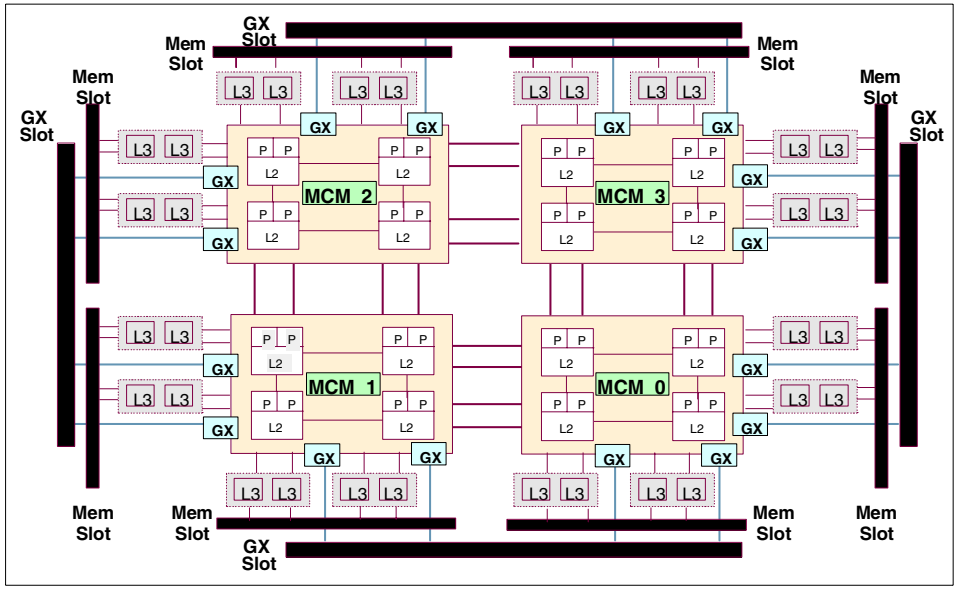


Figure 2-6 Multiple MCM interconnection

If one memory card or two unequal size memory cards are attached to a module, then the L3 caches attached to the module function as two 64 MB L3 caches.

The two L3 caches that act in concert are the L3 caches that would be in front of the memory card. (Note that one memory card is attached to two chips.)

2.4.8 Hardware data prefetch

In addition to out of order execution and the ability to sustain multiple outstanding cache misses, POWER4 systems provide additional hardware to hide memory latency by prefetching data cache lines from memory, L3 cache, and L2 cache transparently into the L1 data cache. The POWER4 processor can prefetch *streams*, which are defined as a sequence of loads from storage that reference at least two or more contiguous data cache lines, in order, either in an ascending or descending pattern (the loads themselves need not be monotonically increasing or decreasing). Eight such streams per processor are supported. Hardware prefetching is triggered by data cache line misses, and then paced by loads to the stream. Pacing prefetches by monitoring loads provides a consumption-driven method to provide timely and effective prefetching.

The prefetch engine typically initiates a prefetch stream after detecting misses to two consecutive cache lines. Figure 2-7 shows the sequence of prefetch operations in the steady-state after a ramp-up phase. L1 prefetches are one cache line ahead of the cache line currently being loaded from in the program. L2 prefetches, which prefetch cache lines from the L3 cache (or memory) into the L2 cache, are five cache lines ahead, which is sufficient to hide the latency between the L3 cache and the L2 cache. Finally, L3 prefetches, which prefetch data from the memory into the L3 cache, are 17 to 20 lines ahead of the current cache line being loaded from in the program. L3 prefetches are usually done as logical 512-byte lines, for example, four 128-byte lines at a time. This increases the efficiency of the transactions and need only be performed every fourth line referenced.

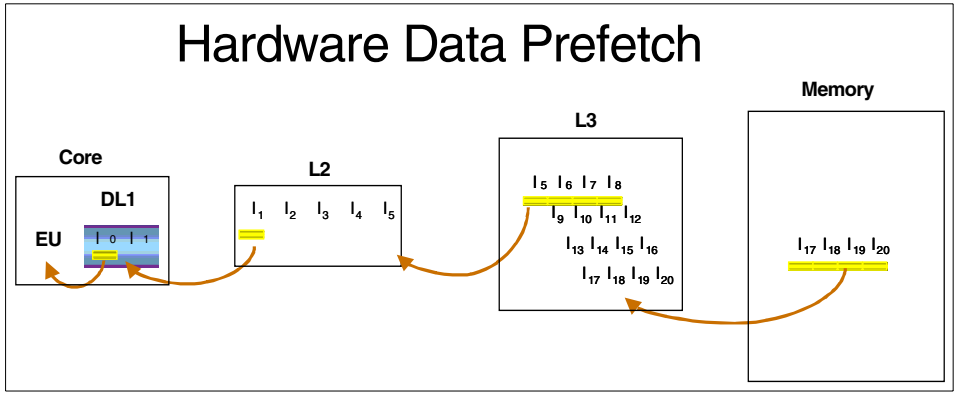


Figure 2-7 Hardware data prefetch operations

To begin a stream, the prefetch engine either increments or decrements the real address of a cache line miss (so that it is the address of the next or the previous cache line) and places that address in the prefetch filter queue. The decision whether to increment or decrement is based upon the offset within the line corresponding to the load operand. As new cache misses occur, if the real address of the new cache miss matches one of the guessed addresses in the filter queue, a stream has been detected. If the prefetch engine has fewer than eight streams active, the new stream is installed in the prefetch request queue and the prefetching ramp-up sequence is begun. Once placed in the prefetch request queue, a stream remains active until it is aged out. Normally a stream is aged out when the stream reaches its end and other cache misses displace its entry in the filter queue.

The hardware prefetch engine issues prefetches only within a real page since it does not carry information about the effective to real address mapping. Hence, page boundaries curtail prefetching and end streams. If a prefetchable storage reference pattern crosses a page boundary, a new stream is started at the beginning of the new real page according to the startup logic described above. Since this results in a performance penalty that can be significant, POWER4 systems support, in addition to the standard 4 KB page, an additional page size of 16 MB (concurrently with 4 KB pages). Applications which place data into 16 MB pages can significantly improve prefetching performance by essentially eliminating this penalty associated with stream re-initialization at page boundaries.

2.4.9 Memory/L3 cache command queue structure

Each L3 cache controller has eight all-purpose coherency processors and eight special-purpose coherency processors. In the desired mode in which four L3 cache arrays are operating in shared mode and therefore appear as one logical, interleaved L3 cache, there are a total of 32 all-purpose coherency processors and 32 special-purpose coherency processors. Coherency processors are busy processing a request until the operation is complete. Special-purpose coherency processors handle primarily cache line writes to memory. For many workloads, the majority of requests to the L3 cache will be read requests or data prefetch requests, and hence the all-purpose coherency processors performance will essentially determine the overall performance of the L3 cache and memory subsystem.

2.5 I/O structure

Figure 2-8 on page 23 shows the I/O structure in POWER4 systems. The POWER4 GX bus is attached to a Remote I/O (RIO) bridge chip. This chip transmits the data across two one-byte wide RIO buses to PCI Host Bridge (PHB) chips.

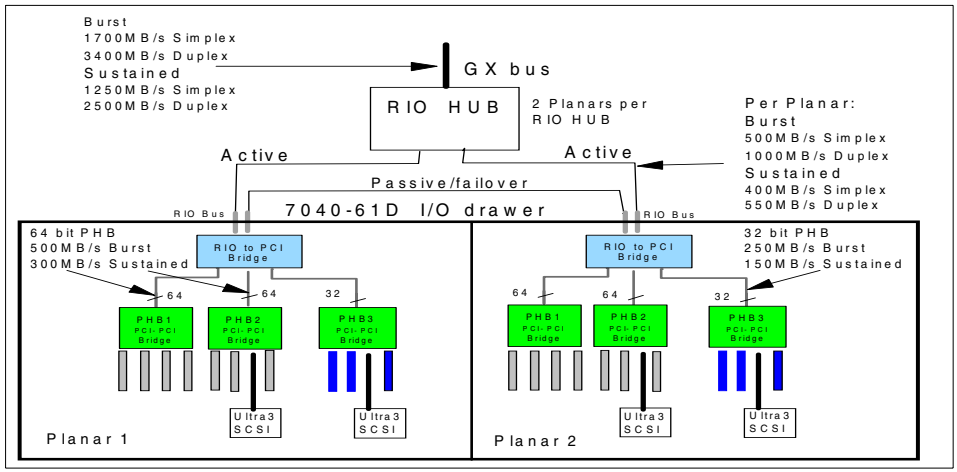


Figure 2-8 I/O structure

Two separate PCI buses attach to PCI-PCI bridge chips that further fan the data out across multiple PCI buses. When multiple nodes are interconnected to form clusters of systems, the RIO Bridge chip is replaced with a chip that connects to the switch. This provides increased bandwidth and reduced latency compared to switches attached using the PCI interface.

2.6 The POWER4 Performance Monitor

The POWER4 design includes powerful performance monitoring facilities that can collect data on various system events and provide valuable performance data. The performance monitor facilities enable the counting of up to eight concurrent events, and counting can be started and stopped and the results retrieved by software. Counters can be frozen until a user-selected trigger event occurs and then incremented, or they can be incremented until a trigger event occurs and then be frozen. It enables the monitoring of classes of instructions selected by the instruction matching facility, or the random selection of instructions for detailed monitoring, as well as count start/stop event pairs that exceed a selected time-out value threshold.

AIX 5L contains application program interface (API) code for customer use in enabling and using the performance monitor facilities from their applications. Use of the POWER4 Performance Monitor API is discussed in Section 5.3, “The performance monitor” on page 101.

3

POWER4 system performance and tuning

This chapter provides a guide for Fortran or C programmers who have a general understanding of tuning techniques to tune their programs for POWER4. The following major topics are discussed within:

- Tuning for scientific and technical numerically intensive applications
- Tuning for non-numerically intensive or commercial applications
- General system level aspects of tuning

For more information on the general aspects of tuning, see *Optimization and Tuning Guide for Fortran, C, and C++*, SC09-1705.

3.1 Tuning for numerically intensive applications

Before describing specific tuning techniques, this section first reviews the tuning process and discusses those aspects of POWER4 microarchitecture that particularly influence the performance of numerically intensive scientific and technical programs.

3.1.1 The tuning process for numerically intensive applications

For an existing program, the following steps summarize the tuning process in approximate order of importance. Taking these guidelines into account when writing a new program should significantly reduce the need for tuning at a later stage.

1. If I/O is a significant part of the program, tuning for this is an important but separate activity from computational tuning. Some guidelines for efficient I/O coding are given in Chapter 4, “Optimizing with the compilers” on page 69.
2. Use the best set of compiler optimization flags. See Section 4.1, “POWER4-specific compiler options” on page 69.
3. Locate the hot spots in the program (profiling). This step is very important. Do not waste time tuning code that is infrequently executed.
4. Use the MASS library and ESSL (and maybe other performance-optimized libraries) when possible. These libraries are discussed in Chapter 6, “Performance libraries” on page 113.
5. Make sure that the generic *common sense* tuning guidance given in Chapter 4, “Optimizing with the compilers” on page 69 has been followed.
6. Hand tune the code to the POWER4 design. This will be discussed in the rest of this chapter.

3.1.2 Hand tuning overview for numerically intensive programs

Hand tuning for cache-based RISC architecture computers such as a pSeries 690 Model 681 is divided into two parts:

1. Avoid the negative.

   Tune to avoid or minimize the impact of a cache and memory subsystem that is necessarily slower than the computational units. Basic techniques for doing this include:

   – Stride minimization
   – Encouragement of data prefetch streaming
   – Avoidance of cache set associativity constraints
   – Data cache blocking

2. Exploit the positive.

   Tune to maximize the utilization efficiency of the computational units, in particular the floating-point units.

   Techniques for CPU tuning include:

   – Unrolling inner loops to increase the number of independent computations in each iteration to keep the pipelines full.

– Unrolling outer loops to increase the ratio of computation to load and store instructions so that loop performance is limited by computation rather than data movement.

It will be assumed that the reader has a basic understanding of the concepts of:

- Loading and storing (into and from floating-point registers)
- Stride and what determines it in Fortran and C loops
- Loop unrolling

3.1.3 Key aspects of the POWER4 design

This section covers those parts of the POWER4 design that are relevant to tuning the performance of floating-point intensive applications. Additional details are provided in Chapter 2, “The POWER4 system” on page 5.

The components described here are:

- The L1, L2, and L3 caches
- The ERAT and TLB
- Data prefetch streaming
- Floating point and load/store units

The level 1, 2, and 3 caches

A brief description of the caches follows.

The L1 instruction cache

The L1 instruction cache (I-cache) is 64 KB and is direct mapped. It can be of considerable importance for commercial applications such as transaction processing.

For computationally intensive applications, it does not usually have a significant impact on performance because such applications usually consist of highly active loops (DO-loops in Fortran or for-loops in C) that contain relatively few instructions. The amount of data handled is usually much larger than the space taken by the instruction stream.

Tuning for the I-cache consists mainly of ensuring that active loops do not contain a very large number of instructions.

The L1 data cache

Each processor has a dedicated 32 KB L1 data cache. It is two-way set associative with a first-in-first-out (FIFO) replacement algorithm. These concepts and the implications for tuning are explained fully in “Structure of the L1 data cache” on page 28.

The L2 cache

Each POWER4 chip has a dedicated L2 (data and instruction combined) cache 1440 KB in size. The pSeries 690 Model 681 and pSeries 690 Turbo have two processors per chip that share the L2 cache. The pSeries 690 HPC feature has one processor per chip that, therefore, has the L2 cache dedicated. Cache coherence is maintained across the entire pSeries 690 Model 681 system at the L2 level.

The L3 cache

Four POWER4 chips are combined into an multi-chip module (MCM) each of which has a 128 MB Level 3 cache. For pSeries 690 Model 681 systems with more than one MCM, the L3 caches on remote MCMs are accessible with a modest performance penalty. This applies even if the system is partitioned using LPAR.

The L3 cache is eight-way set associative.

General cache considerations

The high bandwidth from L2 to L1 is more than enough to feed the floating-point units. Thus, the primary difference from a performance point of view between L1 and L2 is latency. A load/store between floating-point register and L1 has a latency of about 4 cycles; between registers and L2 it is approximately 14 cycles.

The tuning recommendation for dense (as opposed to sparse) computation is therefore to block data for the L2 cache and to structure the data access (array leading dimension, for example) for the L1. This tuning advice will be explained in subsequent sections.

An application whose performance is dominated by latency (such as the pointer-chasing code described in Section 3.1.6, “Cache and memory latency measurement” on page 47) may need to be blocked for L1 for best performance.

Structure of the L1 data cache

There are two concepts, *cache lines* and *set associativity,* that are key to understanding the structure of the pSeries 690 Model 681 data cache, discussed in the following sections.

Cache lines

Conceptually, memory is sectioned into contiguous 128-byte lines, each one starting on a cache-line boundary whose hardware address is a multiple of 128. The cache is similarly sectioned and all data transfer between cache and memory is in units of these lines.

If, for example, a particular floating-point number is required to be copied (loaded) into a floating-point register to be used in a computation, then the whole cache line containing that number is transferred from memory to cache.

Set associativity

The L1 data cache is mapped onto memory, as shown in Figure 3-1. Each column in one of the diagrams is called a *congruence class,* and any particular line from memory may only be loaded into a cache line in a particular congruence class, that is into one of only two locations.

The POWER4 L1 data cache is two-way set associative with 128 congruence classes. Each cache line is 128 bytes. In total, the L1 data cache can contain 32,768 bytes of data.

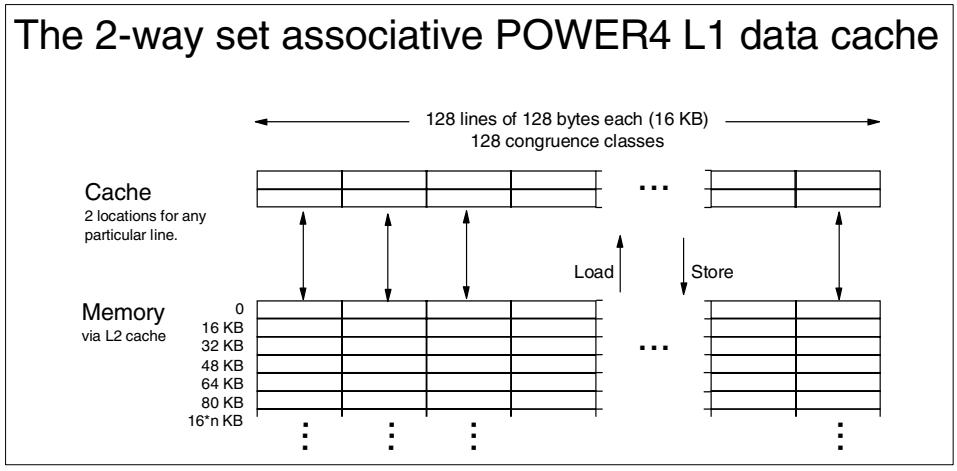


Figure 3-1 The POWER4 L1 data cache

When a new line is loaded into L1, it displaces the oldest of the two lines in the congruence class (FIFO replacement).

The set associative structure of the cache can lead to a reduction in its effective size. Suppose successive data elements are being processed that are regularly spaced in memory (that is with a constant *stride*). With the POWER4 cache, the worst case is when the stride is exactly 16 KB or a multiple of 16 KB. In this case, all elements will lie in the same congruence class and the effective cache size will be only two lines. This effect happens, to a lesser extent, with any stride that is a multiple of a power of 2 less than 16 KB.

Characteristics of the L2 cache

The size of the L2 cache is 1440 KB per POWER4 chip, and this is shared between the two processors in the chip. As with the L1 data cache, the cache line size is 128 bytes. The replacement policy is pseudo-LRU (least recently used) so frequently accessed cache lines should be readily maintained in the cache. The L2 cache is a combined data and instruction cache. Instruction caching aspects of the L2 cache are not considered here.

The L2 cache is divided into three equal parts, each under control of a separate L2 cache controller. The particular portion a line is stored is in is determined from the real memory address using a hashing algorithm. Sixteen consecutive double-precision Fortran array elements (138 bytes) are held in the same cache line, and therefore under control of the same cache controller. The 17th element will be in a different cache line and the hashing algorithm guarantees it will be stored under control of a different cache controller. This has implications for the optimization of store processing when accessing arrays sequentially.

Loads are processed by loading a cache line from L2 into the L1 data cache 32 bytes at a time. This means that the L2 cache can load the equivalent of four double-precision floating-point data elements per cycle, which is double the capability of the processor to issue load instructions. Prefetched data will be loaded into the L1 at the same rate.

The L2 cache is a store-in cache, which means that stores are always written to the L2 cache whether there is a hit in the L2 cache or not. This is in contrast to the store-through L1 data cache where a store miss will not result in the data being written into this level. Stores are passed to the L2 cache interface 8 bytes at a time. The rate at which stores can be accepted by the interface depends on whether the stores are to the same L2 section or not. See 3.1.7, “Selected fundamental kernel performance within on-chip cache” on page 49 and 3.1.8, “Other tuning considerations” on page 51 for discussions concerning store performance.

Once the store has been accepted by the interface unit, the store instruction is released by the processor, freeing up resources. Note that store data is never written into the caches until they have been completed, such as, made visible to the program, by the processor. Completion in this sense is separate and later than execution.

In the case where the cache line is not already present in the L2 cache (an L2 cache miss), then it must be loaded from either memory, another chip's L2 cache, or the L3 cache to ensure that the L2 cache contains the latest copy of this cache line. Depending on whether the line already exists in another L2 cache on another chip, some coherency processing may be required to ensure that the local chip has permission to modify the line. Once the line is updated in the L2 cache, then it is marked as *dirty* and will eventually be written out to memory and potentially to L3 cache.

The ERAT and TLB

The instruction stream addresses data using a 64-bit effective addresses (EA). To access the data in memory, the EA is first converted to an 80-bit virtual address (VA) and then to a 64-bit real address (RA). The translation lookaside buffer (TLB) holds the 1024 entries organized in a 4-way set-associative structure. It contains previously translated EAs to RAs and other information on a page basis, either 4 KB or 16 MB page sizes. For 4 KB pages, the TLB addresses a total of 4 MB of (not necessarily contiguous) memory. Data that is within a page addressed by the TLB will not take the overhead of a TLB miss when the EA is accessed. The ERAT is effectively a cache for the TLB. It is a 256-entry 2-way set-associative array. All ERAT entries are based on 4 KB pages pages, even if 16 MB pages are used.

The TLB addresses a greater amount of memory (at least 4 MB) than the L2 cache (1.41 MB). Therefore, any program that is tuned to take any advantage of the L2 cache is unlikely to experience serious overheads due to TLB misses (this is different from POWER3 where the TLB addressed 1 MB but L2 was 4 MB or 8 MB). It is still possible on POWER4 to construct situations involving high strides that will create a TLB miss and not a cache miss, but tuning for the TLB is beyond the scope of this document. For codes in which a blocking strategy is used, empirically determining the blocking factors will also include ERAT and TLB effects.

Prefetch data streaming

The POWER4 design provides a prefetch mechanism that can identify streams as defined in Section 2.4.8, "Hardware data prefetch" on page 21. Each POWER4 microprocessor can support up to eight independent prefetch streams. In contrast, the POWER3 processor supported four independent prefetch streams. Note that there is no prefetch on store operations.

The prefetch mechanism is based on real addresses. Therefore, whenever a real address reference crosses a page boundary, the prefetch mechanism is stopped. Two consecutive cache line misses on the subsequent page are required to restart the mechanism. Large pages are supported in AIX 5L only through shared memory segments. More general large page support will be available in a future release of AIX 5L. There can be performance benefits because the prefetch mechanism can operate over much larger arrays before crossing page boundaries.

The floating-point units and maximum GFLOPS

To achieve the maximum floating-point rate possible on a single pSeries 690 Model 681 processor, the delays due to the memory subsystem have to be eliminated and the program must reside in the L1 cache.

The following key facts summarize the way the FPUs perform:

- A single pSeries 690 Model 681 processor has two FPUs (sharing a single L1 cache) that can operate independently. The two FPUs see only floating-point registers. There are a total of 72 physical registers. An assembler program can address 32 architected registers and these are mapped onto the physical registers through a hardware process known as *renaming*. The 72 physical registers serve both FPUs. They all have 64 bits. floating-point computation is carried out only with data in these registers. They are all 64-bits wide. All floating-point arithmetic instructions are register-to-register operations, logically using only floating-point registers as sources and targets.
- Data is copied into the registers from the L1 cache (loaded) and copied back to the L2/L1 cache (stored) by two load/store units.
- For data in the L1 or L2 cache, loads or stores of floating-point double-precision (REAL*8) variables can be done by each load/store unit at the rate of one per cycle, but for loads, there is a latency before the FPU can use the data for computation. This latency is approximately four cycles if the data is in L1, or 14 cycles if it is in L2 but not L1. For maximum performance, it is important that loaded data is in L1, because the compiler will assume the L1 latency.
- Single precision (REAL*4) variables use the same register set as REAL*8. Each variable occupies an entire 64-bit register (there is no ability to pack two REAL*4s into a single register).
- The basic computational floating-point instruction is a double-precision multiply/add, with variants multiply/subtract, negative multiply/add, and negative multiply/subtract. There are also single precision variants.

A single add, subtract, or multiply (not divide) is done using the same hardware as a multiply/add and takes the same amount of time. A multiply/add counts as two floating-point operations. For example, a program doing only additions might run at half the MFLOPS rate of one doing alternate multiplies and adds.

The assembler acronym for the double-precision floating-point multiply/add is FMA. This term will be used extensively as shorthand for any of the variants of this basic floating-point instruction.

The computational part of an FMA takes six cycles.

The worst case would be a sequence of wholly dependent 6-cycle FMAs (where a result of one FMA is needed by the next) where only one of the FPUs would be active. This would run at the rate of one FMA per six cycles.

A sequence of *independent* FMAs, however, can be pipelined and the throughput can then approach the peak rate of two FMAs per cycle (one per FPU).

- Divides are very costly and are not pipelined.
- A fundamental aspect of RISC architecture is that the functional units can run independently. Therefore, FMAs can run in parallel with load/stores and other functions.

Conditions for approaching peak GFLOPS

When considering a numerically intensive loop, the following applies to the instruction stream within the loop:

- Operate efficiently within L1 and L2 caches.
- No divides (or square roots or function calls and so on).
- To achieve peak megaflops, loops must contain FMAs only, therefore using floating-point adds or subtracts with multiplies.
- FMAs must be independent and at least 12 in number to keep two pipes of depth six busy.
- The loop should be *FMA-bound*. That is, cycles needed for instructions other than FMAs (mainly load/stores) should be less than that needed for FMAs so that they can be overlapped with FMAs and effectively hidden. In principle, they could be equal to the FMA cycles, but, in practice, peak performance is approached most easily if there are fewer.

- The performance of floating-point intensive applications on 1.3 GHz POWER4 is typically between two and three times faster than on 375 MHz POWER3 but is usually somewhat less than that as indicated merely by a comparison of clock rate ratios. This is because it is more difficult to approach peak performance on POWER4 than on POWER3 because of factors such as:
  - The increased FPU pipeline depth
  - The reduced L1 cache size
  - The higher latency (in terms of processor cycles) on the higher level caches

3.1.4 Tuning for the memory subsystem

There are four basic tuning techniques (some of these techniques may be done by the compilers) that will be discussed in this section, namely:

- Stride minimization
- Encouragement of data prefetch streaming
- Structuring for L1 set associativity
- Data cache blocking

Stride minimization

Sequential accessing of data is beneficial for two reasons:

- It ensures that, once a line is loaded into cache, all other operands in the same cache line will also be referenced. If the data is accessed with a large stride, less data from the cache line will be referenced. If the stride is greater than 16 for double precision words, or 32 for single precision words, only one number in each line will be referenced. There will then be a high probability that, when the other numbers in the line are accessed at a later stage, the line will no longer be in cache leading to the overhead of a cache miss.
- Sequentially accessed (forwards or backwards - stride 1 or -1) data can start one of the eight hardware prefetching streams. Other low-value strides may also start a prefetching stream provided that they are contiguous cache line references.

Fortran arrays are stored in memory in column major order, C arrays in row-major order. Coding nested loops to access data the *right* way so that multi-dimensioned arrays are accessed sequentially (stride 1) is the most basic tuning technique of all.

The following examples illustrate this:

```
Correctly tuned stride 1 sequential access

Fortran
do i=1,n
  do j=1,n
    a(j,i)=a(j,i)+b(j,i)*c(j,i) ! Left subscript same as inner loop var.
  enddo
enddo

C
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    {a[i][j]+=b[i][j]*c[i][j];} /* Right index same as inner loop variable
*/
```

If the nesting order of the loops is changed, the arrays are then accessed with a large stride.

In this simple case, the compilers will reverse the order of the loops for you. However, it is sound coding practice not to rely on the compiler and always to code loops in the correct order.

It is, of course, not always possible to code so that all arrays are accessed stride 1. For example, the following is a typical matrix multiply code fragment:

```
do i=1,n
   do j=1,n
     do k=1,n
       d(i,j)=d(i,j)+a(j,k)*b(k,i)
     enddo
   enddo
 enddo
```

No matter how the loops are coded, one or more arrays will have non-unit stride. In this case, data cache blocking may be necessary as described in “Data cache blocking” on page 38.

Encouragement of data prefetch streaming

Data prefetching is implemented in the POWER4 processor hardware so that prefetching is transparent to the application: it does not require any software assistance to be effective. There are, however, situations where the performance of an application can be improved with code tuning to more fully exploit the capabilities of the hardware prefetch engine. These situations arise when:

- There are too few or too many streams in a performance-critical loop
- The length of the streams in a performance-critical loop is too short.

The POWER4 data prefetch design was optimized for loops with four to eight concurrent hardware streams. Figure 3-2 on page 37 shows the performance for a series of loops with one to eight streams per loop. Note that increasing the number of streams from one to eight can improve data bandwidth out of the L3 cache and memory by up to 70 percent, and that most of the improvement comes from increasing the number of streams from one to four.

The number of streams in a loop can be increased by *fusing* adjacent loops (a capability which the XL compilers posses with the -qhot optimization) or by midpoint bisection of the loop. Fusing simply means combining two or more compatible loops into a single loop. For example:

```
DO I=1,N
  S = S + B(I) * A(I)
ENDDO
DO I=1,N
  R(I) = C(I) + D(I)
ENDDO
```

may be combined into:

```
DO I=1,N
  S = S + B(I) * A(I)
  R(I) = C(I) + D(I)
ENDDO
```

Midpoint bisection of a loop doubles the number of streams but halves its vector length. Consider the standard dot-product loop:

```
DO I=1,N
  S = S + B(I) * A(I)
ENDDO
```

This loop contains two streams corresponding to the two arrays on the right hand side of the expression. Midpoint bisection doubles the number of streams by starting two more streams at the halfway point of each of the arrays, as shown in the following:

```
NHALF = N/2
S0=0.D0
S1=0.D0
DO I=1,NHALF
 S0 = S0 + A(I)*B(I)
 S1 = S1 + A(I+NHALF)*B(I+NHALF)
ENDDO
IF(2*NHALF.NE.N) S0 = S0 + A(N)*B(N)
S = S0+S1
```

For this example, in situations where the data is being reloaded from beyond the L2 cache, the break-even vector length is approximately 220. Loops with vector lengths beyond 220 which have been midpoint bisected as shown have superior performance by up to 20 percent.

When a loop has more than eight streams, reducing the number of streams per loop may also boost overall performance. Since only eight of the streams can be prefetched (as there are only eight prefetch request queues), streams beyond eight will be reloaded on a demand basis. It may be possible to split the loop into two or more loops, each with eight or fewer streams. This may or may not involve introducing extra temporary vectors to allow the loop to be split. In any event, profiling or loop timing should always be done within the application to check whether the tuning, either to increase or decrease the number of streams per loop, had a positive overall effect on performance.

Increasing vector length can significantly improve performance as well, simply due to the fact that there is a fixed overhead resulting from loop unrolling and prefetch stream acquisition. In some cases, increasing the vector length of an application is under direct control of the programmer, such as in those in which explicit integration of a variable permits operations on groups of entities of arbitrary size. In these situations, there is often a trade-off between cache reuse and vector length, so it is again advisable to determine the optimal vector length empirically.

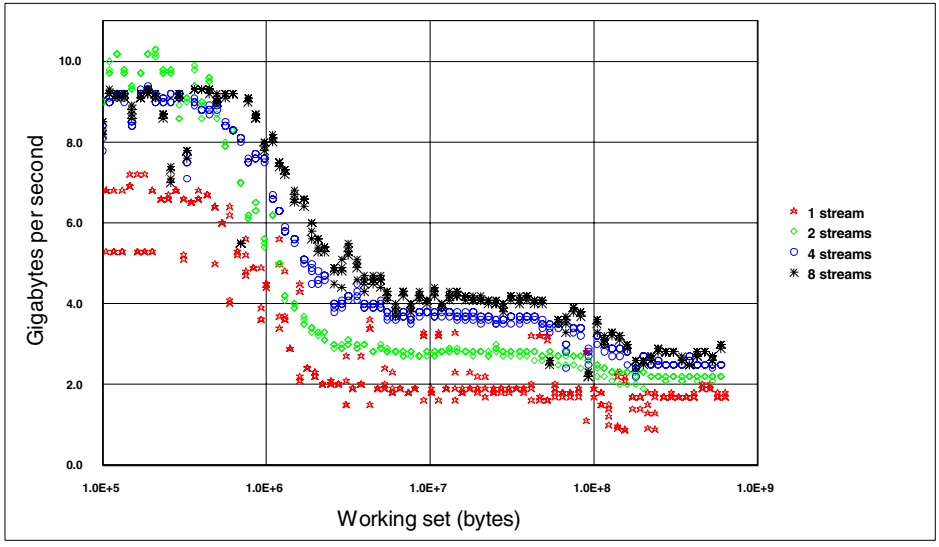


Figure 3-2 POWER4 data transfer rates for multiple prefetch streams

Structuring for L1 set associativity

In cases where it is not possible to access arrays sequentially, the stride is typically determined by the leading dimension of the array. For example, consider the following loop.

```
real*8 a(2048,75)
  .
  .
do i=1,75
  a(100,i)=a(100,i)*1.15
enddo
```

This updates the 100th row of a Fortran array. The row is 75 elements long, so 75 cache lines will be accessed (if this were a column, only 5 cache lines would be accessed). The L1 cache has a total of 256 lines. So, if, for example, this section of the array has been recently accessed, you might hope to find these lines in the cache.

However, the leading dimension of the array determines the stride for array A to be 2048 REAL*8 numbers or 16384 bytes. These map to a single congruence class in the L1 cache so that only two elements of A can be held in L1. At best, only the first two lines (of the 75) accessed could possibly be in the L1 cache.

Changing the leading dimension to 2064 (that is, 2048 plus a single cache line of 16 REAL*8 numbers) would cause the 75 lines to map to different congruence classes and all 75 lines would fit. With a two-way set associative cache, a leading dimension of 2056 (2048 plus half a cache line) would also work. But 2046 would work for any level of set associativity, including direct mapping.

The general rule is:

Avoid leading dimensions that are a multiple of a high power of two.

Any odd number of cache lines is ideal, that is for 128-byte cache lines, any odd multiple of 16 for REAL*8 arrays or any odd multiple of 32 for REAL*4 arrays.

Data cache blocking

The data cache blocking idea is basic: if your arrays are too big to fit into cache, then process them in blocks that do fit into cache. Generally with POWER4, it is the 1440 KB L2 cache that needs to be large enough to contain the block.

