IBM

e business software

Performance Programming with IBM pSeries Compilers

March 27, 2002

Bob Blainey

blainey@ca.ibm.com

Agenda

- **Part 1**
  - PowerPC Architecture
  - pSeries processors
  - AIX performance tools
  - Performance libraries
  - Performance coding
  - Q&A
- **Part 2**
  - Review of pSeries compiler products
  - Tutorial on performance controls
  - Programming for performance
  - A peek inside the compiler
  - A closer look at Power 4 optimization
  - Q&A

PowerPC Architecture: Registers

- 32 General Purpose Registers (64 bits wide)
  - Used for address and integer computation
  - Note that some compliant implementations have 32-bit GPRs
- 32 Floating Point Registers (64 bits wide)
  - Used for floating point computation
- 64-bit Count Register (CTR)
  - Used for loop control and indirect branching
- 64-bit Link Register (LR)
  - Used for subroutine call-return and indirect branching
- 32-bit Condition Register (CR)
  - Normally accessed as 8 4-bit condition fields (eq,gt,lt,ov) to compute and branch on conditions
- 32-bit Exception Register (XER)
  - Contains overflow and carry information.
  - Also used for string copy assist instructions
- 32-bit Floating Point Status and Control Register (FPSCR)
  - Contains floating point control information (rounding mode, exception enable)
  - Also contains exception status information

PowerPC Architecture: Instruction Summary

- Branch Processor
  - branch and branch conditional
    - relative and absolute forms (24 bit unconditional, 16 bit conditional)
    - branch and link form (set LR to IAR+4): used for calls
  - branch conditional to LR, branch conditional to CTR
    - used for indirect calls, switches, etc
  - condition register logic and manipulation
- Floating Point Processor
  - load/store
    - single and double forms
    - base-displacement (16-bit), base-index and base-update forms
  - arithmetic (single and double)
    - move, negate, absolute value, negative absolute value
    - add, substract, multiply, divide
    - multiply-add, multiply-subtract, negative multiply-add, negative multiply-subtract
    - convert to single, convert to/from integer
  - compare ordered and unordered
  - FPSCR manipulation

PowerPC Architecture: Instruction Summary *(continued)*

- Fixed Point Processor
  - load/store
    - base-displacement (16-bit), base-index and base-update forms
    - byte, halfword, word and doubleword widths
    - sign extend and zero load forms (except byte)
    - byte reverse forms
    - multiple and string forms
  - load-and-reserve and store-conditional
    - atomic load/store used for test-and-set, compare-and-swap
  - sync, eieio: enforce instruction ordering, flush stores for SMP
  - arithmetic:
    - add, subtract: register and immediate forms, carrying, non-carrying and extended
    - multiply: register and immediate forms, low and high word and doubleword forms
    - divide: word and doubleword, signed and unsigned forms
  - compare: register and immediate, signed and unsigned
  - trap instructions: word and doubleword, register and immediate
  - logic: and, or, xor, eqv, nand, andc, sign-extend, count-leading-zeroes
  - rotate and shift

PowerPC Architecture: Optional Instructions

- PowerPC architecture evolved from the Power and Power 2 architectures
  - Some new instructions added (single precision floating point, 64 bit, synchronization, cache control)
  - Some instructions removed (some string forms, load/store pair)
- Architecture needed to accommodate existing implementations so certain instructions marked as *optional*.
- Optional instructions are implemented on all modern pSeries processors, including RS64, Power 3 and Power 4.
  - Store floating point as integer word: used for float-to-int32 conversion
  - Square root double and single
  - Estimate instructions
    - Float reciprocal estimate single
    - Float reciprocal square root estimate (double)
    - Used to seed inline expansions of divide, square root
  - Floating point select (conditional assignment)

AIX Performance Tools

- Monitoring tools
  - ps: Report on running processes including resource usage
  - topas: General purpose system performance monitor
  - iostat: Disk I/O performance monitor (also includes CPU usage)
  - vmstat: Virtual memory and CPU usage monitor
  - truss: System call and signal monitor
- Tuning tools
  - fdpr: Binary optimizer based on profile feeback
  - schedtune: Tune process and thread scheduling
  - vmtune: Tune virtual memory subsystem and filesystem parameters
- CPU performance tools
  - gprof: Provides a function level application execution profile based on sampling
  - xprofiler: GUI-based profiler similar to gprof - also provides source line execution profile
  - tprof: System or application level sampling profiler - can also monitor kernel activity
  - alstat: Monitor alignment exceptions
  - emstat: Monitor instruction emulation
- More info
  - http://www.redbooks.ibm.com/redbooks/SG246039.html

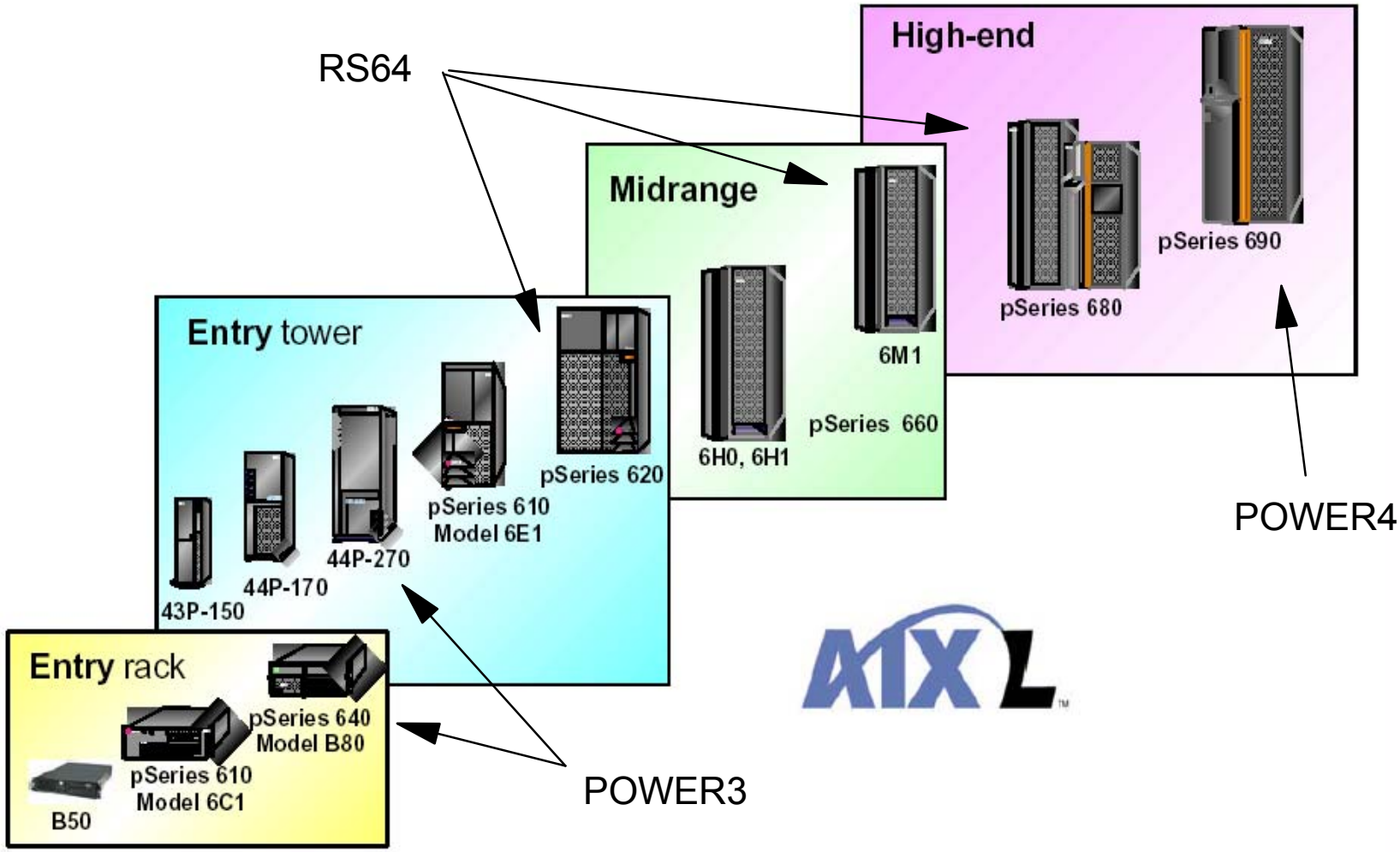
Performance Libraries

- Engineering and Scientific Subroutine Library (ESSL)
  - ESSL has over 400 high-performance subroutines specifically tuned for pSeries
  - Parallel ESSL has over 100 high-performance subroutines designed for SP systems up to 512 nodes
  - BLAS, ScaLAPACK and PBLAS compatibility
  - Linear Algebraic Equations, Eigensystem Analysis, Fourier Transforms, Random Numbers
  - http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/essl.html
- Math Acceleration Subsystem (MASS)
  - High performance versions of a subset of Fortran intrinsic functions (also callable from C/C++)
  - Sacrifices a small amount of accuracy for increased speed
  - Scalar and vector versions available
  - http://techsupport.services.ibm.com/server/mass?fetch=home.html
- Modular I/O (MIO) Library
  - Application-level analysis and tuning of sequential I/O
  - http://www.research.ibm.com/actc/Opt_Lib/mio/mio_doc.htm

General Performance Coding Guidelines

- Minimize stride of data access
  - Reformat multidimensional data
  - Reorder loop nests
  - Copy strided data where there is reuse
  - Try to limit the step of indirect addressing (eg. sort indices)
- Make optimal use of data prefetch facilities
  - Split loops when too many streams are being referenced
  - Use CACHE_ZERO for store streams
- Avoid cache and TLB set associativity conflicts
  - Realign or copy data to avoid referencing data whose addresses differ by a large power of 2
- Data cache blocking
  - To exploit reuse, block computations where possible to ensure that referenced data resides in the largest on-chip cache (L1 on POWER3, L2 on POWER4)
- Unroll loops
  - Creates larger opportunity for code scheduling
  - Can balance computation for superscalar execution, increasing computational intensity
- Keep code as simple as possible

eServer pSeries at a glance



IBM

RS64 (Pulsar, SStar) Microarchitecture

- Systems: pSeries 620 Model 6F1 (up to 4w), pSeries 660 Model 6M1 (8w), pSeries 680 (up to 24w)
- 4-way superscalar processor
  - one load/store unit (LSU), one floating point unit (FPU), two fixed point units (FXUs)
  - 5-stage pipeline
  - simple branch prediction with very fast branch mispredict recovery (often free)
  - in-order execution, 2-way multithreaded (SStar)
- On-chip 128K 2w L1 instruction and data caches
- On-chip L2 cache control with support for up to 8MB (16MB) of off-chip L2 cache
- RS64 III
  - 34 million transistors, 140 mm^2 die
  - 1.8V 0.22 micron copper CMOS 7S process
  - 14.4 GB/s L2 bandwidth
- RS64 IV
  - 44 million transistors, 128 mm^2 die
  - 1.6V 0.18 micron copper SOI CMOS 8S process
  - 19.2 GB/s L2 bandwidth
- More info
  - http://www.research.ibm.com/journal/rd/446/borkenhagen.html

RS64 IV (SStar) Overview

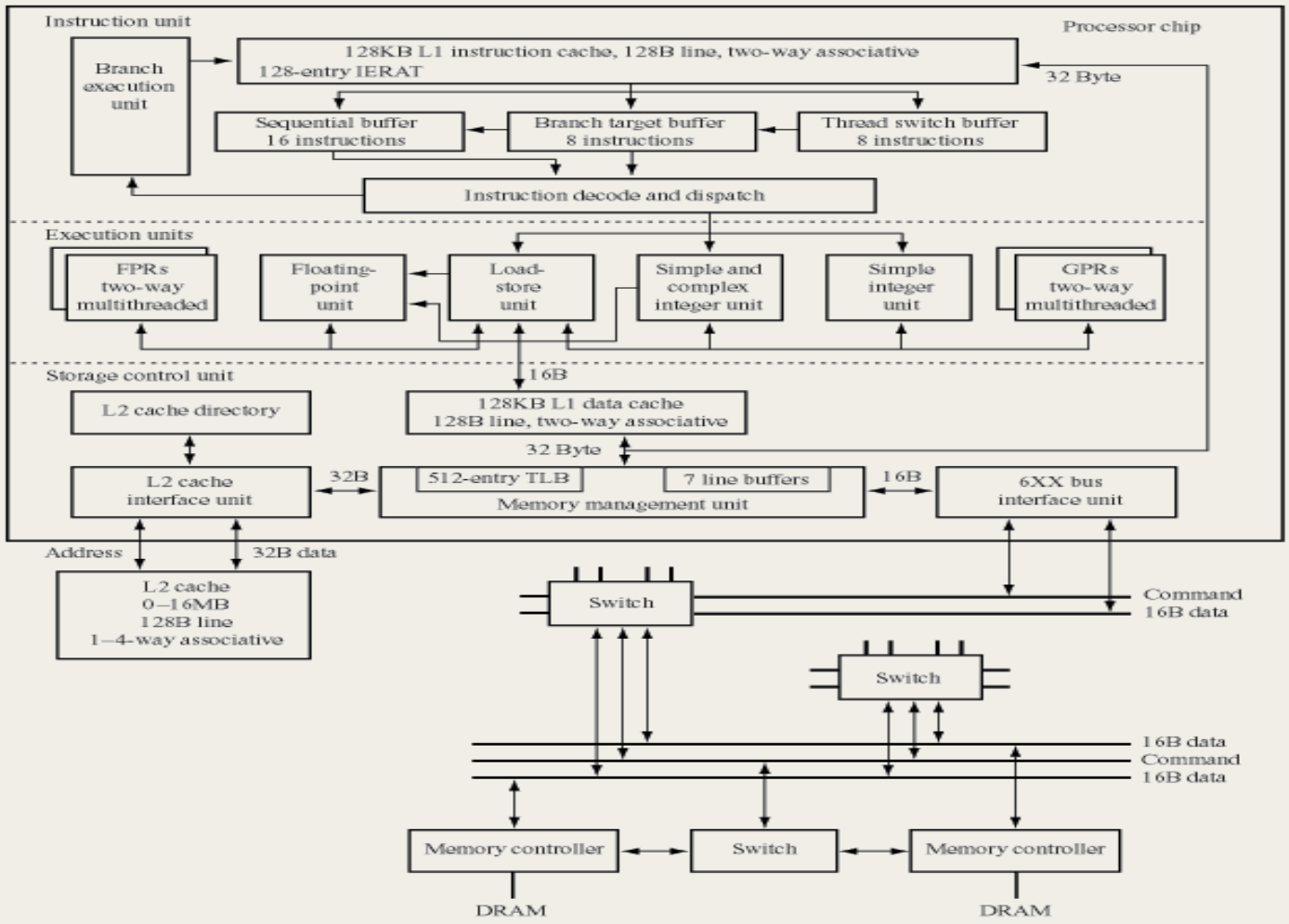


Figure 2

Processor and memory system.