There are two factors that determine if using blocking will be effective:

- When all arrays are accessed stride 1
- When each data item is used in more than one arithmetic operation

The combination of these factors produces four scenarios:

- All arrays are stride 1 and no data reuse. There is no benefit from blocking.

```
! Summed dot products. Note each element of A and B used just once.
do j=1,n
  do i=1,n
    s = s + a(i,j)*b(i,j)
  enddo
endd
```

- Some arrays are not stride 1 and there is no data reuse. Blocking will be moderately beneficial.

```
! Summed dot products with transposed array.
do j=1,n
  do i=1,n
    s = s + a(j,i)*b(i,j)
  enddo
endd
```

- All arrays are stride 1 and there is much data reuse. Blocking will be moderately beneficial.

```
! Matrix multiply transpose.
do i=1,n
   do j=1,n
     do k=1,n
       d(i,j)=d(i,j)+a(k,j)*b(k,i)
     enddo
   enddo
 enddo
```

- Some arrays are not stride 1 and there is much data reuse. Blocking will be essential.

```
! Matrix multiply.
do i=1,n
   do j=1,n
     do k=1,n
       d(i,j)=d(i,j)+a(j,k)*b(k,i)
     enddo
   enddo
 enddo
```

The following example shows how matrix multiply should be blocked.

```
!3 blocking loops
do ii=1,n,nb
  do jj=1,n,nb
    do kk=1,n,nb
!
!     In-cache loops
      do i=ii,min(n,ii+nb-1)
         do j=jj,min(n,jj+nb-1)
           do k=kk,min(n,kk+nb-1)
             d(i,j)=d(i,j)+a(j,k)*b(k,i)
           enddo
         enddo
      enddo
!
    enddo
  enddo
enddo
```

In this example, the size of the blocks of each matrix is NB x NB elements. For blocking to be effective, it must be possible for the L2 cache to hold three such blocks. On an pSeries 690 HPC, the process will have the whole L2 cache available. On a non-HPC model, it may be sharing L2 with another process or thread so that only half the cache is available. The relatively complicated structure of the cache may also require NB to be smaller than a simple size calculation would suggest. In practice, the right way to fix NB is to vary it and measure the performance to achieve the optimum value. However, if a non-HPC machine is being used, these measurements should not be run stand-alone if, in practice, the application will be run when another application or thread is competing for L2.

Note that, although this code leads to in-cache performance, it does not lead to maximum GFLOPS. The reason for this is explained in the next section.

Blocking usually needs to be done by hand rather than leaving it to the compiler.

3.1.5 Tuning for the FPUs

In contrast to tuning for the memory subsystem, the compiler is generally very successful at tuning for the FPUs and often there is little extra that can be achieved by hand tuning. Some exceptions to this are highlighted in this section.

Inner loop unrolling and instruction scheduling

To keep the FPU pipelines busy, the following conditions must apply in the inner instruction loop:

- There must be enough (at least 12) independent FMAs in the compiled loop.
- Loads must precede FMAs in the instruction stream by at least four cycles to overcome the L1 cache latency.
- The total number of architected registers used must not exceed 32. If this happens, the compiler must generate *spill* coding that stores the register values and reloads them later.
- The number of rename registers needed must not exhaust the hardware pool available.
- The number of loads and stores must be less than or equal to the number of FMAs, otherwise the load/store time dominates.

Techniques for dealing with the last item - load/store bound loops - are discussed in “Outer loop unrolling to increase the FMA to load/store ratio” on page 41.

The basic technique for achieving multiple independent FMAs is inner loop unrolling. While this can be done by hand, it produces convoluted coding and usually there is no point since the compiler will do it for you efficiently and reliably. If you unroll manually, there is a danger that the compiler will unroll again. This may cause register spilling or other overheads and it may be beneficial to use the -qnounroll compiler flag.

To help the compiler to avoid register spilling, you should avoid coding too many unnecessary temporary scalar variables in the loop.

Apart from the items noted, you must rely on the compiler to produce the optimum instruction stream unless assembler language is used. This is easier than might be imagined, since advantage can be taken of the -S compiler option. This will produce a file from the Fortran with a .s suffix that may be assembled with the **as** command and linked into the program. Identifying the inner loop of the routine and editing it to improve the instruction stream is then quite possible for the experienced programmer without the necessity to fully learn assembler language. Nevertheless, most programmers will not choose to do this and further advice is beyond the scope of this publication.

Outer loop unrolling to increase the FMA to load/store ratio

In cases where the inner loop is load/store bound (loads + stores greater than FMAs) it may be possible to significantly improve performance by increasing the ratio of FMAs to loads and stores in the loop. This is only possible in *data re-use* cases and the basic technique is usually outer loop unrolling.

This section considers two cases: one simple loop that the compiler does not handle successfully, and then blocked matrix multiply coding.

Simple loop benefitting from hand unrolling

Consider the following loop:

```
do i = 1,n
   do j = 1,n
      y(i) = y(i) + x(j)*a(j,i)
   end do
end do
```

This loop is already well structured in that the inner loop both has stride 1 and is a sum-reduction (`y(i)` is a scalar). This means that the number of loads and stores needed in the inner loop is minimized because the scalar value `y(i)` can be held in a single register and stored just once after the inner loop is complete. Iteration of the inner loop needs just two loads (for `x(j)` and `a(j,i)`) and zero stores. If the loop order were reversed (with the inner loop on I), there would be two loads needed (for `y(i)` and `a(j,i)`) plus one store (for `y(i)`). In addition, there would be poor stride on `a(j,i)`.

However, the loop is load/store bound because there are more load and store instructions than FMAs. Therefore, as it stands, the performance of this loop will be limited by the effective rate at which the load/store unit can operate.

The compiler will successfully unroll the inner loop on J. This is necessary in order to populate the inner loop with independent FMAs rather than dependent ones. However, this does nothing to alter the FMA to load/store ratio.

The solution, in this case, is to unroll the outer loop on I. With this simple loop, the compiler may optimize the code for you with the -qhot option, but, generally, it is more reliable to do outer-loop unrolling by hand.

The following code shows the loops unrolled to depth 4 (tidy-up coding omitted for cases where n is not a multiple of 4).

```
do i = 1,n,4
   s0 = y(i)
   s1 = y(i+1)
   s2 = y(i+2)
   s3 = y(i+3)

   do j = 1,n
      s0 = s0 + x(j)*a(j,i)
      s1 = s1 + x(j)*a(j,i+1)
      s2 = s2 + x(j)*a(j,i+2)
      s3 = s3 + x(j)*a(j,i+3)
   enddo
```

```
      y(i)   = s0
      y(i+1) = s1
      y(i+2) = s2
      y(i+3) = s3
   enddo
```

Note the introduction of the temporary scalar values, S0, S1, S2, and S3. This is very important because usually, whenever the inner loop contains anything more complicated than a single subscripted scalar, the compiler may not recognize that they are scalars and may generate unnecessary loads and stores. Generally speaking, introducing temporary scalars to make the scalar nature of array elements clear to the compiler is good coding practice. This does not contradict previous advice to avoid the introduction of unnecessary scalar variables. In this case, it is necessary for the compiler to recognize that `y(i)`, `y(i+1)`, `y(i+2)`, and `y(i+3)` are scalar in the inner loop.

The load/store to FMA ratio is reduced because the element `x(j)` is now re-used three times in the inner loop. So, now, for four of the original iterations, there are five loads rather than eight. Clearly, as the unrolling depth increases, the load/store to FMA ratio reduces asymptotically from two to one.

The actual performance depends on the compiler optimization flags and the depth of hand-unrolling. Selected results for a 1.1 GHz machine are shown in Figure 3-3 on page 44. The label depth refers to the unrolling depth of the outer loop of the hand-tuned code. The x-axis refers to dimension n. At n=64 the data just exceeds the size of the L1 cache. Without hand-unrolling, the compiler does not take advantage of the L1 cache. The top two lines are with different compilers but the main reason for the difference in performance is that the top line is compiled for -qarch=pwr4 rather than pwr3.

Note the “L1 cache peak” for the (top three) hand unrolled lines as the array size is increased. The untuned code (the bottom line) does not show this peak.

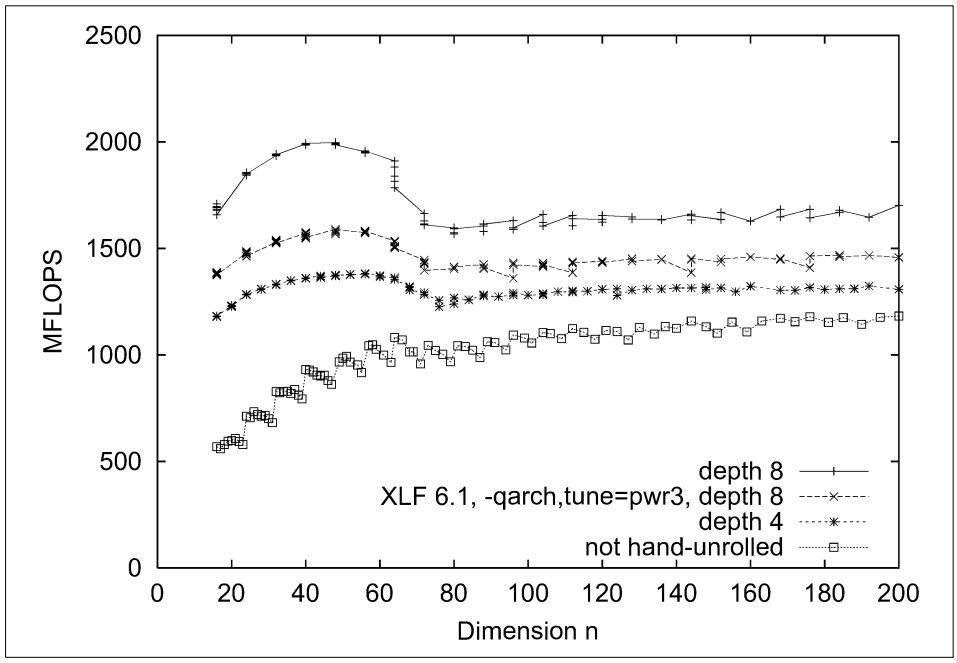


Figure 3-3 Outer loop unrolling effects on matrix-vector multiply (1.1GHz system)

M x N unrolling for matrix multiply

The following is the heart of the blocked matrix multiply code. The blocking loops have been omitted for clarity.

```
do i=ii,min(n,ii+nb-1)
  do j=jj,min(n,jj+nb-1)
    do k=kk,min(n,kk+nb-1)
      d(i,j)=d(i,j)+a(j,k)*b(k,i)
    enddo
   enddo
enddo
```

As with the previous example, having the inner loop on `k` (rather than `i` or `j`) minimizes the number of loads and stores. The array element `d(i,j)` is a scalar in the inner loop, since it does not depend on the inner loop variable, `k`, so the inner loop is a sum reduction. The scalar may be held in a register during iteration and only stored after the inner loop is complete. The inner loop requires just two loads (for `a(j,k)` and `b(k,i)`) whereas if `i` or `j` were the inner loop variable, there would be two loads plus one store.

Let us recast the loop so as to make the scalar nature of `d(i,j)` explicit.

```
do i=ii,min(n,ii+nb-1)
  do j=jj,min(n,jj+nb-1)
    s = d(i,j)
    do k=kk,min(n,kk+nb-1)
      s = s + a(j,k)*b(k,i)
    enddo
    d(i,j) = s
   enddo
enddo
```

As with the previous example, introduction of the variable S is sound coding practice.

Although the number of load/stores in the inner loop has been minimized, the loop is nevertheless clearly load/store bound. There are two loads and only one FMA. This can be effectively transformed into an FMA-bound loop by unrolling the outer two loops. If the outer loop is unrolled to depth M and the middle loop to depth N, then the number of loads is `m+n` and the number of FMAs is `m*n`. Unrolling 2x2 makes the loop balanced (load/stores = FMAs). Anything more makes it FMA-bound. The following code shows 5x4 unrolling. This requires 29 architectured registers (20 for the holding of the 20 partial sums in the temporary scalar variables and 9 for holding the elements of A and B). Anything higher would exceed the number of architectured registers.

```
do i=ii,min(n,ii+nb-1),5
  do j=jj,min(n,jj+nb-1),4
    s00 = d(i+0,j+0)
    s10 = d(i+1,j+0)
    s20 = d(i+2,j+0)
    s30 = d(i+3,j+0)
    s40 = d(i+4,j+0)
    s01 = d(i+0,j+1)
    s11 = d(i+1,j+1)
    s21 = d(i+2,j+1)
    s31 = d(i+3,j+1)
    s41 = d(i+4,j+1)
    s02 = d(i+0,j+2)
    s12 = d(i+1,j+2)
    s22 = d(i+2,j+2)
    s32 = d(i+3,j+2)
    s42 = d(i+4,j+2)
    s03 = d(i+0,j+3)
    s13 = d(i+1,j+3)
    s23 = d(i+2,j+3)
    s33 = d(i+3,j+3)
    s43 = d(i+4,j+3)
    do k=kk,min(n,kk+nb-1)
```

```
        s00 = s00 + a(j+0,k)*b(k,i+0)
        s10 = s10 + a(j+0,k)*b(k,i+1)
        s20 = s20 + a(j+0,k)*b(k,i+2)
        s30 = s30 + a(j+0,k)*b(k,i+3)
        s40 = s40 + a(j+0,k)*b(k,i+4)
        s01 = s01 + a(j+1,k)*b(k,i+0)
        s11 = s11 + a(j+1,k)*b(k,i+1)
        s21 = s21 + a(j+1,k)*b(k,i+2)
        s31 = s31 + a(j+1,k)*b(k,i+3)
        s41 = s41 + a(j+1,k)*b(k,i+4)
        s02 = s02 + a(j+2,k)*b(k,i+0)
        s12 = s12 + a(j+2,k)*b(k,i+1)
        s22 = s22 + a(j+2,k)*b(k,i+2)
        s32 = s32 + a(j+2,k)*b(k,i+3)
        s42 = s42 + a(j+2,k)*b(k,i+4)
        s03 = s03 + a(j+3,k)*b(k,i+0)
        s13 = s13 + a(j+3,k)*b(k,i+1)
        s23 = s23 + a(j+3,k)*b(k,i+2)
        s33 = s33 + a(j+3,k)*b(k,i+3)
        s43 = s43 + a(j+3,k)*b(k,i+4)
      enddo
      d(i+0,j+0) = s00
      d(i+1,j+0) = s10
      d(i+2,j+0) = s20
      d(i+3,j+0) = s30
      d(i+4,j+0) = s40
      d(i+0,j+1) = s01
      d(i+1,j+1) = s11
      d(i+2,j+1) = s21
      d(i+3,j+1) = s31
      d(i+4,j+1) = s41
      d(i+0,j+2) = s02
      d(i+1,j+2) = s12
      d(i+2,j+2) = s22
      d(i+3,j+2) = s32
      d(i+4,j+2) = s42
      d(i+0,j+3) = s03
      d(i+1,j+3) = s13
      d(i+2,j+3) = s23
      d(i+3,j+3) = s33
      d(i+4,j+3) = s43
    enddo
  enddo
```

As with all hand unrolling operations, extra “tidy-up” coding is necessary where the array dimensions are not multiples of (in this case) 5 and 4. The tidy-up coding is omitted for clarity.

Together with blocking, this technique provides the best performance for matrix-multiply kernel. Matrix factorization structured to use the rank-n update, which is an operation identical to matrix-multiply but which updates the target matrix, is also optimized using this unrolling technique. See Section 6.1.2, “Performance examples using ESSL” on page 115 for the performance of ESSL DGEMM, which uses similar optimization techniques.

3.1.6 Cache and memory latency measurement

Most of the examples so far in this chapter have been in connection with structured data that can usually be accessed sequentially and for which data prefetch streaming gives excellent performance even for very large amounts of data that do not fit into the cache. Some applications, however, access data in a much more random way and, for these applications, data streaming cannot be used.

The key performance factor for such an application is the latency, that is, the time before the computational units can make use of a data item. The latency is very different depending on which cache holds the data or whether it is in memory. To study this, the following loop was used:

```
ip1=ia(1)
do i=2,n
   ip2=ia(ip1)
   ip1=ip2
enddo
```

The data in the INTEGER*8 array `ia` was a random sequencing of the integers from 1 to N, subject to the constraint that following the pointers as shown would traverse the whole array. This ensured that each iteration was dependent on the previous one and that data streaming could not operate. As usual, the loop was iterated many times so that, if the whole of the `ia` array fitted into a particular cache, it would be the latency of that cache that was being measured.

The results in Figure 3-4 have been normalized to present the latency in terms of numbers of cycles. Since a 1.3 GHz pSeries 690 HPC was used, the numbers should be divided by 1.3 to get latency in nanoseconds. In the graph, the numbers of bytes increase uniformly on a logarithmic scale.

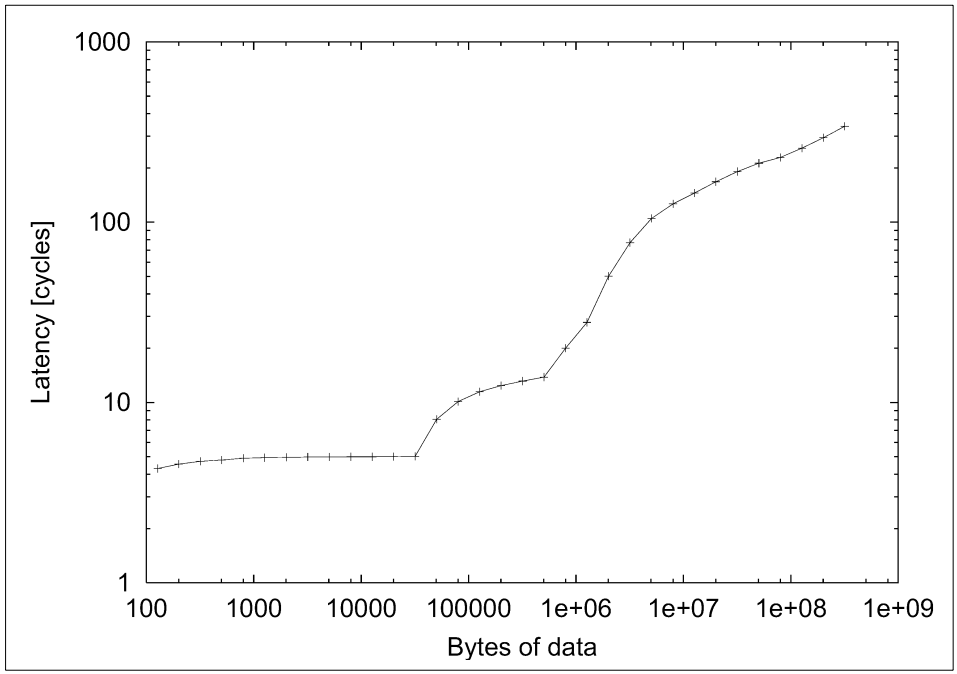


Figure 3-4 Latency in machine cycles to access N bytes of random data

The following conclusions can be drawn from this graph:

- Latency for the L1 cache is around 4-5 cycles. The figures increase sharply when bytes exceed about 32000, the size of the L1 cache.
- Latency for the L2 cache is around 11-14 cycles but seems to increase to over 20 cycles as the cache becomes full at around 1500000 bytes.
- When data spills out of L2 cache, the combined L3 cache and memory subsystem cause a fairly graceful increase in latency to a value of at least 340 cycles corresponding to memory latency. It is difficult to discern the L3 cache latency separately from these figures. With a large volume of random data, some will be in L3 and some will be in memory and this blurs the effect. If the data had been structured non-randomly to ensure that data would not be in cache unless it would all fit, the L3 cache effect might have been clearer. However, the random distribution used is probably more realistic.

3.1.7 Selected fundamental kernel performance within on-chip cache

Table 3-1 shows the measured performance of a set of fundamental loops on a pSeries 690. These measurements serve as a reference for achievable performance levels on the machine; both absolute performance in cycles per iteration for the loop, and performance relative to a 375 MHz POWER3-II processor, are given. Since the POWER4 processor has two levels of on-chip cache, results are shown for loops contained within each level: the L1 data cache and the L2 cache. The vector length, which is also the inner-loop limit, is shown for each set of data. An outer repetition loop has been used to obtain accurate timings. The inner loop is often unrolled by the compiler to minimize branch instructions, break floating-point instruction dependencies, and to allow for more flexibility in scheduling instructions for maximum performance. All of the loops were compiled using the -qarch=pwr4 and -O3 flags with the development version of XL Fortran Version 7.1.1 available at the time of publication.

Table 3-1 Performance of various fundamental loops

ID	Kernel	L1 data cache contained results			L2 cache contained results		
		Vector length	Cycles per iteration	Performance relative to POWER3 Model 270	Vector length	Cycles per iteration	Performance relative to POWER3 Model 270
1	x(i)=s	2000	1.7	2.0	40000	1.8	5.2
2	x(i)=y(i)	1000	1.7	2.8	20000	2.1	4.6
3	x(i)=x(i)+s*y(i)	1000	1.7	3.0	20000	2.2	2.9
4	x(i)=x(i)+y(i)	1000	1.7	3.0	20000	2.2	2.9
5	s=s+x(i)	2000	0.9	2.2	40000	1.7	3.1
6	s=s+x(i)*y(i)	1000	1.3	3.0	20000	1.9	2.8
7	x(i)=sqrt(y(i))	1000	18.1	2.1	20000	18.1	2.1
8	x(i)=1.0/y(i)	1000	15.1	2.2	20000	15.1	2.2
9	x(i)=a(i)+x(i-1)	1000	6.5	1.7	20000	6.5	1.7
10	s=s+y(i)*a(ix(i))	800	2.0	3.1	16000	2.5	3.2

1. Loop 1 has only stfd (double-precision floating-point store) instructions in the inner loop. As discussed in Section 2.3.7, “Store instruction processing” on page 14, store data is placed in the SDQ and the data then proceeds to the proper SSQ and STQ until it is finally written into the L2 array. Store performance is determined by the rate at which the STQ can be drained, and since there is an STQ per L2 cache controller, it depends on how stores are

distributed across the three L2 controllers. The loop measured is a straightforward stride 1 store pattern in which the compiler has unrolled the inner loop by eight and has roughly scheduled the stores within the loop from highest address to lowest (that is, in reverse order). The performance of stores is relatively flat for vector lengths up through the size of the L2 cache because of the store-through design, which always sends updates through to the L2 cache.

2. Loop 2 is the copy loop, consisting of an *lfd* and *stfd* per iteration. The performance of this loop is still determined by the store performance. Cache lines corresponding to load instructions are always reloaded into the L1 data cache on an L1 data cache miss; cache lines to which stores are directed are not.
3. Loop 3, commonly known as DAXPY, is load/store bound like the first two loops, but adds an *fmadd*. Since the vector being stored has been updated with a multiple of the other vector, the line being stored into must first be reloaded, and will then reside in the L1 data cache. Still, the modified data is stored-through to the L2 cache.
4. Loop 4 is identical to DAXPY, but without the multiply/add. Therefore it has the same execution performance. Since the arithmetic instruction is an *fadd* rather than an *fmadd,* the work done is half that of DAXPY.
5. Loop 5 is the sum reduction of a vector. The compiler unrolls the loop by eight producing eight partial sums (which are accumulated in registers), and totals the partial sums at the conclusion of the loop. This breaks the interdependence among the *fadd* operations, which would otherwise determine the performance of the loop, and the resulting performance is determined by the rate at which a single stream of floating-point loads can execute.
6. Loop 6 is commonly known as DDOT, or dot product. Just as in the case of loop 5, the sum reduction is split into eight partial sums to remove the floating-point arithmetic interdependence. The resulting performance is determined by the rate at which two streams of floating-point loads can be completed.
7. Loop 7 shows the average performance of floating-point double-precision square root. Floating-point square-root instructions may execute on either floating-point unit, but are not pipelined. Independent work can execute in the other floating-point unit concurrently, including another floating-point square-root instruction. Since both execution units can work in parallel, and a floating-point double-precision square root normally takes 36 cycles, the average time is approximately 18 cycles.
8. Loop 8 shows the average performance of floating-point double-precision divide. Floating-point divide instructions may execute on either floating-point

unit but are not pipelined. Again, two divides execute in parallel, reducing the average time to 15 cycles.

9. Loop 9 exposes the six-cycle dependent operation latency in the floating-point execution unit. The loop represents a true mathematical recurrence: each operation requires the result from the previous operation. Thus, the execution time is limited by the effective pipeline depth of six. The performance ratio relative to POWER3 is simply half the ratio of the processor frequencies, since the dependent operation latency in the POWER3 is three.
10. Loop 10 is an indirect DDOT in which one of the vectors is independently addressed using a vector of integer indices. This is the crux of the sparse-matrix-vector multiply. Each iteration of the loop requires the index to be loaded (as an integer), and that value to be shifted so as to become a byte-oriented offset rather than doubleword index, and the shifted result is used to load the double-precision element of vector *a.* This is multiplied by the stride 1 vector *y* and accumulated into the scalar *s*. The compiler breaks dependencies on the arithmetic by using eight partial sums, just as in DDOT. The dependent chain of load-shift-(indirect) load is carefully scheduled to avoid stalls.

3.1.8 Other tuning considerations

In this section, the topics of tuning for L2 cache access and a discussion of the branch prediction mechanism are provided.

Tuning for L2 cache access

Blocking for an L2 cache is discussed in “Data cache blocking” on page 38.

Improving store performance to L2 cache

Store performance can be improved with some extra effort to distribute the stores across the three L2 controllers. The following loop is a simple way to accomplish this, and will perform as much as 40 percent faster than the code given in the table for vector lengths greater than around 90.

```
nlim=(n/48)*48
do ii = 1,nlim,48
   do i=ii,ii+15
      x(i)=c0
      x(i+16)=c0
      x(i+32)=c0
   enddo
end do
do i=nlim+1,n
   x(i)=c0
end do
```

3.2 Tuning non-floating point applications

In the following sections we discuss aspects of tuning that are relevant to non-numeric applications. However, you should bear in mind that many of the aspects discussed in Section 3.1, “Tuning for numerically intensive applications” on page 25 are also relevant.

When tuning applications, you should determine whether to tune for throughput or for response time, depending on the type of application. Different approaches may be required in either case and, rather than studying the subject in detail here, we suggest referring to some of the books written on the subject.

Once the approach has been determined, we recommend the following steps:

- If the application is CPU bound, identify the critical parts of the application code using profiling (Section 5.5, “Locating hot spots (profiling)” on page 110) and determine whether the critical code can be improved.
- If the application is paging, identify how much memory is being used and what it is being used for. Consider using tools such as **vmstat** and **svmon** (refer to the AIX commands documentation). If the memory is allocated by the application, it may be possible to adjust this using configuration files. If there is not enough system memory, you could use **vmtune** (see Section 3.3, “System tuning” on page 54).
- If the application is disk or I/O bound, identify the hot disks or volumes (**iostat**, **svmon**, **filemon**). You may need to change the way I/O is performed, for example use asynchronous I/O instead of synchronous I/O or you may simply be able to move files from hot disks to disks that are less busy.
- If the application is network bound, investigate this with tools such as **netstat**, **netpmon**, and **nfsstat**. Tune network parameters with the **no** command.
- If your application is still not performing satisfactorily, start again at the top.

Chapter 4, “Optimizing with the compilers” on page 69 provides a number of suggestions for tuning code.

3.2.1 The load/store and integer units

Loads, stores, and integer operations form the majority of non-floating point instructions executed.

The load/store performance is documented in Section 8.1, “Memory to memory copy” on page 155. Ultimately, the load/store performance depends on the size of the units, that is bytes, 32-bit words or 64-bit words, and using larger units may positively affect performance.

There is a small penalty in load/store performance when data items cross 32-byte and 64-byte boundaries. Where possible, data structures should be organized so that they start on double word or word boundaries.

Note that integer divide instructions are relatively slow compared to other arithmetic instructions.

3.2.2 Memory configurations

pSeries 690 Model 681 systems support four memory controllers per MCM. Physically, the memory subsystem is implemented using memory books where each book contains two memory controllers, synchronous memory interfaces (SMIs) and DIMMs. Each controller can support up to 16 DIMMs. For a detailed description, see Chapter 2, “The POWER4 system” on page 5.

Memory is interleaved across controllers. Interleaving addresses is a function of the L3 cache controllers and the L3 cache to which the memory controllers are attached and is implemented by the L3 cache controller on the POWER4 chip.

Assuming an MCM has two equal-sized memory books attached to it, real memory is interleaved across the four memory controllers. Then physical memory on the next MCM is allocated and so on. The operating system is responsible for mapping real memory to virtual memory.

If you have only one memory book attached to an MCM, the L3 cache configures itself as a 64 MB shared L3 connected to one memory book, plus a 64 MB shared L3 with no backing storage. Memory is interleaved across the two controllers on the book. The bus between the L3 and the book must then process twice the traffic compared to the same amount of memory spread over two books, thus reducing memory bandwidth.

If an MCM has two books of different sizes installed, they operate independently with each book being two-way interleaved.

In the current release of AIX 5L Version 5.1, pages are allocated to a process from any memory book. In a future update to AIX 5L Version 5.1, pages will be allocated from memory attached to the MCM where the process is running. This memory affinity, combined with process affinity, will provide an improvement in application performance for most classes of applications.

3.3 System tuning

In this section we discuss the aspects of the system that are relevant to application performance running on the pSeries 690 Model 681. We begin with a review of the virtual memory architecture before examining large-page support. We then examine some of the system tuning parameters that can have large effects on application performance. Many of these are beyond the ability of the application programmer or user to modify directly because they require root authority to change, but it is useful to understand their possible effects.

3.3.1 POWER4 virtual memory architecture overview

This section provides an overview of the POWER4 virtual memory architecture for those readers who may be unfamiliar with it.

The architecture is an extension of the POWER3 architecture. Unlike POWER3, it provides two page sizes. The default page size is 4 KB but the hardware and operating system provide a large page (16 MB), which can be advantageous in certain circumstances. See Section 3.3.2, “Small and large page sizes” on page 58.

Program structure

POWER programs access memory through segment-based addresses. A segment-based address is calculated using a segment register (pointing to some storage) and a segment offset. In the 32-bit environment there are 16 segment registers and each can reference a segment of up to 256 MB. Some registers are reserved to address kernel memory. Other registers can be used for several purposes.

By default, segment 2 holds process data (Figure 3-5 on page 55). This includes any constants and non-stack variables and they are allocated from the bottom of the segment upwards. Stack space is allocated from the top of the segment downwards.

Programmers can prevent the stack overwriting non-stack data by limiting the size of the stack. This can be done by calling the linker (**`ld`**) with a -S option. The programmer can also use the shell **`ulimit`** command (ksh: **`ulimit`**, csh: **`limit`**) to limit the size of the stack and/or data area at run time.

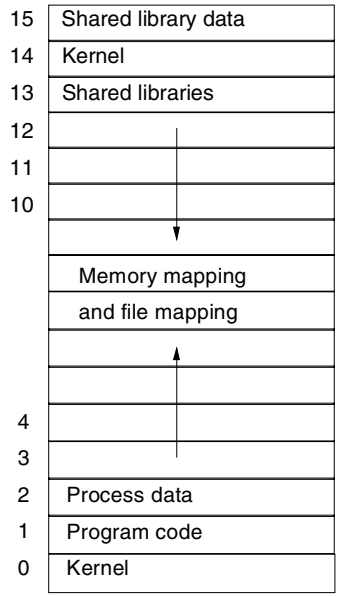


Figure 3-5 32-bit environment segment register usage

Alternatively, you can call the linker specifying the -bmaxdata option. This has two effects: it specifies a maximum size for the user data area and it causes the user data to be placed in segment 3 (and subsequent segments as required up to a maximum of 8 segments for 32-bit programs) while the stack is placed in segment 2.

32-bit programs that need to access more than 256 MB of data can do so by using a contiguous set of segment registers. These programs need to be compiled with the -bmaxdata option. For example, using -bmaxdata:0x80000000 enables the maximum possible data space of 2 GB.

In the 64-bit environment, there are effectively an unlimited number of segment registers. Note that in the 64-bit environment, -bmaxdata should not be used because it will limit the addressable data space.

Each segment consists of a number of pages (of the same size). By default, pages are 4 KB.

Program virtual memory is mapped onto physical memory in units of pages. The operating system maintains a map (page table) of virtual to physical memory for each process. Entries in the map are called page table entries (PTEs).

A PTE provides information about a corresponding page frame (which can be 4 KB or 16 MB in size). Pages of both sizes can co-exist on a system though a segment can only have pages of one size. Each PTE contains a number of status and protection bits as well as address information.

AIX Version 4.3 and AIX 5L Version 5.1 executables

Note that 32-bit executables compiled under AIX Version 4.3 will run unchanged under AIX 5.1. Any code compiled in 64-bit mode under AIX Version 4.3 must be re-compiled before it can be used on AIX 5L Version 5.1. This means that:

- AIX Version 4.3 64-bit executables must be re-compiled from the source code to run under AIX 5L Version 5.1, not just relinked.
- AIX Version 4.3 64-bit object modules or library files cannot be linked with AIX 5L Version 5.1 object modules or library files. The AIX Version 4.3 64-bit modules must be re-compiled.

Note that you cannot link 32-bit together with 64-bit object modules under either operating system release.

Address translation

Figure 3-6 gives an overview of the steps in the address translation process.