POWER3 Microarchitecture

- Example systems with POWER3 processor
  - pSeries 640 Model B80 (4w), SP High Node (up to 16w)
- 8-way superscalar processor
  - 2 LSUs, 2 FPUs, 3 FXUs, 1 BRU
  - In-order dispatch and completion, out-of-order execution
- On-chip 64K 128w L1 data cache, 32K 128w L1 instruction cache
- Up to 8MB off-chip L2 cache
- 128 entry 2w TLB
- Automatic instruction and data prefetch (up to 4 streams)
- Power 3
  - 15 million transistors, 270 mm^2 die
  - 2.5V 0.25 micron CMOS 6S2 process
  - 6.4 GB/s L2 bandwidth
- Power 3-II
  - 23 million transistors, 163 mm^2 die
  - 0.22 micron copper CMOS 7S process
  - 6.4 GB/s L2 bandwidth
- More info
  - http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power3wp.html
  - http://www.redbooks.ibm.com/redbooks/SG245155.html

Power 3-II Overview

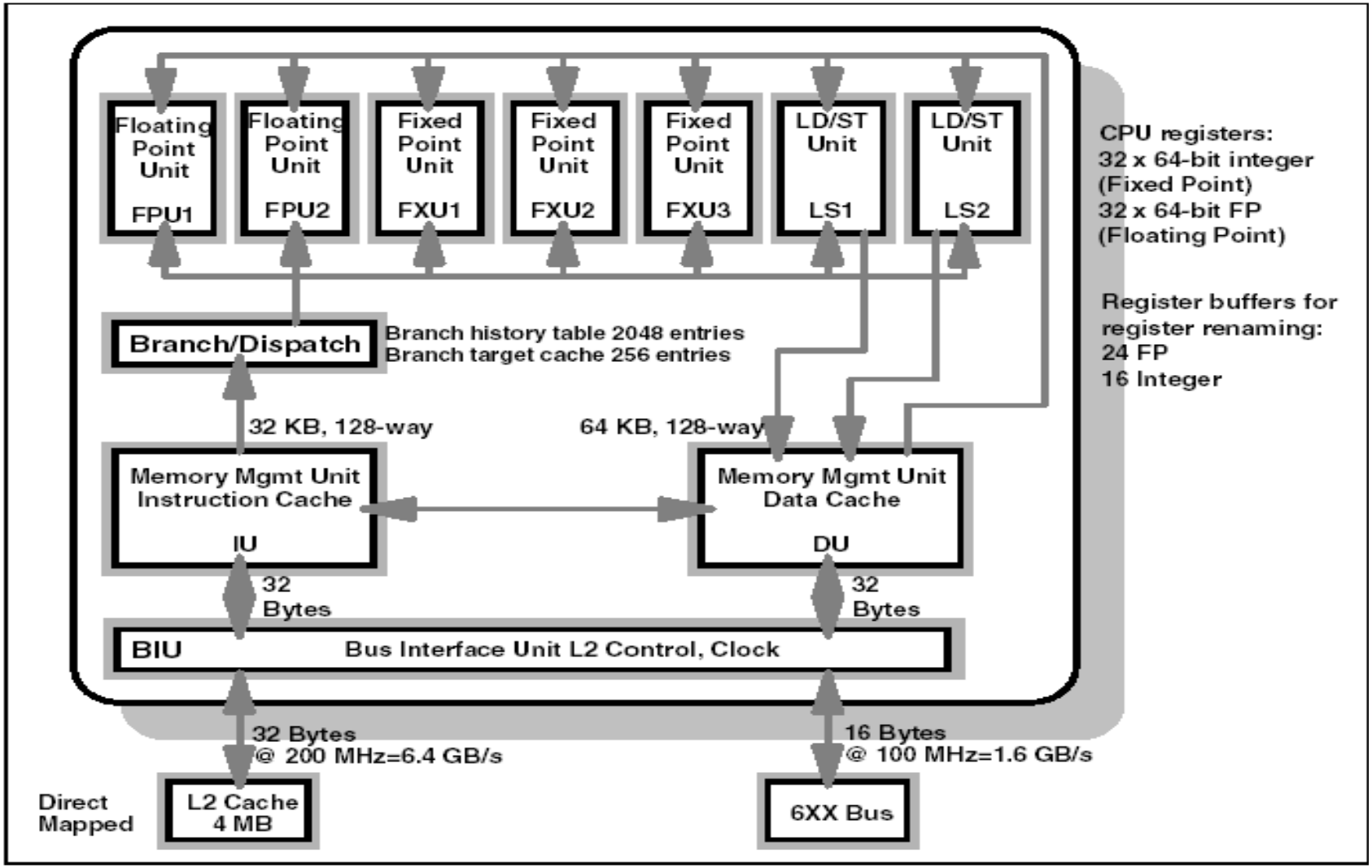
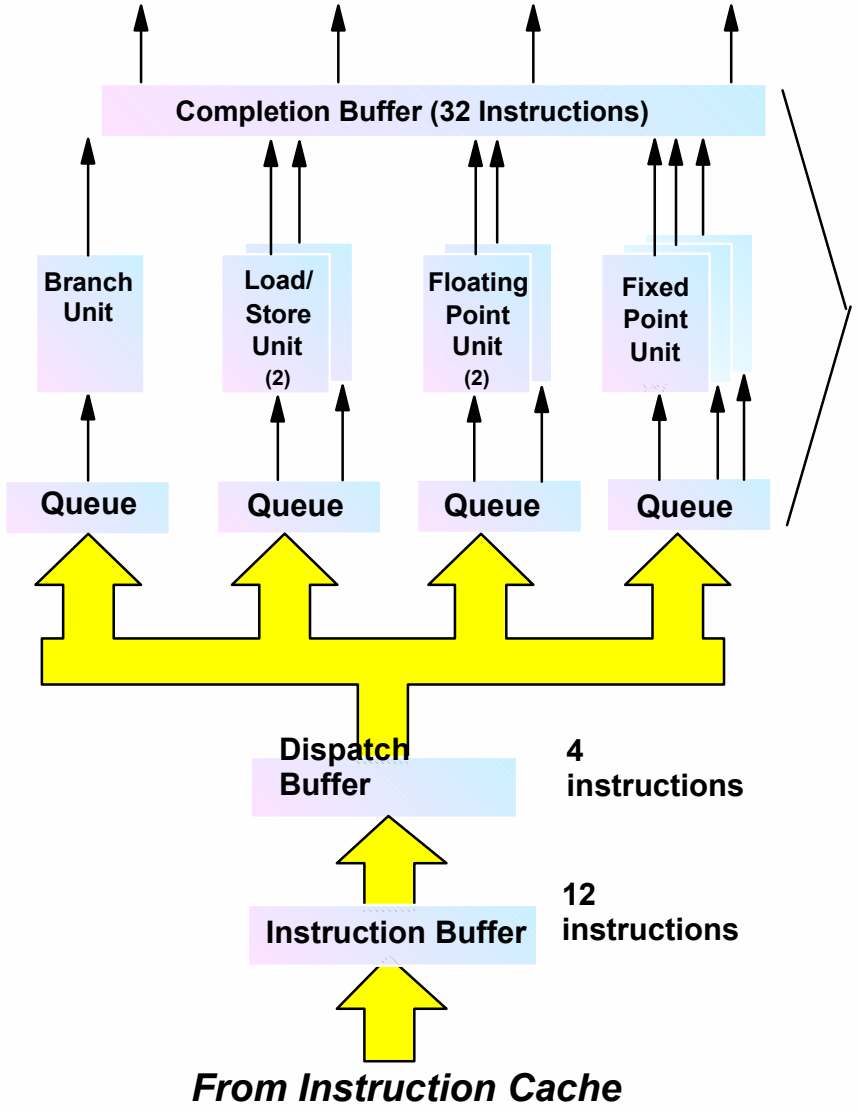


Figure 3. Model 170 - POWER3-II 400 MHz Block Diagram

Execution Core

- Three fixed point units
  - Two units implement single cycle operations
  - One unit for complex, multi-cycle instructions
- Two floating point units
  - Double precision data path
  - Three cycle latency, one cycle throughput
  - Each unit contains divide and square root sub-units
  - 24 real and 32 virtual rename buffers
- Two load / store units
  - Each unit calculates one load or store / cycle
  - Loads processed speculatively
  - 16 entry store queue
- Branch unit
  - 2048 entry branch history table
  - 128 x 2 entry branch target cache
  - Four pending predicted branches

Decode-to-completion bandwidth

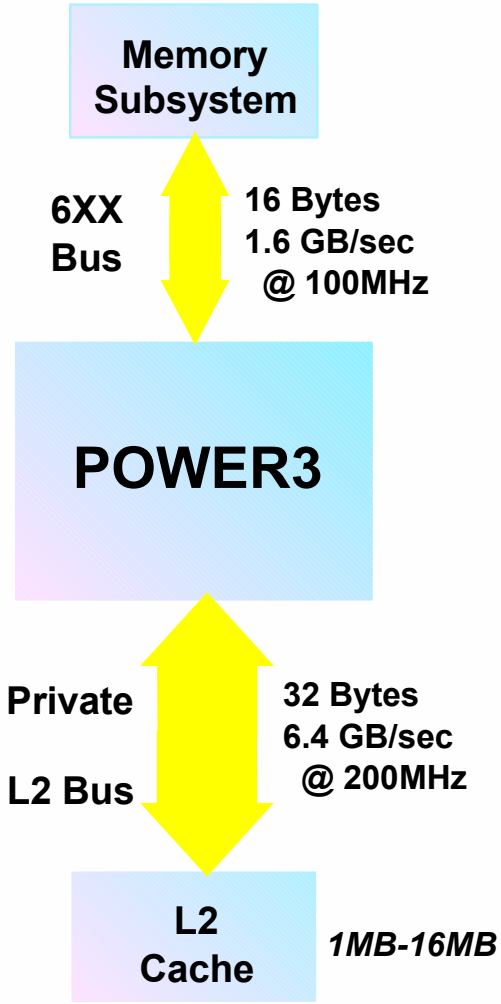


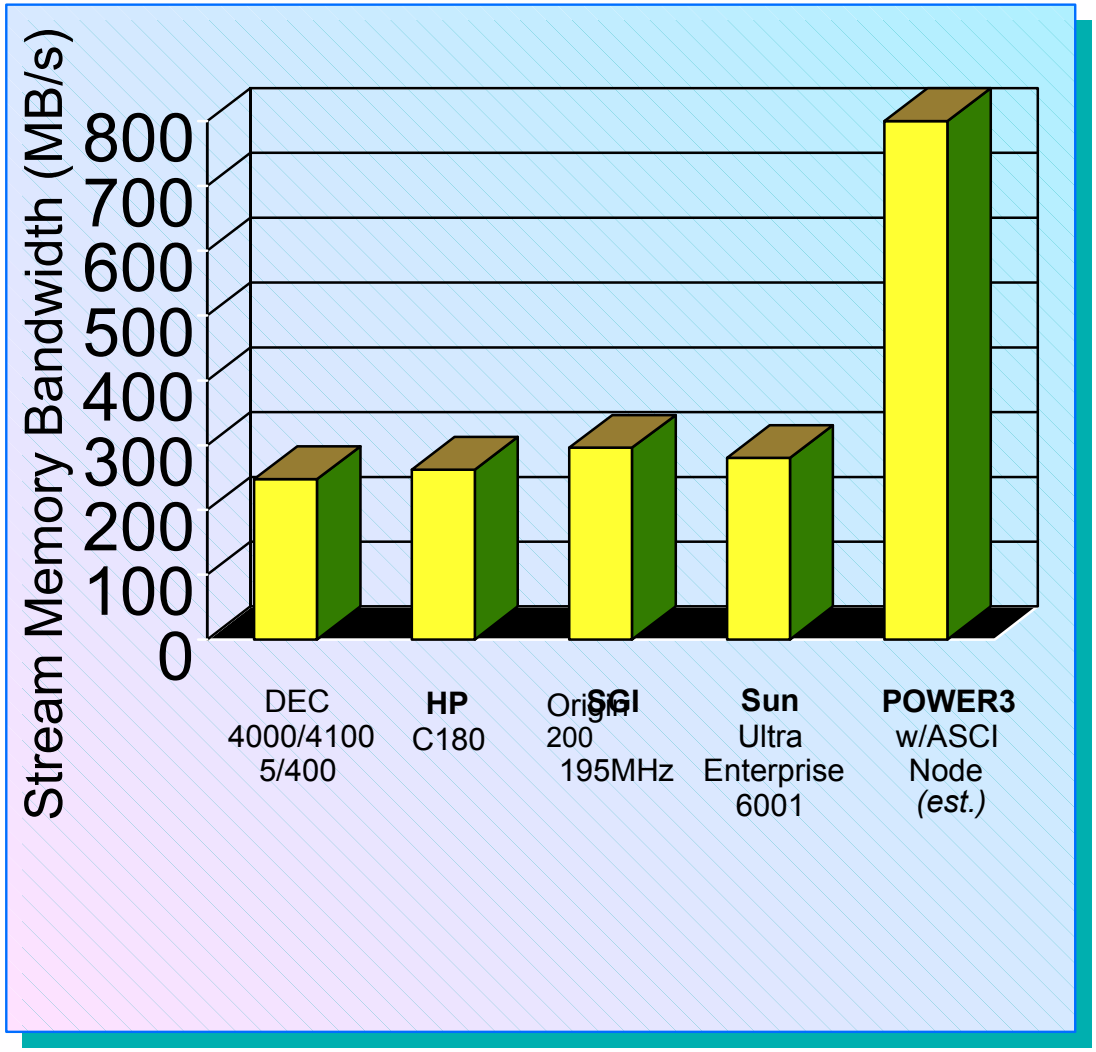
Four instructions per cycle completed (in order)