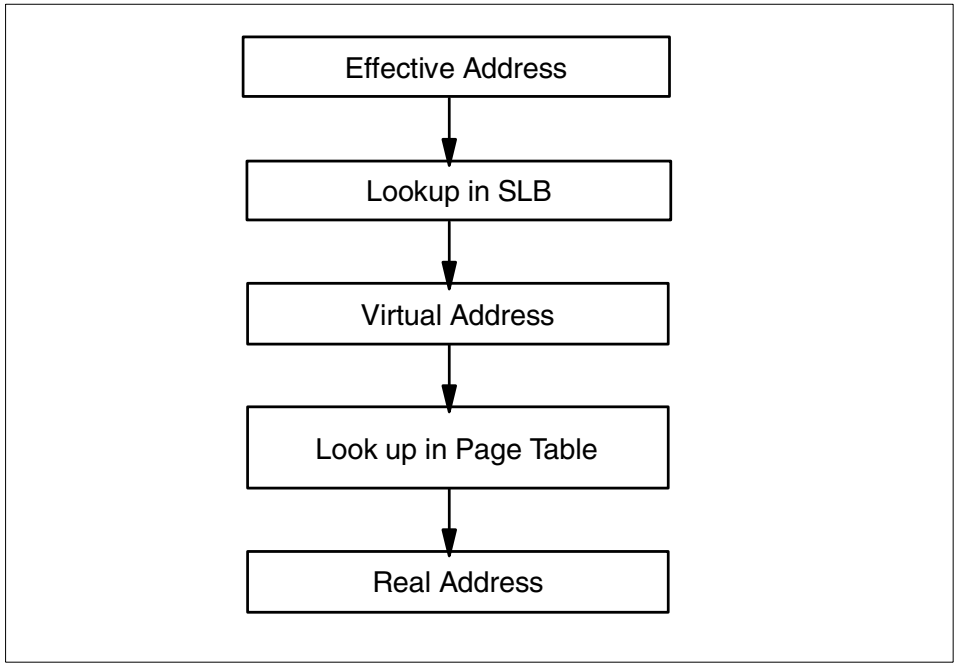


Figure 3-6 POWER address translation

An effective address (EA) is the address of data or an instruction generated by the processor during the decode of an instruction. The EA specifies a segment register and offset information within the segment.

Address translation occurs in two steps: EA to virtual address (VA) and VA to real address (RA). If the EA cannot be translated, a storage exception occurs. While there are a number of different reasons for exceptions, programmers need only concern themselves with those caused by invalid data addresses. In these cases, the operating system will send a signal to the offending process and typically terminate it.

Conversion of a 64-bit effective address to a corresponding virtual address is performed by looking up the segment identifier (ESID) in the Segment Lookaside Buffer (SLB). The SLB is a cache of ESIDs and corresponding virtual segment identifiers (VSIDs) maintained by the operating system and referenced by the hardware. Each SLB entry also contains a *valid* bit and various flags. The 80-bit virtual address is formed by concatenating the VSID with the page and byte address from the EA as shown in Figure 3-7.

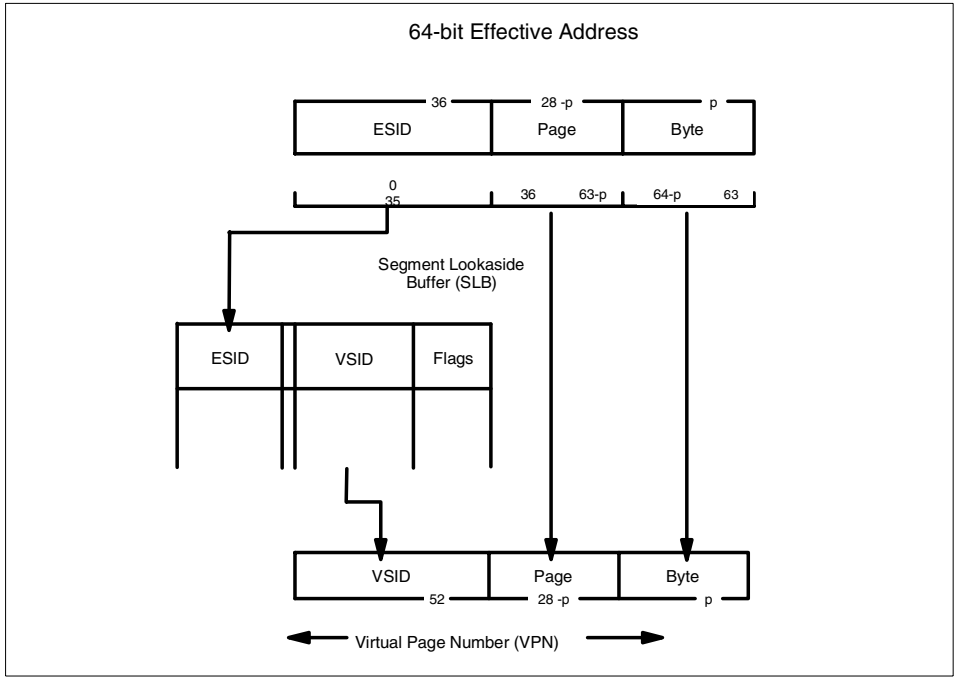


Figure 3-7 Translation of 64-bit effective address to 80-bit virtual address

Conversion of the 80-bit virtual address to its corresponding real address is done by hardware lookup in the page table. The page table is maintained by the operating system and its base address is held in a hardware register. The virtual page number (VSID + page number) is used to construct an index into the page table. The real address for the base of the page is extracted from the page table Entry.

3.3.2 Small and large page sizes

Historically, the PowerPC Architecture supported the mapping between virtual and physical memory at a granularity of 4 KB pages. POWER4 systems introduce a new PowerPC Architecture feature that provides an alternate large-page size that can be used in addition to the 4 KB base-page size. The pSeries 690 Model 681 system supports a 16 MB large-page size. The implementation involves the selective use of large virtual/physical memory pages to back the process private data segment(s). A process can contain a mixture of small (4 KB) and large (16 MB) pages at a 256 MB virtual segment granularity. All pages within a 256 MB segment have the same size.

The primary benefit from large-page support is improved performance for applications. This refers to applications that access a large amount of memory in a sequential manner or have significant gather/scatter components (such as large, randomly accessed user data spaces). Large pages can improve performance for these applications by reducing the translation lookaside buffer (TLB) miss rate. POWER4 systems use memory data prefetching (and other techniques) to minimize memory latencies. Data prefetching starts when a new page is accessed and grows more aggressive as the page continues to be sequentially accessed. However, data prefetching must be restarted at page boundaries. The use of large pages can improve performance by reducing the number of prefetch startups.

An update of AIX 5L Version 5.1, targeted for mid-2002, introduces a usage model that allows existing applications to use large pages without requiring source code changes and/or recompilations. The need for investment protection also dictates that the large-page data support must not impact source or binary compatibility for existing applications and the kernel extensions they depend upon if large-page support is not used by these applications. With the initial release of AIX 5L Version 5.1, which supports the pSeries 690 Model 681, there is already a low-level shmat/shmget interface, which will also be enhanced with future releases.

Note: At the time of writing this document the AIX implementation of large-page support was still under development. The following description is subject to change.

Large-page data areas

Large pages will be used for the data areas of the user address space. For technical applications, these areas consist of the user heap and main program BSS and data storage areas. These are the critical data areas for C programs, since the user heap supports malloc storage, BSS holds uninitialized program data, and data storage holds both initialized and (small) uninitialized data.

These are also the critical data areas for Fortran programs because the Fortran storage classes that require large pages reside within these areas, as follows:

Static Static variables reside in the data storage area. Large, uninitialized static variables reside in BSS.

Common If a common block variable is initialized, the whole block resides in the data storage area; otherwise, the whole block resides in BSS.

Controlled This storage class is used for allocatable arrays. Controlled variables reside in the user heap.

Large pages are not required for other areas of the user address space, so 4 KB pages are used. These consist of the process stack, library data storage area, mmap regions, and user text. At the cost of 16 MB of physical memory resource per page, large pages would provide little or no benefit to applications if used for the process stack or library data because both typically represent small quantities of data. For a single-threaded process automatic and controlled automatic Fortran storage classes reside within the process stack and therefore do not use large pages. Typically, the amount of data is small.

Use of large pages for thread stacks within a multi-thread process is of value and can provide benefit through larger TLB coverage. The use of large pages for thread stacks is supported through the AIX pthreads library, which places thread stacks within the user heap.

It is not planned to support large pages for mmap regions. The support of large pages for user text data is not relevant for technical applications.

Some technical and commercial (for example database) applications do map and use shared memory segments within their user address spaces and can benefit from large pages. This is supported by the implementation of a large-page shmat/shmget interface. An application must be modified in order to use large pages for shared memory segments. This is because a special option must be specified at the time a shared memory segment is created if large pages are to be used for the segment.

In a future release of AIX 5L large pages will be allocated preferentially from physical memory that is close to the processor (or MCM) that initiated the request. This memory affinity is intended to hide the non-uniformity in latency and bandwidth (primarily the latter) of the memory subsystem.

Large pages are pinned (cannot be paged out or stolen) to memory the entire time an application executes. The large-page memory pool is a limited system resource. A failure will occur when a large-page application tries to allocate a large page and none are available.

Large page application support

Although the 64-bit kernel is the strategic AIX kernel, large-page data support is also provided in the 32-bit kernel.

A new bit flag will be maintained within the XCOFF and XCOFF64 executable file headers to record the large-page data attribute of a program. If the flag is set, this indicates that the program uses large pages; otherwise, it only uses small pages. It is deemed better to fail a technical application that requests an additional large page when none is available than have it silently execute with 4 KB pages.

The **ldedit** command will provide the ability to set and unset the large page flag of an executable file without the need for source code changes, recompiling, or relinking. It will also support setting maxdata and maxstack. The **dump** command will be modified to display the status of a program's large page data flag.

The large page data usage is inherited over fork(). Check the manual pages for details on the memory duplication scheme. Large page data usage is not inherited over exec().

Because of different page protection requirements, the data model for 32-bit large page data applications is slightly different from the existing 32-bit process models (default and large memory) shown in Figure 3-5 on page 55. You can inspect a program's memory layout with the help of the **svmon** command.

Large-page command support

An extension of the **vmtune** command will be provided to select at boot time the number of memory segments (256 MB) that will hold large pages.

A system administrator has the ability to control usage of the large-page memory pool by user ID (such as with the **chuser** and **mkuser** commands). This prevents unprivileged users causing privileged large-page data applications to fail due to running out of large pages.

At this time, there is no Workload Manager (WLM) support provided to manage large-page physical memory or large-page applications. Large pages are neither pageable nor swappable. They are essentially pinned pages that are treated as unmanaged resources by the WLM.

The commands **ps**, **vmstat**, and **svmon** have been extended to report on large-page usage.

Large-page performance observations

General large-page support for Fortran and C application is not available in AIX 5L Version 5.1. It is expected that when large-page support is available, uniprocessor performance for memory-bound kernels such as DAXPY will increase significantly. This is primarily due to the increased efficiency of data prefetching long vectors in large pages (see Section 2.3.8, “Fixed-point execution pipeline” on page 15).

3.3.3 AIX system parameters

System factors that can influence application performance include:

- Hardware configuration
  - CPU configuration

    The speed, number of CPUs and the particular type of pSeries 690 Model 681 processor module installed are, of course, extremely important to application performance. In the POWER4 or POWER4 Turbo modules, the L2 caches are each shared between two CPUs. In the POWER4 HPC modules, the L2 caches are not shared. However, these are factors that cannot be adjusted or tuned for a given machine configuration, but are hardware characteristics of which the programmer should be aware.

- Memory configuration

  Similarly, the memory configuration is also important. As described in Section 3.2.2, “Memory configurations” on page 53, the particular memory configuration of a particular machine determines the overall memory bandwidth available to the CPUs. This is not a factor that can be adjusted for a given machine configuration, but the programmer should be aware of the possible effects.

- Storage configuration

  A detailed discussion of possible storage configurations that can be attached to the pSeries 690 Model 681 is outside the scope of this book. However, for I/O dependent applications, the underlying storage configuration can have a very significant effect on application performance. These can include General Parallel Filesystem (GPFS) or AIX Journaled Filesystem (JFS) configurations spread across multiple physical disks, which could include SSA, various types of SCSI disk, or fiber-attached disks. For this reason, awareness of the target storage configuration and the available memory configuration may favor programming choices that trade memory use for I/O.

- Software configuration
  - Paging space configuration

    In the scientific and technical computing domain, it is common for the application mix on a machine to be selected and controlled so as to avoid paging. With the advent of AIX Version 4.3.2 the paging allocation algorithm only allocates space in paging space when it is necessary to free up a page in memory. This means that for a system that is under no pressure for real memory pages, the paging space utilization will be very small. The **`lsps -a`** command might show one percent utilization. For this reason, and in order to save disk space, it is becoming common to configure paging space that is somewhat smaller than real memory. Large memory systems may be running applications that consume large amounts of memory. If so, it is important to consider the effect if multiples of these jobs are ever started such that memory becomes overcommitted, possibly exhausting paging space. A control mechanism, for example a job scheduling system such as LoadLeveler or AIX Workload Manager, should be considered. Alternatively, the paging space should be made large enough to accommodate such an event.

– 32- or 64-bit kernel

  In general, if the main application or applications are 64-bit applications, then it is slightly better to use the 64-bit AIX kernel. For 32-bit applications then it is slightly better to use the 32-bit kernel. However, the overhead of running 64-bit applications on the 32-bit kernel (remapping system calls to 32-bit calls, and reshaping the data structures for these calls) is handled in the kernel and is small. The overhead of running 32-bit applications on the 64-bit kernel (reshaping data structures in system calls) is likewise small.

  Note that 64-bit applications from AIX Version 4.3.3 must be recompiled to run under AIX 5L Version 5.1, whether they will be run on the 64-bit kernel or not.

– Kernel parameters

  Certain kernel tuning parameters can have a large effect on the performance of certain applications, depending on the application's use of memory and files.

  The **vmtune** command (provided in the AIX fileset bos.adt.samples) provides a number of parameters that can be adjusted to suit particular applications:

  • Page replacement selection of file or application pages

    As demand for memory increases, the AIX Virtual Memory Manager (VMM) must occasionally reassign pages in use by programs to maintain a minimum number of free pages. The vmtune parameters minperm and maxperm set thresholds that determine the pool from which the AIX VMM page replacement algorithm will select pages to be reassigned.

    For the purposes of this discussion, allocated memory pages can be considered to be one of two types. File pages are pages containing data from files mapped into memory by AIX. Computational pages are pages allocated to running programs.

    When the percentage of real memory occupied by file pages falls below the minperm value, the page replacement algorithm steals both computational and file pages.

    When the percentage of real memory occupied by file pages is greater than the maxperm value, the page replacement algorithm steals only file pages.

When the percentage of real memory occupied by file pages is between the minperm and maxperm values, the page replacement algorithm can steal pages from both computational and file pages. It will normally steal only file pages, unless the repage rate for file pages is higher than that for computational pages. If so, it will steal both types of pages.

The default settings for these parameters are approximately minperm=20, maxperm=80. That is 20 percent and 80 percent of real memory.

Consider, for example, a program that uses a lot of memory for computation, but that also writes out large files sequentially. As the program writes out to the file, more and more file pages will be created in memory. Once the number of file pages reaches 80 percent of real memory, an application's computational pages will be largely protected from being stolen by the page replacement algorithm. Below this level, an application may find that its computational pages are being stolen to make way for file pages. If the working set size of the program is larger than 20 percent of real memory (100 - maxperm), then its performance may well suffer as its computational pages are stolen to make way for file pages. The larger the program, the greater this effect.

Therefore, such an application could well benefit from setting minperm and maxperm lower than their default values, and in the case of maxperm possibly much lower.

There is a third parameter, related to minperm and maxperm: strict_maxperm. This makes the maxperm setting a hard limit rather than a threshold.

For example, to set the threshold below which VMM page replacement will steal computational pages to 5 percent and the threshold above which it will steal only file pages to 20 percent, the following command would be used:

```
/usr/samples/kernel/vmtune -p5 -P20
```

To set the maxperm threshold as a hard limit using strict_maxperm:

```
/usr/samples/kernel/vmtune -h 1
```

- Memory page replacement parameters

  The minfree, maxfree, mempools, and lrubuckets parameters may need to be adjusted together to reduce memory scanning overhead in a busy system and to maintain a large enough free list to readily satisfy demands from programs allocating memory. Recommendations for tuning these parameters can be found in the *AIX Performance Management Guide* (product manual, available on the Web) and the *AIX 5L Performance Tools Handbook,* SG24-6039.

  The maxfree parameter should be at least maxpgahead (see the maxpgahead kernel parameter description that follows in this section) greater than minfree. It is worth experimenting with larger values for minfree and maxfree than the defaults when trying to smooth out peaks and troughs of mixed workloads.

  Page replacement in AIX is performed by the lrud daemon. From AIX 4.3.3 onwards, this daemon is multi-threaded, and the system memory is divided into a number of pools. The number of pools is specified by the mempools parameter. On a large memory, SMP system, this allows memory scanning to be performed more efficiently than with one large memory pool.

  Each memory pool can be further subdivided into a number of sections called buckets. The size of these buckets is specified by the lrubuckets parameter. These buckets are scanned individually by the lrud using the VMM page replacement algorithm. This involves a two pass process where unreferenced pages are marked in the first pass, and, if a free page is not found a second pass is made and pages still marked as unreferenced will be replaced. On a large, busy system, with a single bucket across all of memory, this two pass memory scan would be too great an overhead. The subdivision of memory into buckets reduces this overhead.

- I/O pacing with min_pout and max_pout

  These parameters are of importance in improving the performance of both single, large applications performing sequential I/O, and multiple jobs that perform I/O.

  min_pout and max_pout are system attributes that control I/O pacing. That is, max_pout sets a maximum threshold for pending I/O requests per file. Above this level, an application generating large numbers of I/O requests will be put into a sleep state until the number of pending I/O requests falls to or below the min_pout value. The default settings are zero for both values, which means no checking, but this can allow a

high I/O volume application to saturate the system's capabilities and seriously affect the performance of other applications. However, enabling checking with too low values for these parameters can reduce the performance of such a high I/O volume application.

Therefore, where multiple applications must share a system, one approach to setting I/O pacing would be to set these values high (several thousand), and measure the effect on all types of workload on the system. The aim should be to get these values as high as possible for maximum throughput of the high I/O volume of the application while not reducing the performance of other workloads.

These parameters can be set with the **chdev -l sys0** command, using the appropriate attribute.

- Read ahead with minpgahead and maxpgahead

  These values control the amount by which the VMM will schedule pages in advance of the current page when reading files sequentially. When sequential access is detected, the read-ahead mechanism brings in two pages, and at each confirmation the number of pages read ahead is doubled up to maxpgahead. For applications that perform large amounts of serial I/O, it may be advantageous to set a relatively large maxpgahead value (the default value is 8).

  However, the underlying I/O subsystem should be taken into account when selecting this value. For example, if the file is stored on file systems striped across multiple devices then a higher value may be appropriate than if it is stored on a single disk device.

- max_coalesce

  This parameter is an attribute of logical disk drives that sets the maximum number of bytes to be transferred to the disk by the device driver in a single operation. When using SSA RAID arrays for sequential I/O, this value should be set to the number of disks across which the data is striped multiplied by 64 KB.

- Sequential and random write-behind

  Files mapped into memory are partitioned into 16 KB clusters (four pages when using small pages). When writing sequentially, all four pages in a cluster will be modified one after another. The parameter numclust specifies the number of such clusters before the current cluster, which the VMM will allow before scheduling the writing of their modified pages. By default this is set to 1, which means that modified pages from sequential files should not accumulate in memory. For randomly written files, this mechanism does not apply. There is another parameter, maxrandwrt, which sets a maximum number of modified (also known as dirty) pages for a given file. Once this number is

exceeded, then the VMM will schedule these pages for writing. It should be noted that when the syncd daemon runs, these modified pages will be written to disk anyway, but these parameters can prevent the buildup of modified pages between runs of syncd to the extent that syncd running affects system performance. These parameters can be modified using the **vmtune** command.

- **lgpg_regions**, **lgpg_size**

  As discussed in Section 3.3.2, “Small and large page sizes” on page 58, the use of large pages for virtual memory has the potential to significantly improve the performance of certain applications. In order for an application to use large pages, there must have been large pages defined to the system at system IPL. The lgpg_regions parameter specifies the number of large pages to be made available at the next reboot. The lgpg_size parameter specifies the size of these pages, and for the IBM @server pSeries 690 Model 681 POWER4 machines this would be 16 MB specified in bytes. An example of a sequence of commands and actions to define 8 GB of large pages and make them available might be as follows:

  ```
  vmtune -g 16777216 -L 512
  bosboot -a
  shutdown -Fr
  ```

  The exact usage of the **bosboot** command would depend on the particular system being configured for large pages.

3.3.4 Minimizing variation in job performance

Various factors can affect the consistency of the performance of a job from run to run. These include:

- System factors
  - Competing jobs

    Multiple jobs running in the system simultaneously can compete for resources such as CPU, memory, and I/O bandwidth. One approach to reducing the variability introduced by running multiple jobs on the system is to carefully select jobs that require different types of resources to be run together. Once jobs have been characterized in this way, the running of the job mix can be controlled using a job scheduling system such as LoadLeveler. In practice, many large applications have requirements for all the above types of resource. Another approach is to use the AIX Workload Manager to guarantee resources to a particular job, and perhaps sharing the remaining resource between other jobs in the system. For examples of the effects of multiple jobs running on the system, see Chapter 8, “Application performance and throughput” on page 153.

For more information on AIX Workload Manager, see *AIX 5L System Management Concepts: Operating System and Devices*, and *AIX 5L System Management Guide: Operating System and Devices.* For more information on LoadLeveler, see *Using and Administering IBM LoadLeveler for AIX*, SA22-7311.

- External I/O performance

  If a job is dependent on the performance of shared storage facilities that are heavily utilized at certain times, then it may experience variation in runtime. In this situation, it may be possible to trade memory use for I/O to reduce the dependency on the external I/O performance.

- System software levels

  Occasionally, different system software levels will implement different default values for certain tuning parameters. This can cause unexpected variation in job performance. Software updates and fixes should therefore be checked and tested carefully for such changes.

- Application factors
  - Processor binding

    In order to guarantee the sharing of L2 cache between certain threads, or to guarantee that threads are using dedicated L2 cache, you may have bound threads or processes to specific processors. Depending on the thread scheduling scope (see Section 7.1.1, “SMP runtime behavior” on page 126), and the numbers of threads and processors, this could introduce variation in runtime behavior.

  - Variation in data

    The previous factors apply to variations in runtimes for the same job running with the same data. It is also the case that variations in the data input to the job can cause variability, even though the problem to be solved is the same size, and the program may take longer to converge to a solution. With parallel jobs variations in input data may lead to hotspots where certain processors have more work to do than others, leading to an overall increase in the runtime of the job. An approach to resolving this, which is outside the scope of this book, is to implement a dynamic load balancing design in the parallelization of the program.

4

Optimizing with the compilers

In this chapter we describe the features of the XL Fortran and C and C++ compilers that relate to optimization for the POWER4 processors. We begin with the optimization options available with particular emphasis on those that benefit applications running on POWER4 processors. In subsequent sections the particular techniques and considerations for improving performance using the compiler are discussed.

4.1 POWER4-specific compiler options

In this section some useful XL Fortran compiler options that can be used to improve performance are presented. We then focus on options with specific benefits on POWER4 microarchitecture machines. Finally, we make some recommendations for initial attempts at optimization.

It should be noted that, when specifying conflicting compilation options on the command line, the last option wins. For example, consider the following command:

```
xlf -O3 -qsource -qlist -o monte -O2 monte.f
```

The optimization flag -O is specified twice with two different levels. In this case, optimization level two would be used because this is specified last. This also applies to those options that are implied by another option. See the description of -O4 and -O5 in the following section.

4.1.1 General performance options

In the following sections, useful Fortran, C and C++ compiler general performance options are discussed.

XL Fortran options

The following options are provided by the XL Fortran compiler. For more details see the *XL Fortran for AIX User's Guide*, SC09-2866.

- -O, -O2, -O3, -O4, -O5

  The -O flag is the main compiler optimization flag, and can be specified with several levels of optimization. -O and -O2 are currently equivalent.

  At -O2, the XL Fortran compiler's optimization is highly reliable and usually improves performance, often quite dramatically. -O2 avoids optimization techniques that could alter program semantics.

  -O3 provides an increased level of optimization. It can result in the reordering of associative floating-point operations or operations that may cause runtime exceptions. This could slightly alter program semantics. This can be prevented through the use of the -qstrict option together with -O3. At this optimization level, the compiler can also replace divides with reciprocal multiplies. -O3 is often used together with -qhot, -qarch, and -qtune.

  -O4 provides more aggressive optimization and implies the following options:

  – -qhot
  – -qipa
  – -O3
  – -qarch=auto
  – -qtune=auto
  – -qcache=auto

  -O5 implies the same optimizations as -O4 with the addition of -qipa=level=2.

  In general, increasing levels of optimization require more time (sometimes considerably more time), and larger memory during the compilation. In addition, -O4 and -O5 sometimes need additional space in /tmp (or the location specified by the TMPDIR environment). The recommendation is to have at least 200 MB available, and potentially up to 400 MB.

- -qarch, -qtune, -qcache

  These options allow the compiler to take advantage of particular hardware configurations for the purposes of optimization.

  - -qarch specifies the instruction set architecture of the machine, that is which instructions the compiler will generate. Specifying certain values for this option can generate code that will not run on other machine types. For example, -qarch=pwr2 would generate code that might not run on a POWER4 machine.

    The -qarch=com option generates executable code that will run on any POWER or PowerPC hardware platform. However, this option also prevents the compiler from generating any of the optional PowerPC architecture instructions. In the case of POWER4, these instructions include the two floating-point square root instructions: fsqrt and fsqrts, which are likely to be important in numerically intensive applications.
  - -qtune instructs the compiler to perform optimizations for the specified processor. These can include taking into account instruction scheduling and memory hierarchy for the specified architecture. This option only has an effect when used with an optimization level of -O (or -O2) or greater.
  - The -qcache option is only effective if the -qhot option is also specified explicitly or implicitly with, for example, -O4.

    This option can be used to specify the exact cache hierarchy of the machine. This can be useful if the target machine has a different cache hierarchy from the default. -qcache is designed to describe the complete cache hierarchy of the system including the TLB. If specifying cache configurations with -qcache, then the specifications should be ordered by capacity and to be very precise should include the ERAT, TLB, L1, L2, and L3.

    At present, the compiler uses the line size of the cache for optimization, but a future level of the compiler may use capacity and miss cost more aggressively. At that time, if compiling for machines with different cache hierarchies, then the most conservative specification would be the larger line size, the smaller capacity, the smaller associativity level, and the larger cost.

  If the program will be compiled and run on the same machine, then -qarch=auto should be used or -qarch should be set to the specific processor. The default setting is -qarch=com in 32-bit mode, and -qarch=ppc in 64-bit mode. The compiler will then automatically select default settings for -qtune and -qcache appropriate to the processor architecture selected. If the compilation machine is different from the target machine, then it can be useful to specify the target architecture for -qarch and -qtune.

For example, if compiling and running on a POWER4 pSeries 690 Model 681, then use -qarch=auto -qtune=auto, or -qarch=pwr4 -qtune=pwr4. If compiling on this machine but executing on an RS/6000 SP 375 MHz POWER3 High Node, then -qarch=pwr3 -qtune=pwr3 should be used. Different combinations of these two options can be used to specify the machines on which the executable will run, but produce code that is optimized for one of the target machine types.

- -qhot

  The -qhot option performs high-order transformations to maximize the efficiency of loops and array language. It can optionally pad arrays for more efficient cache usage and can generate calls to vector intrinsic functions such as square root and reciprocal. As with -O3, some of the transformations can slightly alter program semantics, but this can be avoided by also using -qstrict. The -qhot option is made less effective by the -C array bounds checking option, but remains active. Note that -qhot is selected by default when -O4, -O5, or -qsmp=auto options are specified.

- -qalias

  The -qalias option can be used to tell the compiler about the types of aliasing that may be found in the program where an area of storage may be referred to by more than one name. The compiler may be able to perform additional optimization with this information, for example for programs that violate parameter aliasing rules (see the discussion of -qalias in the *XL Fortran User's Guide*, SC09-2866).

  Compiling with -O2 -qalias=nostd may give better performance than using no optimization at all.

- -qalign

  The -qalign option specifies the alignment of data objects in storage. There are two suboptions:

  – -qalign=4k, which causes certain objects over 4 KB to be aligned on 4 KB boundaries and can be useful for optimizing I/O when using data striping.

  – -qalign=struct, which can specify the alignment of derived type objects such as structures.

- -qassert

  The -qassert option can be used to give the compiler information about loop dependencies and iteration counts, which may allow additional optimizations.

- -qcompact

  The -qcompact option reduces optimizations that increase the size of the executable. For current systems with large memories this is not commonly used. However, it can be useful in the rare cases where -O3 generates code that performs worse than code generated with -O2. In these cases, -O3 -qcompact is often better.

- -qpdf, -qfdpr

  The -qpdf option enables profile-directed feedback. This is a two-step process where profile information from a typical run or set of runs is used for further optimization.

  -qfdpr generates object files containing the necessary information for use with the AIX Feedback Directed Program Restructuring (FDPR) command.

- -qipa

  The -qipa option can improve basic optimization by doing analysis across procedures. This must be specified at both compile and link stages, and there are various suboptions to give the compiler more information about the characteristics of procedures within the program, and how to handle references to procedures that have not been compiled with -qipa.

- -qsmp

  The -qsmp option is used for shared memory parallelization of certain loops within a program. It is possible to make the compiler use the minimum optimization necessary to achieve parallelization by using -qsmp=noopt.

  Shared memory parallelization is covered in more detail in Section 7.1, “Shared memory parallelization” on page 126.

- -qstrict, -qstrict_induction

  These options prevent the compiler (options -O3, -qhot and -qipa) from performing optimizations that could alter the semantics of the program and potentially producing results that differ from unoptimized code. -qstrict_induction applies to such optimizations on loop counter variables. Both of these options can result in reduced performance.

- -qunroll

  The -qunroll option allows the compiler to unroll loops within a compilation unit. By default, with optimization level 2 (-O, or -O2) the compiler performs loop unrolling if analysis indicates that it will be beneficial. If such unrolling actually reduces performance for a procedure, then -qnounroll could be used to turn it off for a particular procedure. Loops where it is beneficial to unroll within this procedure could then be marked with the UNROLL compiler directive. See Section 4.2, “XL Fortran compiler directives for tuning” on page 80

- -Q

  The -Q option allows the compiler to inline functions and procedures. That is, the compiler can move the code from the inlined program unit into the code of the calling unit and potentially achieve further optimizations by doing so. This option can also take the names of functions or procedures to be inlined, or those to be excluded from inlining.

- -qlibansi, -qlibessl, -qlibposix

  These options specify that any references to functions that have the same name as a library function are references to that function.

  -qlibansi ANSI C library.

  -qlibessl ESSL library. See Section 6.1, “The ESSL and Parallel ESSL libraries” on page 114

  -qlibposix POSIX 1003.1 library.

- -qnozerosize

  The -qnozerosize option tells the compiler that there are no zero-sized objects in the program that can improve performance in some programs by removing the need to check for them.

- -g

  The -g option is not a performance flag. It generates symbol and line number information in the object files that can be used for debugging. However, it is important to note that compiling with -g has almost no effect on performance. It does not prevent optimizations performed by the compiler.

- -p, -pg

  These options are used to generate monitoring information when producing runtime profiles of a program. See Section 5.5, “Locating hot spots (profiling)” on page 110 for more details.

Visual Age C and C++ options

With the following exceptions, all the options mentioned previously are also valid when used with the C and C++ compilers:

- -qhot
- -qnozerosize
- -qlibessl, -qlibposix
- -qsmp is supported by the C compiler, but is not supported by the C++ compiler. However, C++ can declare (as extern “C”) and call C functions that are coded with shared memory parallelism through OpenMP pragmas.

In addition, the following performance-related options exist for these compilers:

- -qalias=ansi

  The -qalias=ansi option specifies the use of type-based aliasing during optimization. This is synonymous with the obsolete -qansialias, and allows the compiler to make assumptions about the types of objects accessed via pointers.

- -qfold

  The -qfold option evaluates constant floating-point expressions at compile time.

- -qinline

  The -qinline option is equivalent to the -Q option described in the Fortran options.

- -qunroll=n

  The -qunroll option accepts a value n, where n is the depth to which the compiler should unroll inner loops. The default value of n is four, and the maximum value is eight. This option takes effect when an optimization level of -O2 or higher is specified.

4.1.2 Options for POWER4

This section describes specific optimization actions performed by the compiler for POWER4 microarchitecture machines.

Compiler options that perform specific optimizations for POWER4 microarchitecture machines are as follows:

- -qarch=pwr4
- -qtune=pwr4
- -qcache=auto

  or

  ```
  -qcache=level=1:type=i:size=64:line=128:assoc=0:cost=13 \
  -qcache=level=1:type=d:size=32:line=128:assoc=2:cost=11 \
  -qcache=level=2:type=c:size=1440:line=128:assoc=8:cost=125
  ```

  Note that the cost value for the L2 cache miss above is derived from an average for data misses across the various L3 caches.

4.1.3 Using XL Fortran vector-intrinsic functions

The compiler is capable of generating calls to specially optimized vector versions of intrinsic functions. These are included in the libxlopt.a library included with XL Fortran, and the standard linkage sequences for the various invocations of the Fortran compiler (for example xlf, xlf90, xlf90_r) include this library. Calls to intrinsic functions can often make up a significant percentage of the CPU usage profile. For example, one weather modelling program spends 22 percent of its time in the intrinsic functions.

These calls may be generated using the -qhot compiler option and will be satisfied from the libxlopt.a library. Certain other options will prevent the generation of these calls: -qhot=novector, or -qstrict. For example, the following code outline could generate vector-intrinsic function calls when compiled with -qhot:

```
do i=1,n
   c(i)=cos(a(i))
   .
   .
end do
```

Vector versions of the following functions exist with examples of the calls provided in the following list:

- Cosine

  `cos(a(i))`

- Division (not strictly speaking a function, but a vector division function exists in libxlopt.a)

  `a(i)/b(i)`

  At present, although this function exists, the compiler does not generate this function call, but uses a combination of a vrec or a vsrec function call and multiply instructions.

- Exponential

  `exp(a(i))`

- Natural logarithm

  `log(a(i))`

- Reciprocal

  `1.0/a(i)`

- Reciprocal square root

  `1.0/sqrt(a(i))`

- Sine

  `sin(a(i))`
- Square root

  `sqrt(a(i))`
- Tangent

  `tan(a(i))`

There are two versions of each function, a double-precision version and a single-precision version. The compiler will generate the appropriate call.

These functions are derived from the MASS library functions (see Section 6.2, “The MASS libraries” on page 117 for more information on the MASS library). Since the XL Fortran release schedule is separate from the freely available MASS library, the libxlopt.a versions may lag behind the MASS library versions. This means that any improvements in the performance of these routines, for example by algorithm changes that take advantage of the POWER4 architecture, are likely to be available in the MASS library first.

Also note that the use of these functions is subject to the same considerations as the use of the MASS library functions.

Examples of the speedups that can be seen with the vector-intrinsic functions are shown in Table 4-1.

Table 4-1 Vector-intrinsic function speedups

Function	Speedup (double precision)	Speedup (single precision)
cos	3.94	3.90
div	not generated	not generated
exp	4.60	4.55
log	5.80	5.74
reciprocal	1.10	2.17
rsqrt	2.26	6.23
sin	4.03	3.85
sqrt	1.09	2.17
tan	4.27	3.79

The double-precision numbers were generated with the following program:

```
program cos_test

        integer m,n
        parameter ( n = 1000 )
        parameter ( m = 10000 )
        real*8 a(n)
        real*8 b(n)
        real*8 fns
        real*8 time1,time2,rtc
        real*8 ctime
        integer i,j

        ctime=0.0d0

        call random_seed
        call random_number(a)
        call random_number(b)

        do i=1,m
          time1=rtc()
          do j=1,n
            b(j)=cos(a(j))
          end do
          time2=rtc()
          ctime=ctime+(time2-time1)
          call dummy(b,a,n)
        end do

        fns=float(m*n)

        write(6,998)ctime,fns/(ctime*1.0e6)
998     FORMAT('Cosine: intrinsic time (s) = ',F6.2,
     &                ' cos/s = ',F8.2)

        stop
        end
```

For the other vector-intrinsic functions, the call to cos_test in the preceding example was replaced with the appropriate function or operation.

The dummy subroutine does nothing, but calling it prevents the compiler from optimizing away the loop.

4.1.4 Recommended options

The recommended starting compiler options for the POWER4 microarchitecture are:

```
-O3 -qarch=pwr4 -qtune=pwr4
```

If the executable must execute on POWER3 and POWER4 machines, but the performance on the POWER4 machine is most important, then -qarch=pwr3 and -qtune=pwr4 should be specified instead.

The -qhot option may give significant performance benefits, at the cost of additional compile time. It is also essential for certain other optimization flags to take effect, including -qcache.

If the program makes extensive use of the intrinsic functions listed in Section 4.1.3, “Using XL Fortran vector-intrinsic functions” on page 76 and the programmer does not wish to modify the code to use the MASS library functions, then there may be considerable benefit from using the vector versions from libxlopt. The -qhot option should then be used. In this case, the benefit from the vector-intrinsic functions can be determined by comparing the effect of compiling with -qhot and compiling with -qhot=novector.

Note: As described in Section 4.1.1, “General performance options” on page 70, -O4 implies -qhot.

4.1.5 Comparing C and Fortran compiler code generation

This section compares C and Fortran compiler code generation.

Numeric intensive code

The IBM Fortran and C compilers share common technology and, in particular, a common back-end optimizer. To investigate potential variations, we examined the code generated by the C and Fortran compilers for simple loops (DDOT and DAXPY). The C code was written using arrays instead of pointers for similarity with Fortran.

Using only the -O2 compiler option, the Fortran compiler generates unrolled loops while the C compiler does not. The Fortran examples run faster than the C examples.

Using the recommended compiler options (-O3 -qarch=pwr4 -qtune=pwr4), the code generated by the compilers was essentially the same. The execution times for the C and Fortran loops were within 1 percent variation. This is as expected since the back-end optimizer is common to both compilers.

The essential code is shown below:

```
ddot.f                          ddot.c
do j=1,iterations               for (i=0;i<iterations;i++) {
   c1=0.0d0                        c1=0.0;
   do i=1,array_size               for (j=0;j<array_size;j++) {
      c1 = c1 + x(i) * y(i)           c1 += x[j] * y[j];
   end do                          }
   call dummy(x,c1)                dummy(x,c1);
end do                          }
```

The example for DAXPY is similar to the code sample above:

```
x(i) = x(i) +c1 * y(i)          x[j] += c1*y[i];
```

Non-numeric intensive code

We made a brief investigation of the impact of compiler options on non-numeric C code by compiling applications with -O3 -qarch=com and -O3 -qarch=pwr4 -qtune=pwr4 and compared execution time. In one case, we compared the performance of the UNIX utility **nroff** and in another we compared a string manipulation script written in Perl (where the Perl compiler was compiled with the different options). In both the nroff and Perl cases, there was no difference in execution time.

We compiled the FASTA program (see Section 8.6, “FASTA genetic sequencing program” on page 168) with both -qarch=com and -qarch=pwr3 -qtune=pwr3 and re-ran the arp_arath test. Execution times of the com, pwr3, and pwr4 versions were within 1 percent variation. Since this program performs large amounts of I/O, we consider these execution times equivalent.

Note that -qarch=com will ensure that the compiler uses only standard PowerPC instructions. Optional instructions such as the hardware floating-point square root and non-PowerPC instructions such as the POWER2 loadquad instruction will not be used. This can have a significant impact on performance.

4.2 XL Fortran compiler directives for tuning

A number of XL Fortran compiler directives exist that the programmer can use to improve performance without extensive modification of source code. These directives can be activated at compile time by specifying the -qdirective option with the trigger expression that has been used. These directives are discussed in the following sections.

4.2.1 Prefetch directives

Prefetch directives are directives that generate specific machine instructions for accessing memory locations. They can be used to influence the hardware prefetch mechanism so that, for example, data that will be needed later in the execution begins to be prefetched before it is actually needed. Not all of these prefetch directives have an effect on all machine architectures.

- PREFETCH_BY_LOAD

  This generates a load byte and zero (lbz) instruction for a memory location. It can be used to trigger prefetching for data that may be loaded or stored later.

  As discussed in Section 2.3.2, “Instruction fetch, group formation, and dispatch” on page 9, load misses are entered into the prefetch filter queues, and on confirmation will automatically initiate prefetching. This is not the case for store misses. By using the PREFETCH_BY_LOAD directive and specifying a data element to be stored, it is therefore possible to precede the store miss with a load miss that will be entered into the prefetch filter queue. A second prefetch directive to the next cache line in the desired direction will initiate prefetching of the cache lines where the data will be stored.

  This directive (and the related technique of multiplying a data element to be stored by 0.0) was quite useful on the POWER3 architecture machines. With POWER4, it is less useful with the exception of store only or initialization operations. An example of its usage follows:

```
do i=1,n
!P4_bl* PREFETCH_BY_LOAD(x(i+17))
   x(i) = s
.
.
end do
```

  In this example, where x is a double precision floating-point array, the prefetch directive generates a load instruction for a data element in the next cache line beyond x(i). Subsequent iterations will issue loads for consecutive elements of x in this cache line, with a new cache line being referenced every 16 iterations. The exact offset from i used in the prefetch directive depends on the size of the loop.

  This directive is not always beneficial and can be detrimental to performance. The use of this directive inserts extra load instructions into the executable code that must be scheduled and completed among the other instructions, and for which the data will be loaded into L1 cache. This will not benefit the store operation, and may replace data that would otherwise be reused from L1 cache.

- PREFETCH_FOR_LOAD, PREFETCH_FOR_STORE

  Each of these directives generates a cache line touch instruction (dcbt and dcbtst respectively). They will cause a cache line to be loaded into L1, but will not by themselves initiate hardware prefetching. However, they are treated like load misses and generate entries in the prefetch filter queue. Subsequent directives targeting consecutive cache lines will therefore initiate prefetching.

  These directives have an advantage over the PREFETCH_BY_LOAD directive in that the instructions generated do not have to wait for the cache line to be loaded for completion.

4.2.2 Loop-related directives

These directives are used to specify the characteristics of do loops in Fortran and instruct the compiler to perform certain optimizations in relation to do loops. They can also be used in association with automatic parallelization using the -qsmp option.

- ASSERT

  This directive can be used to specify likely iteration counts and dependency information between iterations (not within an iteration) for a specific do loop.

- INDEPENDENT

  This directive indicates that the iterations of a do loop can be performed in any order.

- UNROLL(n)

  This directive indicates that the compiler may unroll the following loop to depth n. If the compiler can unroll the specified loop, then it should do so. This is most useful for unrolling a particular loop in a compilation unit while preventing other loops from being unrolled with the -qnounroll compiler flag. Another use of this directive is to specify a different depth to unroll from that which the compiler would select automatically at optimization level -O2 and above.

- CNCALL

  This directive indicates to the compiler that no dependencies between iterations exist for procedures called by the following loop.

- PERMUTATION

  This directive indicates to the compiler that one or more integer arrays have no repeated values. This would be used where the integer array was being used to index another array.

4.2.3 Cache and other directives

In this section, the cache_zero and light_sync directives are discussed.

- CACHE_ZERO

  This directive generates a dcbz instruction that zeros a cache line without needing to first load the cache line from memory. It could be used for efficient initialization of storage, or as a mechanism for establishing cache lines that will be overwritten without the need for them to be loaded into cache first (either by a load or store miss). However, this instruction should be used with care. It modifies the whole cache line, so the programmer should make sure that only data elements that it intended to set to zero are in this cache line and that no other processor requires access to the cache line until this operation is complete. For example, consider:

  ```
  CACHE_ZERO(x(1))
  ```

  This will cause the cache line containing x(1) to be set to zeros. There is no certainty that x(1) will be at the beginning of a cache line, and it could be anywhere in the cache line. It is, therefore, essential to check the location of this element with, for example:

  ```
  MOD( LOC( x(1) ), 128 )
  ```

  On POWER4 microarchitecture machines, this directive is likely to be of benefit only when the 128-byte line to be zeroed is in memory and not anywhere in the cache hierarchy. If the data is in L1 or L2 caches, then using this directive is likely to result in significant degradation in performance. If the data is in L3 cache, then there is likely to be a slight degradation in performance. However, when the programmer is sure that the data is not in cache, for example in an initialization near the beginning of a program, then this directive does give a performance benefit.

- LIGHT_SYNC

  This directive allows synchronization between multiple processors without waiting for a confirmation from each processor. This can reduce the performance impact of synchronization between processors. It generates a lightweight sync instruction, which is a special case of the sync instruction. It can be used to guarantee the ordering of loads and stores relative to a specific processor. This has some use in the pthread programming model where for example, one thread updates a value that is then used by a second thread. The lightweight sync can be used to ensure that the second thread does not access the value until the first thread has updated it.

  ```
  Thread 1                    Thread 2
  flag=0
  .                           do while ( flag .ne. 1 )
  .                              .
  .                              .
  ```

```
x=newvalue                      .
lightweight sync                .
flag=1                      end do
.                           y=x
```

The lightweight sync between the two store operations (shown in the preceding example) in Thread 1 means that these operations must be completed in order. This means that when Thread 2 polls the flag and finds that it has been set to one, the update of x must be complete and so it is safe to use this value.

4.3 The object code listing

This section reviews the process of obtaining an object code listing from the compiler. The object code listing shows the instruction sequences generated by the compiler and can assist the programmer in a number of ways:

- Understanding the impact of the compiler options used, such as the optimization and unroll flags.
- Identifying instructions that may be platform specific and therefore cause the code to fail on other platforms.
- Identifying potential problems with the compiler.

Obtaining an object code listing from a Fortran, C, or C++ compilation is simple. Invoke the compiler with the -qlist option. This will generate a file with the same prefix as the module being compiled but with an .lst extension.

For example, consider the following C program:

```
1  #include <stdio.h>
2  main()
3  {
4     printf("Hello world.\n");
5  }
```

Compiling with the -qlist option generates hello.lst. This file contains several sections depending on the language and compiler:

- The options section lists the compiler options (both default and those specified on the command line) that were used for the compilation.
- The file table section lists any included source files.
- The source section (only present if -qsource is specified) lists line-numbered source code and warnings and errors from the compiler.
- The compilation epilogue section provides a summary of the number of source lines processed, errors, warnings and other messages.

- The object section lists the pseudo-assembler code generated.

Note: Compiling with the -S flag will generate assembler code that can be read by the assembler, as (1) and used to generate a .o file.

The pseudo-assembler in the object section is somewhat easier to read (than that generated with -s) and the listing includes line numbers that are normally invaluable in understanding the listing. In addition to the assembler listing, the object section also contains a map showing register usage.

The following is part of the object section for the hello world program:

```
  | 000000                              PDEF     main
2 |                                     PROC
0 | 000000 mfspr    7C0802A6   1       LFLR     gr0=lr
0 | 000004 stw      93E1FFFC   0       ST4A     #stack(gr1,-4)=gr31
0 | 000008 stw      90010008   2       ST4A     #stack(gr1,8)=gr0
0 | 00000C stwu     9421FFC0   0       ST4U     gr1,#stack(gr1,-64)=gr1
0 | 000010 lwz      83E20004   1       L4A      gr31=.+CONSTANT_AREA(gr2,0)
4 | 000014 ori      63E30000   2       LR       gr3=gr31
4 | 000018 bl       4BFFFFE9   0       CALL     gr3=printf,1,gr3,printf",gr1
                                      ,cr[01567]",gr0",gr4"-gr12",fp0"-fp13"
4 | 00001C ori      60000000   1
5 | 000020 addi     38600000   0       LI       gr3=0
5 |                               CL.1:
5 | 000024 lwz      80010048   1       L4A      gr0=#stack(gr1,72)
5 | 000028 mtspr    7C0803A6   2       LLR      lr=gr0
5 | 00002C addi     38210040   1       AI       gr1=gr1,64
5 | 000030 lwz      83E1FFFC   0       L4A      gr31=#stack(gr1,-4)
5 | 000034 bclr     4E800020   2       BA       lr
```

The left-hand column in the example shows the corresponding source line number. Column two contains the relative instruction address and column three contains the instruction. The right-hand column contains the instruction operands.

Column five is a number indicative of the number of cycles to execute the instruction. A zero means the instruction can be overlapped with previous instructions. Note, these numbers should not be used to estimate execution time from cycle times, because they do not accurately reflect the POWER4 microarchitecture.

In the hello world example, locate the five instructions on line zero (which doesn't exist in a program). These instructions set up the stack and return address for the program. Instructions for line four set up for and then call the printf function. Instructions for line five restore the link register (program return address), collapse the stack, and exit.

Optimized code frequently shows more complex behavior. The optimizer will move code sequences, unroll loops, and use a number of other techniques that make it more difficult to interpret the object code listing. However, the line numbers associated with each instruction are preserved and you can identify code for given source lines without completely understanding why the compiler has generated the specific sequences. Consider the following example:

```
1     #define NX 1000000
2
3     main()
4     {
5        int j;
6        double f1;
7
8        double  e[NX], q[NX];
9
10       f1 = 1.5;
11
12       for (j=0;j<NX;j++)
13       {
14          q[j] = e[j] + f1*q[j];
15       }
16    }
```

This example was compiled with optimization for POWER4 and no loop unrolling. In this case we specified no loop unrolling to keep the object code small. The command used to compile is as follows:

```
xlc -O3 -qarch=pwr4 -qtune=pwr4 -qnounroll -qlist loop.c
```

The corresponding segment of the object list produced from this command is as follows:

```
   | 000000                          PDEF     main
 3 |                                 PROC
 0 | 000000 addis    3CE0000F   1    LIU      gr7=15
12 | 000004 addi     38600000   1    LI       gr3=0
 0 | 000008 addis    3D80FF0C   1    LIU      gr12=-244
10 | 00000C lwz      80A20004   1    L4A      gr5=.+CONSTANT_AREA(gr2,0)
 0 | 000010 addi     398CDBC0   1    AI       gr12=gr12,-9280
 0 | 000014 addi     38074240   1    AI       gr0=gr7,16960,ca"
```

```
 10| 000018 lfs      C0650000   1     LFS      fp3=+CONSTANT_AREA(gr5,0)
  0| 00001C mtspr    7C0903A6   1     LCTR     ctr=gr0
  0| 000020 stwux    7C21616E   1     ST4U     gr1,#stack(gr1,gr12)=gr1
  0| 000024 addis    3C810001   1     CFAA     gr4=32760,gr1,1
  0| 000028 addi     38848038   1
  0| 00002C addi     38848000   1     AI       gr4=gr4,-32768,ca"
  0| 000030 addis    3CC1007A   1     CFAA     gr6=8026200,gr1,1
  0| 000034 addi     38C67898   1
  0| 000038 addi     38C699A0   1     AI       gr6=gr6,-26208,ca"
 14| 00003C lfd      C8260008   1     LFL      fp1=q[]0(gr6,8)
 14| 000040 lfdu     CC040008   1     LFDU     fp0,gr4=e[]0(gr4,8)
 14| 000044 fmadd    FC03007A   1     FMA      fp0=fp0,fp3,fp1,fcr
  0| 000048 bc       4340001C   0     BCF      ctr=CL.26,taken=0%(0,100)
  0| 00004C ori      60000000   2
 12|                                 CL.3:
 14| 000050 lfdu     CC240008   1     LFDU     fp1,gr4=e[]0(gr4,8)
 14| 000054 lfd      C8460010   1     LFL      fp2=q[]0(gr6,16)
 14| 000058 stfdu    DC060008   1     STFDU    gr6,q[]0(gr6,8)=fp0
 14| 00005C fmadd    FC0308BA   1     FMA      fp0=fp1,fp3,fp2,fcr
  0| 000060 bc       4320FFF0   0     BCT      ctr=CL.3,taken=100%(100,0)
  0|                                 CL.26:
 14| 000064 stfdu    DC060008   1     STFDU    gr6,q[]0(gr6,8)=fp0
 16| 000068 lwz      80210000   1     L4A      gr1=#stack(gr1,0)
 16| 00006C bclr     4E800020   0     BA       lr
```

To locate a loop, we can look for a BCT instruction that branches back to a label and confirm this by checking line numbers. In our example, there are two BCT instructions. The relevant one is the second one with the additional hint *taken=100%*. Note that some instructions associated with the loop appear to be outside the loop code. This is caused by the instruction scheduling knowledge built in to the optimizer.

Before entering the loop, the loop counter is loaded using a mtspr (move to special register) instruction at address `01C` and the single precision constant f1 is loaded into fp3 and converted to double (at `018`). We also set up registers pointing to arrays e and q (`020` through `038`).

Starting at address `03C`, q[j] is loaded into fp1. The lfd instruction loads a double (8 bytes) into a floating-point register. Then the lfdu instruction loads e[j] into fp0 and also updates the register pointer to e[j]. Note that the address of e[j] is not computed from j but rather by incrementing by the size of a double. A floating-point multiply/add is initiated to generate the new value for q[j] in fp0.

The following points are related to the inner loop processing:

- The lfdu loads fp1 with the next value of e[j], updating the pointer.
- The lfd loads fp2 with the next element in the array q[j]. Note carefully the offset of 16 bytes from the pointer (gr6,16) and compare this with the previous lfd, where the offset was 8 bytes (gr6,8).
- By this time, the previous FMA has completed and put its result in fp0. The stfdu stores the new value of q[j] and then updates the pointer. Note the stfdu also uses (gr6,8).
- We then initiate an FMA for the e[j] and q[j] just loaded.
- The bc conditional branch tests the counter and branches back to CL.3 if appropriate.
- If we did not branch, we still need to store the result of the last FMA, hence the stfdu following CL.26.

From this basic example, we can see how the compiler can optimize code by use of registers as pointers and by appropriate scheduling of possibly overlapping instructions. In production code, optimized code can appear very complex.

A complete description of the instruction set can be found on the AIX Extended Documentation CD. It can also be found on the Web site:

http://www.ibm.com/servers/aix/library/techpubs.html

4.4 Basic coding practices for performance

In this section we list coding practices that can help the compiler to generate more efficient code.

4.4.1 Language-independent tips

- Do not excessively hand-optimize your code (for example, unrolling or inlining). This often confuses the compiler (and other programmers) and makes it difficult to optimize for new machines.
- Avoid unnecessary use of globals and pointers. When using them in a loop, load them into a local before the loop and store them back after.
- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using the -qipa option.
- Use register-sized integers (long in C/C++ and INTEGER*4 or INTEGER*8 in Fortran) for scalars.

- For large arrays or aggregates of integers, consider using 1- or 2-byte integers or bit fields in C or C++.
- Use the smallest floating-point precision appropriate to your program. Use long double, REAL*16, or COMPLEX*32 only when extremely high precision is required.
- Obey all language aliasing rules (try to avoid -qassert=nostd in Fortran and -qalias=noansi in C/C++).
- Use locals wherever possible for loop index variables and bounds. In C/C++, avoid taking the address of loop indices and bounds.
- Keep array index expressions as simple as possible. Where indexing needs to be indirect, consider using the PERMUTATION directive.

4.4.2 Fortran tips

- Use the [mp]xlf90[_r] or [mp]xlf95[_r] driver invocations where possible to ensure portability. If this is not possible, consider using the -qnosave option.
- When writing new code, use module variables rather than common blocks for global storage.
- Use modules to group related subroutines and functions.
- Use INTENT to describe usage of parameters.
- Limit the use of ALLOCATABLE arrays and POINTER variables to situations that demand dynamic allocation.
- Use CONTAINS in subprograms only to share thread local storage.
- Avoid the use of -qalias=nostd by obeying Fortran alias rules.
- When using array assignment or WHERE statements, pay close attention to the generated code with -qlist or -qreport. If performance is inadequate, consider using -qhot or rewriting array language in loop form.

4.4.3 C and C++ tips

- Use the xlc[_r] invocation rather than cc[_r] when possible.
- Always include string.h when doing string operations and math.h when using the math library.
- Pass large class/struct parameters by address or reference and pass everything else by value where possible.
- Use unions and pointer type-casting only when necessary and try to follow ANSI type rules.

- If a class or struct contains a double, consider putting it first in the declaration. If this is not possible, consider using -qalign=natural.
- Avoid virtual functions and virtual inheritance unless required for class extensibility. These are costly in object space and function invocation performance.
- Use volatile only for truly shared variables.
- Use const for globals, parameters and functions whenever possible.
- Do limited hand-tuning of small functions by defining them as inline in a header file.

4.4.4 Inlining procedure references

Inlining involves replacing a procedure reference with a copy of the procedure's code, so that the overhead of referencing the procedure, and of returning from it, is eliminated. In certain situations inlining can enable the compiler to perform more optimization than without inlining.

The general advice is to avoid inlining in areas of a program that are infrequently executed and to ensure that small functions are inlined in frequently executed areas. Do not inline large functions. Inlining does not always improve performance; therefore you should test the effects of this option on your code.

A program with inlining might slow down because of larger code size resulting in more cache misses and page faults, or because there are not enough registers to hold all the local variables in some combined routines (check the compiler output for register spills).

Inlining by the compiler is controlled through the -Q and -O options and the suboptions of the -qipa (not available for C++). You must specify at least optimization level -O (equivalent to -O2) for -Q inlining to take effect. In Fortran, by default, -Q only affects a procedure if both the caller and callee are in the same source file or set of files that are connected by INCLUDE directives. To turn on inline expansion for calls to procedures in different source files, you must also use the -qipa option.

The compiler decides whether to inline procedures based on their size. Other criteria might help to improve performance. For procedures that are unlikely to be referenced in a standard execution (for example, error-handling or debugging procedures), you might selectively disable inlining by using the -Q-*names* option. For procedures that are referenced within hot spots, specify the -Q+*names* option to ensure that those procedures are always inlined.

Getting the right amount of inlining for a particular program may require some trials. As a good starting point, consider identifying a few subprograms that are called most often, and inline only those subprograms.

To verify whether the compiler has inlined the call of a certain procedure or not, you can check whether the call has disappeared in the object listing (-qlist). The following example shows how a call to a subroutine (named foo) may appear:

```
9| 000038 bl       4BFFFFC9   0     CALL gr3=foo,3,a",gr3,#1",gr4,i",gr5,
fcr",foo",gr1,cr[01567]",gr0",gr4"-gr12",fp0"-fp13",mq",lr",fcr",xer",fsr",ca",
ctr"
```

In C++ there are two methods to define a functions as inline: by using the inline keyword or by defining (not just declaring) member functions within the class definition. Inline functions are not necessarily inlined by the compiler, and functions that are not defined as inline may still be inlined, depending on the optimization level and the -Q compiler flag.

4.4.5 Structuring code for optimal grouping

The grouping of the internal microprocessor instructions is important in order to exploit the potential performance of the different hardware execution units for a specific calculation. As a general rule, it is desirable to fill out all four slots (five in case of a branch) of an instruction group. Instructions that have to be first or last in a group may prevent optimal grouping. Flushing instruction groups and refetching instructions for reordering is a worst case situation to be avoided if possible.

There are no means to influence instruction grouping from the C or Fortran language level directly. The compiler has to cope with the requirements of grouping. Only when writing assembler code is it possible to arrange the order of instructions in order to optimize grouping.

Writing suitably structured high-level code might help the compiler to generate an instruction stream, which can be grouped nicely. The key issues to be considered carefully are proper alignment and data dependencies and these tuning techniques are beneficial for overall performance anyway.

4.5 Tuning for 64-bit integer performance

Given a program that uses 64-bit integer data types, you need to compile with the -q64 option in order to exploit the 64-bit integer hardware support of POWER3 and POWER4. Note that specifying the -q64 compiler option does not affect the default setting for -qintsize.

In 64-bit mode, the use of INTEGER(8) induction (loop counter) variables can improve performance. The XLF 7.1 compiler automatically converts induction variables declared as INTEGER or INTEGER(1) or (4) to INTEGER(8) unless -qSTRICT_INDUCTION is specified. It is no longer necessary to set the size of default INTEGER and LOGICAL data entities (-qintsize) to 8 bytes in order to achieve this goal without source code changes. In this case the usage of -qintsize=8 could increase the memory consumption and bandwidth requirements unnecessarily.

Figure 4-1 shows some performance implications. A simple add operation, B(I) = A(I) + C, is selected. In the context of this example 32-bit denotes 32-bit array elements and 32-bit address space; 64-bit indicates 64-bit integer elements. Fetching data from L1 and L2 cache, the 64-bit version with hardware support (-q64) is not slower than the 32-bit version. The 32-bit version is faster when going out to L3 cache and to memory. Twice the number of elements are kept in L2 and L3 cache, so the performance degradation is delayed.

Without the 64-bit integer hardware support, the performance of the operation with 64-bit operands is significantly worse. One should expect twice the number of load, store, and add instructions. But the object code listing reveals that 64-bit emulation is more complex. In addition, addic (add with carry) instructions are generated, which probably lead to inefficient instruction grouping.

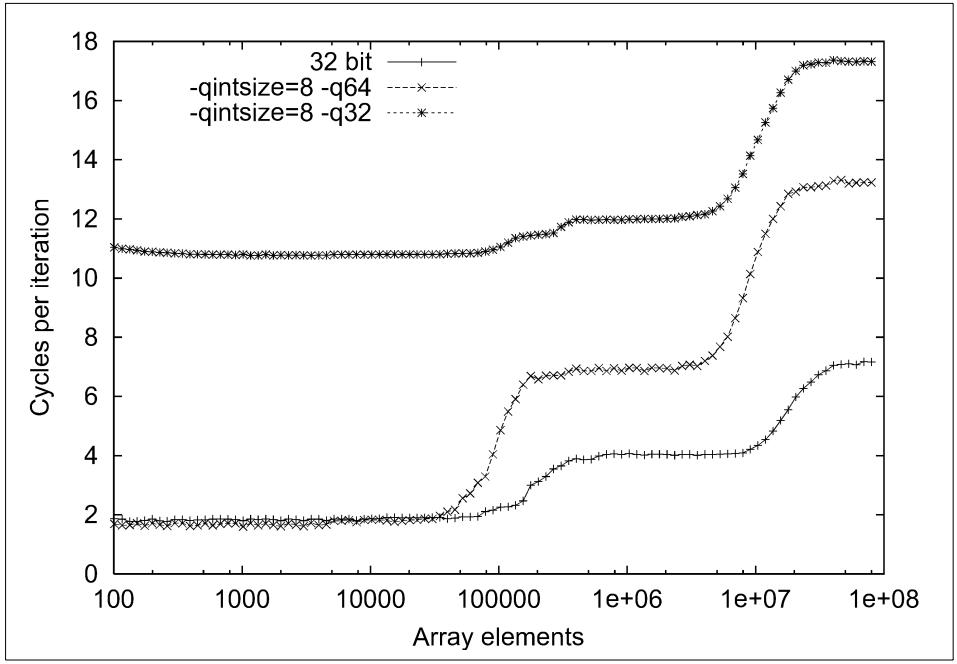


Figure 4-1 Integer computation: B(I)=A(I)+C

5

General tuning guidelines

This chapter covers general code tuning and application optimization techniques that are not specific to POWER4 microarchitecture. It is intended to be a repository of recommended coding practices.

5.1 Hand tuning code

Many of the following tips and advice can be found in the IBM Fortran and C compiler manuals.

5.1.1 Local or global variables?

Use local variables, preferably automatic variables, as much as possible. The compiler can accurately analyze the use of local variables, but it has to make several worst-case assumptions about global variables. These assumptions tend to hinder optimization. For example, if you write a function that uses global variables heavily, and that function also calls several external functions, the compiler assumes that every call to an external function could change the value of every global variable. If you know that none of the function calls affects the global variables that you are using, and you have to read them frequently with function calls interspersed, copy the global variables to local variables and then use these local variables. The compiler can then perform optimization that it could not otherwise perform.

In C, if you must use global variables, use static variables with file scope rather than external variables wherever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.

To access an external variable, the compiler has to make an extra memory access to obtain the address of the variable. When the compiler removes extraneous address loads, it has to use a register to keep the address. Using many external variables simultaneously takes up many registers. Those that cannot fit into the available registers during optimization are spilled into memory. The C compiler organizes all elements of an external structure so that they use the same base address and hence base address register. Therefore, you should group external data into structures or arrays wherever it makes sense to do so. Do not group together data whose address is taken (either explicitly using an ampersand (&) or implicitly, including arrays passed as parameters and C++ class objects passed as *this* parameters) in the same structure with other data whose address is not taken.

In C, because the compiler treats register variables the same as it does automatic variables, you do not gain performance by declaring register variables. Note that this differs from other vendors' implementations, where using the register attribute can greatly affect program performance. However, declaring a variable as register is a good hint to the compiler and means the variable cannot be dereferenced.

5.1.2 Pointers

Keeping track of pointers during optimization is difficult and in some cases impossible. Using pointers inhibits most memory optimization (such as dead store elimination and store motion).

Using the C #pragma disjoint preprocessor directive to list identifiers that do not share the same physical storage can improve the runtime performance of optimized code.

5.1.3 Expressions

The Fortran compiler is good at recognizing identical expressions but not permutations of them. For example, in the code:

```
x = a + b + c + d
y = a + c + b + d
```

the compiler will only load the variables a, b, c, and d into registers once. However, it evaluates the expressions separately and stores the results from separate registers. Wherever possible, write identical expressions specifying variables in the same order.

5.1.4 Data type conversions

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations.

Do not use floats (datatype real) for loop variables.

While this is not a performance issue, when comparing floating-point numbers for equality, bear in mind that values may vary in the least significant bit, depending on how the value is calculated. Where appropriate, test that the unsigned difference between the values is less than an acceptable threshold of accuracy.

5.1.5 Tuning loops

There are a variety of techniques, basically good coding practice, that can be applied to tuning loops. These techniques are not POWER4 specific. They are described in the Fortran context but also apply to C.

- Keep the size of do loops manageable

  Loops should not spread over many lines of code. If they do, there probably exists a better algorithm. Large loops also make program maintenance more difficult.

- Access data sequentially (unit stride)

  Wherever possible, organize data arrays so that elements are accessed with unit stride to improve cache utilization. Note, in Fortran arrays, elements a(1,n), a(2,n), a(3,n) and so on are in sequential memory locations. In C arrays, the order is reversed, that is a[n][1], a[n][2] and a[n][3] are in sequential locations.

- Minimize loop invariant IF statements in loops

  Reduce the loop instruction path length by moving IF statements outside the loop and coding two separate loops. This is more important for small loops where the IF test may be a significant contributor to the loop execution time.

- Avoid subroutine and function calls in loops (give routine its own loop)

  Avoid subroutine calls within loops (where possible) to save the cost of the branch and link instructions. Use code inlining instead.