Eight Instruction execution (out of order)

Four instruction per cycle dispatch (in order)

POWER3 System Level Bandwidth



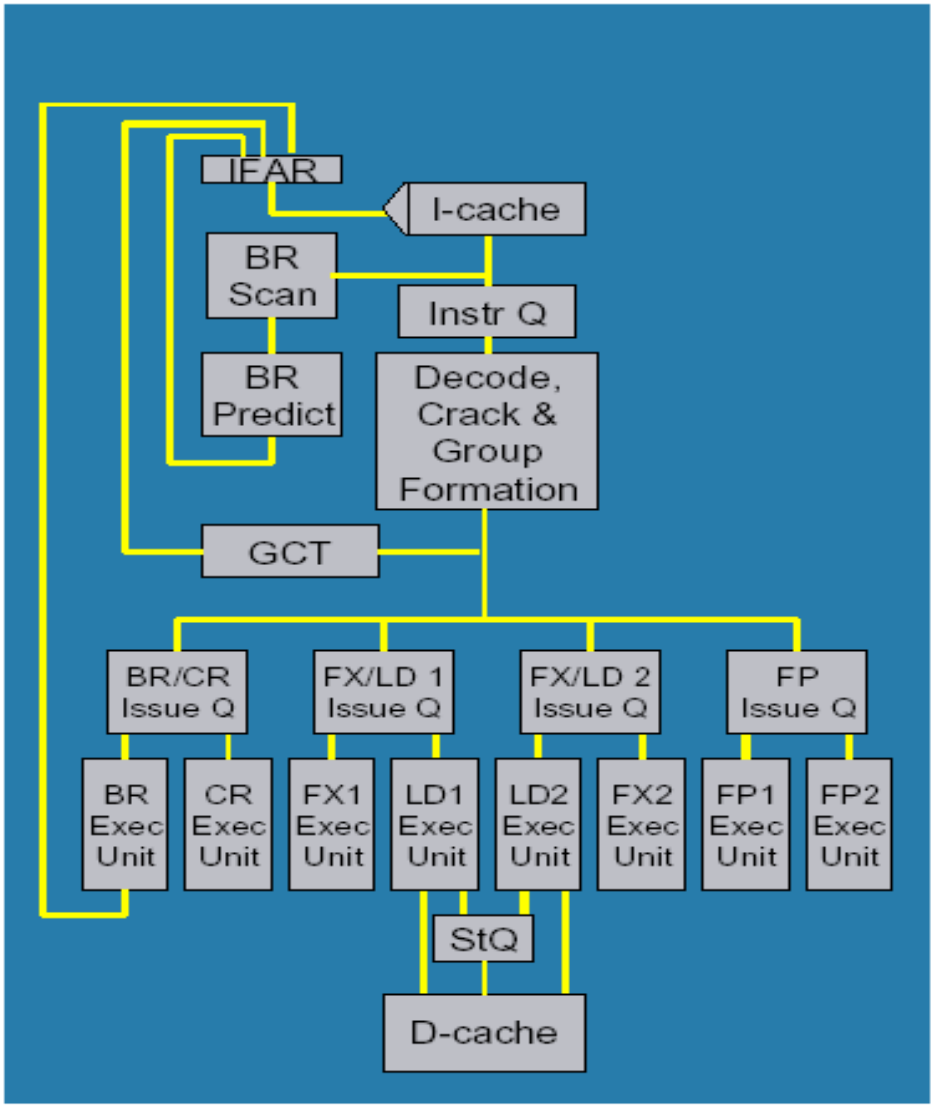


Performance data as of 9/97, as reported in corporate websites and other public sources; except IBM data, which is estimated.

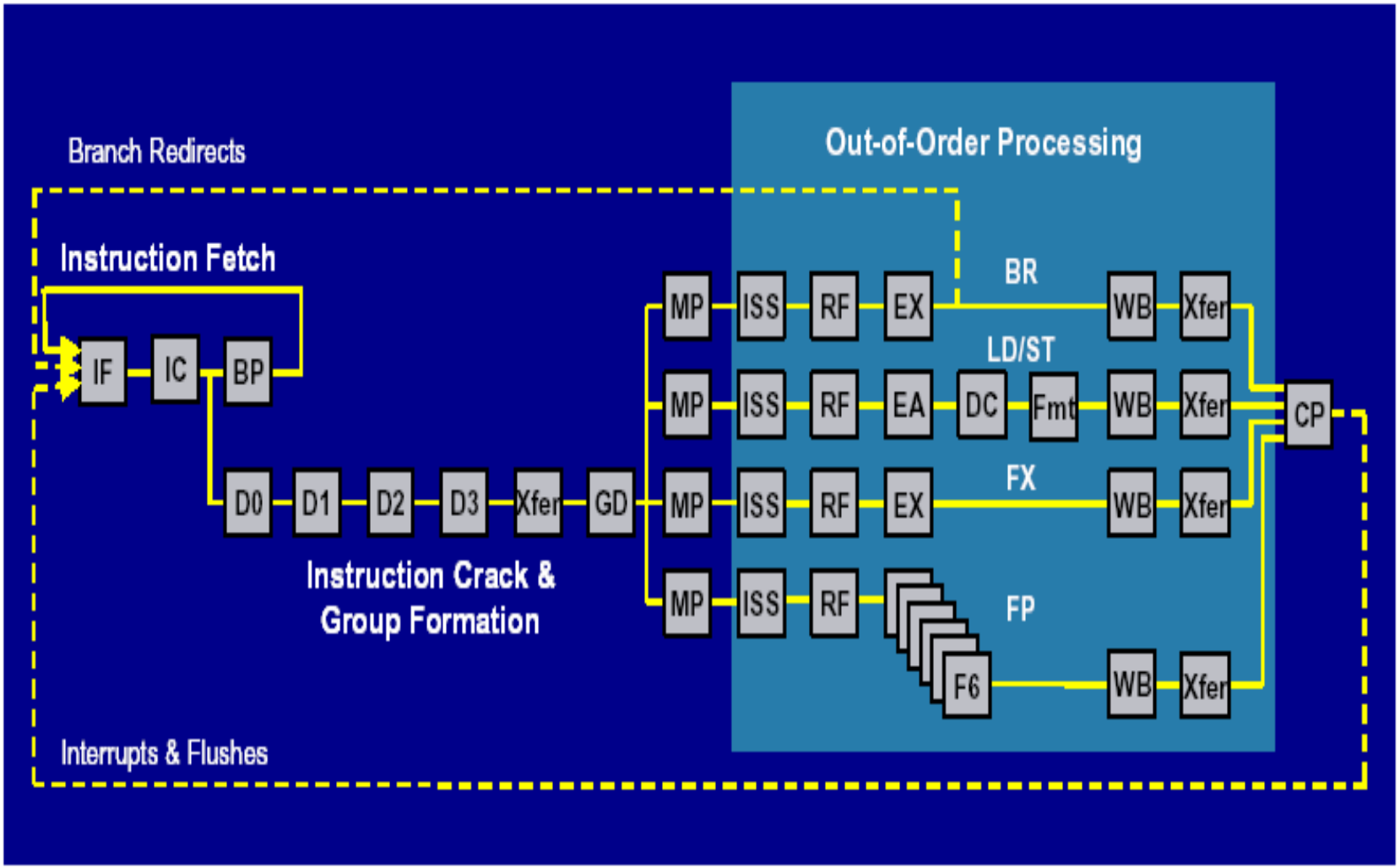
POWER4 Microarchitecture

- Example systems with POWER4 processor
  - pSeries 690 (up to 32w)
- 8-way superscalar processor
  - 2 LSUs, 2 FPUs, 2 FXUs, 1 branch unit (BRU), 1 condition register logic unit (CRLU)
  - 15 to 20 stage pipelines
  - Dynamic branch address and direction prediction
  - Out-of-order execution of instruction groups
- 2 cores per chip (1 core in HPC models)
- On-core 64K direct-mapped L1 instruction cache, 32K 2w L1 data cache
- On-chip 1.5MB 8w L2 cache (shared by two cores)
- On-chip L3 cache control supporting off-chip 32MB 8w L3 cache
- Support for 4KB and 16MB page sizes
- Automatic instruction and data prefetch (up to 8 streams, delayed ramp-up)
- POWER4
  - 170 million transistors, 400 mm^2 die
  - 1.6V 0.18 micron copper SOI CMOS 8S process
  - 100 GB/s L2-L1 bandwidth, 35 GB/s L2-L2 bandwidth, 10GB/s L3-L2 bandwidth
- More info
  - http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html
  - http://www.redbooks.ibm.com/redbooks/SG247041.html