Alternatively, replace the loop with its subroutine call for each element with a subroutine call, passing an array parameter, where the subroutine contains a loop operating on each element.

- Simplify array subscripts

  Avoid writing complex expressions for array subscripts if possible, particularly expressions involving loop variables. Complex expressions may cause the compiler to compute the array index even if the increment is fixed. For fixed increments, the compiler can generate array element addresses with register add instructions instead of a more complex calculation.

- Use INTEGER loop variables

  Integer loop variables simplify loop counter optimization, since the counter can be kept in a register and also used as an address index. Do not use INTEGER*8 loop variables unless in 64-bit mode. Use of REAL loop variables is strongly discouraged because both of the following affect performance:

  – Calculation of the loop variable requires one or more floating-point operations

  – Use of the loop variable as an index requires conversion

  In addition, the test for loop termination may be inexact and the loop may be executed one fewer or one more time than expected.

  In C, declare loop variables as type long. Long variables are the natural or register size in 32-bit and 64-bit environments, that is they are 32 or 64 bits accordingly. Loop variable arithmetic using the register size has significantly better performance than using, for example, integer (32-bit) loop variables in a 64-bit environment. Note that in the 64-bit environment, the C compiler will optimize integer loop variables to long if you specify -O3 (or greater) and do not specify -qstrict_induction.

- Avoid the following constructs within loops:

  – Flow control statements such as GOTO, STOP, PAUSE, RETURN computed GOTO, ASSIGN, or ASSIGNED GOTO

  – EQUIVALENCE data items

  These constructs impair the ability of the compiler to optimize the loop.

- Avoid non-optimizable data types such as LOGICAL*1, BYTE, INTEGER*1, INTEGER*2, REAL*16, COMPLEX*32, CHARACTER, and INTEGER*8 in 32-bit mode.

  These data types do not correspond to the native hardware types and require additional instructions for each operation, impacting performance.

For performance-critical do loops, avoid the following:

- Access data with large stride

  Large strides reduce the effectiveness of the cache

- Do few iterations of the loop

  For very small numbers of loop iterations, it may be preferable to unroll the loop by hand.

- Include I/O statements

  I/O can introduce indeterminate delays in processing. I/O function calls will also prevent automatic parallelization of loops.

Examples

These are some examples of how to correct some inefficient coding practices that have been found in real codes:

Removal of invariant IF

```
      Untuned
      -------
DO I=1,N
  IF(D(J).LE.0.0)X(I)=0.0
  A(I)=B(I)+C(I)*D(I)
  E(I)=X(I)+F*G(I)
ENDDO
```

```
       Tuned
       -----
IF(D(J).LE.0.0)THEN
  DO I=1,N
    A(I)=B(I)+C(I)*D(I)
    X(I)=0.0
    E(I)=F*G(I)
  ENDDO
ELSE
  DO I=1,N
    A(I)=B(I)+C(I)*D(I)
    E(I)=X(I)+F*G(I)
  ENDDO
ENDIF
```

The compiler will recognize that the IF test is invariant within the loop but will not generate two versions of the loop as in the tuned example.

Boundary condition IF testing

A frequent requirement is to perform a different calculation for the first and/or last iteration of a loop. If the loop is performance-critical, then it is important to treat these special cases separately and remove the IF code from the main loop:

```
     Untuned                          Tuned
     -------                          -----
DO I=1,N                        A(1)=B(1)+C(1)*D(1)
  IF(I.EQ.1)THEN                X(1)=0.0
    X(I)=0.0                    E(1)=F*G(1)
  ELSEIF(I.EQ.N)THEN            DO I=2,N-1
    X(I)=1.0                      A(I)=B(I)+C(I)*D(I)
  ENDIF                           E(I)=X(I)+F*G(I)
  A(I)=B(I)+C(I)*D(I)           ENDDO
  E(I)=X(I)+F*G(I)              X(N)=1.0
ENDDO                           A(N)=B(N)+C(N)*D(N)
                                E(N)=1.0+F*G(N)
```

Repeated intrinsic function calculation

In this example, the untuned code calls SIN() N^2 times, whereas in the tuned code, it is called N times and saved in a separate array. In the inner loop, the call is replaced by a significantly cheaper load.

```
     Untuned                          Tuned
     -------                          -----
DO I=1,N                        DIMENSION SINX(N)
  DO J=1,N                         .
    A(J,I)=B(J,I)*SIN(X(J))     DO J=1,N
  ENDDO                           SINX(J)=SIN(X(J))
ENDDO                           ENDDO
                                DO I=1,N
                                  DO J=1,N
                                    A(J,I)=B(J,I)*SINX(J)
                                  ENDDO
                                ENDDO
```

Replacing divides by reciprocal multiply

This optimization can sometimes be done automatically by the compiler by specifying at least -O3 optimization level.

Since divides are costly, any loop that divides by the same value more than once can be easily optimized by taking the reciprocal of the value and then multiplying by the reciprocal, as in this example:

```
   Untuned                  Tuned
   -------                  -----
DO I=1,N                 DO I=1,N
  A(I)=B(I)/C(I)           OC=1.0/C(I)
  P(I)=Q(I)/C(I)           A(I)=B(I)*OC
ENDDO                      P(I)=Q(I)*OC
                         ENDDO
```

In practice, any improvement will depend on the ratio of divides to loads and stores. For trivial loops, there is no benefit for *reals* but there is a benefit for integers.

The following example shows a similar method that has been used when there are two (or more) different divisors:

```
   Untuned                  Tuned
   -------                  -----
DO I=1,N                 DO I=1,N
  A(I)=B(I)/C(I)           OCD=1.0/( C(I)*D(I) )
  P(I)=Q(I)/D(I)           A(I)=B(I)*D(I)*OCD
ENDDO                      P(I)=Q(I)*C(I)*OCD
                         ENDDO
```

Here, two divides have been replaced by one divide and five multiplies. In the untuned case, the compiler can take advantage of the multiple FPU pipelines, whereas in the tuned case, the code is dependent on a single floating-point divide.

Array dimensions that are high powers of two

The following discussion is an extension of what was described in “Set associativity” on page 29.

The following code elements illustrate a problem that can arise with array dimensions:

```
      integer nx,nz
      parameter (nx=2048,nz=2048)
      real    p(2,nx,nz)
...
...
do 25 ix=2,nx-1
              do 20 iz=2,nz-1
                   p(it1,ix,iz)=  -p(it1,ix,  iz)
     &                         +s*p(it2,ix-1,iz)
     &                         +s*p(it2,ix+1,iz)
     &                         +s*p(it2,ix,  iz-1)
     &                         +s*p(it2,ix,  iz+1)
20            continue
25         continue
```

The second dimension of p multiplied by the first dimension (in this case two) is precisely one half of the size of the L1 cache. Array elements p(it2,ix-1,iz) and p(it2,ix+1,iz) will (normally) be found in the same cache line as p(it2,ix,iz). However, accessing p(it2,ix,iz-1) and p(it2,ix,iz+1) will displace this cache line because each of these elements map to the same congruence class. In this example, there are five loads to the same congruence class but only two cache lines available, because L1 is two-way associative.

A simple solution is to increase the dimension of the array, as in

```
      integer nx,nz
      parameter (nx=2048,nz=2048)
      real    p(2,2080,nz)
```

Note that we are simply changing the dimension of the array, not the number of elements accessed. In this example, the application performance increased by a factor of two.

You can also get multiple loads to the same congruence class in loops that access a large number of arrays because there are only 128 classes. In this case, it is possible to improve performance by splitting the loop into multiple loops and relocating array accesses into these separate loops.

5.2 Using pre-tuned code

Do not spend time duplicating tuning work that has already been done. If your program performs standard functions, such as matrix multiply, equation solving, other BLAS functions, FFTs, convolution, and so on, then modify your code to call the equivalent ESSL function. ESSL is described in 6.1, “The ESSL and Parallel ESSL libraries” on page 114, and contains probably the most highly tuned code available for RS/6000 and pSeries numerically intensive functions. Other commercially and publicly available libraries, such as NAG, IMSL, LAPACK, and so on, have also been tuned for cache-based superscalar architectures.

5.3 The performance monitor

The POWER3 and POWER4 processor designs (as well as RS64) include hardware performance monitoring facilities. These facilities provide access to counters that record highly detailed information about processor behavior and instruction execution. At the lowest level, the interface consists of special-purpose registers that control the state of counters and multiplexors within the processor. These registers are only accessible at the operating system level; therefore a programming interface is provided that accesses these registers using a kernel extension.

The hardware provides eight counters each of which can count the number of occurrences of one event. Events are things that happen inside the processor such as the completion of an instruction or a load from a cache line. Events are platform specific, therefore, certain events may exist on one processor type but not another.

The programming interface provides a set of C routines to specify which events should be counted, whether they should be counted for the kernel, the user, or at the process level. Counting can be turned on or off within a program, thereby providing a very accurate mechanism for determining processor usage in specific parts of an application.

The API and documentation are provided on the AIX 5L installation media.

There is also a command, **pmcount**, which will execute a command or script. You can specify countable events as options to **pmcount**. Using another set of options, **pmcount** will display event numbers and their definitions for the current hardware platform.

The POWER3 and RS64 implementations allow the counting of events in any counter for which the event is defined (although not all combinations may be meaningful in the sense that the set of multiplexors used to accumulate into a specified counter may not produce a meaningful result).

Event counting has been refined to provide a number of groups of events for each processor type. The definition of a group is simply a set of eight events and the particular counters on which they are counted. Combinations of events in a group are meaningful.

A group may be specified as an option to **pmcount** in place of a set of events.

The increased level of complexity of the POWER4 design means that it is more difficult to guarantee meaningful results from counting events. Therefore, only counting by groups is supported.

The following example illustrates some of the techniques that may be useful in programming the API:

```
#include <stdio.h>
#include <sys/time.h>
#include <malloc.h>
#include <stdlib.h>
#include “pmapi.h”

#define STRIDE_MAX 4096
#define NUM_LOOPS 100

void timevalsub(struct timeval *, struct timeval *);
void timevaladd(struct timeval *, struct timeval *);
void invalidate_tlb();

main(int argc, char **argv)
{

   int       i,j, testcount, /* various loop variables */
             rc,             /* return code */
             stride, group_no;/* parameters */
   char  *progname;
   float     x,
             array[512][513];

   /* timestamps for loop start, end */
   struct timeval     loop_start, loop_end, total;

   /* process monitor data structures */
   pm_info_t       myinfo;
   pm_groups_info_tmygroupinfo;
```

```
pm_prog_t       myprog;
pm_data_t       mydata;


stride=1;
total.tv_sec=0; total.tv_usec=0;

/* make sure all pages in array exist to minimize later timing issues */
for (i=0;i<512;i++) {
   for (j=0;j<513;j++) {
      array[i][j]=0.0;
   }
}

progname = *argv;

if (argc == 3 ) {
   argv++;
   group_no=atoi(*argv);
   argv++;
   stride=atoi(*argv);
} else {
   printf("usage: %s group stride\n",progname);
   exit(1);
}

/* initialize API. Allow all possible events. */
if ((rc = pm_init(PM_VERIFIED|PM_UNVERIFIED|PM_CAVEAT,
          &myinfo,&mygroupinfo)) > 0) {
             pm_error("pm_init", rc);
             exit(-1);
}

/* set up counting modes for call to pm_set-program_mythread() */

myprog.mode.w = 0;
/* count in user mode, not kernel mode */
myprog.mode.b.user = 1;
myprog.mode.b.kernel = 0;
/* defer starting counting until we call pm_start_mythread */
myprog.mode.b.count = 0;

/* set is_group to say we're counting groups rather than events */
myprog.mode.b.is_group = 1;
/* since we're counting groups, put the group number into events[0].
   The API won't look at other events[] structures. */
myprog.events[0]=group_no;

if ((rc=pm_set_program_mythread(&myprog)) != 0 ) {
```

```
        pm_error("Calling pm_set_program_mythread",rc);
        exit(1);
    }

    testcount=NUM_LOOPS;

    while (testcount-- > 0 ) {
        invalidate_tlb();
        gettimeofday(&loop_start,NULL);
        /* Start counting. We don't want to include the overhead of the
         invalidate_tlb() and gettimeofday() calls so we start and stop
         counting accordingly */
        if ((rc=pm_start_mythread()) != 0 ) {
            pm_error("Calling pm_start_mythread",rc);
            exit(1);
        }
        for (i=0;i<512;i++) {
            for (j=0;j<512;j+=stride) {
                array[i][j]=array[i][j] * 1.5;
            }
        }

        /* Stop counting but don't reset the counters. Therefore, counting
         will simply continue on the next call to pm_start_mythread() */

        if ((rc=pm_stop_mythread()) != 0 ) {
            pm_error("Calling pm_stop_mythread",rc);
            exit(1);
        }
        gettimeofday(&loop_end,NULL);
        timevalsub(&loop_end,&loop_start);
        timevaladd(&total,&loop_end);
    }

    /* retrieve counter data */
    if ((rc=pm_get_data_mythread(&mydata)) != 0 ) {
        pm_error("pm_get_data_mythread",rc);
        exit(1);
    }

    x=(total.tv_sec*1000000)+total.tv_usec;
    printf("Time (usecs) = %8.2f\n",x/NUM_LOOPS);
    for (i=0;i<8;i++)
        printf("Counter %d = %-8lld\n",
            i+1,mydata.accu[i]/NUM_LOOPS);

    return(0);
}
```

Running this program produces the output below:

```
bu30b$ ./pm_example 5 8
Inside pm_set_program_mythread: prog->events[0] is 5.
  prog->mode.b.is_group is 1.
Time (usecs) =   375.19
Counter 1 = 249
Counter 2 = 215
Counter 3 = 0
Counter 4 = 7966
Counter 5 = 0
Counter 6 = 0
Counter 7 = 0
Counter 8 = 0
bu30b$
```

The first two lines of output are generated by the pm_set_program_mythread() call, apparently as diagnostic information. The program prints the elapsed time and counter values. We previously used **pmcount** to identify the groups and counters. Group 5 counts information on sources of data. The definitions for the individual counters used in this example are as follows:

Counter 1 The number of times data was loaded from L3

Counter 2 The number of times data was loaded from memory

Counter 3 The number of times data was loaded from L3.5

Counter 4 The number of times data was loaded from L2

Counter 5 The number of times data was loaded from L2 partition 1 in shared mode

Counter 6 The number of times data was loaded from L2 partition 2 in shared mode

Counter 7 The number of times data was loaded from L2 partition 1

Counter 8 The number of times data was loaded from L2 partition 2

Thus, in this example, we can see the relative sources of data for the calculation. Other groups can be used to identify the efficiency of the prefetch mechanism, floating-point unit and so on.

The API described above is provided in C. There is no Fortran API. However, it is a reasonable task to write a suitable, simplified API.

Subroutines to initialize the performance monitor, start and stop counting, and print results are required. Here is some pseudo-code that implements these subroutines.

```
#include <sys/types.h>
#include “pmapi.h”
int pminit(int group)
{
   pm_info_t    pm_myinfo;
   pm_groups_info_t   pm_mygroupinfo;
   pm_prog_t    myprog;

   pm_init(PM_VERIFIED|PM_UNVERIFIED,&pm_myinfo,&pm_mygroupinfo);
   myprog.mode.b.user=1; myprog.mode.b.kernel=0; myprog.mode.b.count=0;

   myprog.mode.b.is_group=1; myprog.events[0]=group;
   pm_set_program_mythread(&myprog);
   return(0);
}

int pmstart()
{
   pm_start_mythread();
   return (0);
}

int pmstop()
{
   pm_stop_mythread();
   return(0);
}

int pmprint()
{
   int i;
   pm_data_t    my_data;
   pm_get_data_mythread(&my_data);
   for (i=0;i<8;i++)
   {
      printf(“Counter %d = %-8lld\n”,i+1,my_data.accu[i]);
   }
   return(0);
}
```

You will need to compile these functions and save the object file for later use. The -c option tells the compiler that the source file is not a complete program and it should stop after the compilation stage and not attempt to link. For example:

```
xlc -O3 -c -o pm_subroutines.o mysourcefilename.c
```

Here is how you might use these subroutines to monitor a Fortran program. Note that we have passed the group to be monitored by value explicitly because Fortran, by default, passes parameters by reference:

```
program pm_test

       integer pminit,pmstart,pmstop,pmprint,i,j
       integer result,group
       real*8 x,a(512,512)

       group=5
       result = pminit(%VAL(group))

       result = pmstart()

       do i=1,512
         do j=1,512
           a(i,j) = a(i,j) *1.5
         end do
       end do
       result = pmstop()
       result = pmprint()

       end program
```

To compile this program, we need to include the C subroutines and the performance monitor libraries:

```
xlf -O3 -o pm_test pm_subroutines.o -lpmapi -L/usr/pmapi/lib pm_test.f
```

Note that the performance monitor has not been tested in LPAR environments.

5.4 Tuning for I/O

If I/O is a significant part of the program, it may well dominate the overall run time and render CPU tuning unproductive. Some guidelines for improving I/O efficiency in Fortran and C are discussed in the following sections. However, the best advice is simply to eliminate or minimize I/O as much as possible. If I/O is your performance bottleneck, then using the best hardware and software options (high-speed storage arrays, striping over multiple devices and adaptors, and asynchronous I/O, for example) may be the best tuning options. A detailed discussion of these subjects is outside the scope of this publication.
Large-memory SMP systems are capable of generating large amounts of I/O, but different I/O subsystems have different performance characteristics, so it is difficult to make specific recommendations.

Asynchronous I/O

Programs normally perform I/O synchronously. That is, execution continues after the operating system has completed the I/O. AIX also supports asynchronous I/O. In this case, a program executes an I/O call that returns immediately. The program can then perform other useful work. The operating system will perform the I/O and inform the program when it's complete. There are a variety of techniques for the program to detect that the I/O has finished.

Taking advantage of asynchronous I/O can result in reduced run time because you can overlap computation and I/O. The degree of improvement will depend on the amount of I/O the program performs.

Implementing asynchronous I/O will require program changes and the degree of difficulty will vary from program to program.

In Fortran (introduced in XL Fortran Version 5), a program can open a file with the ASYNC qualifier. Read and writes will be performed asynchronously. The program needs to be changed to issue a wait for each asynchronous read or write. a description of asynchronous I/O, including a discussion of error handling, can be found in the XL Fortran Language documentation.

In C, asynchronous I/O is only supported for unbuffered I/O. The program changes required for asynchronous I/O are typically more complex than those required in Fortran. Refer to "Asynchronous I/O Overview", in the *AIX Version 4.3 Kernel Extensions and Device Support Programming Concepts* documentation.

Direct I/O

Direct I/O is a form of synchronous I/O. By default, the operating system transfers data between the application program and a file using intermediate buffers. For example, for file system files, the operating system caches file data and this typically improves I/O performance. Using direct I/O data is transferred between the device and the application's data buffers without intermediate buffering. This can sometimes lead to degraded performance, typically with file system files.

Paging I/O

Paging is a special case of I/O. You can measure paging rates using **`vmstat`**. This command displays, among other statistics, the paging rates for a specified time interval. It displays these statistics for the whole system, which must be taken into account when evaluating the effect of a particular application. A certain amount of paging during startup or when the program changes from one phase to another is to be expected. However, any measurable paging rate over a sustained period during program execution is an indication that you are over-committing memory or are on the edge of doing so. This is likely to cause

serious performance problems. The only solution is to reduce the level of memory over-commitment. Either tune the program to use less memory, or run on a computer with more memory (or fewer users). It should be noted that from AIX Version 4.3.2 on, the paging space allocation algorithm only allocates a page of paging space when it actually needs to write to that page. This means that it is common for the amount of paging space configured on a large memory system to be considerably less than the size of memory. In this situation any significant amount of paging can have more serious effects than poor performance of an application, as the system can quickly reach a state where virtual memory is exhausted and thrashing ensues.

The **topas** command is also a useful real time monitor of system I/O activity.

C unbuffered and buffered I/O

C programs can make use of two techniques for I/O. These are buffered I/O, also referred to as streams, and unbuffered I/O. Unbuffered I/O is implemented by calls to operating system functions and offers the greatest opportunity for performance at a cost in coding complexity. Buffered or stream I/O is implemented by standard library functions that provide a higher level interface. Refer to the “Input and Output Handling Programmer’s Overview” in *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs*.

Fortran I/O

Some guidelines for efficient I/O in Fortran follow:

- Reduce the number of calls to the I/O subsystem.

  For example, the following three ways of writing the whole of a 2-D array to a sequential file differ very considerably in performance. As well as performing very slowly, Case 3 will create a file almost twice as large as Case 1 (if A is REAL*8) because of the extra record length indicators.

```
      DIMENSION A(N,N)
         .
         .

Case 1. Best. 1 record of N*N values.
      WRITE(1)A

Case 2. N records, each of N values.
      DO I=1,N
         WRITE(1)(A(J,I),J=1,N)
      ENDDO

Case 3. Worst. N*N records, each of one value.
      DO I=1,N
         DO J=1,N
```

```
            WRITE(1)A(J,I)
          ENDDO
        ENDDO
```

- Use long record lengths when reading or writing files in a sequential fashion.

  Use at least 100 KB if possible, preferably 2 MB or more. This allows the I/O to access the underlying devices more effectively.

- Prefer Fortran unformatted I/O to formatted.

  This reduces binary to decimal conversion overhead.

- Prefer Fortran direct files to sequential.

  This avoids Fortran record length and overflow checking. A Fortran direct file in AIX is a simple sequential series of data bytes. A Fortran sequential file has record length indicators at both ends of each record.

- Use asynchronous I/O to overlap computation with I/O activity.

- If you write a large temporary file sequentially and need to read through it again at a later stage in processing, make it a direct access file and then try to read the end records of the file first. Ideally, read it sequentially backwards. This is because AIX will automatically use memory to buffer the file. Assuming the file is larger than memory, after the write is completed, memory is likely to contain a large number of buffers corresponding to the last part of the file. If you then read these records, AIX will supply them to the program from memory without physically reading the disk. If you read the file forwards, the incoming records from the front of the file will flush out the in-memory buffers before you reach them.

5.5 Locating hot spots (profiling)

Profiling tells you how the CPU time used by a program during execution is distributed over the code. It identifies the active subroutines and loops so that tuning effort can be applied most effectively.

It is important to understand that a profile relates just to the particular run of the program for which the profile was obtained. The same program run with different data may produce a different profile. Some numerically intensive programs produce very consistent profiles with widely varying sets of input data. Others produce quite different profiles when the data is changed.

From the point of view of the person tuning the code, the ideal situation is a consistent profile with very pronounced concentrations of time spent in a few routines. Tuning effort can then be concentrated on those routines.

The AIX tools available for profiling the programs include:

- The AIX **prof** and **gprof** commands
- The AIX **tprof** command

The **prof** and **gprof** commands provide profiling at the procedure (subroutine and function) level. The **tprof** command uses the AIX trace facility to interrupt your program at each tick (10 milliseconds) of the AIX CPU clock and construct a trace table that contains the hardware instruction address register. At the end of your program execution, **tprof** creates a report (using the trace table) showing the number of ticks that relate to each line of your source code.

To use **prof** and **gprof**, do the following:

1. Compile your program with the -p or -pg option in addition to the normal compiler options
2. Run the program (this produces the gmon.out file)
3. Run **prof** or **gprof** by entering:

   `prof > filename`

   or

   `gprof > filename`

The standard output, *filename*, of **prof** will contain the following information:

- The percentage of the program's CPU time used by the procedure.
- The time in seconds required for all references to the procedure.
- The cumulative total of seconds required for all procedures in the list.
- The number of times the procedure was called and the time required to perform each call.

The output of **gprof** contains all the information provided by **prof**, and in addition the timing information of the calling tree for the procedures in the program.

To use **tprof** on a program myprog.f, do the following:

1. Compile your program with the -g option
2. Run **tprof** on the program:

```
tprof -p myprog -x "myprog params"
```

This procedure creates two output files, namely __myprog.all and __t.myprog.f. The first file shows all the processes involved in running your program and provides a count of the timer ticks associated with each process. It also lists the percentage of ticks that are associated with the complete program. The second file is only produced if you compile your program with the -g option. It is an annotated version of your source file, that indicates the CPU ticks associated with each line of the source code being executed.

For more details on how to use **prof**, **gprof**, and **tprof**, see *Optimization and Tuning Guide for Fortran, C, and C++*, SC09-1705.

By far the most user-friendly and powerful tool, providing graphically assisted profiling down to the Fortran or assembler statement level, is **xprofiler**. **xprofiler** is a supported IBM tool distributed as part of the IBM Parallel Environment for AIX licensed program product (5765-D93). The specific fileset component that supplies this tool is ppe.xprofiler. If you are running on a workstation where PE is not installed, your profiling option is to use **prof**, **gprof**, or **tprof**.

To use **xprofiler**, compile and link as for **gprof** with -g -pg options together with -O3 or whatever other optimization you are using. It is important to use the same optimization options as you will use for production, since changing the optimization is highly likely to also change the profile.

Then simply run the executable against the chosen test data. This will produce the standard gmon.out file containing the profiling data. Then run **xprofiler**. Graphics will appear showing the subroutine tree of the program, with each subroutine represented by a rectangle. The area of each rectangle is roughly proportional to the CPU time spent in that routine, giving an immediate visual indication of hot-spot locations. Clicking on a rectangle will produce a set of options, one of which creates a source code listing with each statement annotated with the amount of CPU time (in units of 1/100 of a second) used. This enables the active loops to be easily identified.

6

Performance libraries

In this chapter we discuss performance-enhancing techniques that take advantage of highly tuned variants of commonly needed operations.

Scientific and technical computational problems often contain common mathematical constructs, such as matrix-vector multiply or matrix-matrix multiply, which use a large portion of an application's computational time. Many of these common constructs have been extensively researched and tuned for efficient computation. Basic linear algebra subprograms (BLAS) that compute many of these common constructs are available from various sources. For example, you can download the source or precompiled BLAS from:

`http://www.netlib.org/blas`

Subprograms for solving systems of linear equations, eigenvalue problems, singular value problems, and so forth can be found in the public domain LAPACK libraries:

`http://www.netlib.org/lapack`

LAPACK uses BLAS calls whenever possible to simplify its use and to be able to take advantage of any available optimized BLAS libraries.

The public domain BLAS or LAPACK are not highly tuned for a particular architecture and locally compiled versions seldom approach peak GFLOPS rates. An alternative is to download the package from the automatically tuned linear algebra software (ATLAS) site:

http://math-atlas.sourceforge.net

With ATLAS, you must first generate a tuned library for a system by running an extensive testing suite on a quiet system. The adjustable parameters in the tuned library are determined by extensively testing processor speed, cache sizes and speed, and memory size and speed. Some impressive performance results can be obtained in this manner. However, the resulting library may not be well tuned if it is subsequently used on a somewhat different machine configuration.

Another alternative is to use the IBM Engineering and Scientific Subroutine Library (ESSL) and the parallel version, Parallel ESSL. These libraries are highly tuned for IBM hardware, having been tested on many different PowerPC processor configurations. ESSL and Parallel ESSL are discussed in The ESSL and Parallel ESSL libraries section that follows.

A different type of specialized performance tuning is applying faster, but slightly less accurate versions of Fortran intrinsic functions such as SIN, LOG, and EXP. IBM has produced tuned versions of functions like these, which can be found in the MASS library. MASS can be downloaded from:

http://www.rs6000.ibm.com/resource/technology/MASS

MASS is discussed in detail in Section 6.2, “The MASS libraries” on page 117.

6.1 The ESSL and Parallel ESSL libraries

The Engineering and Scientific Subroutine Library (ESSL) family of products is a state-of-the-art collection of mathematical subroutines. Running on IBM pSeries servers and IBM RS/6000 workstations, servers and SP systems, the ESSL family provides a wide range of high-performance mathematical functions for a variety of scientific and engineering applications.

The ESSL family includes:

- ESSL for AIX, which contains over 400 high-performance mathematical subroutines tuned for IBM UNIX hardware.
- Parallel ESSL for AIX, which contains over 100 high-performance mathematical subroutines specifically designed to exploit the full power of RS/6000 SP hardware with scalability of up to 512 nodes.

Complete information on the ESSL and Parallel ESSL libraries, including information on obtaining them, can be found at:

http://www-1.ibm.com/servers/eserver/pseries/software/sp/essl.html

6.1.1 Capabilities of ESSL and Parallel ESSL

ESSL provides a variety of mathematical functions, such as:

- Basic Linear Algebra Subprograms (BLAS)
- Linear Algebraic Equations
- Eigensystem Analysis
- Fourier Transforms

ESSL products are compatible with public domain subroutine libraries such as Basic Linear Algebra Subprograms (BLAS), Scalable Linear Algebra Package (ScaLAPACK), and Parallel Basic Linear Algebra Subprograms (PBLAS). Thus, migrating applications to ESSL or Parallel ESSL is straightforward.

Both ESSL and Parallel ESSL have SMP-parallel capabilities. The term *parallel* in the Parallel ESSL product name refers specifically to the use of MPI message passing, usually across the SP switch. For SMP parallel use within a single pSeries 690 Model 681, Parallel ESSL is not required. An SMP-parallel example (DGEMM) for the pSeries 690 Model 681 is provided in Figure 6-1 on page 116.

6.1.2 Performance examples using ESSL

ESSL V3.3 and Parallel ESSL V2.3 are available for the IBM @server pSeries 690 Model 681 and contain highly optimized routines tuned for the POWER4 processor. Significant optimizations have been done in ESSL to effectively use the L1 and L2 cache, maximize data reuse in the caches, and minimize memory bandwidth requirements. Any application that can be formulated with BLAS calls, especially BLAS3 calls such as SGEMM or DGEMM, will benefit greatly from the ESSL library.

Attention: Some performance numbers reported here used a pre-release version of ESSL that was the latest available at the time this document was written. Readers should perform their own studies to establish firm performance metrics.

Single processor DGEMM

The ESSL version of DGEMM was used to perform matrix-matrix multiplies on a single POWER4 processor on a pSeries 690 Turbo system. The measured GFLOPS as a function of matrix size are shown in Figure 6-1. The best performance is greater than 3.6 GFLOPS and the performance is excellent for a wide range of matrix sizes.

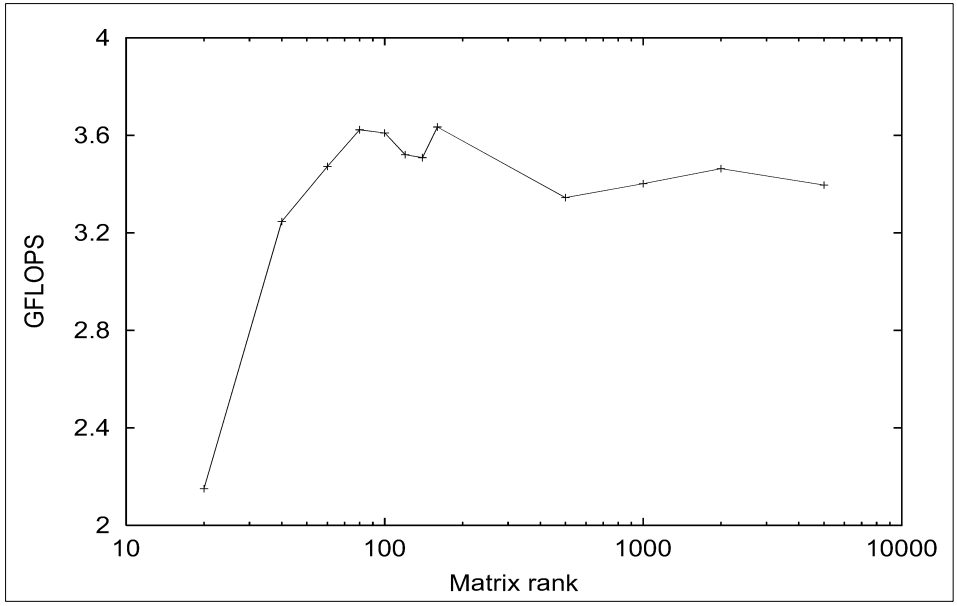


Figure 6-1 ESSL DGEMM single processor GFLOPS

SMP-parallel DGEMM - optimal POWER4 GFLOPS from ESSL

Table 6-1 lists the performance of SMP parallel DGEMM on a pSeries 690 Turbo for square REAL*8 matrices.

Table 6-1 DGEMM throughput summary

Parallelism	2000x2000 REAL*8	10000x10000 REAL*8
32 CPU pSeries 690 Turbo		
8-way	25.80	not measured
16-way	47.86	not measured
24-way	69.00	not measured
32-way	84.30	96.13

A sustained rate of over 96 GFLOPS was measured on a 1.3 GHz pSeries 690 Turbo for 32-way parallel DGEMM on 10000x10000 matrices, as provided in Table 6-1 on page 116. This is the highest performance seen for a single job on a single pSeries 690 Turbo during the preparation of this publication. Also see Section 8.4.1, “ESSL DGEMM throughput performance” on page 161 for the performance observed for multiple copies of a single processor DGEMM application.

6.2 The MASS libraries

The mathematical acceleration subsystem (MASS) library provides high-performance versions of a subset of Fortran intrinsic functions. These versions sacrifice a small amount of accuracy to allow for faster execution. Compared to the standard mathematical library, libm.a, the MASS library differs, at most, only in the last bit. Thus, MASS results are sufficiently accurate in all but the most stringent conditions.

There are two basic types of functions available for each operation:

- A single instance function
- A vector function

The single instance function simply replaces the libm.a call with a MASS library call. The vector function is used to produce a vector of results given a vector operand. The vector MASS functions may require coding changes while the single instance functions do not.