POWER4 Core

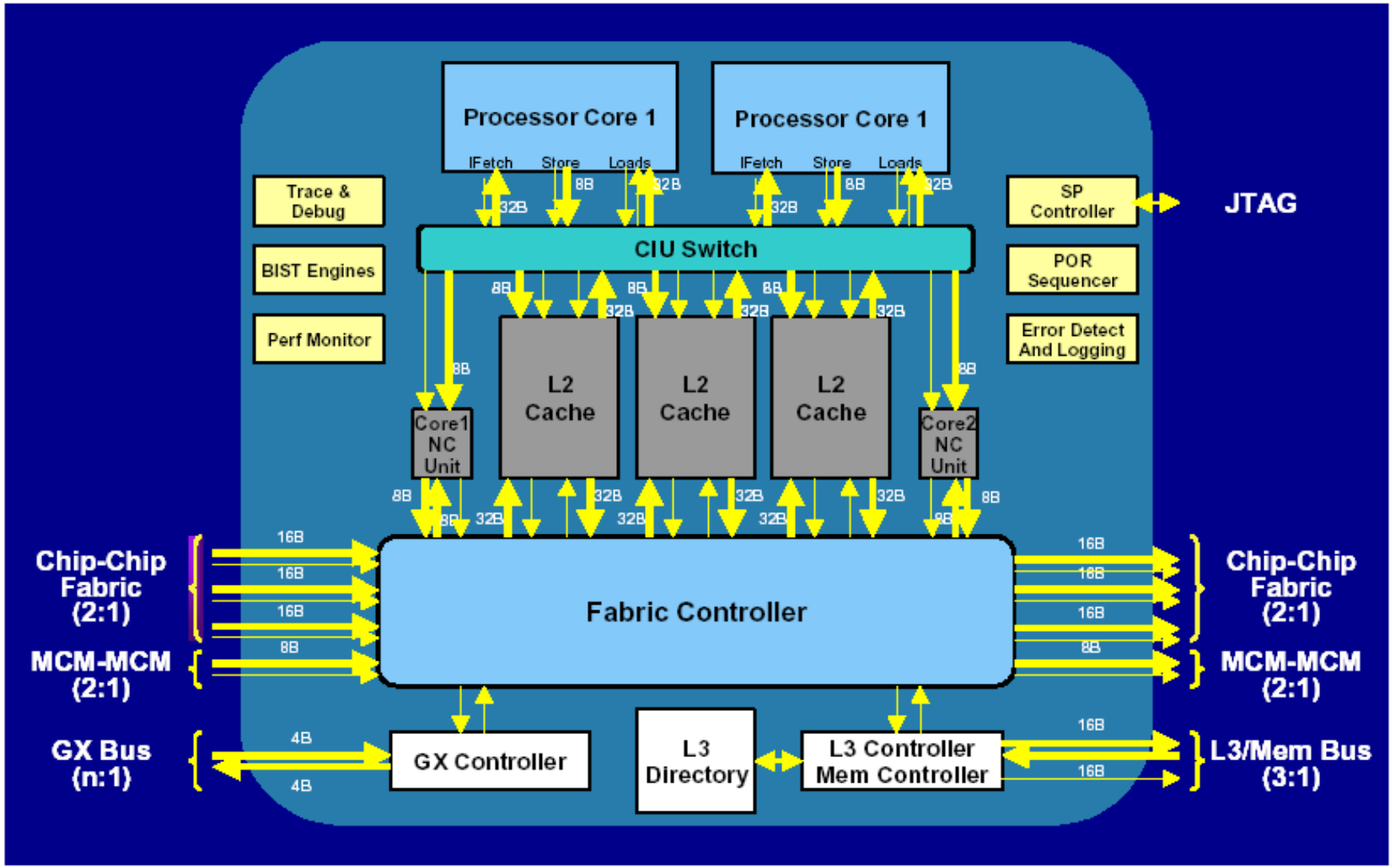


POWER4 Pipeline

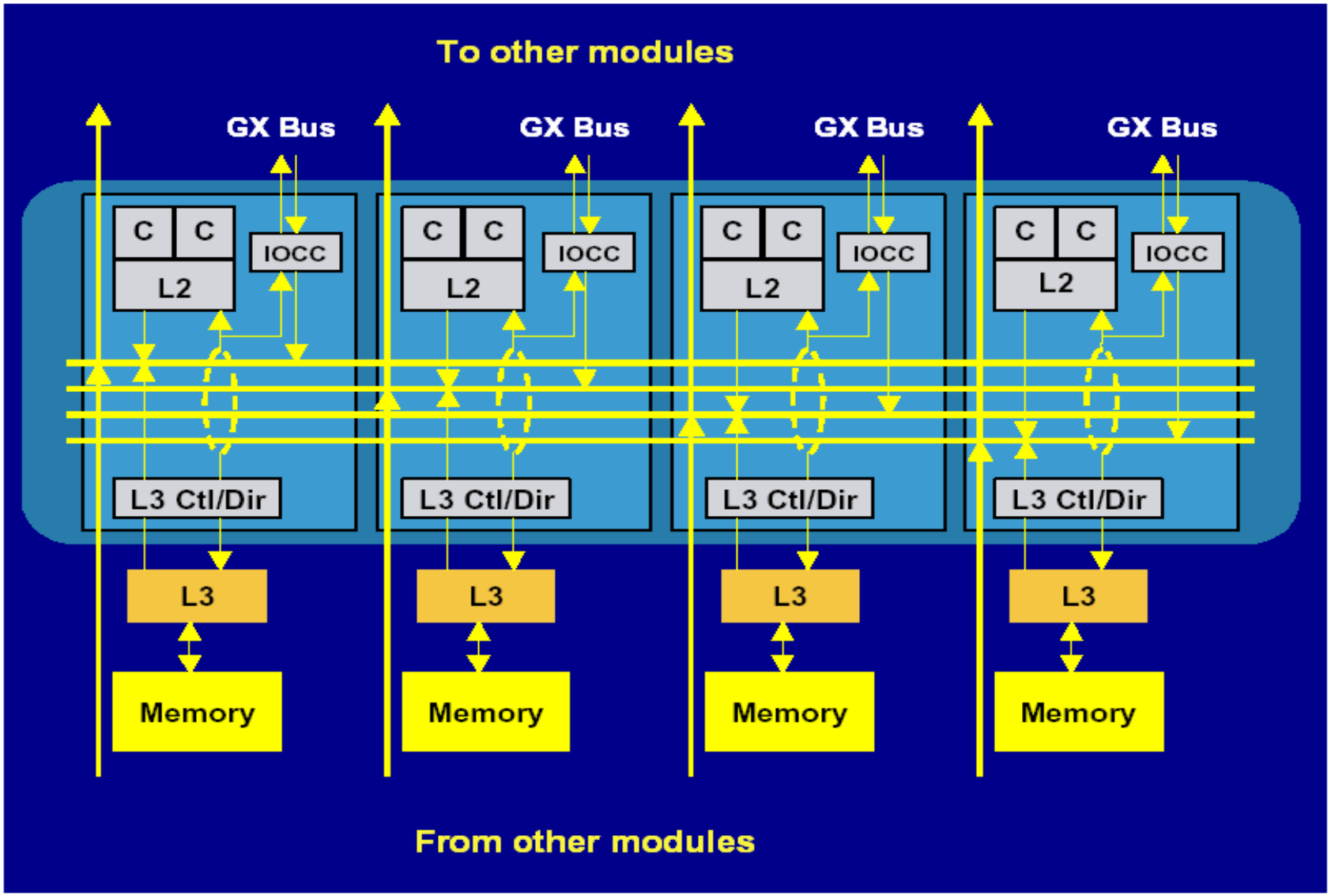


IBM

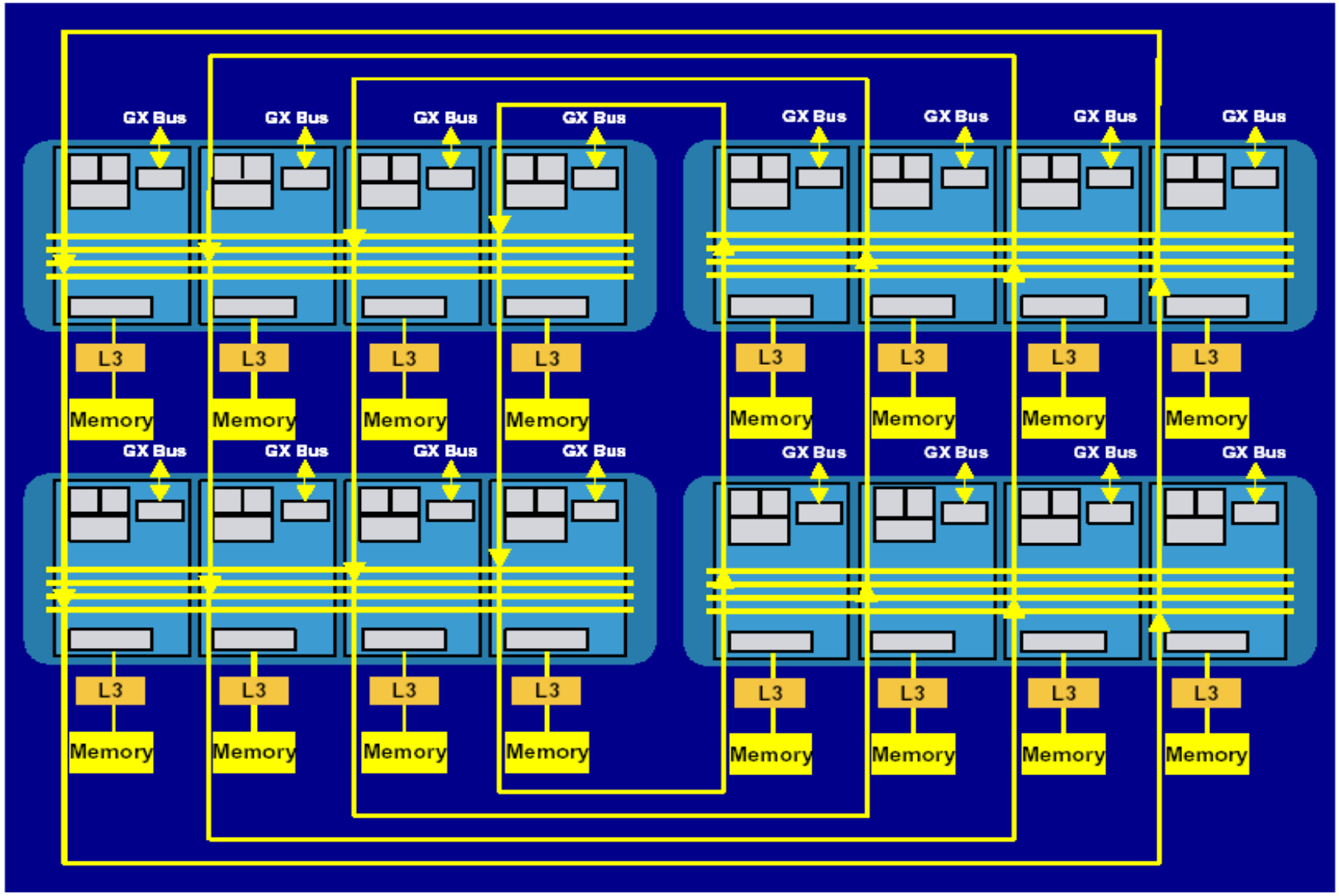
POWER4 Chip


Processor Core 1
Processor Core 1
IFetch
Store
Loads
Trace & Debug
BIST Engines
Perf Monitor
CIU Switch
SP Controller
JTAG
POR Sequencer
Error Detect And Logging
L2 Cache
L2 Cache
L2 Cache
Core1 NC Unit
Core2 NC Unit
Chip-Chip Fabric (2:1)
MCM-MCM (2:1)
GX Bus (n:1)
Fabric Controller
GX Controller
L3 Directory
L3 Controller Mem Controller
L3/Mem Bus (3:1)

POWER4 Multi-Chip Module (MCM)



POWER4 MCM Interconnect



Key POWER4 System Attributes

- Memory System
  - L3 is interleaved across MCM on 512 Byte boundaries
  - Memory is interleaved across MCM on 512 Byte boundaries
  - 4-way interleaving requires matched memory cards on the MCM
  - Peak BW is 27.7 GB/s per direction per MCM (measured at L3)
  - Peak DRAM BW is 25.6 GB/s * 2 ports per MCM (measured at DDR SDRAMs)
- CPU Performance
  - 1100 MHz or 1300 MHz available
  - Out of order
    - max of ~200 instructions in flight
  - Superscalar
    - up to 5 instructions per cycle
    - two floating-point multiply-add units
  - Aggressive hybrid branch prediction
  - Max sustainable FP performance about 75% of peak: L2-contained DGEMM, with compiler-generated code (due to limited rename resources)
  - L1 Dcache
    - 2-way associative
    - FIFO replacement policy
    - write-through

Unique Features of POWER4 & p690

- Shared L2 cache
- Shared L3 cache
- Interleaved memory
- Hardware Prefetch
- Multiple Page Size support

Shared L2 cache

- The L2 cache has plenty of bandwidth for two cores
- Sharing the L2 increases the L2 miss rate
  - fixed capacity
  - fixed associativity
- What to do?
  - select blocking factors for about 512 kB blocks per process
  - can block to approximately 1.4MB per process on HPC models
  - fiddle with offsets to minimize conflict misses

Shared L3 cache

- The L3 cache is a front-side cache
- L3 reduces latency for non-prefetchable accesses
  - good for gather/scatter kernels
  - good for store-dominated kernels
- L3 BW roughly matches memory BW for prefetchable kernels
- Fixed Associativity: 8-way
- What to do?
  - pay careful attention to offsets and conflicts

Interleaved Memory

- Memory is interleaved 4 way within MCM
  - only if 2 memory cards match in size
- Each MCM is associated with a contiguous quarter of the total address range
  - each page is interleaved within an MCM
  - consecutive pages can be on any MCM

Multiple Page Sizes

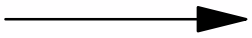
- AIX currently provides random page placement for small pages
  - local page placement AIX 5.1D
  - "first touch" policy
- Large pages are currently accessible through a System V shared memory segment interface
  - 256 MB contiguous segments
- Local placement of large pages AIX5.1D
  - same policy as for small pages
  - no page-to-page contiguity forced
- Fixed pool of pinned large pages (boot option)

Hardware Prefetch

- POWER4 has hardware prefetch on sequences of load misses
  - ascending or descending
  - 8 streams
  - 12 stream filter
- Ramped Initialization
  - L2 to L1 prefetches
  - L3 to L2 prefetches
  - Memory to L3 prefetches
- New dcbt variant starts a stream immediately
- Prefetch helps small page performance
- Prefetch works better with large pages
  - amortize slow startup

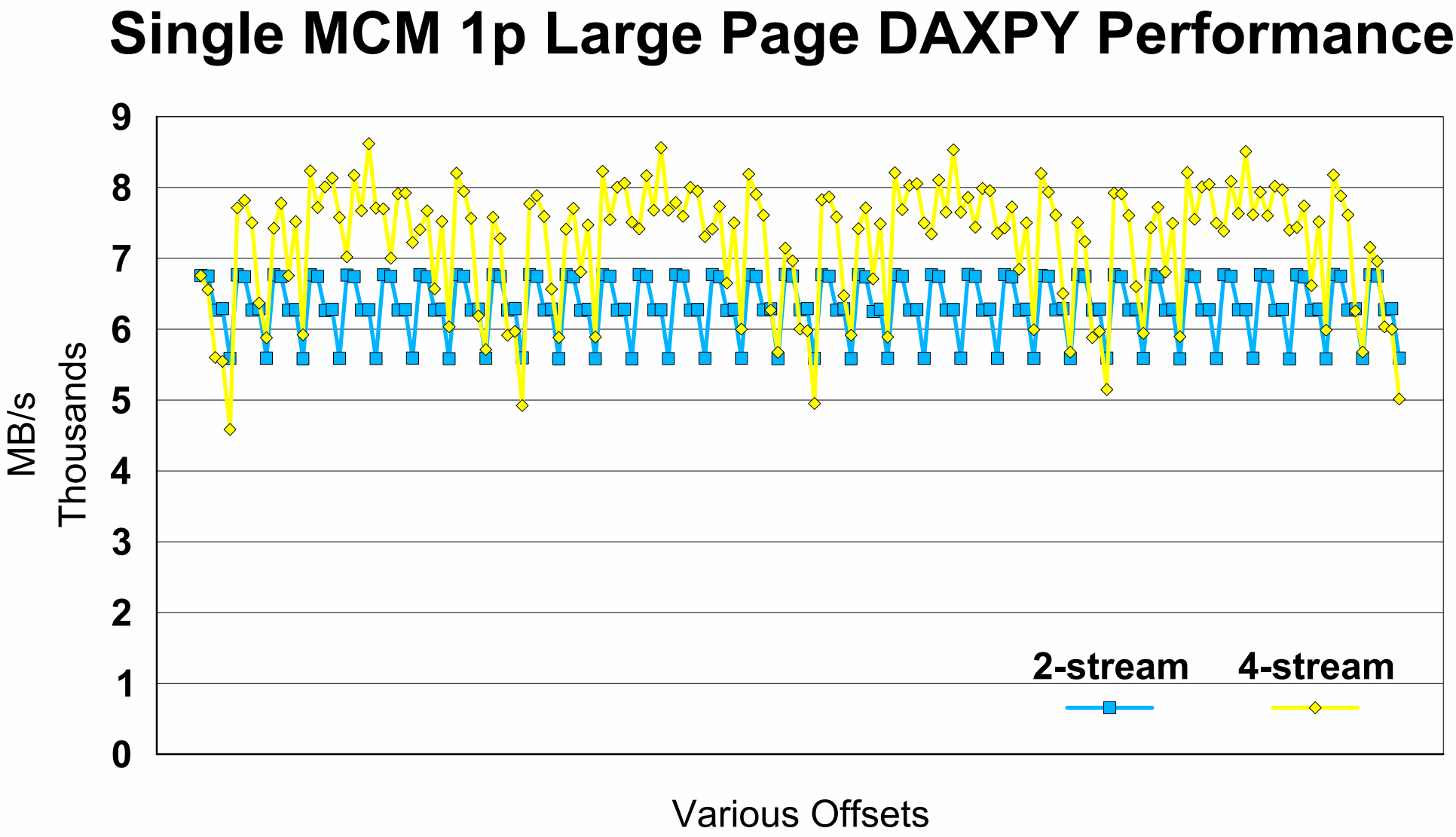
Coding for Hardware Prefetch: Bisection

```
* single stream example
sum = 0.0
do i=1,N
    sum = sum + a(i)
end do
```



```
* bisect to increase number
* of streams
sum1 = 0.0
sum2 = 0.0
do i=1,N/2
    sum1 = sum1 + a(i)
    sum2 = sum2 + a(i+N/2)
end do
sum = sum1 + sum2
```

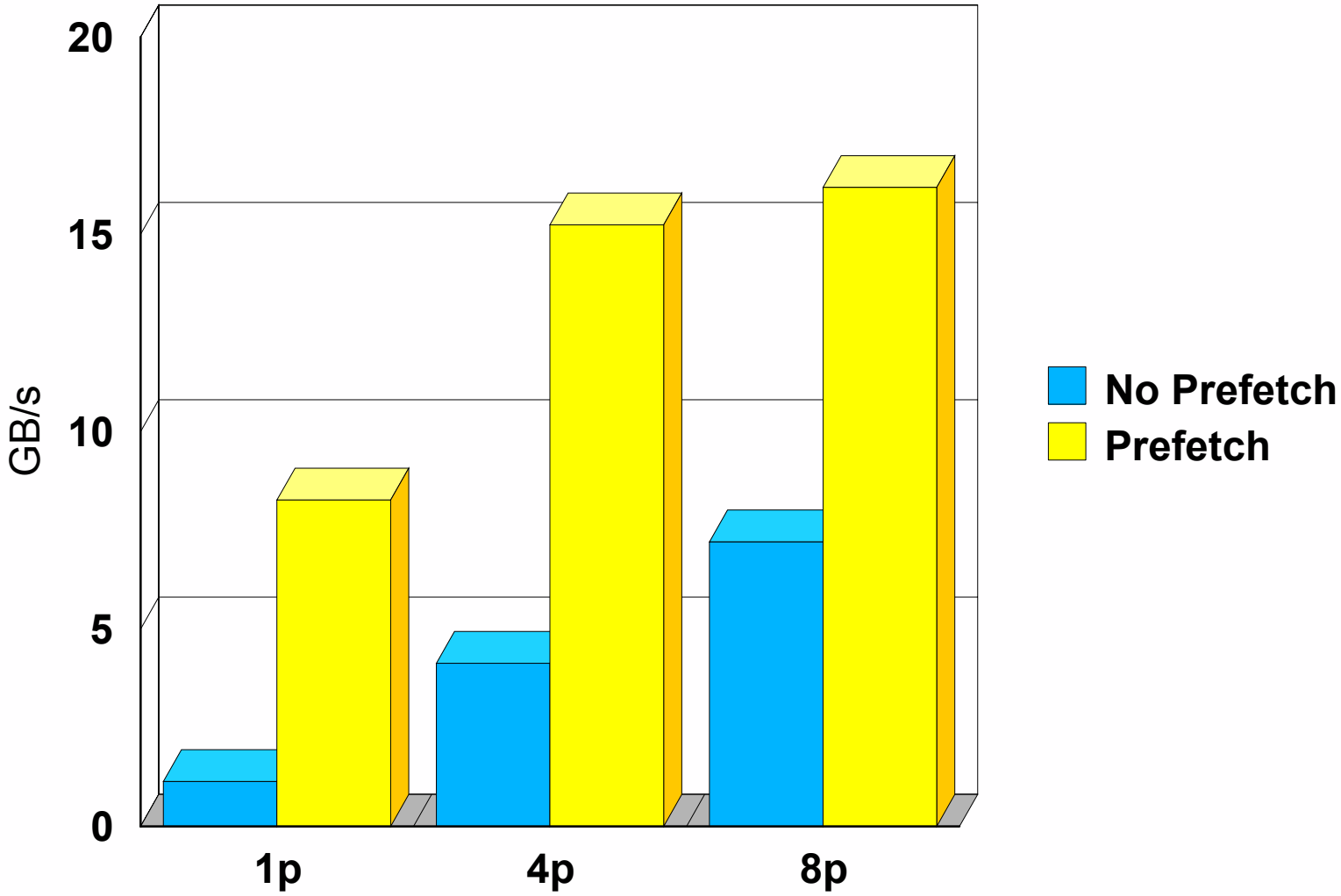
Bisecting uniprocessor DAXPY



Coding for Prefetch: Fission

- Hardware can prefetch up to 8 streams
- Hardware can monitor up to 12 streams
- If there is no re-use, splitting loops to reduce number of streams to 8 or less can improve throughput
- If there is any re-use, don't do this!

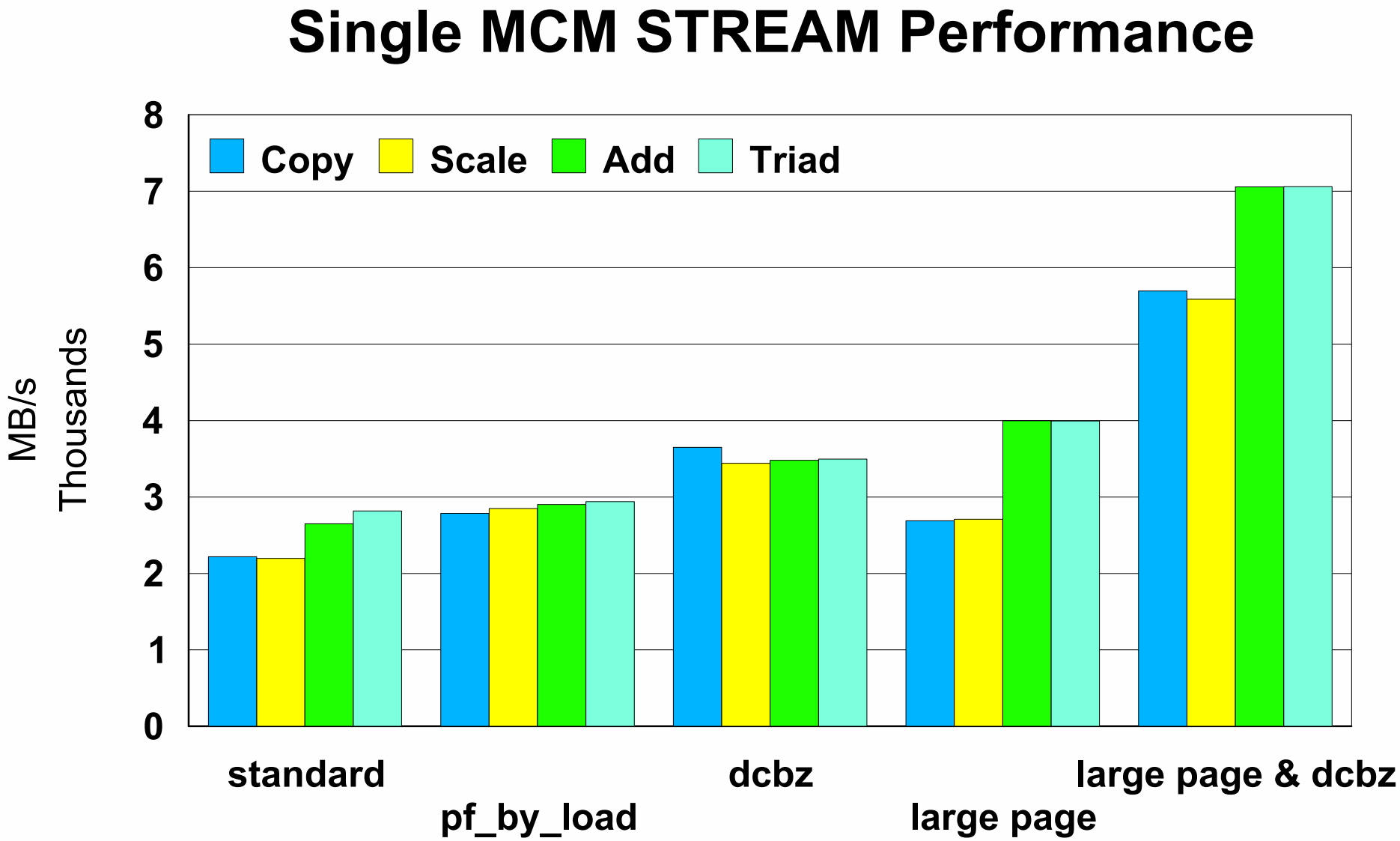
Effect of Hardware Data Prefetch on Large Page DAXPY Bandwidth



Coding for Prefetch: STREAM

- The POWER4 hardware prefetch engine prefetches load-miss streams, not store-miss streams
- Store miss performance can be boosted in several ways
  - load the data before storing
    - not great since the L1 cache is write-through
  - data cache block touch for store (dcbtst)
    - also pollutes L1 cache (quirk of implementation)
  - data cache block zero (dcbz)
    - avoids store miss from L2 entirely
    - seems more effective with large pages

STREAM Benchmark Tuning



IBM

Key POWER4 Features from a Compiler Perspective

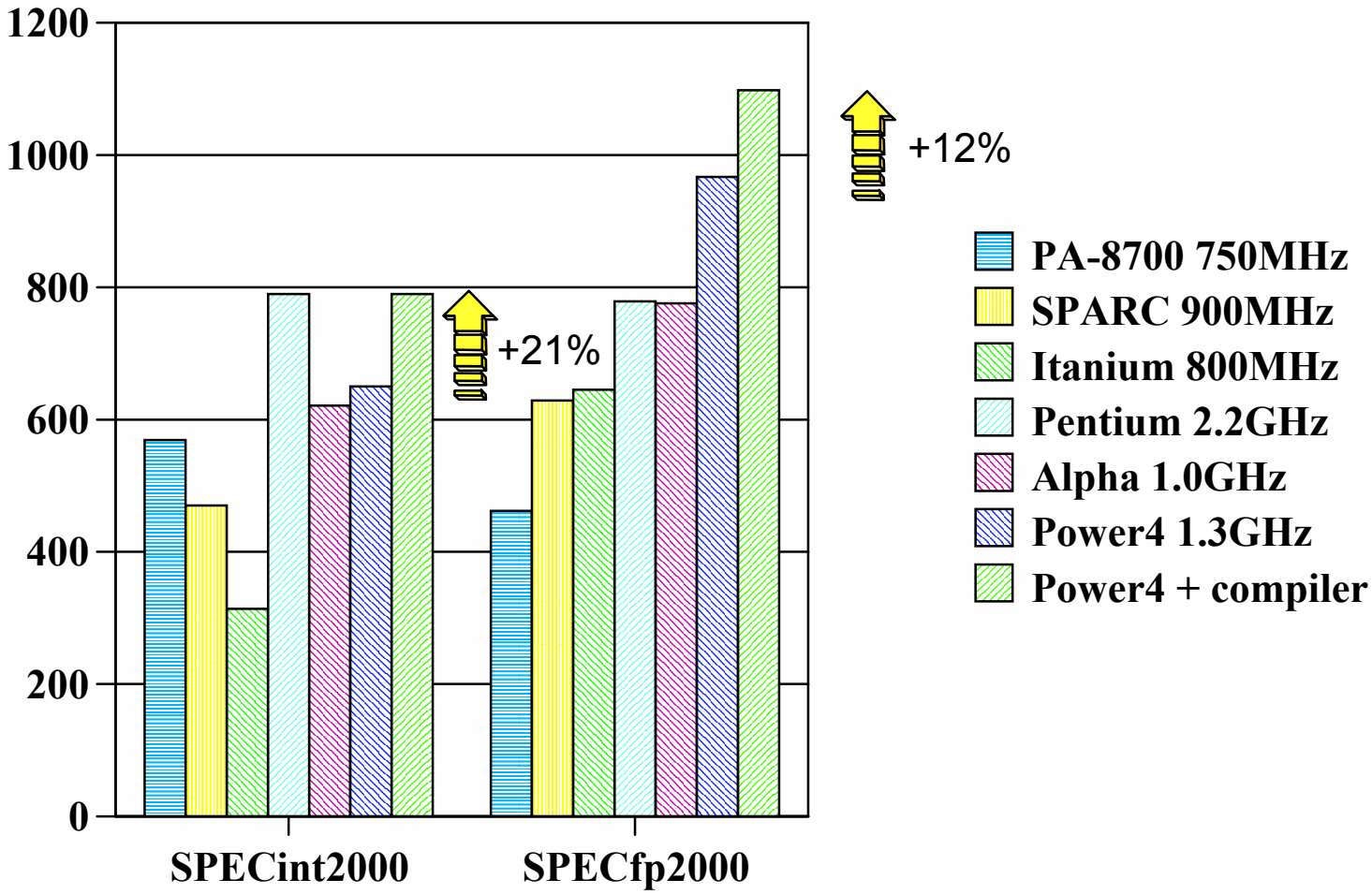
- Very long floating point pipelines, no bypasses
- Aggressive out-of-order execution
- Deep instruction fetch and decode pipeline - microcoding and cracking
- 16M page support
- 8 prefetch buffers, stream touch instructions
- Store-through L1 D-cache
- FIFO L1 D-cache
- L2 cache shared between two cores
- Excellent dynamic branch prediction, new branch annotations
- Dedicated execution unit for CR logic