6.2.1 Installing and using the MASS libraries

The MASS libraries can be downloaded from:

```
http://www.rs6000.ibm.com/resource/technology/MASS
```

This site also has extensive documentation and should be referred to for more detailed explanations.

The download file is a compressed tar file that can be unpacked into /usr/lpp and the resulting library files linked to /usr/lib, or the tar file may be unpacked into any other location for inclusion at link time. There are separate libraries for the single instance functions and the vector functions.

The following is an example using MASS. If libmass.a and the other libraries are installed in /home/somebody/mass, it is used as:

```
xlf90 -c -O3 -qarch=pwr4 -qtune=pwr4 myprogram.f
xlf90 -o myjob -L/home/somebody/mass -lmass myprogram.o
```

All references to SINE, LOG, EXP and other functions in myprogram.f will have been satisfied from the single instance functions in libmass.a rather than the normally chosen functions in libm.a.

Some of the functions available in the MASS library have now been included in the XL Fortran runtime environment. This means that at higher specified levels of compiler optimization, a Fortran intrinsic function or operation may be replaced with a faster version found in /usr/lib/libxlopt.a even if the MASS libraries have not been installed. If you wish to track exactly which version of an intrinsic has been used you can produce a detailed, sorted cross reference map using -bsxref:myxref when creating the executable.

The following is an example using the MASS library with Fortran code:

```
real(8) a(*)
...
do i=1,n
   a(i)=1.0d0/a(i)
enddo
```

This code would be rather expensive using the hardware divide function and may be replaced using the vector MASS reciprocal approximation function vrec as:

```
call vrec(a,a,n)
```

Using the vector form, the speedup for this example is approximately 2.25 for n>~50. See Table 6-2 on page 119 for more information.

The executable is linked as:

```
xlf90 -o myjob -bsxref:myxref -L/home/somebody/mass -lmassv myprogram.o
```

Examination of the file myxref shows that vrec has been loaded from libmassv.a.

However, if you are using XL Fortran Version 7.1 or later, compiling and linking as:

```
xlf90 -c -O3 -qhot -qarch=pwr4 -qtune=pwr4 myprogram.f
xlf90 -o myjob -bsxref:myxref myprogram.o
```

you will find that a version of vrec has been loaded from libxlopt.a. Several other functions are recognized and may be substituted by the compiler such as exp, sin, cos, sqrt, and reciprocal square root.

6.2.2 Description and performance of MASS libraries

Table 6-2 lists the functions available in the MASS libraries and an approximate measure of performance. The performance numbers are based on POWER3 measurements. Similar speedups are expected on POWER4. While MASS functions are somewhat less accurate than the standard function, errors are mostly less than 1 bit.

Table 6-2 Mass library functions and performance

Function	mass call	speedup	massv call	speedup[a]
64-bit exponential	exp	2.37	vexp	6.7
32-bit exponential	exp	2.37	vsexp	9.7
64-bit natural log	log	1.57	vlog	10.4
32-bit natural log	log	1.57	vslog	12.3
64-bit sine or cosine	sin,cos	2.25[b]	vsin,vcos	7.2[b]
32-bit sine or cosine	sin,cos	2.17[b]	vssin,vscos	9.75[b]
64-bit sine and cosine	sin,cos	2.42[b]	vsincos[c]	10.0[b]
32-bit sine and cosine	sin,cos	2.08[b]	vssincos	13.2[b]
64-bit tangent	tan	2.13	vtan	5.84
32-bit tangent	tan	2.02	vstan	5.95
64-bit inverse tangent of complex number	atan2	4.75	vatan2	16.5
32-bit inverse tangent of complex number	atan2	4.70	vsatan2	16.7
Truncate to whole number	dint	1.0	vdint	7.86
Convert to nearest whole number	dnint	2.0	vdnint	7.06
64-bit reciprocal	n/a		vrec	2.6
32-bit reciprocal	n/a		vsrec	3.8
64-bit square root	sqrt		vsqrt	1.2
32-bit square root	sqrt		vssqrt	2.3
64-bit reciprocal square root	rsqrt	1.34	vrsqrt	6.2
32-bit reciprocal square root	rsqrt	1.34	vsrsqrt	13.2
Real raised to real power	x**y	2.35	N/A	N/A

[a] Per result for vector length 1000
[b] Speedup for data range [-1,1]
[c] See libmassv.f in installation directory for usage

6.3 Modular I/O (MIO) library

The Modular I/O (MIO) library was developed by the Advanced Computing Technology Center (ACTC) of the Watson Research Center at IBM to address the need for an application-level method for optimizing I/O. Applications frequently have very little logic built into them to provide users the opportunity to optimize the I/O performance of the application. The absence of application level I/O tuning leaves the end user at the mercy of the operating system to provide the tuning mechanisms for I/O performance. Typically, multiple applications are run on a given system that have conflicting needs for high-performance I/O resulting, at best, in a set of tuning parameters that provide moderate performance for the application mix.

The MIO library allows users to analyze the I/O of their application and then tune the I/O at the application level for a more optimal performance for the configuration of the current operating system.

Sequential access, predominantly reads, of very large files (tens of gigabytes) is a common pattern of I/O, for example, in implicit finite element analysis codes. Applications that are characterized by this I/O pattern tend to benefit minimally from operating system buffer pools. Large operating system buffer pools are ineffective since there is very little, if any, data reuse and system buffer pools typically do not provide prefetching of user data. However, the MIO library can be used to address this issue by invoking a prefetching (pf) module that will detect the sequential access pattern and asynchronously preload the needed data into a smaller cache. The pf cache need only be large enough to contain enough pages to maintain sufficient read ahead. The pf module can optionally use direct I/O, which will avoid an extra memory copy to the system buffer pool and also frees the system buffers from the one-time access of the I/O traffic, allowing the system buffers to be used more productively. Our early experiences with the aix module have consistently demonstrated that the use of direct I/O with the pf module is highly beneficial to system throughput.

The MIO library consists of four I/O modules that may be invoked at run time on a per-file basis. The modules currently available are:

mio The interface to the user program

pf A data prefetching module

trace A statistics gathering module

aix The MIO interface to the operating system

For each file that is opened with MIO there are a minimum of two modules invoked: the mio module, which converts the user MIO calls (MIO_open, MIO_read, MIO_write, to name a few) into the internal calling sequence of MIO, and the aix module, which converts the internal calling sequence of MIO into the appropriate system calls (open, read, write, for example). Between the mio and aix module invocations the user may specify the invocation of the other modules, pf and trace.

For applications that use the POSIX standard open, read, write, lseek, and close I/O calls the application programmer should only need to introduce #define's to direct the I/O calls to use the MIO library. MIO is controlled through four environment variables. Among other things, these variables determine which modules are to be invoked for a given file when MIO_open is called.

As an example, the output of a MIO trace invocation is shown for a simple program. It opens a file, truncating it back to zero bytes in length, and then writes 100 records of 16 KB. The file is then read forwards with 100 reads of 16 KB, and then read backwards with 100 reads of 16 KB.

```
MIO statistics file : Wed Feb  9 16:03:17 2000
hostname=v01n01.vendor.pok.ibm.com
program=a.out
MIO library built Feb  1 2000 12:53:59  : with aio calls
MIO_STATS   =example.mio
MIO_DEBUG   =OPEN
MIO_FILES   = *.dat [ trace/stats ]
MIO_DEFAULTS= trace/kbytes


Opening file file.dat
   modules=trace/stats
================================================================================

Trace close : mio <-> aix : file.dat : (4800/1.80)=2659.71 kbytes/s
     demand rate=2611.47 kbytes/s=4800/(1.85-0.02))
     current size=1600   max_size=1600
  mode =0640  sector size=4096
  oflags =0x302=RDWR  CREAT  TRUNC
  open              1     0.03
  write           100     0.03       1600       1600      16384      16384
  read            200     1.65       3200       3200      16384      16384
  seek            101     0.00
  fcntl             1     0.00
  close             1     0.12
  size            100
================================================================================
```

For more information about MIO, refer to the following Web site:

http://www.research.ibm.com/actc/Opt_Lib/mio/mio_doc.htm

The MIO library was shipped first with the AIX Version 4.3 and 5L Bonus Pack in July 2001. More information on this is found at the following Web site:

http://www.ibm.com/servers/aix/products/bonuspack

6.4 Watson Sparse Matrix Package (WSMP)

The Watson Sparse Matrix Package (WSMP) is a high-performance, robust, and easy-to-use software package for solving large sparse systems of linear equations using a direct method on pSeries servers, RS/6000 workstations, and the RS/6000 SP. It can be used as a serial package, in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing environment, where each node can either be a uniprocessor or a shared-memory multiprocessor.

WSMP is comprised of two parts, both of which are bundled in the same library. Part I of WSMP replaces the older software called WSSMP for the solution of symmetric sparse systems of linear equations. Part II of the WSMP library deals with the solution of general sparse systems of linear equations. Currently, WSMP does not support the solution of general/unsymmetrical sparse systems in a message-passing parallel environment. WSMP does not have out-of-core capabilities. The problems must fit in the main memory for reasonable performance.

Technical papers related to the software, some example programs, and information about the latest updates can be obtained from the following Web site:

http://www.cs.umn.edu/~agupta/wsmp.html

IBM Research intends to provide a version of WSMP compiled for POWER4 when the hardware and compiler become available.

For solving symmetric systems, WSMP uses a modified version of the multifrontal algorithm for sparse Cholesky factorization and a highly scalable parallel sparse Cholesky factorization algorithm. The package also uses scalable parallel sparse triangular solvers and an improved and parallelized version of the previously released package WGPP for computing fill-reducing orderings. Sparse symmetric factorization in WSMP has been clocked at up to 3.6 GFLOPS on an RS/6000 workstation with four 375 MHz POWER3 CPUs and 90 GFLOPS on a 128-node SP with two-way SMP 200 MHz POWER3 nodes.

For solving general sparse systems, WSMP uses a modified version of the multifrontal algorithm for matrices with an unsymmetrical pattern of nonzeros. WSMP supports threshold partial pivoting for general matrices with a user-defined threshold. WSMP automatically exploits SMP parallelism on an RS/6000 workstation or SP node with multiple CPUs and this parallelism is transparent to the user. On an RS/6000 with four 375 MHz POWER3 CPUs, WSMP has been clocked at up to 2.4 GFLOPS for factoring general sparse matrices with partial pivoting.

7

Parallel programming techniques and performance

There are several methods available to the application programmer to achieve parallel execution of a program and more rapid job completion compared to running on a single processor. These methods include:

- Directive-based shared memory parallelization (SMP)
- Compiler automatically generated shared memory parallelization
- Message passing based shared or distributed memory parallelization
- POSIX threads (pthreads) parallelization
- Low-level UNIX parallelization using fork() and exec()

Each of these techniques has been used to produce efficient parallel codes. The best technique to use is highly dependent on the application, the programmer's skills and preferences, portability requirements for the application, and the target machine's characteristics.

In this chapter we discuss shared memory parallelization, both directive-based and automatic, message passing based parallelization using the MPI standard, and pthread parallelization.

7.1 Shared memory parallelization

Shared memory parallelization describes parallelization that can take place in a computer in which all memory used by a program is locally addressable from within the program. For current IBM computers and the current AIX 5L operating system, this means running on a single node. In the future, non-uniform memory access (NUMA) may be available in which memory on separate, remote nodes may be addressable locally from within a single program.

A detailed description of shared memory parallelization, or SMP programming, can be found in *Scientific Applications in RS/6000 SP Environments*, SG24-5611. A brief overview is given in this section. All discussions refer to the OpenMP standard implementation of SMP parallelism.

7.1.1 SMP runtime behavior

Shared memory parallelization is implemented by creating user threads that are scheduled to run on kernel threads by the operating system. This parallel job flow is illustrated in Figure 7-1 on page 127.

A single thread is created when a program starts. Additional threads are created when the first parallel region is entered. After all parallel work for a thread is completed, it spin waits for the next parallel section for a period, but it consumes processor time while waiting. After the spin wait time has expired and if a yield wait time has been specified, the thread can yield its place on the kernel thread to another runable thread. If the yield wait time has expired and no new parallel region has been entered, the thread goes to sleep. Reactivating a thread from a sleep state is more costly than if the thread is in a yielded state.

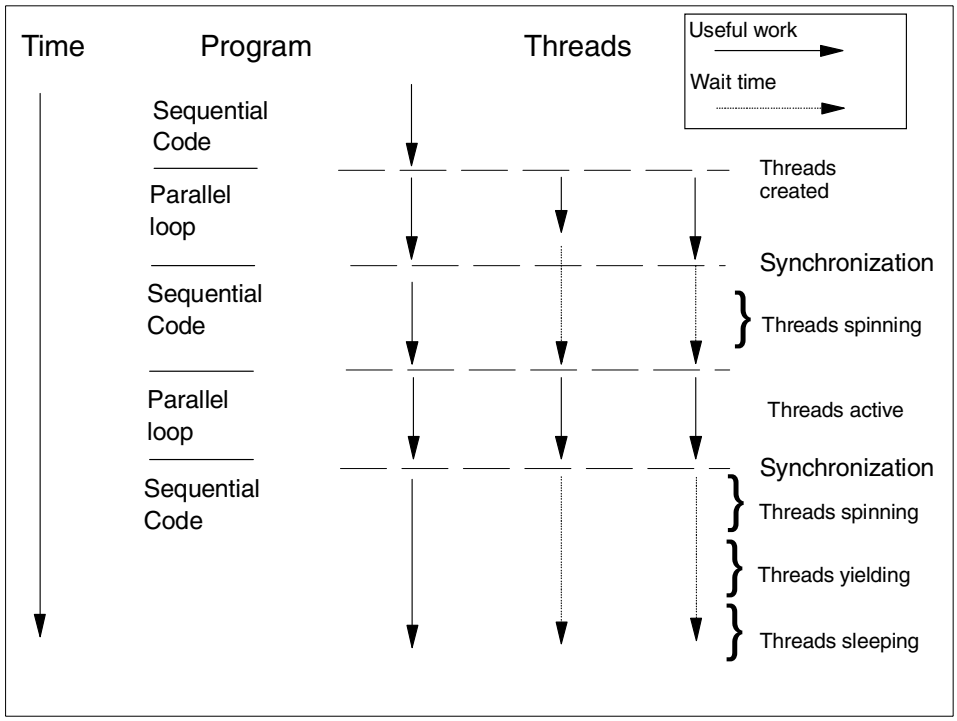


Figure 7-1 Shared memory parallel job flow

There are some important environment variables that can affect parallel performance at run time. Different settings would be appropriate on a busy machine compared to a quiet machine. Some of the more important environment variables are:

- AIXTHREAD_SCOPE = S or P (default = P)

  The thread contention scope can be system (S) or process (P). When system contention scope is used, each user thread is directly mapped to one kernel thread. This is appropriate for typical scientific and technical applications in which there is a one-to-one ratio between threads wanted and processors wanted. Process contention scope is best when there are many more threads than processors. When process contention scope is used, user threads share a kernel thread with other (process contention scope) user threads in the process.

- OMP_DYNAMIC = FALSE or TRUE (default = TRUE)

  The OMP_DYNAMIC environment variable disables or enables dynamic adjustment of the number of threads available for the execution of parallel regions. If this variable is TRUE, the runtime environment can adjust the number of threads it uses for executing parallel regions so it makes the most efficient use of system resources. The dynamic checking can add a small amount of overhead, so for benchmarking, scaling tests, or if an application depends on a specific number of threads, this variable should be set to FALSE.

- SPINLOOPTIME=n (default = 40)

  If a user thread cannot acquire a lock (which is necessary to begin a parallel loop, for example), it will attempt to spin for up to SPINLOOPTIME times. Once the spin count has been exhausted, the thread will go to sleep waiting for a lock to become available unless the YIELDLOOPTIME is set to a number greater than zero. You want to spin rather than sleep if you are waiting for a previous parallel loop to complete, provided there is not too much sequential work between the loops. If YIELDLOOPTIME is set, upon exhausting the spin count, the thread issues the yield() system call, gives up the processor, but stays in a runable state rather than going to sleep. On a quiet system, yielding is preferable to sleeping since reactivating the thread after sleeping costs more time. For benchmarking or scaling tests, SPINLOOPTIME can be very large, for example 100000 or more. On a busy system, it should not be too large or much processor time that could otherwise be shared with other jobs is consumed spinning. The best value to use depends on various system characteristics such as processor frequency, and several values should be tested to achieve optimal tuning.

- YIELDLOOPTIME = n (default = 0)

  YIELDLOOPTIME controls the number of times that the system yields the processor when trying to acquire a busy spin lock before going to sleep. The processor is yielded to another kernel thread, assuming there is another runable one with sufficient priority. YIELDLOOPTIME is only used if SPINLOOPTIME is also set.

- MALLOCMULTIHEAP (default = not set)

  Multiple heaps are useful so that a threaded application can have more than one thread issuing memory allocation subroutine calls. With a single heap, all threads trying to do a malloc(), free(), or realloc() call would be serialized (that is, only one thread can do malloc/free/realloc at a time) which could have a serious impact on multi-processor machines. With multiple heaps, each thread gets its own heap, up to 32 separate heaps.

- SMP stack size (default = 4 MB/thread)

  For 32-bit OpenMP applications, the default limit on stack size per thread is rather small and if it is exceeded it will result in a runtime error. Should this occur, the stack size may be increased using the XLSMPOPTS environment variable with:

  ```
  export XLSMPOPTS=stack=n
  ```

  where n is the stack size in bytes. However, the total stack size for all threads cannot exceed 256 MB (one memory segment). This limitation of one segment does not apply to 64-bit applications.

7.1.2 Shared memory parallel examples

Shared memory parallelization (SMP) programming can be done at a very high level such as:

```
      SUBROUTINE EXAMPLE(M,N,A,B)
      REAL(8) A(N),B(N)
!$OMP PARALLEL DO PRIVATE(J), DEFAULT(SHARED)
      DO J=1,M
      CALL DOWORK(J,N,A,B)
      ENDDO
      ...
```

The subroutine `DOWORK` and all subsequent subroutine calls must be carefully checked to ensure they are, in fact, thread safe. This high level of parallelization is usually the most efficient, and is recommended when possible.

It is also common to use shared memory parallelization at a low level, although scaling efficiencies are often quite limited when little work is done in a parallel region. The ease of implementation is an attractive feature of low-level parallelization. The discussion and examples that follow demonstrate parallelism at the loop level.

We have tested three loops from the solver of a computational fluid dynamics code and use them as examples. The loops are:

```
LOOP A

      DO  J=1,NX
        Q(J)=E(J)+F2*Q(J)
      ENDDO

LOOP B

      DO J=1,NX
        I1=IL(1,J)
```

```
          I2=IL(2,J)
          I3=IL(3,J)
          I4=IL(4,J)
          I5=IL(5,J)
          I6=IL(6,J)
          E(J)=Y3(J)*Q(J)-(
        *              Q2(1,J)*Q(I1)+Q2(2,J)*Q(I2)+
        *              Q2(3,J)*Q(I3)+Q2(4,J)*Q(I4)+
        *              Q2(5,J)*Q(I5)+Q2(6,J)*Q(I6))
          F3=F3+Q(J)*E(J)
        ENDDO

LOOP C

        DO J=1,NX
          Z0(J)=Z0(J)+X2*Q(J)
          B1(J)=B1(J)-X2*E(J)
          T1=B1(J)
          E(J)=T1*DBLE(C1(J))
          F1=F1+T1*E(J)
          F4=F4+ABS(T1)
        ENDDO
```

The declarations are:

```
REAL(8) Z0(NX),B1(NX),E(NX),Q(0:NX)
REAL(4) Y3(NX),Q2(6,NX),C1(NX)
INTEGER(4) IL(6,NX)
```

In this example, NX is typically 100000 to 10000000.

Loop A is a simple multiply/add loop. Loop B is a complicated loop with 20 memory loads, a single store, and a reduction sum. Six of the memory references, such as `Q(I1)` are indirect address references. Loop C is a moderately complicated loop with five memory loads, three stores, and two reduction sums.

7.1.3 Automatic shared memory parallelization

Automatic shared memory parallelization is successful when the compiler can recognize parallel code constructs and safely produce efficient parallel code. The IBM XL Fortran Version 7.1 compiler has state-of-the-art capabilities for automatically parallelizing Fortran programs. A major concern with automatic parallelization is the potential that a loop with little work or few iterations is parallelized and runs more slowly than it would had it remained sequential. However, when a large Fortran code is well written and it is compiled for automatic parallelization, good speedups can be realized with very little effort.

The three example loops are automatically parallelized with:

```
xlf90_r -c -qsmp=auto -qnohot -qreport=smplist -O3 -qarch=pwr4 -qtune=pwr4
-qfixed sub.f
```

The option -qsmp=auto initiates automatic parallelization and it also implies -qhot. The option -qnohot was used to be consistent with the directive-based SMP runs. The option -qreport=smplist reports the line number of each successfully parallelized loop. The resulting file, sub.lst, has additional information including reasons why parallelization may have been unsuccessful for a loop. The xlf90_r compiler invocation should be used rather than xlf90 to ensure the resulting object code is thread safe. Performance results are shown in Section 7.1.5, “Measured SMP performance” on page 132.

7.1.4 Directive-based shared memory parallelization

Directive-based shared memory parallelization is more labor intensive than automatic parallelization, but it does allow for more control over which loops get parallelized and more options for scheduling individual loops.

For the example loops, the following directives were used:

```
LOOP A

!$OMP PARALLEL DO PRIVATE(J),DEFAULT(SHARED),SCHEDULE(GUIDED)

LOOP B

!$OMP PARALLEL DO PRIVATE(J,I1,I2,I3,I4,I5,I6)
!$OMP* REDUCTION(+:F3)
!$OMP*  DEFAULT(SHARED),SCHEDULE(GUIDED)

LOOP C

!$OMP PARALLEL DO PRIVATE(J,T1)
!$OMP* REDUCTION(+:F1,F4)
!$OMP*  DEFAULT(SHARED),SCHEDULE(GUIDED)
```

The loops use guided scheduling, which initially divides the iteration space into one chunk equal to NX divided by N and then exponentially decreases the chunk size to a minimum size of 1. This scheduling algorithm, which allows for a processor that found more data in L1 or L2 cache to get another chunk of data quickly while a processor requiring many L3 or memory references is working, is often most efficient.

Compiling for directive-based parallelization uses the following command options:

```
xlf90_r -c -qsmp=omp -O3 -qarch=pwr4 -qtune=pwr4 -qfixed sub.f
```

The option `noauto` is implied when `-qsmp=omp` is used. However, with `-qsmp=omp`, `-qhot` is not implied.

7.1.5 Measured SMP performance

The three example loops were run from within an application and timed using realistic data for 200 repetitions with NX set to 1000000. The environment settings used were:

```
export AIXTHREAD_SCOPE=S
export SPINLOOPTIME=100000
export YIELDLOOPTIME=40000
export OMP_DYNAMIC=false
export MALLOCMULTIHEAP=1
```

All results were run on a two-MCM, eight-processor pSeries 690 HPC. The results for each of the three loops are shown separately in Table 7-1, Table 7-2, and Table 7-3 on page 133.

Table 7-1 Loop A parallel performance elapsed time

Processors	-O3	-qsmp=auto	speedup	-qsmp=omp	speedup
1	1.211	1.378	0.88	1.238	0.98
2		0.815	1.49	0.706	1.72
4		0.481	2.52	0.431	2.81
6		0.364	3.33	0.335	3.61
8		0.307	3.94	0.262	4.62

Table 7-2 Loop B parallel performance elapsed time

Processors	-O3	-qsmp=auto	speedup	-qsmp=omp	speedup
1	4.190	4.758	0.88	4.268	0.98
2		2.000	2.10	2.068	2.03
4		1.143	3.67	1.146	3.66
6		0.885	4.73	0.883	4.75
8		0.787	5.32	0.720	5.82

Table 7-3 Loop C parallel performance elapsed time

Processors	-O3	-qsmp=auto	speedup	-qsmp=omp	speedup
1	2.795	3.056	0.91	2.830	0.99
2		1.430	1.95	1.541	1.81
4		0.906	3.08	0.933	3.00
6		0.719	3.89	0.736	3.80
8		0.633	4.42	0.588	4.75

The data shows that for these test loops there is little difference between automatic and manual parallelization. Some overhead due to parallelization can be seen comparing the single processor results. Note that the compiler may be using different optimization strategies when creating parallel code as well.

The reduction sums in loops B and C require the creation of critical sections in which only one processor can update the reduced variable at a time. These critical sections can significantly reduce parallel efficiency if the amount of work in the loop is too small or too many processors are used.

The conclusions from this analysis are:

- SMP parallelization does result in improved run times.
- SMP parallelization is easy to implement.
- Overall speedups are limited for small loops, especially when there are reduction sums.

7.2 MPI in an SMP environment

This section examines how existing MPI programs, written for distributed memory systems, can make the best use of both SMP and distributed memory systems.

We do not attempt to provide a detailed discussion of distributed memory parallelization or the use of MPI and refer the reader to the *IBM Parallel Environment for AIX* product documentation, and the *IBM Redbook Scientific Applications in RS/6000 SP Environments*, SG24-5611.

In the following discussion, processes executing in parallel and communicating using MPI calls are referred to as tasks. A number of different scenarios are considered:

- MPI only

  The MPI implementation in IBM Parallel Environment for AIX (PE) can use several protocols for communication between tasks.

  Internet Protocol (IP) can be used between tasks on the same node and between tasks on different nodes. This incurs relatively high latencies and IP overheads.

  In the IBM RS/6000 SP environment with nodes attached to one of the types of SP switch, then another protocol known as *user space* can be used for communication between tasks. Depending on the type of switch involved and the version and release of PE, there may be restrictions on the number of user space tasks allowed per node. At the time of writing, the SP Switch2 with PE 3.1 can support up to 16 user space tasks per POWER3 node. User space significantly reduces the overhead and latency when compared to IP, but it may still be higher between processes on the same node than using shared memory.

  MPI communication calls can also use shared memory for message passing between MPI tasks on the same node. The PE MPI library is capable of using shared memory automatically. In a cluster or SP configuration of POWER3 nodes, then IP or user space would be used between tasks on other nodes.

  In this case, overall performance can still be limited by communication between the nodes. This could be reduced for group operations (such as broadcast) by having one processor per node handle all the internode communication. This process would use shared memory to collect and distribute data to other processes on the same node.

  Since the different tasks on the same node are different processes, they have different address spaces and the shared memory MPI library will communicate though a shared memory segment. This mean a double copy of the data (into and out of the shared memory segment). It would be possible for each task to keep its data in the shared memory segment and not use MPI for this communication but this would require some degree of reprogramming. The advantage of using the PE shared memory MPI library is that no reprogramming is required.

In order to use shared memory for communication calls within a shared memory machine one of the following two procedures should be followed:

- Use the following PE environment variable settings:

```
export MP_SHARED_MEMORY=yes
export MP_WAIT_MODE=poll
```

  MP_WAIT_MODE is not essential in order to use shared memory, but setting it to poll is recommended for performance in most scenarios where MPI tasks will use shared memory.

- Use the following command line arguments either with the parallel program or with the **poe** command depending on the way the parallel program is started:

```
-shared_memory yes -wait_mode poll
```

- MPI and SMP Fortran

  In this scenario, also known as the hybrid or mixed-mode programming model, there are fewer MPI tasks than processors per node. Shared memory parallelization techniques such as OpenMP directives can be used to execute sections of the code between MPI calls in parallel. This means that each MPI task has multiple threads executing in parallel, and the aim would be to keep all of the processors busy all of the time. In practice, it will be difficult to achieve this during the MPI communication phases of the program. However, the benefit of this programming model is that it can be used to reduce the amount of communication traffic between nodes, especially during global communications, by reducing the total number of MPI tasks. This could be especially important for large multi-processor systems such as 32-way POWER4 systems clustered together.

  The overhead of shared memory parallelization is similar to that of MPI data transfers, so it is desirable to parallelize at a sufficiently coarse granularity to keep the effect of this overhead small. Some recoding may be required to achieve this hybrid parallelization.

- MPI and explicit large chunk threads

  In this scenario, there is only one MPI process per node. The initial process (or master thread) creates threads which, instead of issuing MPI calls, use pthread techniques to transfer data between themselves and the master thread. The master thread uses MPI to transfer all data between the nodes.

  Data does not have to be copied between threads since they all use the same address space. Synchronization can be achieved either with standard pthread calls, or, with even less overhead, by using spin loops and the atomic fetch_and_add function (which guarantees that only one thread at a time can update a variable).

The total number of messages between nodes is reduced and hence delays due to latency are reduced. Since the master thread handles all messages, it should perhaps be coded to do less work than the other threads.

However, all of this may imply considerable reprogramming. The program may have used the MPI task ID to create its arrays and organize its data. The threads will have to arrange this differently, because they share the same task ID, and are using the same address space.

The advantages and disadvantages of these scenarios are summarized in Table 7-4.

Table 7-4 Advantages and disadvantages of message passing techniques

Programming model	Advantages	Disadvantages
MPI only	No program changes. Same coding for calls between all tasks, uses shared memory on same node.	Double copy between processes on same node.
Hybrid mode	MPI exchanges reduced. Can reduce off node communication.	May not be possible to fully use the CPUs. Some reprogramming required.
MPI and large chunk threads	MPI exchanges reduced. Exchanges and overhead between threads reduced.	Considerable reprogramming may be required.

To summarize, all of the scenarios can be useful depending on the particular application requirements and the target environment. Descending the table, the efficiency of the solution increases, but the amount of reprogramming required also increases.

To gain addressability to 8 GB with a 32-bit MPI, the sPPM ASCI benchmark code used the Hybrid mode. More information about this can be obtained from:

`http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/sppm_readme.html`

7.3 Programming with threads

The thread programming paradigm is a flexible, low-level model of distributing the work of a given application into multiple streams of execution that share a single memory address space. Each thread can execute its own function and can be controlled independently. In the context of high-performance computing, threads are used to distribute a workload onto multiple processors of an SMP system, rather than to dispatch many threads onto a single processor, as is common for graphical user interfaces.

There is a standardized application interface for threads called Pthreads (POSIX threads) that is part of the UNIX specification. The redbook *Scientific Applications in RS/6000 SP Environments*, SG24-5611, provides a compact introduction to Pthreads for multi-processor applications on the AIX platform. The corresponding AIX reference manual *General Programming Concepts: Writing and Debugging Programs* (part of the AIX Programming Guides) can be found at:

`http://www.ibm.com/servers/aix/library/techpubs.html`

Programming explicitly with threads is not recommended for the casual user. In many cases the benefits of multiple threads can be more easily obtained by using the automatic parallelization capabilities of the compiler or OpenMP directives.

7.3.1 Basic concepts

Threads can be described as light-weight processes. Each thread has its own private program counter, stack, and registers. The memory state and file descriptors are shared. For a brief overview of the usage of Pthreads a simple *hello world* program is shown in Example 7-1 on page 138. Although this program does no complicated work, it provides a useful template for thread creation.

A Pthread program begins to execute as a single thread. Additional threads are created and terminated as necessary to concurrently schedule work onto the available processors. In this example, the initial thread creates three worker threads, which will print *hello* messages and terminate. As will be familiar to message passing programmers, it is a good practice for the master thread (or MPI task) to take part in the computation. This yields good load-balancing when N threads are dispatched on N processors.

A threaded application should be compiled and linked with the _r-suffixed invocation of the C compiler, for example xlc_r, which defines the symbol _THREAD_SAFE and links with the Pthreads library.

Example 7-1 Pthread version of a hello world program

```
#include <pthread.h>
#include <stdio.h>

void * thfunc(void * arg)
{
   int id;
   id = *((int *) arg);
   printf("hello from thread %d \n", id);
   return NULL;
}

int main(void)
{
   pthread_t thread[4];
   pthread_attr_t attr;
   int arg[4] = {0,1,2,3};
   int i;

   /* setup joinable threads with system scope */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create N-1 worker threads */
   for (i=1; i<4; i++) {
      pthread_create(&thread[i], &attr, thfunc, (void *) &arg[i]);
   }

   /* let the master thread also take part in the computation */
   thfunc((void *) &arg[0]);

   /* wait for all other threads to finish */
   for (i=1; i<4; i++) {
      pthread_join(thread[i], NULL);
   }
   return 0 ;
}
```

Threads are created using the pthread_create function. This function has four arguments: A thread identifier, which is returned upon successful completion, a pointer to a thread-attributes object, the function that the thread will execute, and the argument of the thread function. The thread function takes a single pointer argument (of type void *) and returns a pointer (of type void *). In practice, the

argument to the thread function is often a pointer to a structure, and the structure may contain many data items that are accessible to the thread function. In this example, the argument is a pointer to an integer, and the integer is used to identify the thread.

The previous simple example creates a fixed number of threads. In many applications, it is useful to have the program decide how many threads to create at run time while providing the ability to override the default behavior by setting an environment variable. For example, for OpenMP programs the default is to create as many threads as processors are available. In AIX, you can get the number of online processors by calling the sysconf routine from libc, as shown in Example 7-2.