Compiler Improvements for POWER4

- Instruction scheduling for dispatch
- Register constrained modulo scheduling of single block inner loops
- Avoid microcoded and some cracked instructions
- Generate stream touch instructions
- Optimize store streams with dcbz
- Allocate boolean variables to condition register bits
- Optimize dependent conditional branches using CR logic
- Eliminate small branch sequences using CA bit
- Tune loop optimization for 8 prefetch buffers
- 4 word procedure and loop alignment
- Use static branch prediction override with PDF
- Inline glue and use CTR cache for pointer calls
- Bias CR allocation to get same source/target for CR logic

* Note: Not all optimizations included in current compiler release

SPEC CPU2000 Results: Regatta vs. Competition



* Measurements not done using official SPEC run rules

Questions & Answers

IBM Compiler Products for pSeries

- **Latest versions**
  - C for AIX, Version 5.0.2
  - VisualAge C++ for AIX, Version 5.0.2
  - VisualAge C++ for AIX, Version 6.0 beta
  - XL Fortran for AIX, Version 7.1.1
- **Older, supported versions**
  - XL Fortran for AIX, Version 7.1.0
  - VisualAge C++ Professional for AIX, Version 4.0 (until 12/02)

XL Fortran version 7.1.1

- Fortran 77/90/95 compiler with many extensions
- 32 and 64 bit support for serial and SMP
- OpenMP 1.0 support (OpenMP 2.0 coming ...)
- Support for TotalView, xldb, IBM distributed debugger and dbx/pdbx
- Snapshot directive for debugging optimized code
- Portfolio of optimizing transformations
  - Comprehensive path length reduction
  - Whole program analysis
  - Loop optimization for parallelism, locality and instruction scheduling
  - Tuned support for all pSeries processors
- More info: www.software.ibm.com/ad/fortran

C for AIX version 5.0.2

- ANSI C89 compliant compiler (C99 coming soon)
- 32 and 64 bit support for serial and SMP
- Full support for OpenMP 1.0 (participating in OpenMP 2.0 definition)
- Support for TotalView, xldb, IBM distributed debugger and dbx/pdbx
- Snapshot directive for debugging optimized code
- Runtime memory debug support
- Portfolio of optimizing transformations
  - Similar to Fortran support but includes tuned optimizations for C pointers and systems coding styles
- More info: www.software.ibm.com/ad/caix

VisualAge for C++ for AIX version 5.0.2

- Fully compliant ANSI98 C++ compiler
- 32 and 64 bit support
- Batch compiler for traditional build environments and maximal optimization
- Integrated graphical development environment including remote debug and performance visualization
- Support for TotalView, xldb, IBM distributed debugger and dbx/pdbx
- Portfolio of optimizing transformations
  - Subset of transformations available in Fortran and C but has tuned support for all processors
  - Much more coming soon
- More info: www.software.ibm.com/ad/vacpp

VisualAge C++ for AIX 6.0 beta

- Enhanced optimization for Power4 (-qarch=pwr4, -qtune=pwr4)
- Support for -qhot option (loop optimizations)
- Support for -qipa (interprocedural analysis), -O4 and -O5 options (C++)
- More info (and download): http://www.ibm.com/software/ad/vacpp/news/v6beta.html

Tutorial on Performance Controls

- Compiler options
  - Optimization level
  - High order transformations
  - Interprocedural analysis
  - Profile directed feedback
  - Target machine specification
  - Floating point options
  - Program behaviour
  - Diagnostic options
- Directives and pragmas
  - Assertive
  - Prescriptive

Optimization Levels

Fast compile
Full debug support

Low level optimization
Partial debug support

More extensive optimization
Some precision tradeoffs

Interprocedural optimization
Loop optimization
Automatic machine tuning

Example: Simple Matrix Multiply

```
DO I = 1, N1
  DO J = 1, N3
    DO K = 1, N2
      C(I,J) = C(I,J) + A(K,I) * B(J,K)
    END DO
  END DO
END DO
```

Matrix Multiply inner loop code with -qnoopt

38 instructions, 31.4 cycles per iteration

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Matrix Multiply inner loop code with -qnoopt

Necessary instructions

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Matrix Multiply inner loop code with -qnoopt

Necessary instructions **Loop control**

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Matrix Multiply inner loop code with -qnoopt

Necessary instructions Loop control Address computation

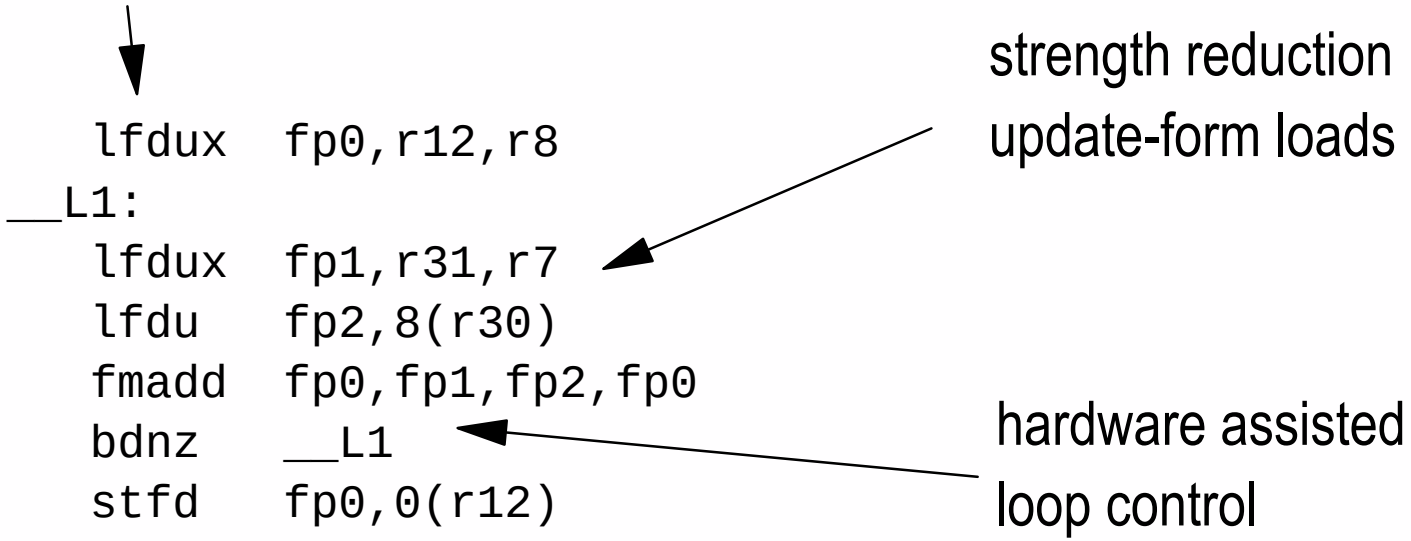
```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Optimization Level -O2 (same as -O)

- **Comprehensive low-level optimization**
  - Global assignment of user variables to registers
  - Strength reduction and effective usage of addressing modes
  - Elimination of unused or redundant code
  - Movement of invariant code out of loops
  - Scheduling of instructions for the target machine
  - Some loop unrolling and pipelining
- **Partial support for debugging**
  - Externals and parameter registers visible at procedure boundaries
  - Snapshot pragma/directive creates additional program points for storage visibility
  - -qkeepparm option forces parameters to memory on entry so that they can be visible in a stack trace

Matrix Multiply inner loop code with -O2

load/store of "C"
moved out of loop



3 instructions, 3.1 cycles per iteration

Matrix Multiply inner loop code with -O2 -qtune=pwr3

```
    lfdux   fp2,r31,r7
    lfdu    fp1,8(r30)
    bdz     __L2
__L1:
    fmadd   fp0,fp2,fp1,fp0
    lfdux   fp2,r31,r7
    lfdu    fp1,8(r30)
    bdnz    __L1
__L2:
    fmadd   fp0,fp2,fp1,fp0
```

pipelined execution

3 instructions, 2.9 cycles per iteration

Optimization Level 3 (-O3)

- **More extensive optimization**
  - Deeper inner loop unrolling
  - Better loop scheduling
  - Additional optimizations allowed by -qnostrict
  - Widened optimization scope (typically whole procedure)
  - No implicit memory usage limits (-qmaxmem=-1)
- **Some precision tradeoffs**
  - Reordering of floating point computations
  - Reordering or elimination of possible exceptions (eg. divide by zero, overflow)

Matrix Multiply inner loop code with -O3 -qtune=pwr3

```
__L1:
    fmadd   fp6,fp12,fp13,fp6
    lfdux   fp12,r12,r7
    lfd     fp13,8(r11)
    fmadd   fp7,fp8,fp9,fp7
    lfdux   fp8,r12,r7
    lfd     fp9,16(r11)
    lfdux   fp10,r12,r7
    lfd     fp11,24(r11)
    fmadd   fp1,fp12,fp13,fp1
    lfdux   fp12,r12,r7
    lfd     fp13,32(r11)
    fmadd   fp0,fp8,fp9,fp0
    lfdux   fp8,r12,r7
    lfd     fp9,40(r11)
    fmadd   fp2,fp10,fp11,fp2
    lfdux   fp10,r12,r7
    lfd     fp11,48(r11)
    fmadd   fp4,fp12,fp13,fp4
    lfdux   fp12,r12,r7
    lfd     fp13,56(r11)
    fmadd   fp3,fp8,fp9,fp3
    lfdux   fp8,r12,r7
    lfdu    fp9,64(r11)
    fmadd   fp5,fp10,fp11,fp5
    bdnz    __L1
```

unrolled by 8

dot product accumulated in
8 interleaved parts (fp0-fp7)
(combined after loop)

3 instructions, 1.6 cycles per iteration

2 loads and 1 fmadd per iteration

Tips for getting the most out of -O2 and -O3

- If possible, test and debug your code without optimization before using -O2
- Ensure that your code is standard-compliant. Optimizers are the ultimate conformance test!
  - In Fortran code, ensure that subroutine parameters comply with aliasing rules
  - In C code, ensure that pointer use follows type restrictions (generic pointers should be char* or void*)
  - Ensure all shared variables and pointers to same are marked volatile
- Compile as much of your code as possible with -O2.
- If you encounter problems with -O2, consider using -qalias=noansi or -qalias=nostd rather than turning off optimization.
- Next, use -O3 on as much code as possible.
- If you encounter problems or performance degradations, consider using -qstrict or -qcompact along with -O3 where necessary.
- If you still have problems with -O3, switch to -O2 for a subset of files/subroutines but consider using -qmaxmem=-1 and/or -qnostrict.

High Order Transformations (-qhot)

- Supported for Fortran (and for C and C++ in 6.0 beta)
- Specified as -qhot[=[no]vector | arraypad[=*n*]]
- Optimized handling of F90 array language constructs (elimination of temporaries, fusion of statements)
- High level transformation (eg. interchange, fusion, unrolling) of loop nests to optimize:
  - ►memory locality (reduce cache/TLB misses)
  - ►usage of hardware prefetch
  - ►loop computation balance (typically ld/st vs. float)
- *Optionally* transforms loops to exploit vector intrinsic library (eg. reciprocal, sqrt, trig) - may result in slightly different rounding
- *Optionally* introduces array padding under user control - potentially unsafe if not applied uniformly

Matrix multiply inner loop code with -O3 -qhot -qtune=pwr3

```
__L1:
   fmadd   fp1,fp4,fp2,fp1
   fmadd   fp0,fp3,fp5,fp0
   lfdux   fp2,r29,r9
   lfdu    fp4,32(r30)
   fmadd   fp10,fp7,fp28,fp10
   fmadd   fp7,fp9,fp7,fp8
   lfdux   fp26,r27,r9
   lfd     fp25,8(r29)
   fmadd   fp31,fp30,fp27,fp31
   fmadd   fp6,fp11,fp30,fp6
   lfd     fp5,8(r27)
   lfd     fp8,16(r28)
   fmadd   fp30,fp4,fp28,fp29
   fmadd   fp12,fp13,fp11,fp12
   lfd     fp3,8(r30)
   lfd     fp11,8(r28)
   fmadd   fp1,fp4,fp9,fp1
   fmadd   fp0,fp13,fp27,fp0
   lfd     fp4,16(r30)
   lfd     fp13,24(r30)
   fmadd   fp10,fp8,fp25,fp10
   fmadd   fp8,fp2,fp8,fp7
   lfdux   fp9,r29,r9
   lfdu    fp7,32(r28)
   fmadd   fp31,fp11,fp5,fp31
   fmadd   fp6,fp26,fp11,fp6
   lfdux   fp11,r27,r9
   lfd     fp28,8(r29)
   fmadd   fp12,fp3,fp26,fp12
   fmadd   fp29,fp4,fp25,fp30
   lfd     fp30,-8(r28)
   lfd     fp27,8(r27)
   bdnz    __L1
```

unroll-and-jam 2x2

inner unroll by 4

interchange "i" and "j" loops

2 instructions, 1.0 cycles per iteration

balanced: 1 load and 1 fmadd per iteration

Vectorization Example

```
SUBROUTINE VD(A,B,C,N)
REAL*8 A(N),B(N),C(N)
DO I = 1, N
  A(I) = C(I) / SQRT(B(I))
END DO
END
```

Vectorization Example pseudocode (slightly edited -qreport output) with -O3 -qhot -qarch=pwr3

```
                SUBROUTINE vd (a, b, c, n)
     3|           IF (n > 0) THEN
     4|             CALL __vrsqrt_630(a,c,&n)
     3|             @CIV0 = 0
       Id=3          DO @CIV0 = @CIV0, n-1
                       ! DIR_INDEPENDENT loopId = 0
     4|                a(@CIV0 + 1) = b(@CIV0 + 1) * a(@CIV0 + 1)
     5|             ENDDO
                   ENDIF
     6|            RETURN
                 END SUBROUTINE vd


Source          Source          Loop Id         Action / Information
File            Line
----------      ----------      ----------      ------------------------------------
          0               4                     Vectorization applied to statement.
```

Tips for getting the most out of -qhot

- Try using -qhot along with -O2 or -O3 for all of your code. It is designed to have neutral effect when no opportunities exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of -qhot, try using -qhot=novector, or -qstrict or -qcompact along with -qhot.
- If possible, report long compile times or poor generated code to IBM through your service representative. If that doesn't work, feel free to contact me.
- If necessary, deactivate -qhot selectively, allowing it to improve some of your code.
- Read the transformation report generated using -qreport (Fortran only for now). If your hot loops are not transformed as you expect, try using assertive directives such as INDEPENDENT or CNCALL or prescriptive directives such as UNROLL or PREFETCH.

Interprocedural Analysis (-qipa)

- Supported for Fortran and C (and C++ in 6.0 beta)
- Can be specified on the compile step only or on both compile and link steps ("whole program" mode)
- Whole program mode expands the scope of optimization to an entire program unit (executable or shared object)
- Specified as -qipa[=level=*n* | inline= | *fine tuning*]
  - *level=0:* Program partitioning and simple interprocedural optimization
  - *level=1:* Inlining and global data mapping
  - *level=2*: Global alias analysis, specialization, interprocedural data flow
  - *inline*=: Precise user control of inlining
  - *fine tuning*: Specify library code behaviour, tune program partitioning, read commands from a file

Interprocedural analysis in depth

- level=0
  - automatic recognition of standard libraries (eg. ANSI C, Fortran runtime, ESSL)
  - localization of statically bound variables and procedures
  - partitioning and layout of code according to call affinity
    - expansion of backend optimizer scope
- level=1
  - procedure inlining
  - partitioning and layout of static data according to reference affinity
- level=2
  - whole program alias analysis
    - disambiguation of pointer references and calls, refinement of call side effect information
  - aggressive intraprocedural optimizations
    - value numbering, code propagation and simplification, code motion (into conditions, out of loops), redundancy elimination
  - interprocedural constant propagation, dead code elimination, pointer analysis
  - procedure specialization (cloning)

Tips for getting the most from -qipa

- When specifying optimization options in a makefile, remember to use the compiler driver (cc, xlf, etc) to link and repeat all options on the link step:
  - LD = xlf
  - OPT = -O3 -qipa
  - FFLAGS=...$(OPT)...
  - LDFLAGS=...$(OPT)...
- -qipa works when building executables or shared objects but always compile 'main' and exported functions with -qipa.
- It is not necessary to compile everything with -qipa but try to apply it to as much of your program as possible.
- When compiling and linking separately, use -qipa=noobject on the compile step for faster compilation.
- Ensure there is enough space in /tmp (at least 200MB) or use the TMP_DIR variable to specify a different directory.
- The "level" suboption is a throttle. Try varying the "level" suboption if link time is too long. -qipa=level=0 can be very beneficial for little cost.
- Look at the generated code. If too few or too many functions are inlined, consider using -qipa=[no]inline

Target Machine Options

- **-qarch**
  - Restricts the compiler to generate a subset of the Power or PowerPC instruction set
  - Specified as -qarch=*isa* where *isa* is one of:
    - *com* (default): Code can run on any RS/6000 - implies -qtune=pwr2
    - *auto*: Code may take advantage of instructions available only on the compiling machine (or similar machines)
    - *ppc*: Code follows PowerPC architecture - implies -qtune=604 (32 bit) or -qtune=pwr3 (64 bit)
    - *pwr3*: Code can run on any Power 3 - implies -qtune=pwr3
    - Lots of others: *pwr, pwr2, 604, pwr4, ...*

Target Machine Options *(continued)*

- **-qtune**
  - Bias optimization toward execution on a given machine
  - Does *not* imply anything about the ability to run correctly on a given machine - only affects performance
  - -qtune=auto generates code that is automatically tuned for the compiling machine (or similar machines)
  - Specified as -qtune=*machine* where *machine* is one of auto, 604, pwr2, p2sc, pwr3, pwr4, rs64c, etc.
- **-qcache**
  - Defines a specific cache/memory geometry
  - Defaults are set through -qtune
  - Specified as -qcache=level=*n*:*cache_spec*, where *cache_spec* includes:
    - type=i|d|c: cache type (instruction/data/combined)
    - line=*lsz*:size=*sz*:assoc=*as*: line/cache size and set associativity
    - cost=*c*: cost (in cpu cycles) of a miss
  - Mainly useful when using -qhot or -qsmp

Getting the most out of target machine options

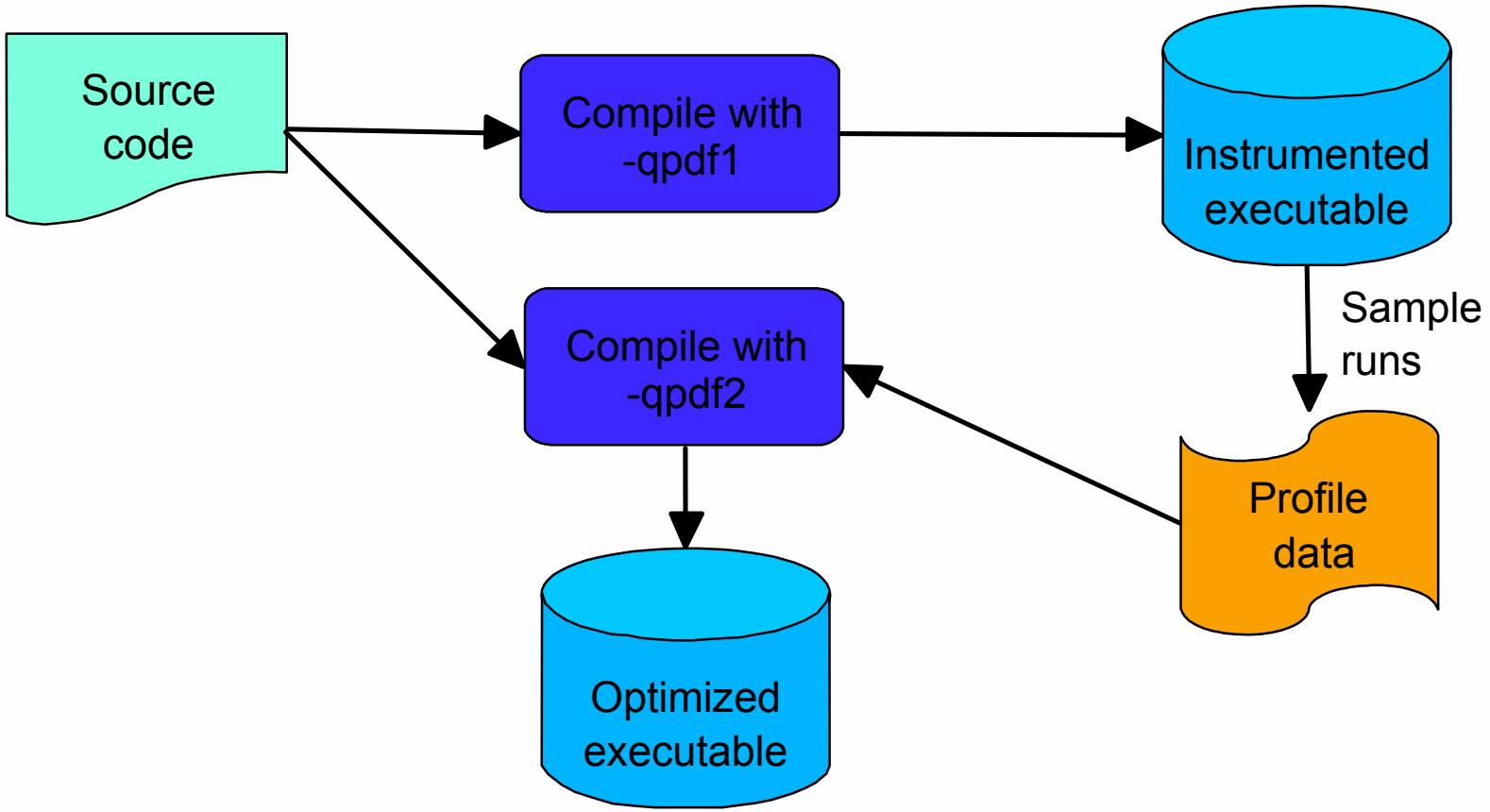
- Try to specify with -qarch the smallest family of machines possible that will be expected to run your code correctly.
  - -qarch=com will generate code that runs anywhere but will have slower integer divides and multiplies and will be unable to exploit single precision floating point
  - -qarch=ppc is better if you don't need to run on Power or Power2 but this will inhibit generation of sqrt or fsel, for example
  - -qarch=ppcgr is a bit better, since it allows generation of fsel but still no sqrt
  - To get sqrt, you will need -qarch=pwr3. This will also generate correct code for Power 4.
- Try to specify with -qtune the machine where performance should be best.
  - If you are not sure, try -qtune=pwr3. This will generate code that should generally run well on most machines.
- Before using the -qcache option, have a look at the options sections of the listing (using -qlist) to see if the current settings are satisfactory. If you do decide to use -qcache, use -qhot or -qsmp along with it.

The -O4 and -O5 macro options

- Optimization levels 4 and 5 automatically activate several other optimization options as a package
- Optimization level 4 (-O4) includes:
  - -O3
  - -qhot (needs 6.0 beta for C/C++)
  - -qipa (needs 6.0 beta for C++)
  - -qarch=auto
  - -qtune=auto
  - -qcache=auto
- Optimization level 5 (-O5) includes everything from -O4 plus:
  - -qipa=level=2

Profile Directed Feedback

- Profile directed feedback (PDF) is a two-stage compilation process that allows the user to provide additional detail about typical program behaviour to the compiler.



Profile Directed Feedback *(continued)*

- Stage 1 is a regular compilation (using an arbitrary set of optimization options) with the -qpdf1 option added.
  - the resulting object code is instrumented for the collection of program control flow and other data
- The executable or shared object created by stage 1 can be run in a number of different scenarios for an arbitrary amount of time
- Stage 2 is a recompilation (only relinking is necessary with Fortran 7.1.1 or C/C++ 6.0) using exactly the same options except -qpdf2 is used instead of -qpdf1.
  - the compiler consumes previously collected data for the purpose of path-biased optimization
    - code layout, scheduling, register allocation
    - (in XLF 7.1.1, C/C++ 6.0) inlining decisions, partially invariant code motion, switch code generation, loop optimizations
- PDF should be used mainly on code which has rarely executed conditional error handling or instrumentation
- PDF usually has a neutral effect in the absence of firm profile information (ie. when sample data is inconclusive)
- However, always use characteristic data for profiling. If sufficient data is unavailable, do not use PDF.

Other Performance Options

- **-qcompact**: specified as -q[no]compact
  - Prefers final code size reduction over execution time performance when a choice is necessary
  - Can be useful as a way to constrain -O3 optimization
- **-qsmallstack** (Fortran 8.1, C/C++ 6.0)
  - Tells the compiler to compact stack storage (may increase heap usage)
- **-qinline**: specified as -qinline[+*names* | *-names*] or -qnoinline
  - Controls inlining of named functions - usable at compile time and/or link time
  - Synonymous with -qipa=inline and -Q
- **-qunroll**: specified as -q[no]unroll
  - Independently controls loop unrolling (implicitly activated under -O2 and -O3)
- **-qinlglue:** specified as -q[no]inlglue
  - Inline calls to "glue" code used in calls through function pointers (including *virtual*) and calls to functions which are dynamically bound
  - Pointer glue is inlined by default for -qtune=pwr4

Other Performance Options *(continued)*

- **-qtbtable**
  - Controls the generation of traceback table information:
  - *-qtbtable=none* inhibits generation of tables - no stack unwinding is possible
  - *-qtbtable=small* generates tables which allow stack unwinding but omit name and parameter information - useful for optimized C++
    - This is the default setting when using optimization
  - *-qtbtable=full* generates full tables including name and parameter information - useful for debugging
- **-q[no]eh** (C++ only)
  - Asserts that no throw is reachable from compiled code - can improve execution time and reduce footprint in the absence of C++ exception handling
- **-q[no]unwind** (Fortran 8.1, C/C++ 6.0)
  - Asserts that the stack will not be unwound in such a way that register values must be accurately restored at call points - usually true in C and Fortran and allows the compiler to be more aggressive in register save/restore

Other Performance Options *(continued)*

- **-qlargepage** (Fortran 8.1, C/C++ 6.0, Power 4 only)
  - Hint to the compiler that the heap and static data will be allocated from large pages at execution time (controlled by environment variable in AIX 5.1D)
  - Compiler will divert large data from the stack to the heap
  - Compiler may also bias optimization of heap or static data references

More Target Machine Options

- **-q64, -q32**: Generate code for 64 bit (4/8/8) or 32 bit (4/4/4) addressing model
  - -q64 generates code with incompatible object formats on AIX V4 and AIX V5. If your code needs to run on both, build two executables or two libraries.
- **-qsmp**: Generate threaded code for a shared-memory parallel machine
  - Supported in Fortran and C (and C++ in 6.0)
  - Specified as -qsmp[=[no]auto:[no]omp:[no]opt:*fine tuning*]
  - *auto* instructs the compiler to automatically generate parallel code where possible without user assistance
  - *omp* instructs the compiler to observe OpenMP 1.0 language extensions for specifying explicit parallelism
  - *opt* instructs the compiler to optimize as well as parallelize. The optimization is equivalent to -O2 -qhot by default. The default setting is -qsmp=opt.
  - *fine tuning* includes control over thread scheduling, nested parallelism and locking

Getting the most out of -qsmp

- Test your programs using optimization and preferably using -qhot in a single-threaded manner before using -qsmp (where practical).
- Always use the "_r" or reentrant compiler invocations when using -qsmp.
- By default, the runtime will use all available processors. Do not set the PARTHDS or OMP_NUM_THREADS variables unless you wish to use fewer than the number of available processors.
- If using a machine or node in a dedicated fashion, consider setting the SPINS and YIELDS variables (suboptions of XLSMPOPTS) to 0.
- -qsmp implies an optimization level of at least -O2. When debugging an OpenMP program, try using -qsmp=noopt (without -O) to make debugging information produced from the compiler more precise.
- If you encounter apparent memory overrun errors with -qsmp, your thread stack size may be too small. This sometimes happens with Fortran 90 code due to temporary storage allocated by the compiler. Use the STACK suboption of XLSMPOPTS to modify the thread stack size.