Example 7-2 Sample code for setting the number of threads at run time

```
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
...
char * penv;
int ncpus, numthreads;
...
/* get the number of online processors */
ncpus = sysconf(_SC_NPROCESSORS_ONLN);
if (ncpus < 1) ncpus = 1;

/* check the NUMTHREADS environment variable */
penv = getenv("NUMTHREADS");
if (penv == NULL) numthreads = ncpus;
else numthreads = atoi(penv);
...
```

A thread terminates implicitly when the execution of the thread function is completed. A thread can terminate itself explicitly by calling pthread_exit. It is also possible for one thread to terminate other threads by calling the pthread_cancel function. The initial thread has a special property. If the initial thread reaches the end of its execution stream and returns, the exit routine is invoked, and, at that time, all threads that belong to the process will be terminated. However, the initial thread can create detached threads, and then safely call pthread_exit. In this case, the remaining threads will continue execution of their thread functions and the process will remain active until the last thread exits. In many applications, it is useful for the initial thread to create a group of threads and then wait for them to terminate before continuing or exiting. This is can be achieved with threads that are joinable (see

pthread_attr_setdetachstate). The AIX default is detached. The function pthread_join suspends the calling thread until the referenced thread has terminated. The system scope attribute is appropriate when N threads are supposed to run on N processors concurrently.

Synchronization

As with OpenMP directive-based parallelization the distinction between threadprivate and shared variables is essential for the correctness and performance of a program. The access to shared variables has to be synchronized to avoid conflicts and to assure correct results. The use of synchronization should be balanced with its degradation of performance and scalability.

A major difficulty of parallel programming for shared memory is to find the right balance of local and global variables, since the scoping defines which variables are private or shared. Contention for global variables, as in a reduction sum, is a major source of performance problems. The introduction of temporary local variables often helps to resolve such problems.

In multi-threaded applications the update of shared memory locations is usually protected with mutex (mutual exclusion) locks. The operating system ensures that access to the shared data is serialized. At a given time only one thread can enter the region between lock and unlock to modify the data. The usage of mutex locks is shown in Example 7-3. This example demonstrates how to construct a basic barrier synchronization function. It is left as an exercise for the reader to study a Pthread programming reference in order to understand this complex construct.

Example 7-3 Usage of mutex locks to modify shared data structures

```
#include <pthread.h>
int barrier_instance = 0;
int blocked_threads = 0;

pthread_mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t sync_cond = PTHREAD_COND_INITIALIZER;

int syncthreads(int nth)
{
   int instance;

   /* the calling thread implements a lock, other threads block */
   pthread_mutex_lock(&sync_lock);

   /* the thread with the lock proceeds */
   instance = barrier_instance;
   blocked_threads++;
```

```
   if (blocked_threads == nth)
   {
      /* notify all threads that the sync condition is met */
      blocked_threads = 0;
      barrier_instance++;
      pthread_cond_broadcast(&sync_cond);
   }

   while (instance == barrier_instance)
   {
      /* release the lock and wait here */
      pthread_cond_wait(&sync_cond, &sync_lock);
   }

   /* all threads call the unlock function and return */
   pthread_mutex_unlock(&sync_lock);

   return(0);
}
```

Frequent mistakes

The most common mistakes of thread programming shown in this section occur more frequently than we would like:

- Process exits before all threads have finished

  The following example of code is not correct because when a process exits or returns from main(), all of the process memory is deallocated and all threads belonging to the process are terminated.

  ```
  #include <pthread.h>
  int main(void) {
     pthread_t tid[NUMBER_OF_THREADS];
     ...
     /* create threads */
     for (i =0; i<NUMBER_OF_THREADS; i++)
        /* each thread calls thread_main with parameter arg. */
        pthread_create (&tid[i], NULL,(void*(*)(void*)) thread_main, (void *)
                                                                      arg);
     exit(0);
  }
  ```

- Parent thread exits before child

  A problem similar to the previous one; do not forget to call the pthread_join routine before exiting.

- Dangling pointer

  The following code fragment is incorrect because `errorcode` resides on the local stack of the thread and will be freed when the thread is destroyed, leading to a dangling pointer.

```
void * thfunc(void * arg)
{
   int errorcode;

   /* do something */
   /* if error condition detected, errorcode = something; */
   pthread_exit(&errorcode);
   ...
}
```

Using Pthreads in Fortran

On IBM systems, a Fortran version of the Pthreads interface is available in addition to the standard C Pthreads interface. This makes it relatively simple to introduce threads into numerically intensive Fortran applications. The reader should recognize that the Fortran interface is not backed by an industry-wide standard. For example, Pthread constructs can be used within OpenMP programs in rare instances when some direct control of thread management or data access synchronization is necessary. In such a mixed mode the OpenMP runtime environment will create and manage all threads used for the execution of OpenMP parallel constructs. Explicit pthread creation is the responsibility of the programmer.

The IBM Fortran version of the Pthreads API is similar to the C version, where the function names and data types from C are preceded with f_. Fortran programs that use explicit Pthread routines must have a statement to include the f_pthread module. In general the -qnosave option is essential for correct behavior of a program. A number of Fortran routines, including f_pthread_create, have call sequences that differ from the standard C version. For example, the f_pthread_create function takes an additional parameter to specify properties of the argument to the thread function. The IBM Fortran implementation of Pthreads is described in the *XL Fortran for AIX Language Reference*, SC09-2867.

7.3.2 Coding and performance considerations

The following performance considerations apply to both Pthread hand-coded programs and OpenMP based programs.

Thread creation

The time required for creation of a thread is of the order of magnitude of 100 microseconds. You should only create/awake/terminate threads that execute for a significantly longer time.

Lock contention

Shared data structures that are modified within an innermost loop of a thread and need to be protected against concurrent access can cause severe performance degradation. The following example shows a loop that counts the number of elements in a vector with a value equal to one. The counter is a shared variable that has to be protected by a mutex lock when incremented. Assuming that every second element is equal to one, this example takes more than 30 times longer to execute on four processors than an equivalent sequential loop, which does not need to call the Pthread lock routines.

```
int shared_count=0;
...
void * thfunc(void *id)
{
   ...
   for (i=start; i<start+incr; i++) {
      if (vector[i] == 1) {
         pthread_mutex_lock(&lock);
         shared_count++;
         pthread_mutex_unlock(&lock);
      }
   }
   ...
}
```

Without increasing the default values of the SMP runtime variables SPINLOOPTIME or YIELDLOOPTIME the performance is even slower. For details on AIX environment variables that determine the runtime behavior of a thread when waiting for a lock (spin, yield, sleep), see Section 7.1.1, “SMP runtime behavior” on page 126.

In this simple case, the performance problem can be resolved with the help of a local counter variable, which turns the tremendous speed down into an expected parallel speedup.

```
int shared_count=0;
...
void * thfunc(void *id)
{
   int private_count=0;
   ...
   for (i=start; i<start+incr; i++) {
      if (vector[i] == 1) {
         private_count++;
      }
   }
   pthread_mutex_lock(&lock);
   shared_count += private_count;
   pthread_mutex_unlock(&lock);
   ...
}
```

Avoiding locks and OpenMP critical sections

In many multi-threaded programs, a barrier synchronization routine can help to reduce extensive use of locks or OpenMP critical sections. For example, suppose that multiple threads are working to fill out different entries of a table, and, once that is done, each thread needs read access to the table for the next step. A barrier synchronization point would ensure that no thread could proceed to the next step until all threads have finished filling out the table. Instead of working directly with the low-level pthread_mutex functions, a higher level thread synchronization function is very useful.

If you cannot avoid a lock or critical section:

- Reduce the amount of time a lock is held. Move all unnecessary code outside a critical section.
- Combine access to shared data in order to reduce the number of single lock/unlock calls.

False sharing

False sharing of a cache line occurs when multiple threads on different processors with private caches modify independent data structures that happen to belong to the same cache line. In this situation, the cache line of a particular CPU is flushed out due to another processor store operations and has to be transferred repeatedly from remote caches. This is likely to happen when, for example, a thread local variable is stored in a global array indexed by the logical thread number. This causes the data to be located close together in memory.

By using appropriate padding, false cache-line sharing can be avoided. Referring to the proceeding example, the following implementation cures the lock contention problem, but suffers from false sharing. On our machine, this leads to a moderate speed down (of about 1.5) on four processors compared to the sequential program.

```
int shared_count=0;
int private_count[4]={0,0,0,0};
...
void * thfunc(void *id)
{
   ...
   myid = *((int *)id);
   for (i=start; i<start+incr; i++) {
      if (vector[i] == 1) {
            private_count[myid]++;
      }
   }
   pthread_mutex_lock(&lock);
   gobal_count += private_count[myid];
   pthread_mutex_unlock(&lock);
}
```

By introducing appropriate padding space, to fill up a cache line of 128 byte, false sharing can be eliminated.

```
struct count{
   int  private_counter;
   char pad[124];
}counter[4];
int shared_count=0;
...
void * count_ones(void *id)
{
   ...
   for (i=start; i<start+incr; i++) {
      if (vector[i] == 1) {
         counter[myid].private_counter++;
      }
   }
   pthread_mutex_lock(&lock);
   shared_count += counter[myid].private_counter;
   pthread_mutex_unlock(&lock);
}
```

For example, a similar false sharing problem can occur when storing a set of mutex lock objects in a global array. The following Fortran code avoids false sharing. The type pthread_mutex_t is declared in /usr/include/sys/types.h. In 64-bit mode it has a different size.

```
use f_pthread
integer, parameter :: maxthreads=8
type plock
   sequence
   type(f_pthread_mutex_t)  :: lock
   integer                  :: pad(19)
end type plock
type(plock)        :: locks(maxthreads)
common /global/ locks
```

As a rule of thumb, in an SMP program, global data whose access is not serialized should not be close together.

Reducing OpenMP overhead

In general, an SMP parallel program does generate some computational overhead. Even when executed by just a single thread a certain amount of overhead compared to the execution of the equivalent sequential (non threaded) version of a program can be observed. For discussion purposes, call this the sequential overhead. Thread-safe system libraries, such as for I/O, that are referenced by an _r-suffixed compiler invocation may also contribute to this overhead.

For a fine-grain OpenMP program, the sequential overhead can be significant. If the overhead exceeds, for example, 30 percent of the elapsed time, this can be an indication of inefficient use of OpenMP directives. If global variables need to be scoped threadprivate this often causes problems.

The following example is taken from a real application. To support the threadprivate pragma (or directive in Fortran) the compiler generates calls to the internal function _xlGetThreadValue. These calls are relatively expensive. In general it is a good idea to reduce the number of calls by packing several threadprivate variables into a single structure. This way we will encounter only one call per dynamic path through each function where the threadprivate variables are referenced. Otherwise, we will encounter one call per threadprivate variable. As an example, consider the following lines of code:

```
static int n_nodes, num_visits;
static Node *node_array;
static int *val, *stack;
static Align_info *align_array;

#pragma omp threadprivate( \
   n_nodes, num_visits, \
   node_array, \
   val, stack, \
   align_array \
)
```

This code could be substituted as follows to improve performance. More code changes in the subsequent code are necessary to make this complete.

```
struct nn{
   int n_nodes, num_visits;
   Node *node_array;
   int *val, *stack;
   Align_info *align_array;
} glob;
#pragma omp threadprivate(glob)
```

7.3.3 The best approach for shared memory parallelization

As discussed, there are many different ways to parallelize a program for shared memory architectures. The appropriate approach depends on several considerations, for example: Does a sequential code already exist or will the parallel program be written from scratch? What are the parallel programming skills of the project team? and so on. The following are some pros and cons of the different paradigms:

- Auto-parallelization by the compiler:
  - Easy to implement (just a few directives)
  - Enables teamwork easily
  - Limited scalability because data scoping is neglected
  - Compiler dependent (even on the release of a particular compiler)
  - Not necessarily portable
- OpenMP directives:
  - Portable
  - Potentially better scalability of the auto-parallelization
  - Uniform memory access is assumed
- RYO (subset or mixture of OpenMP and Pthreads, or UNIX fork() and exec() parallelization, or platform-specific constructs)
  - Might enable teamwork
  - Needs a well-tested concept to assure performance and portability
  - Not necessarily portable
- Pthread:
  - Portable
  - Potentially best scalability
  - Needs experienced programmers

- SMP-enabled libraries
  - Least effort
  - Limited flexibility

7.4 Parallel programming with shared caches

On all models of POWER4 microarchitecture machines currently available, each processor has a dedicated L1 cache. As discussed in Section 2.4.3, “L2 cache” on page 17, the L2 cache organization is that each L2 cache unit is shared between two processors in the pSeries 690 Turbo and pSeries 690 Model 681, but in the pSeries 690 HPC each processor has a dedicated L2 cache. In all models L2 cache remains the level of coherency.

The sharing of the L2 cache raises a number of considerations, such as:

- Processors that share L2 cache will compete for the bandwidth from the L2 cache to L3 and memory. However, the effect of this is unlikely to differ very much from the effect seen by independent processes sharing the bandwidth from L2.
- In the configurations where two processors share L2 cache, and each are accessing different memory addresses, then for each cache line loaded into L2, there may be conflict.
- Interference when accessing the same cache lines

  In the extreme case, two threads of a shared memory application may access the same cache line. In this case, there may be a benefit to the shared L2 cache configuration, since there will be a higher percentage of hits in the L2 cache and there will be fewer cache snooping events.

  An example of this would be the following, rather artificial loop:

```
!$OMP PARALLEL DO PRIVATE(s,j,time1,time2), SHARED(a,b,s1,ttime), &
!$OMP&             SCHEDULE(STATIC,1)
          do i=1,m
            do j=1,n
              b(i,j)=a(i,j)+a(i,j)*c1
            end do
          end do
!$OMP END PARALLEL DO
```

  In this example, when parallelized across two threads, each thread will access alternate elements of the array.

Measurements were performed on a 32-way pSeries 690 Turbo, with the arrays dimensioned to fit in L2 but not in L1, and the results provided in Table 7-5 were obtained.

Table 7-5 Shared memory cache results, pSeries 690 Turbo

Threads	Unshared cache time [s]	Shared cache time [s]	Unshared / shared
1	14.76	14.40	0.98
2	14.11	8.63	0.61
4	7.57	8.02	1.06
8	6.78	5.69	0.84
16	6.60	5.75	0.87

The unshared cache times were obtained by binding the threads of the program to alternate processors. Thread one was bound to processor one, thread two was bound to processor three, and so on. The shared cache times were obtained by binding the threads to adjacent processors.

The times are the average of the time obtained in three separate runs. Apart from the four-thread case, it seems that for this example there is a clear benefit in the shared cache. This is most noticeable with two threads. With a larger number of threads, the amount of work done by the individual threads is reduced and so the overhead of running in parallel starts to dominate the time.

Further examples where the loop appeared similar to the following were also run:

```
!$OMP PARALLEL DO PRIVATE(s,j), SHARED(a,b,s1), &
!$OMP&             SCHEDULE(STATIC,1)
          do i=1,m
             do j=1,n
                s=s+a(i,j)*b(i,j)
             end do
             s1(i)=s
          end do
!$OMP END PARALLEL DO
```

These did not show any difference in speed between shared and dedicated L2 cache configurations.

We also tested two examples where we compared performance of two processes accessing a shared cache line where the cache line was in a single L2 cache or moved between L2 caches.

The test programs we used forked child processes which then bound themselves to specific processors. Each process acquired a semaphore, updated a counter, and released the semaphore. The code kernel is as follows:

```
for (i=0;i<LOOPS;i++)
{
   msem_lock(sem,0);
   curr_value= (*shared_counter)++;
   msem_unlock(sem,0);
}
```

The msem_ routines are part of the AIX libsys.a library. They implement an atomic lock using the lwarx instruction.

In the first example (Table 7-6), the counter and semaphore were in the same cache line.

Table 7-6 Counter and semaphore sharing cache line

Case	Time [s]
Single process (no sharing)	3.36
Two processes. L2 cache shared	8.95
Two processes. L2 caches on same MCM	13.31
Two processes. L2 caches on separate MCMs	13.40

In the second example (Table 7-7), the counter and semaphore were in separate cache lines.

Table 7-7 Counter and semaphore in separate cache line

Case	Time [s]	Ratio to shared counter/semaphore
Single process (no sharing)	3.34	0.99
Two processes. L2 cache shared	8.85	0.98
Two processes. L2 caches on same MCM	12.75	0.96
Two processes. L2 caches on separate MCMs	12.82	0.95

As expected, there is a significant performance benefit when two processes share data in the L2 cache. There is also a benefit in separating data structures and the semaphores that control them.

The previous code example makes use of a *sleeping* semaphore. In addition, the amount of work done on the cache line is relatively small. We created a second example using spin/wait instead of sleeping semaphores and increased the amount of work on the cache line. The semaphore and the shared data structure were in separate cache lines.

The relevant code segments are:

```
struct shared_data {
   long long n;
   int counter;
   int i;
   } *p_shared;

msemaphore   *p_shared_sem;
....
   for (i=0;i<LOOPS;i++)
   {
      /* spin waiting for semaphore */
      while ( ((err=msem_lock(p_shared_sem,MSEM_IF_NOWAIT)) == -1)
            && (errno == EAGAIN));

      (p_shared->counter)++;
      p_shared->n=1;
      /* now do some real work */
      for (j=0;j<200;j++)
         {
         p_shared->n = p_shared->n * (p_shared->n +1);
         }
      msem_unlock(p_shared_sem,0);
   }
```

We observed the results provided in Table 7-8.

Table 7-8 Heavily used shared cache line performance

Case	Time [s]
Single process (no sharing)	38.72
Two processes. L2 cache shared	75.91
Two processes. L2 caches on same MCM	84.57
Two processes. L2 caches on different MCMs	84.51
Four processes on two chips on the same MCM	143.27
Four processes, one on each chip on an MCM	156.03
Four processes, each on a different MCM	156.12

The shared L2 cache enables two processes to share the workload very efficiently. Two processes run in twice the time of one process. When the cache is not shared there is an 18 percent penalty. This is independent of whether or not the processes run on the same MCM.

We also examined the case where four processes ran on two chips, that is, four processors sharing two L2 caches, and compared this with unshared L2 caches. We see a small benefit in sharing the L2 cache but once the L2 cache is not shared, the impact of on or off the MCM is the same as for two processes. When testing four processes, we observed that the run times of the individual processes varied (results above are averages for two or four processes). We saw that three processes would run in approximately the same time and one would run in approximately 60 percent to 75 percent of the others. We were not able to investigate this effect for this document. We assume it is either an artifact of the operating system scheduler or an error in the test program. Note that this effect was not observed in the two process test runs.

8

Application performance and throughput

This chapter examines system performance achievable from running multiple copies of a program or programs compared to a single copy of a program. On a multiple processor machine (or node), throughput issues include:

- Processor utilization

  Oversubscribing processors (for example, 12 concurrent jobs on an 8-processor machine) when running processor-bound jobs usually does not increase total processor time significantly as measured by user processor time and system processor time, since the operating system efficiently schedules the jobs to run. Processor-bound jobs refer to programs that are not bottlenecked by any other major system resources.

- Memory bandwidth utilization

  A 32-processor pSeries 690 Turbo has a very high aggregate memory bandwidth of approximately 200 GB/s. For many workloads, this is sufficient to sustain 32 concurrent processes with a performance per process close to that obtained if the processes were to run standalone.

  However, a standalone process is capable of driving the memory bandwidth at a far greater rate than 1/32nd of the total. As detailed in 8.4.3, “Memory stress effects on throughput” on page 162, on a 32-way pSeries 690 Turbo or a 16-way pSeries 690 HPC, a standalone application can use approximately

1/8th of the total bandwidth. This is a significant benefit of pSeries 690 design in cases where such memory-stressing applications can be run in a mixed workload together with low memory stress applications. However, if 32 copies of a job that does use maximum bandwidth are run concurrently on a pSeries 690 Turbo, each job will necessarily take at least 4 times as long as a standalone job. For 16 processes on a 16-way pSeries 690 HPC, it would be at least twice as long.

Most applications, when run standalone, use far less than the maximum bandwidth and there are many techniques, such as blocking, available for reducing the extent of memory stress. Some of these techniques are described in 3.1.4, "Tuning for the memory subsystem" on page 34. Nevertheless, applications that, run standalone, use more than their proportionate share of the total bandwidth, will necessarily run more slowly when every processor is loaded with them. For such workloads, the pSeries 690 HPC is likely to be a more appropriate configuration than a pSeries 690 Turbo.

- Shared L2 cache

  On a pSeries 690 Model 681 or pSeries 690 Turbo, two processors on a single chip share the 1440 KB L2 cache. When two similar jobs are running on the same chip they can effectively utilize only half of the L2 cache and the L2 to L3 cache bandwidth and it could be anticipated that there will be some performance degradation. In a pSeries 690 HPC, there is only one processor that can access the L2 cache, which implies more predictable behavior.

- I/O channels

  When a program has high I/O requirements the I/O channels and subsystems often prove to be the performance bottleneck. When multiple copies of high I/O jobs are run, performance can seriously degrade unless attention is given to separating or hiding I/O transfers. See Section 6.3, "Modular I/O (MIO) library" on page 120, which describes one useful tool that can be used to hide I/O transfers.

The rest of this chapter shows some examples of throughput testing done on POWER4 pSeries 690 Model 681 systems.

8.1 Memory to memory copy

Figure 8-1 shows performance for a simple copy (a[i] = b[i]) for a range of array sizes and numbers of copies. Maximum throughput is achieved when the array fits into the L2 cache. The system was configured with two MCMs and four memory books.

Tuning by using methods such as

```
a[i] = b[i] + zero*a[i];
```

does not make any significant difference to copy performance, whereas this technique was often beneficial on the POWER3.

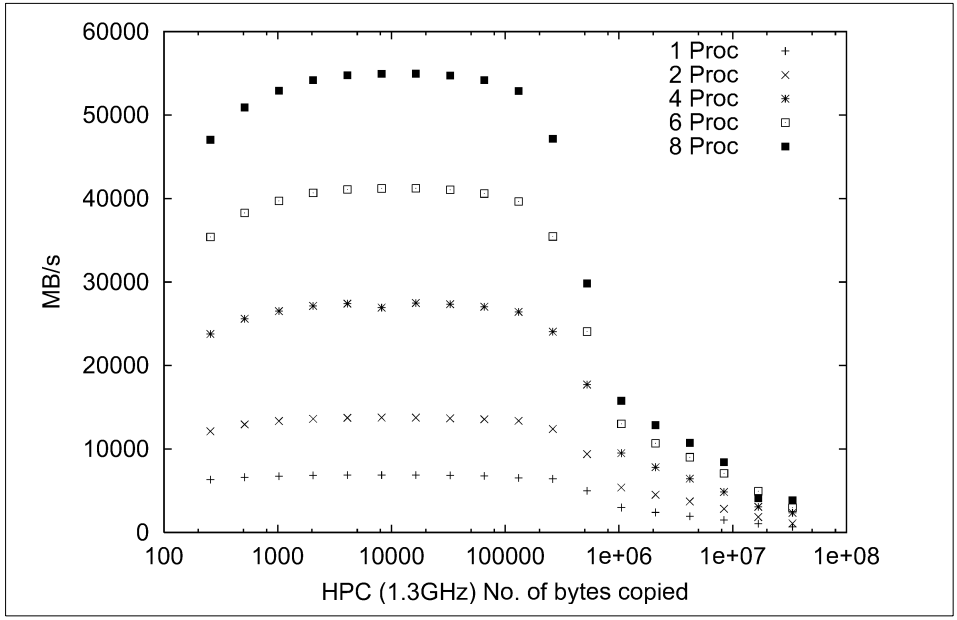


Figure 8-1 Memory copy performance

Figure 8-2 on page 156 shows corresponding performance when using the C library memcpy() function. Performance is less than that achieved in the case above because the load/stores are 8 bytes (Fortran REAL*8) whereas the memcpy() function loads and stores bytes to a word boundary then loads and stores words (4 bytes) and completes the copy with bytes if required.

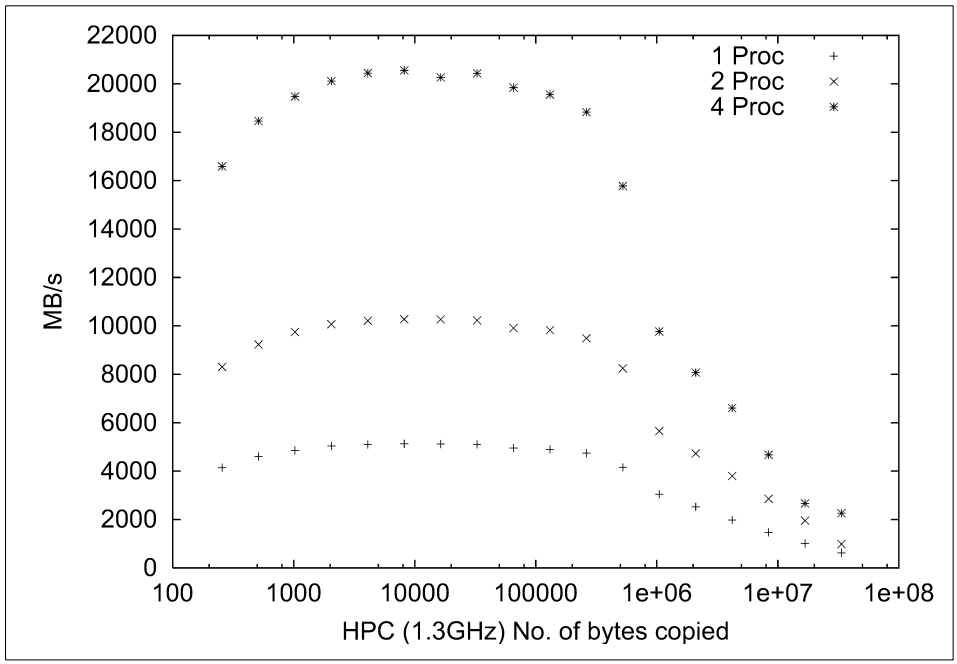


Figure 8-2 C library memcpy performance

The memory subsystem provides a reasonably linear response to increasing processor load as provided in Table 8-1.

Table 8-1 Memory copy performance relative to one CPU

	2 CPUs	**4 CPUs**	**6 CPUs**	**8 CPUs**
16 KB	2.0	3.9	6.0	8.0
32 KB	2.0	4.0	6.0	8.0
128 KB	2.0	4.0	6.1	8.1
1 MB	1.8	3.2	4.4	5.3
2 MB	1.9	3.2	4.5	5.4
4 MB	1.9	3.3	4.7	5.6
8 MB	1.9	3.2	4.7	5.6

8.2 Memory bandwidth limited throughput

In contrast to the performance described in the previous section, this section describes a throughput test that deliberately challenges the total system memory. The program solves for the dot product of two REAL*8 arrays of length N. For this throughput test, N was chosen to be 110000000 to ensure that most of the data would not be resident in L3 cache.

A single copy of this program achieved a 2.3 GB/s transfer rate on a 1.3 GHz processor. When eight copies of this job were run on a two MCM, eight processor pSeries 690 HPC, the aggregate data transfer rate was 11.3 GB/s, a speedup of 4.9. The aggregate transfer rates for job counts of 1 through 8 are shown in Figure 8-3.

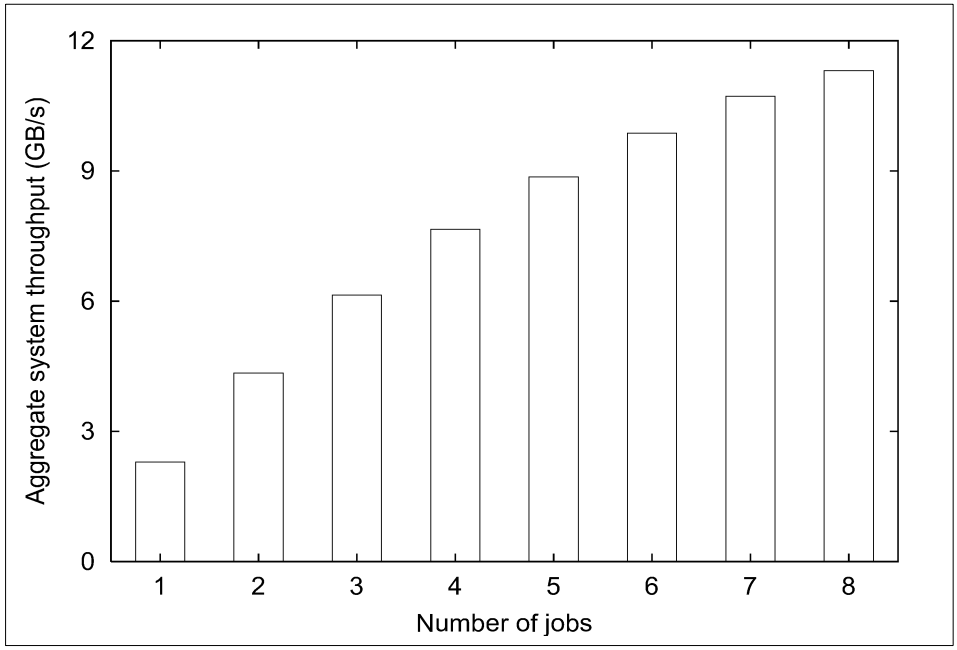


Figure 8-3 System memory throughput for pSeries 690 HPC

A second throughput test using the same program was run on a 32-way four MCM,1.3 GHz pSeries 690 Turbo with 96 GB of real memory. This system had 512 MB of shared L3 cache so the array sizes were increased to 310 million double-precision elements up through 16 processors to ensure most of the data was not in L3 cache. For 32 copies of the program, the array sizes were reduced to 150 million double-precision elements, to prevent the programs from exceeding the real memory on the system. The aggregate system throughput rates are shown in Figure 8-4.

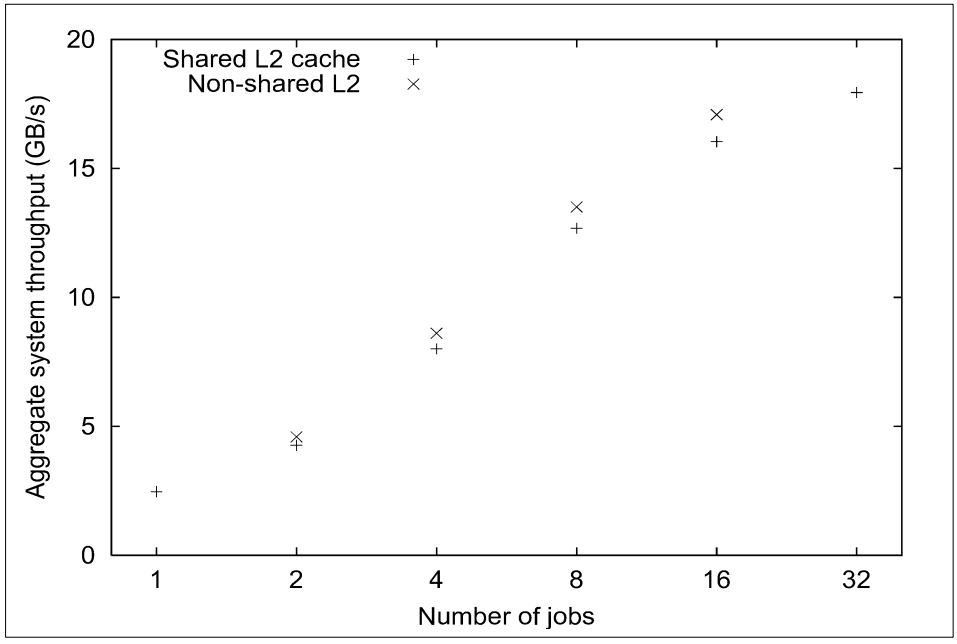


Figure 8-4 System memory throughput on pSeries 690 Turbo

The shared L2 cache and non-shared L2 cache throughput rates for 2 through 16 jobs were obtained using the AIX **`bindprocessor`** command and related system calls. For the shared L2 cache runs, pairs of jobs were bound to processors on the same POWER4 chip. For the non-shared L2 cache runs, at most one job was bound to any POWER4 chip. The non-shared L2 cache performance would be nearly identical to the performance of a pSeries 690 HPC system. As expected, when two jobs share the L2 cache, the system throughput decreased.

It should be noted that this program relies on hardware prefetch streams. The performance of the prefetch streams are highly dependent on the size of memory pages. At the time this document was written, only 4 KB pages were available in AIX 5L. Large page sizes, which will be available in early 2002, are expected to significantly increase the single job and multiple job throughput for these examples.

8.3 MPI parallel on pSeries 690 and SP

This section describes a hydrodynamics benchmark application called *Hydra* from AWE, Aldermaston in the UK and is included as an example to illustrate the comparative performance of pSeries 690 Model 681 and the RS/6000 SP 375 MHz POWER3 SMP High Node both with respect to absolute performance and parallel scalability. The RS/6000 SP 375 MHz POWER3 SMP High Node, is a shared memory unit that contains 16 POWER3-II processors running at a frequency of 375 MHz.

Hydra is written in Fortran with MPI message passing and scales well to at least 512 processors for large problems. It does not use OpenMP or similar paradigms.

The results for two test cases, a medium one called *2mm* and a large one called *1mm* are shown in Table 8-2. All MPI communication is shared memory with the single exception of the 32-way RS/6000 SP 375 MHz POWER3 SMP High Node case where user space MPI (EUILIB=us) was used over the IBM Switch2 connecting two SP nodes.

For this application, conclusions that can be drawn include:

- Up to 32-way parallel, a 1.3 GHz pSeries 690 Model 681 system is between 2.1 and 3.1 times faster than the same number of RS/6000 SP 375 MHz POWER3 SMP High Node processors.
- Scalability characteristics of a single pSeries 690 Model 681 system are similar to that of an RS/6000 SP 375 MHz POWER3 SMP High Node.