Floating Point Options

- **-qfloat**
  - Precise control over the handling of floating point calculations
  - Defaults are *almost* IEEE 754 compliant
  - Specified as -qfloat=*subopt* where *subopt* is one of:
    - *[no]fold*: enable compile time evaluation of floating point calculations - may want to disable for handling of certain exceptions (eg. overflow, inexact)
    - *[no]maf*: enable generation of multiple-add type instructions - may want to disable for exact compatibility with other machines but this will come at a high price in performance
    - *[no]rrm*: specifies that rounding mode may not be round-to-nearest (default is *norrm*) or may change across calls
    - *[no]hsflt*: allow various fast floating point optimizations including replacement of division by multiplication by a reciprocal
    - *[no]rsqrt*: allow computation of a divide by square root to be replaced by a multiply of the reciprocal square root

Floating Point Options *(continued)*

- **-qflttrap**
  - Enables software checking of IEEE floating point exceptions
  - Usually more efficient than hardware checking since checks can be executed less frequently
  - Specified as -qflttrap=imprecise | enable | *ieee_exceptions*
  - -qflttrap=imprecise: check for error conditions at procedure entry/exit, otherwise check after any potentially excepting instruction
  - -qflttrap=enable: enables generation of checking code, also enables exceptions in hardware
  - -qflttrap=overflow:underflow:zerodivide:inexact: check given conditions
  - In the event of an error, SIGTRAP is raised - as a convenience the -qsigtrap option will install a default handler which dumps a stack trace at the point of error

Program Behaviour Options

- **-q[no]strict**
  - Default is -qstrict with -qnoopt and -O2, -qnostrict with -O3, -O4, -O5
  - *-qnostrict* allows the compiler to reorder floating point calculations and potentially excepting instructions
- **-qalias (Fortran)**
  - Specified as -qalias=[no]std:[no]aryovrlp:*others*
  - Allows the compiler to assume that certain variables do not refer to overlapping storage
  - *std* (default) refers to the rule about storage association of reference parameters with each other and globals
  - *aryovrlp* (default) defines whether there are any assignments between storage-associated arrays - try -qalias=noaryovrlp for better performance your Fortran 90 code has no storage associated assignments

Program Behaviour Options *(continued)*

- **-qalias (C, C++)**
  - Similar to Fortran option of the same name but focussed on overlap of storage accessed using pointers
  - Specified as -qalias=*subopt* where *subopt* is one of:
    - *[no]ansi*: Enable ANSI standard type-based alias rules (ansi is default when using "xlc", noansi is default when using "cc")
    - *[no]typeptr*: Assume pointers to different types never point to the same or overlapping storage - use if your pointer usage follows strict type rules
    - *[no]allptrs*: Assume that different pointer variables always point to non-overlapping storage - use only in selected situations where pointers never overlap
    - *[no]addrtaken*: Assume that external variables do not have their address taken outside the source file being compiled

Why the big fuss about aliasing?

- The precision of compiler analyses is gated in large part by the apparent effects of direct or indirect memory writes and the apparent presence of direct or indirect memory reads.
- Memory can be referenced directly through a named symbol, indirectly through a pointer or reference parameter, or indirectly through a function call.
- Many apparent references to memory are false and these constitute barriers to compiler analysis.
- The compiler does analysis of possible aliases at all optimization levels but analysis of these apparent references is best when using -qipa since it can see through most calls.
- Options such as -qalias and directives such as disjoint, isolated_call, CNCALL and INDEPENDENT can have pervasive effect since they fundamentally improve the precision of compiler analysis.

Program Behaviour Options *(continued)*

- **-qassert (Fortran, C)**
  - Specified as -qassert=[no]deps | itercnt=*n*
  - *deps* (default) indicates that some loop has a memory dependence or conflict from iteration to iteration. Try -qassert=nodeps for improved performance when no loops in a file carry a dependence.
  - *itercnt* modifies the default assumptions about the expected iteration count of loops (normally 10).
- **-qintsize (Fortran)**: Define the default size of INTEGER variables
  - Specified as -qintsize=1|2|4|8
  - When using -q64, try -qintsize=8 for improved performance
- **-qignerrno (C,C++)** - Specified as *-q[no]ignerrno*
  - Indicates that the value of *errno* is not needed by the program
  - Can help in optimization of math functions such as sqrt.
  - This is the default with -O3.

Program Behaviour Options *(continued)*

- **-qdatalocal**
  - *-qdatalocal[=vars]*: Specifies that the definitions of all or just the named variables will be statically bound - access to statically bound variables may be faster
- **-qdataimported**
  - *-qdataimported[=vars]*: Specifies that the definitions of all or just the named variables might be dynamically bound
- **-qproclocal**
  - *-qproclocal[=funcs]*: Specifies that the definitions of all or just the named functions will be statically bound - calls to statically bound functions are faster than dynamic or unknown
- **-qprocimported**
  - *-qprocimported[=funcs]*: Specifies that the definitions of all or just the named functions will be dynamically bound
- **-qprocunknown**
  - *-qprocunknown[=funcs]*: Specifies that the definitions of all or just the named functions have unknown linkage
- Note that the usage of -qipa when linking automatically detects the linkage of all variables and functions, obviating the need for these options

Program Behaviour Options *(continued)*

- **-q[no]libansi (C, C++)**
  - Specifies that calls to ANSI standard functions will be bound with conforming implementations
  - When linking with -qipa, this options is not necessary.
- **-ma (C, C++)**
  - Directs the compiler to generate inline code for calls to the *alloca* function.
- **-qproto (C)**
  - Asserts that procedure call points agree with their declarations even if the procedure has not been prototyped.
  - Useful for well behaved K&R C code.
- **-qro,-qroconst (C,C++)**
  - Directs the compiler to place string literals (-qro) or constant values (-qroconst) in read-only storage
- **-qaggrcopy={ovrlap|noovrlap} (C, C++)**
  - Specify whether aggregate assignments may have overlapping source and target locations
  - Default is ovrlap with "cc", noovrlap with "xlc"

Diagnostic Options

- **-qlist**
  - Instructs the compiler to emit an object listing
  - The object listing includes hex and pseudo-assembly representations of the generated code along with traceback tables and text constants
- **-qreport (Fortran)**
  - Specified as -qreport [=smplist]
  - Instructs the high level optimizer to emit a report including pseudo-Fortran along with annotations describing what transformations were performed (eg. loop unrolling, automatic parallelization)
  - Also includes information about data dependences and other inhibitors to optimization

Diagnostic Options *(continued)*

- **-qinitauto**
  - Directs the compiler to emit code that initializes all automatic (stack) variables to a given value
  - *-qinitauto=XX* initializes bytes with the value given in hex
  - *-qinitauto=XXXXXXXX* initializes words with the value given in hex
- **-qcheck=[nullptr|bounds|divzero]**
  - Inserts runtime checks (traps) for null pointer access, array bounds violations and/or divide by zero
- **-qextchk**
  - Generates additional symbolic information to allow the linker to do cross-file type checking of external variables and functions
  - Requires the linker -btypchk option to be active

Directives and Pragmas

- **OpenMP 1.0 -** supported in C and Fortran (and C++ in 6.0)
- **Legacy SMP** directives and pragmas
  - Most of these are superceded by OpenMP - use OpenMP where possible
- **Assertive directives** (Fortran)
  - ASSERT, INDEPENDENT, CNCALL, PERMUTATION
- **Assertive pragmas** (C)
  - *isolated_call*, *disjoint, independent_loop, independent_calls, iterations, permutation, execution_frequency, leaves*
- **Embedded Options**
  - *#pragma options* and *#pragma option_override* in C
  - @PROCESS in Fortran
- **Prescriptive directives** (Fortran)
  - PREFETCH, UNROLL
- **Prescriptive pragmas** (C)
  - *sequential_loop*

Assertive Directives (Fortran)

- **ASSERT** (ITERCNT(*n*) | [NO]DEPS)
  - Same as options of the same name but applicable to a single loop - much more useful
- **INDEPENDENT:** Asserts that the following loop has *no* loop carried dependences - enables locality and parallel transformations
- **CNCALL:** Asserts that the calls in the following loop do not cause loop carried dependences
- **PERMUTATION** (*names*)
  - Asserts that elements of the named arrays take on distinct values on each iteration of the following loop - may be useful in sparse codes

Assertive Pragmas (C)

- *isolated_call* (*function_list*) asserts that calls to the named functions do not have side effects
- *disjoint* (*variable_list*) asserts that none of the named variables (or pointer dereferences) share overlapping areas of storage
- *independent_loop* is equivalent to INDEPENDENT
- *independent_calls* is equivalent to CNCALL
- *permutation* is equivalent to PERMUTATION
- *iterations* is equivalent to ASSERT(ITERCNT)
- *execution_frequency* (*very_low*) asserts that the control path containing the pragma will be infrequently executed
- *leaves* (*function_list*) asserts that calls to the named functions will not return (eg. exit)

Prescriptive Directives (Fortran)

- **PREFETCH**
  - PREFETCH_BY_LOAD (*variable_list*): issue *dummy* loads to cause the given variables to be prefetched into cache - useful on Power machines or to activate Power 3 hardware prefetch
  - PREFETCH_FOR_LOAD (*variable_list*): issue a *dcbt* instruction for each of the given variables.
  - PREFETCH_FOR_STORE (*variable_list*): issue a *dcbtst* instruction for each of the given variables.
- **CACHE_ZERO**
  - Inserts a dcbz (data cache block zero) instruction with the given address
  - Useful when storing to contiguous storage (avoids the L2 store miss entirely)
- **UNROLL**
  - Specified as [NO]UNROLL [(*n*)]
  - Used to activate/deactivate compiler unrolling for the following loop.
  - Can be used to give a specific unroll factor.

Prescriptive Pragmas (C)

- *sequential_loop* directs the compiler to execute the following loop in a single thread, even if the -qsmp=auto option is specified

Compiler Friendly Programming

- Compiler-friendly programming idioms can be as useful to performance as any of the options or directives
- Do not excessively hand-optimize your code (eg. unrolling, 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 -qipa.
- Use register-sized integers (**long** in C/C++ and **INTEGER*4** or **INTEGER*8** in Fortran) for scalars. For large arrays of integers, consider using 1 or 2 byte integers or bitfields in C or C++.

Compiler Friendly Programming *(continued)*

- Use the smallest floating point precision appropriate to your computation. 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.
- Consider using the highly tuned MASS and ESSL libraries rather than custom implementations or generic libraries

Fortran programming 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 which demand dynamic allocation.
- Use CONTAINS 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.

C/C++ Programming 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, 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.

Power 4 Optimization Technology

- **Architecture-neutral and -specific code paths**
  - tuning for arch=ppc and arch=pwr4
- **Precise machine model for scheduling (-O2+)**
  - new instruction scheduler with more detailed modelling capability
  - tuned through extensive experimention on early h/w
- **New loop transformations for deep pipelines (-O3+)**
  - more precise loop unrolling and pipelining
- **New aggressive branch optimizations (-O2+)**
  - branch pattern replacement
  - utilization of branch hints (eg. using profile feedback)
- **Optimized usage of hardware-expanded instructions**
  - eg. load/store update, mtcr, lm/stm
- **Optimized prefetch buffer allocation (-qhot)**
  - utilization of prefetch stream start instructions
  - loop nest fusion and partitioning to optimize # streams

Questions?