Table 8-2 MPI performance results for AWE Hydra code

	2mm test case			**1mm test case**		
Parallelism	**Elapsed seconds**	**Parallel speedup**	**Ratio over NH2**	**Elapsed seconds**	**Parallel speedup over 4-way run**	**Ratio over NH2**
16-processor RS/6000 SP 375 MHz POWER3 SMP High Nodes						
Serial	8776.4	1	1	not measured	N/A	1
2-way	4598.9	1.91	1	not measured	N/A	1
4-way	2331.1	3.76	1	28645	1	1
8-way	1286.2	6.82	1	14065	2.04	1
16-way	754.6	11.63	1	8033	3.57	1
32-way	388.5	22.59	1	3913	7.32	1

	2mm test case			1mm test case		
Parallelism	**Elapsed seconds**	**Parallel speedup**	**Ratio over NH2**	**Elapsed seconds**	**Parallel speedup over 4-way run**	**Ratio over NH2**
64-way	229.0	38.32	1	2080	13.77	1
128-way	141.2	62.16	1	1101	26.02	1
8-processor pSeries 690 HPC, results normalized to 1.3 GHz						
Serial	3297.5	1	2.66	not measured	not measured	not measured
2-way	1701.6	1.93	2.70	not measured	not measured	not measured
4-way	926.9	3.56	2.51	11349	1	2.52
8-way	537.9	6.13	2.39	6307	1.80	2.23
32-processor pSeries 690 Model 681, 1.3 GHz.						
4-way	752.3	3.56*	3.10	not measured	not measured	not measured
8-way	468.9	5.71*	2.74	not measured	not measured	not measured
16-way	272.1	9.84*	2.77	not measured	not measured	not measured
32-way	160.5	16.68*	2.42	1821	N/A	2.15

* Assuming 4-way speedup is same as pSeries 690 HPC, that is, 3.56.

8.4 Multiple job throughput

This section discusses the extent to which the total execution times for different types of jobs increases when multiple jobs are run concurrently. Examples are given of two jobs that only lightly stress the I/O and memory subsystems and hence give excellent throughput scaling. Then results from an artificial job are shown adjusted to provided varying degrees of memory subsystem stress.

We have not been able to investigate throughput effects for I/O intensive jobs. This is a very important subject area, but only a very limited I/O configuration was available to us on the pSeries 690 Model 681 systems we tested. Results from any I/O intensive applications would, therefore, have been unrealistic.

8.4.1 ESSL DGEMM throughput performance

Multiple copies of DGEMM from ESSL (see Section 6.1, “The ESSL and Parallel ESSL libraries” on page 114) were run together on a 32-way pSeries 690 Turbo. Each job multiplied matrices of 5000x5000 REAL*8 numbers, which require 600 MB of memory for the three arrays involved. However, because ESSL blocks the code to achieve good memory locality, and because matrix multiply involves a high ratio of computation to memory access, almost no slowdown was seen when multiple copies were run.

Table 8-3 lists the performance of the jobs in GFLOPS. Any slowdown due to running multiple copies would be evidenced by decreasing values for GFLOPS as the number of jobs increases. However, the multiple job slowdown is very small in all cases, being only 11 percent when 32 concurrent jobs were run on a 32-way pSeries 690 Turbo.

Table 8-3 Effects of running multiple copies of DGEMM

Number of jobs	GFLOPS for 5000x5000 REAL*8 matrices	Slowdown ratio to single job
1.3 GHz 32-way pSeries 690 Turbo		
1	3.417	1
8	3.338	1.02
16	3.253	1.05
24	3.167	1.08
32	3.079	1.11

As can be calculated from Table 8-3, 32 copies of the same program achieve a total performance rate of 98.5 GFLOPS.

8.4.2 Multiple ABAQUS/Explicit job streams

ABAQUS/Explicit is a commercially available structural analysis code from HKS Inc. of Pawtuckett, Rhode Island. It uses an explicit (rather than implicit) solution technique and, therefore, does not perform heavy I/O or memory access operations. The jobs run were HKS’s seven standard timing tests, t1-exp through t7-exp and the time reported is the total elapsed time to run all seven.

In addition to running a single stream of jobs, four and then eight sets of the seven timing jobs were run concurrently on an eight-processor 1.3 GHz pSeries 690 HPC (different from the 1.1 GHz machine used for most of the other measurements in this publication). The times for the three runs are shown in Table 8-4 and are the total elapsed seconds to complete all jobs.

These results show excellent throughput scaling from the pSeries 690 HPC for this application. The 8-stream run, using all processors, takes only 2 percent longer than a single-stream run.

Table 8-4 Multiple ABAQUS/Explicit job stream times

Number of job stream	Elapsed seconds	Slowdown ratio to single stream
1	2439	1
4	2459	1.01
8	2488	1.02

8.4.3 Memory stress effects on throughput

Compared with the previous sections that showed jobs with excellent total throughput, this section describes a *worst case* example of a job that is designed to stress memory as much as possible. Most production applications will stress memory significantly less than this. As will be explained, this study demonstrates the benefits of the pSeries 690 HPC models for high-memory stress applications.

A simple program was used consisting of repeated calls to a subroutine that executed the statement A(I)=B(I)+C(I)*D(I) in a loop. This code stresses memory in much the same way as the dot-product test reported in Section 8.2, “Memory bandwidth limited throughput” on page 157. The results presented here are consistent with those in that section but are presented in a way that focuses on the total throughput obtained by running multiple copies of the job.

Figure 8-5, Figure 8-6, and Figure 8-7 show the interactions between a number of jobs plotted as a function of the total amount of memory accessed by each program. Results are shown for a 16-way RS/6000 SP 375 MHz POWER3 SMP High Node, an 8-way pSeries 690 HPC and a 32 processor pSeries 690 Model 681. First, these figures are discussed individually and then some overall conclusions are drawn.

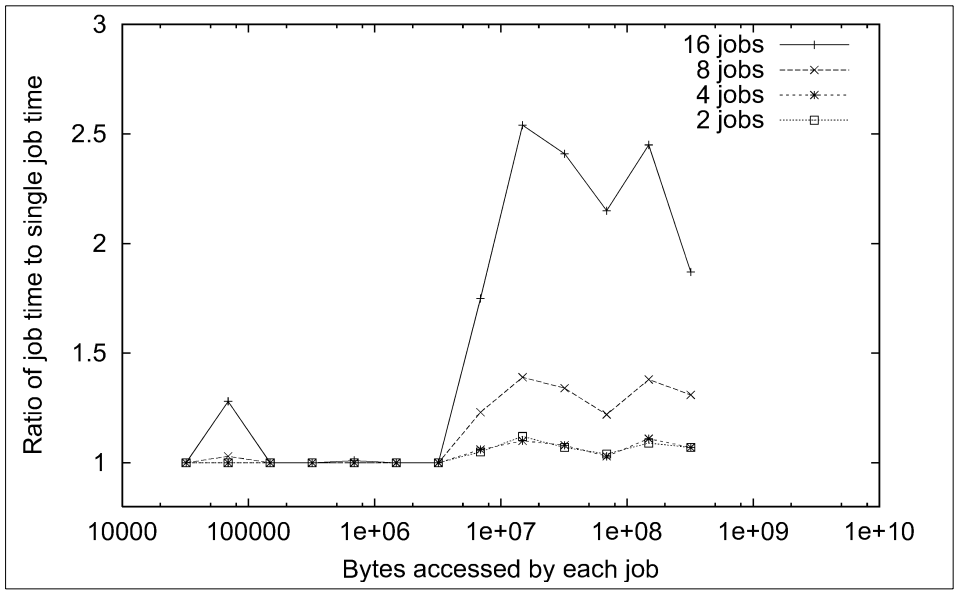


Figure 8-5 Job throughput effects on a 375 MHz POWER3 SMP High Node

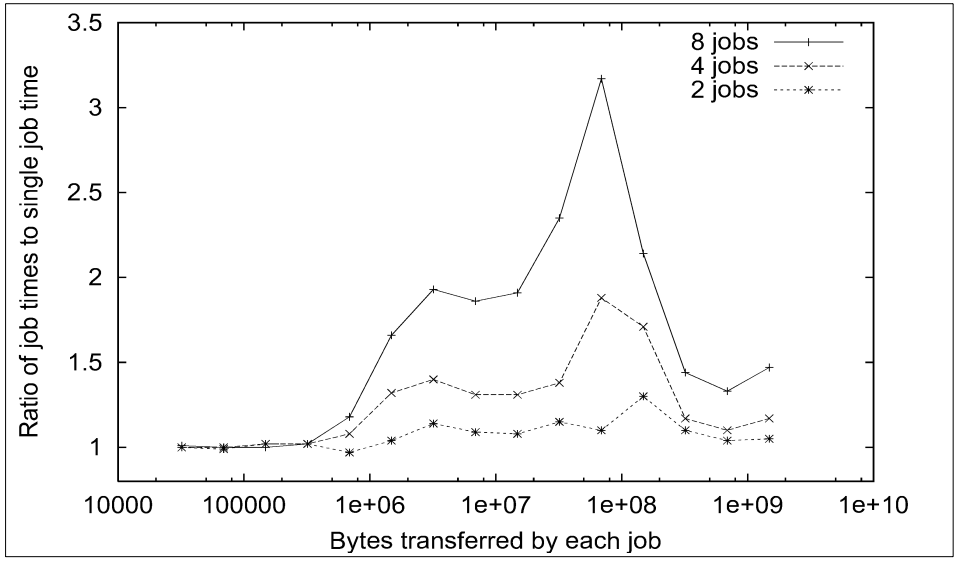


Figure 8-6 Job throughput effects on an eight-way pSeries 690 HPC

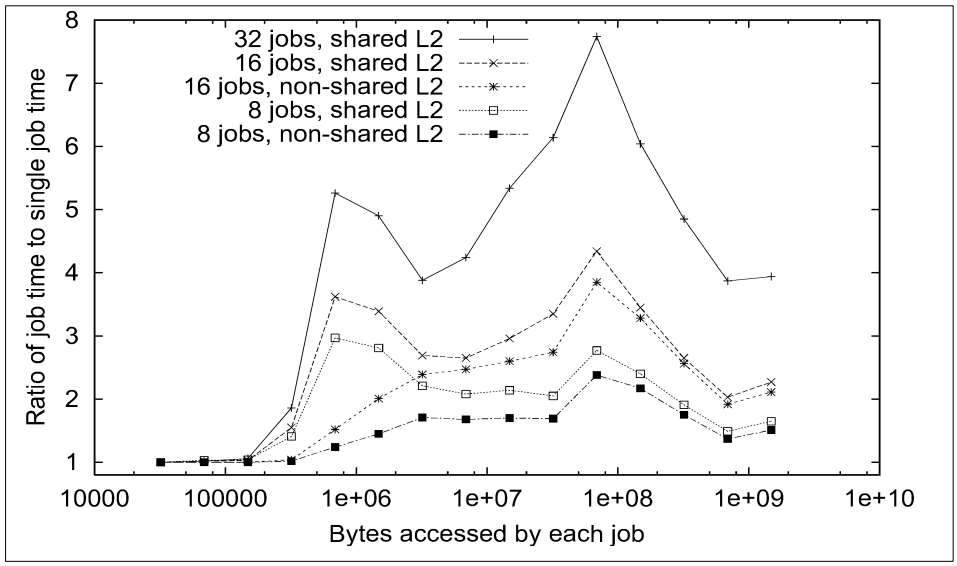


Figure 8-7 Job throughput effects on a 32-way pSeries 690 Turbo

- 16-way RS/6000 SP 375 MHz POWER3 SMP High Node

  Each processor has a local 8 MB L2 cache. The times for multiple jobs start to exceed the single job time when the memory accessed by each job approaches this value.

- pSeries 690 HPC and pSeries 690 Model 681

  On the pSeries 690 HPC, each processor has its own local L2 cache whereas on the pSeries 690 Turbo, even/odd pairs of processors share a local L2 cache. To illustrate the effect of this, on the pSeries 690 Turbo, the 8 and 16 job runs were done in two ways. The shared L2 cache runs were done with the jobs bound sequentially to processors. The non-shared L2 cache runs were done with the jobs bound only to even processors so that no two jobs were ever sharing the cache. The non-shared runs are expected to behave in the same way as a pSeries 690 HPC and it can be seen that the 8 jobs, non-shared L2 graph in Figure 8-7 (pSeries 690 Turbo) is very similar to the 8 jobs graph in Figure 8-6 (pSeries 690 HPC).

Conclusions from the graphs

The following are the conclusions that may be developed from the graphs:

- The graphs for pSeries 690 Model 681 are more complicated than for RS/6000 SP 375 MHz POWER3 SMP High Node because of the presence of the Level 3 cache.

- As a consequence of the pSeries 690 design, in which each processor has a powerful data prefetch engine plus full access the L3 cache and memory bandwidth on its MCM, it takes relatively few jobs as stressful at this to consume the system's resources. This design feature provides the maximum opportunity for a mixture of jobs with arbitrary resource demands to achieve the best possible system throughput. Once a system resource such as L3 cache or memory bandwidth is fully consumed, however, adding more jobs to the system will not improve overall system throughput. This effect is seen on the pSeries 690 Model 681 around the point where each program accesses around 100 MB of data. This happens because of the shared L3 cache (256 MB on the two-MCM pSeries 690 HPC and 512 MB on the four-MCM pSeries 690 Turbo). Multiple jobs can be slowed down by spilling out of L3 cache, necessitating additional accesses to main memory subsystem.
- A similar shared-L2 cache effect can be seen on the three shared L2 lines on the pSeries 690 Turbo (Figure 8-7) around the point where the jobs access around 750 KB of memory and spill out of L2. The pSeries 690 HPC-like lines do not show any such effect.
- Throughput in the *worst case* region where the programs are working wholly outside any cache shows job times of approximately four times single job times for 32 jobs on a 32-way pSeries 690 Turbo, approximately two times for 16 jobs on a 16-way pSeries 690 HPC, and approximately 1.5 times for 8 jobs on an 8-way pSeries 690 HPC.
- In general, throughput is expected to improve for this example with a future update to AIX 5L in which pages are allocated from memory attached to the MCM where the process is running, thus minimizing MCM-to-MCM traffic. (See 3.2.2, “Memory configurations” on page 53).
- The benefit of the pSeries 690 HPC design over pSeries 690 Turbo for a memory stressing job mix is clear.

8.4.4 Shared L2 cache and logical partitioning (LPAR)

FIRE is a commercially available computational fluid dynamics (CFD) analysis code from AVL List GmbH, Austria (http://www.avl.com). There are optimized versions of FIRE available for scalar, vector, and parallel (shared and distributed memory) architectures. For the following study, the SMP Version V7.3, compiled with XLF 6.1 for the POWER3 platform, was used.

AVL provides several standard test cases for benchmark purposes. In the following, the test cases water (water cooling jacket; 284,000 cells) and ext3d (external flow; 711,000 cells) are investigated. A sequential job needs about 300 MB (water) or 620 MB (ext3d) of memory, respectively. FIRE is a memory bandwidth demanding application. The time for I/O is negligible.

The following machines were used, which not only differ by clock frequency but also by memory speed and micro code level:

hpc 8-CPU pSeries 690 Model 681 HPC at 1.1 GHz (memory at 328 MHz)

turbo 32-CPU pSeries 690 Model 681 Turbo at 1.3 GHz (memory at 400 MHz)

lpar 8-CPU pSeries 690 Model 681 HPC at 1.0 GHz (memory at 400 MHz, a development system, two logical partitions of four processors each)

Performance impact of shared versus non-shared L2 cache

The performance impact of a shared L2 cache can be studied when binding two threads of a two-CPU parallel job to two processors that either belong to the same POWER4 chip or to different chips. It turns out that the shared cache has very little impact on performance with respect to FIRE. Table 8-5 contains the job execution times on a pSeries 690 Model 681 Turbo.

Table 8-5 FIRE benchmark: Impact of shared versus non-shared L2 cache

Elapsed time [s]	water	ext3d
Sequential (pSeries 690 Turbo)	228.8	525.6
Two-CPU -- shared (pSeries 690 Turbo)	127.1	303.7
Two-CPU -- not shared (pSeries 690 Turbo)	125.9	297.8

Impact of partitioning on single job performance

Logical partitioning is expected to introduce a little overhead on memory access. However, it is possible to distribute a throughput workload across several LPARs in order to isolate single jobs or groups of jobs. Whether throughput performance can benefit from partitioning depends on how the physical resources are mapped onto the different LPARs.

For the following benchmark a pSeries 690 Model 681 HPC system is divided into two LPARs. Each LPAR consists of one MCM (four CPUs). Only the MCM's local memory was assigned to its LPAR (this configuration is not supported through standard hardware management console function, a system reset after an LPAR reconfiguration was required to achieve this through trial and error). Results for a single sequential job running in a LPAR are presented in Table 8-6. Timings normalized to 1.3 GHz are given in parentheses. Note, partitioning does

not degrade single job performance. A slight benefit was even observed, which might be close to the bounds of the experimental error. The benefit is likely due to the chosen memory affinity. The ratio between clock frequency and memory frequency also bias the measurement.

Table 8-6 FIRE benchmark: Uniprocessor, single job versus partitioning

Elapsed time [s]	water	ext3d
no LPAR pSeries 690 HPC	260.1 (220.1)	627.0 (530.5)
LPAR 1 (64-bit kernel)	274.9 (211.5)	648.8 (499.1)
LPAR 2 (32-bit kernel)	283.0 (217.7)	659.2 (507.1)

Impact of partitioning on throughput performance

Running eight sequential FIRE jobs on an eight-way machine is the throughput scenario that puts the most stress on the memory subsystem. For the particular setup of this benchmark, it is observed that partitioning can reduce the interference between different processes of a throughput workload and therefore improve the throughput performance. The results are presented in Table 8-7. Timings normalized to 1.3 GHz are given in parentheses.

Table 8-7 FIRE benchmark: Throughput performance versus partitioning

Elapsed time [s]	water	ext3d
Single job, no LPAR (pSeries 690 Turbo)	228.8	525.6
Single job, no LPAR (pSeries 690 HPC)	260.1 (220.1)	627.0 (530.5)
Eight jobs, no LPAR (pSeries 690 HPC)	452.8 (383.1)	1004.3 (849.8)
Four jobs using LPAR 2 LPAR 1 idle	415.9 (319.9)	905.6 (696.6)
Four jobs using LPAR 1 Four jobs using LPAR 2	409.5 (315.0) 421.8 (324.5)	919.7 (707.5) 926.4 (712.6)

8.5 Genetic sequencing program

A genetic sequencing program was run on a number of systems including POWER4 to determine relative performance. The program is written in C and comprises a mixture of floating-point arithmetic, character manipulation and file I/O. Table 8-8 lists performance results on the different systems. Use of POWER4 specific optimization provides a noticeable benefit compared to -qarch=com.

Table 8-8 Performance on different systems

System and compiler flags	Elapsed Time
POWER3 -O3 (375 MHz)	22m 21s
S80 -O3 (450 MHz)	45m 22s
POWER4 -O3 -qarch=com (1.3 GHz HPC)	11m 42s
POWER4 -O3 -qarch=pwr4 -qtune=pwr4 (1.3GHz HPC)	10m 42s

8.6 FASTA genetic sequencing program

The FASTA program suite provides a number of utilities for local sequence alignment of DNA or protein sequences against corresponding sequence databases. The FASTA utility uses a fast, heuristic algorithm. The search utility uses a Smith-Waterman algorithm. Comparison tests for two well-known sequences, arp_arath (536AA) and metr_salty (276AA), were run against the Swiss-Prot Release 39 database using both algorithms. Note that these utilities do significant amounts of I/O. The sequence database is approximately 250 MB. Table 8-9 provides a single-processor performance comparison against POWER3.

Table 8-9 Relative performance of FASTA utilities

Sequence	POWER3	POWER4	Speedup
arp_arath (fasta)	26.47	15.54	1.7
arp_arath(ssearch)	453.72	300.30	1.5
metr_salty(fasta)	20.84	12.09	1.7
metr_salty(ssearch)	230.45	153.03	1.5

8.7 BLAST genetic sequencing program

BLAST (Basic Local Alignment Search Tool) is a suite of applications for searching DNA sequence databases. The BLAST algorithm makes pairwise comparisons of sequences, seeking regions of local similarity rather than optimal global alignment. BLAST 2.2.1 can perform gapped or ungapped alignments.

blastn DNA sequence queries can be performed against DNA sequence databases.

tblastn Protein sequence query performed against a DNA sequence database dynamically translated in all six reading frames.

As with the FASTA tests, the BLAST programs perform varying and typically significant amounts of I/O. Relative performance on POWER3 and POWER4 for blastn and tblastn are provided in Table 8-10 and Table 8-11, respectively:

Table 8-10 Blastn results

Query	POWER3	POWER4	Ratio
nt.2655203	180	81	2.2
nt.3283410	70	31	2.3
nt,5764416	10	4	2.5

Table 8-11 Tblastn results

Query	POWER3	POWER4	Ratio
nt.1177466	170	68	2.5
nt.129295	63	23	2.7
nt,231729	95	35	2.7

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 173.

- *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, SG24-5155
- *Scientific Applications in RS/6000 SP Environments*, SG24-5611
- *AIX 5L Performance Tools Handbook*, SG24-6039

Other resources

These publications are also relevant as further information sources:

- *IBM RISC System/6000 Technology*, SA23-2619
- *XL Fortran for AIX User’s Guide*, SC09-2866
- *XL Fortran for AIX Language Reference*, SC09-2867
- *Optimization and Tuning Guide for Fortran, C, and C++*, SC09-1705
- *Accelerating AIX* by Rudy Chukran, Addison-Wesley, 1998
- *AIX Performance Tuning* by Frank Waters, Prentice -Hall, 1996
- You can access all of the AIX documentation through the Internet at the following URL: `http://www.ibm.com/servers/aix/library`

  The following types of documentation are located on the documentation CD that ships with the AIX operating system:

  – User guides
  – System management guides
  – Application programmer guides
  – All commands reference volumes
  – Files reference
  – Technical reference volumes used by application programmers

Referenced Web sites

These Web sites are also relevant as further information sources:

- AIX and RS/6000 SP manuals

  http://www.ibm.com/servers/aix/library/techpubs.html
- MIO library

  http://www.research.ibm.com/actc/Opt_Lib/mio/mio_doc.htm
- Watson Sparse Matrix Package (WSMP)

  http://www.cs.umn.edu/~agupta/wsmp.html
- AIX Bonus Pack

  http://www.ibm.com/servers/aix/products/bonuspack
- CFD (computational fluid dynamics) application FIRE

  http://www.avl.com
- SPPM

  http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/sppm_readme.html
- BLAS

  http://www.netlib.org/blas
- LAPACK

  http://www.netlib.org/lapack
- ATLAS

  http://math-atlas.sourceforge.net
- MASS

  http://www.rs6000.ibm.com/resource/technology/MASS
- ESSL

  http://www-1.ibm.com/servers/eserver/pseries/software/sp/essl.html
- HKS Abaqus

  http://www.abaqus.com
- SPEC

  http://www.specbench.org
- TPC

  http://www.tpc.org
- STREAM

  http://www.cs.virginia.edu/stream

- NAS

  http://www.nas.nasa.gov//NAS/NPB

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

This document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others

Abbreviations and acronyms

ABI	Application Binary Interface
AFPA	Adaptive Fast Path Architecture
AIX	Advanced Interactive Executive
ANSI	American National Standards Institute
APAR	Authorized Program Analysis Report
API	Application Programming Interface
ASCI	Accelerated Strategic Computing Initiative
ASCII	American National Standards Code for Information Interchange
ASR	Address Space Register
AUI	Attached Unit Interface
BCT	Branch on Count
BIST	Built-In Self-Test
BLAS	Basic Linear Algebra Subprograms
BOS	Base Operating System
CAD	Computer-Aided Design
CAE	Computer-Aided Engineering
CAM	Computer-Aided Manufacturing
CATIA	Computer-Graphics Aided Three-Dimensional Interactive Application
CDLI	Common Data Link Interface
CD-R	CD Recordable
CE	Customer Engineer
CEC	Central Electronics Complex
CFD	Computational Fluid Dynamics
CGE	Common Graphics Environment
CHRP	Common Hardware Reference Platform
CIU	Core Interface Unit
CLIO/S	Client Input/Output Sockets
CMOS	Complimentary Metal-Oxide Semiconductor
CPU	Central Processing Unit
CWS	Control Workstation
D-Cache	Data Cache
DAD	Duplicate Address Detection
DAS	Dual Attach Station
DASD	Direct Access Storage Device
DFL	Divide Float
DIMM	Dual In-Line Memory Module
DIP	Direct Insertion Probe
DMA	Direct Memory Access
DOE	Department of Energy
DOI	Domain of Interpretation
DPCL	Dynamic Probe Class Library
DRAM	Dynamic Random Access Memory
DSA	Dynamic Segment Allocation
DSE	Diagnostic System Exerciser
DSU	Data Service Unit
DTE	Data Terminating Equipment
DW	Data Warehouse
EA	Effective Address
EC	Engineering Change

ECC	Error Checking and Correcting
EEPROM	Electrically Erasable Programmable Read Only Memory
EFI	Extensible Firmware Interface
EHD	Extended Hardware Drivers
EIA	Electronic Industries Association
EISA	Extended Industry Standard Architecture
ELF	Executable and Linking Format
EPOW	Environmental and Power Warning
ERAT	Effective-to-Real Address Table
ERRM	Event Response resource manager
ESID	Effective Segment ID
ESSL	Engineering and Scientific Subroutine Library
ETML	Extract, Transformation, Movement and Loading
F/C	Feature Code
F/W	Fast and Wide
FBC	Fabric Bus Controller
FDPR	Feedback Directed Program Restructuring
FIFO	First In/First Out
FLIH	First Level Interrupt Handler
FMA	Floating-point Multiply/Add operation
FPR	Floating-Point Register
FPU	Floating-Point Unit
FRCA	Fast Response Cache Architecture
FRU	Field Replaceable Unit
GAMESS	General Atomic and Molecular Electronic Structure System
GCT	Global Completion Table
GFLOPS	Billion FLoating-point Operations Per Second
GPFS	General Parallel File System
GPR	General-Purpose Register
HACWS	High Availability Control Workstation
HiPS	High Performance Switch
HiPS LC-8	Low-Cost Eight-Port High Performance Switch
HPF	High Performance Fortran
HPSSDL	High Performance Supercomputer Systems Development Laboratory
Hz	Hertz
I-CACHE	Instruction Cache
I/O	Input/Output
I2C	Inter Integrated-Circuit Communications
IA	Intel Architecture
IAR	Instruction Address Register
IBM	International Business Machines
ID	Identification
IDE	Integrated Device Electronics
IDS	Intelligent Decision Server
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IFAR	Instruction Fetch Address Register
IHV	Independent Hardware Vendor
IM	Input Method

INRIA	Institut National de Recherche en Informatique et en Automatique
IPL	Initial Program Load
IRQ	Interrupt Request
ISA	Industry Standard Architecture, Instruction Set Architecture
ISO	International Organization for Standardization
ISV	Independent Software Vendor
ITSO	International Technical Support Organization
L1	Level 1
L2	Level 2
L3	Level 3
LAPI	Low-Level Application Programming Interface
LED	Light Emitting Diode
LFD	Load Float Double
LID	Load ID
LLNL	Lawrence Livermore National Laboratory
LMQ	Load Miss Queue
LP	Linear Programming
LP64	Long-Pointer 64
LPP	Licensed Program Product
LRQ	Load Reorder Queue
LRU	Least Recently Used
MASS	Mathematical Acceleration Subsystem
MAU	Multiple Access Unit
Mbps	Megabits Per Second
MBps	Megabytes Per Second
MCAD	Mechanical Computer-Aided Design
MCM	Multi-chip Module
NCU	Non-Cacheable Unit
MDI	Media Dependent Interface
MES	Miscellaneous Equipment Specification
MFLOPS	Million FLoating-point Operations Per Second
MII	Media Independent Interface
MIP	Mixed-Integer Programming
MLD	Merged Logic DRAM
MLR1	Multi-Channel Linear Recording 1
MODS	Memory Overlay Detection Subsystem
MP	Multiprocessor
MPI	Message Passing Interface
MPP	Massively Parallel Processing
MPS	Mathematical Programming System
MST	Machine State
NTF	No Trouble Found
NUMA	Non-Uniform Memory Access
NUS	Numerical Aerodynamic Simulation
NVRAM	Non-Volatile Random Access Memory
NWP	Numerical Weather Prediction
OACK	Option Acknowledgment
OCS	Online Customer Support
ODBC	Open DataBase Connectivity
OEM	Original Equipment Manufacturer
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OSL	Optimization Subroutine Library
OSLP	Parallel Optimization Subroutine Library
P2SC	POWER2 Single/Super Chip

PBLAS Parallel Basic Linear Algebra Subprograms

PCI Peripheral Component Interconnect

PDT Paging Device Table

PDU Power Distribution Unit

PE Parallel Environment

PEDB Parallel Environment Debugging

PHB Processor Host Bridge

PHY Physical Layer

PID Process ID

PII Program Integrated Information

PIOFS Parallel Input Output File System

PMU Performance Monitoring Unit

POE Parallel Operating Environment

POSIX Portable Operating Interface for Computing Environments

POST Power-On Self-test

POWER Performance Optimization with Enhanced Risc (Architecture)

PPC PowerPC

PPM Piecewise Parabolic Method

PSE Portable Streams Environment

PSSP Parallel System Support Program

PTF Program Temporary Fix

PTPE Performance Toolbox Parallel Extensions

PVC Permanent Virtual Circuit

QP Quadratic Programming

RAM Random Access Memory

RAN Remote Asynchronous Node

RAS Reliability, Availability, and Serviceability

RFC Request for Comments

RIO Remote I/O

RIPL Remote Initial Program Load

RISC Reduced Instruction-Set Computer

ROLTP Relative Online Transaction Processing

RPA RS/6000 Platform Architecture

RPL Remote Program Loader

RPM Red Hat Package Manager

RSC RISC Single Chip

RSCT Reliable Scalable Cluster Technology

RSE Register Stack Engine

RSVP Resource Reservation Protocol

RTC Real-Time Clock

SAR Solutions Assurance Review

SAS Single Attach Station

ScaLAPACK Scalable Linear Algebra Package

SCB Segment Control Block

SCO Santa Cruz Operations

SDQ Store Data Queue

SDRAM Synchronous Dynamic Random Access Memory

SEPBU Scalable Electrical Power Base Unit

SGI Silicon Graphics Incorporated

SHLAP Shared Library Assistant Process

SID Segment ID

SIT Simple Internet Transition

SLB Segment Look-aside Buffer, Server Load Balancing

SLIH	Second Level Interrupt Handler
SLR1	Single-Channel Linear Recording 1
SM	Session Management
SMB	Server Message Block
SMI	System Memory Interface
SMP	Symmetric Multiprocessor
SOI	Silicon-on-Insulator
SP	Service Processor
SP	IBM RS/6000 Scalable POWER Parallel Systems
SP	Service Processor
SPCN	System Power Control Network
SPEC	System Performance Evaluation Cooperative
SPM	System Performance Measurement
SPR	Special Purpose Register
SPS	SP Switch
SPS-8	Eight-Port SP Switch
SRQ	Store Reorder Queue
SRN	Service Request Number
SSA	Serial Storage Architecture
SSC	System Support Controller
SSQ	Store Slice Queue
SSL	Secure Socket Layer
STE	Segment Table Entry
STFDU	Store Float Double with Update
STP	Shielded Twisted Pair
STQ	Store Queue
SUID	Set User ID
SUP	Software Update Protocol
SVC	Switch Virtual Circuit
SVC	Supervisor or System Call
SYNC	Synchronization
TCE	Translate Control Entry
Tcl	Tool Command Language
TCQ	Tagged Command Queuing
TGT	Ticket Granting Ticket
TLB	Translation Lookaside Buffer
TOS	Type Of Service
TPC	Transaction Processing Council
TPP	Toward Peak Performance
TTL	Time To Live
UDI	Uniform Device Interface
UIL	User Interface Language
ULS	Universal Language Support
UP	Uniprocessor
USLA	User-Space Loader Assistant
UTF	UCS Transformation Format
UTM	Uniform Transfer Model
UTP	Unshielded Twisted Pair
VA	Virtual Address
VESA	Video Electronics Standards Association
VFB	Virtual Frame Buffer
VHDCI	Very High Density Cable Interconnect
VLAN	Virtual Local Area Network
VMM	Virtual Memory Manager
VP	Virtual Processor
VPD	Vital Product Data
VPN	Virtual Private Network
VSD	Virtual Shared Disk
VT	Visualization Tool
XCOFF	Extended Common Object File Format
XLF	XL Fortran

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

IBM Redbooks

The POWER4 Processor Introduction and Tuning Guide

(0.2”spine)
0.17”<->0.473”
90<->249 pages

The POWER4 Processor Introduction and Tuning Guide

Comprehensive explanation of POWER4 performance

Includes code examples and performance measurements

How to get the most from the compiler

This redbook is designed to familiarize you with the IBM @server pSeries POWER4 microarchitecture and to provide you with the information necessary to exploit the new high-end servers based on this architecture.

The eight to 32-way symmetric multiprocessing (SMP) pSeries 690 Model 681 will be the first POWER4 system to be available. Thus, most analysis presented in this publication refers to this system.

Specifically, this publication will address the following issues:

- POWER4 features and capabilities
- Processor and memory optimization techniques, especially for Fortran programming
- AIX XL Fortran Version 7.1.1 compiler capabilities and which options to use
- Parallel processing techniques and performance
- Available libraries and programming interfaces
- Performance examples of commonly used kernels

While this publication is decidedly technical in nature, the fundamental concepts are presented from a user point of view and numerous examples are provided to reinforce these concepts.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7041-00 ISBN 0738423556