



Exploiting the Dual FPU in Blue Gene/L

March 2006 (Updated June 2006)

Introduction

The Blue Gene[®]/L supercomputer consists of up to 65,536 compute nodes. Each compute node contains 2 PowerPC 440 (PPC440) processors, each enhanced with a specially designed dual Floating Point Unit (FPU). This dual FPU is also known as the “Double Hummer” FPU. Each of the two FPU units contains 32 64-bit floating point registers for a total of 64 FP registers per processor.

The PPC 440 and the dual FPU

In addition to the regular PowerPC floating point instructions (operating on the *Primary* registers), new *parallel* floating point instructions have been added to operate simultaneously on both the *Primary* and *Secondary* registers. Some of the new dual FPU instructions perform identical operations on each set of registers in parallel. Other instructions allow operands to be *copied* from one register set to the other, or perform complex *cross* operations optimized for complex arithmetic. A set of load/store instructions has also been added to perform loads and stores to both sets of FP registers with a single instruction.

Since the PPC440 chip can issue at most one load/store and one FPU operation per cycle, the parallel instructions have the potential to double the floating point performance of the chip. The IBM Mathematical Acceleration Subsystem (MASS) library (and the vector MASSV library), and the IBM Engineering and Scientific Software Library (ESSL) take advantage of the parallel instructions to fully utilize the dual FPU. Hand written code using the parallel instructions can easily access this performance increase. New *builtin* functions have been added to the IBM XL C and C++ compilers to generate the parallel instructions. *Intrinsic* functions have been added to the IBM XL Fortran compiler.

The IBM XL compilers will automatically generate parallel FPU instructions, but doubling the floating point performance benefit is not usually achieved for arbitrary floating point code.

How much benefit can you expect from a second FPU?

John D. McCalpin gave a keynote talk at the 3rd IEEE Workshop in Workload Characterizations (<http://www.cs.virginia.edu/~mccalpin/wwc-keynote.html>), where he found that most “real applications” and much of SPEC 2000 FP benchmarks show that approximately 40% of instructions issues are load/store operations and about 20% are floating point operations. Using this data, assuming completely independent operations with perfect scheduling and no cache interference or stalls, adding a second load/store

unit and a second floating point unit would allow cutting approximately 60% of instructions issued in half to 30%, increasing the instructions issued per cycle by 42%. In the “real world”, the above assumptions would not hold, and the actual performance increase would be smaller. For BlueGene/L, the “second” load/store unit may only be used for parallel floating point load/stores, lowering the possible benefit.

There are obvious counter-examples, where the percentage of load/store and floating point instructions issued is close to 100%, and the speedup can be close to 2. Examples of this speedup include vector and matrix operations, as well as the LINPACK benchmark. These examples generally process floating point data in regular patterns, such as arrays of floating point values

Limitations of the BG/L dual FPU

Floating Point Registers

While an IBM POWER5[®] processor has only 32 FP registers, it does contain 2 independent floating point units, as well as 2 independent load/store units. A Power 5 processor may issue instructions to all four of those units every cycle. The PPC440 is limited to issuing at most one floating point operation and one load/store instruction per cycle. There are 64 FP registers available; however these registers are not independently addressable. The encoding of registers in the PowerPC architecture allows only 5 bits to name a register, suitable for addressing 32 registers. To overcome this limitation with the BG/L double FPU, the new parallel instructions use the 5 bits to address a *register pair*. A register pair *N* consists of the *N*th register in the primary register set and the corresponding *N*th register in the secondary register set. This pairing obviously violates the *independent* assumption in the previous section.

Parallel load/store

A major benefit of the dual FPU is the ability to issue parallel load/store instructions. As only one load or store instruction may be issued each cycle, the maximum memory accessed by a PPC440 (an 8 byte floating point operand) can be doubled, allowing 16 bytes to be loaded or stored per cycle. The compiler’s use of the parallel load/store instructions must be conservative. On the PPC440, any floating point load or store whose operands cross a cache line boundary (32 bytes) will take an alignment trap. Normally floating point operands are aligned on an 8 byte boundary, so no alignment trap will occur using a single floating point load or store. A 16 byte load from an arbitrary 8 byte boundary will cause an alignment trap 25% of the time. As an alignment trap may cause thousands of cycles of delay, it is important to avoid parallel loads and stores if the operand cannot be proven to be aligned on a 16 byte boundary.

The parallel load/store instructions also have a further restriction. Like AltiVec[™] load/store instructions, these instructions use the base/index instruction format, with no displacement. Any non-zero displacement must be allocated in an index register. This increases register pressure for the integer registers, causing more spill. Modification of the compiler to force all floating point load/stores to use the base/index form showed that for SPEC 2000 FP programs on an IBM Power 4, the slowdown was no more than 5%.

Since many load/stores will be to primary registers without this restriction, the real effect should be much smaller.

Single precision arithmetic

The parallel instructions added for the dual FPU calculate all operations in double precision. It is possible to process single precision computations using double precision instructions. While this increases the range of values over single precision operations, it is not possible to deliver the bitwise exact same results generated by single precision expressions using double precision, unless each double precision operation is immediately followed by a round-to-single-precision operation. On the PPC440 FPU, this additional rounding would add 5 cycles of latency to each parallel operation, negating the benefits of the parallelization. For this reason, single precision calculations are not parallelized automatically by the compiler.

IEEE FP Exceptions

In a similar vein, the parallel operations of the dual FPU do not signal IEEE exceptions. Any program using the -qsigtrap compiler option to detect IEEE exceptions will not be parallelized.

Compiler generation of dual FPU code

The IBM XL compilers will use the dual FPU in several ways:

- Even without optimization, complex arithmetic will use the parallel instructions to speed up calculations. Structure assignments and **memcpy** will use the parallel load/store instructions if the alignment and size are multiples of 16.
- At -O2 and up, the compiler will attempt to convert floating point calculations within a single block to parallel operations using a Superword Level Parallelism¹ (SLP) algorithm. Alignment information is propagated within a procedure, and heuristics are used to detect situations where generating parallel code may necessitate too many moves between primary and secondary registers.
- -qhot=simd (default with -qarch=440d and -qhot/-O4/-O5) will do loop analysis to generate parallel code across basic blocks, versioning loops for alignment, and rewriting loops to parallelize as much as possible.
The same framework used by the XL compilers for AltiVec and Cell Broadband Engine™ *Single Instruction Multiple Data* (SIMD) code generation is used for BlueGene, treating the double FPU as a 2-element vector.
- Linking with -O5 enables more loop analysis, and allows whole program alignment propagation, reducing the overhead for loop versioning for alignment and for overlap.

¹ *Exploiting Superword Level Parallelism with Multimedia Instruction Sets* Samuel Larsen and Saman Amarasinghe

Achieving doubled floating point performance using the dual FPU

The IBM XL compiler can most easily utilize the dual FPU on BG/L when compiling code processing vectors of doubles accessed with stride 1. An example of code that parallelizes well is:

```
subroutine daxpy (a,b,c,n)
  real*8 a(n),b(n),
  do 10 i = 1,n
    a(i) = a(i) + b(i) * c
10  continue
end
```

Compiling this with `-O5 -qarch=440d2`, the compiler will generate (in pseudo-code):

```
if (n is large enough && a is 16 byte aligned && b is 16 byte aligned) {
    Use parallel instructions to load/compute/store
} else {
    Load/compute/store using single FPU
}
```

Each loop is then unrolled enough times to cover the latency of the FPU (5 cycles), and scheduled to overlap the load/stores and the computation as much as possible. For this subroutine, each *floating-point multiply-add* (FMA) operation is fed by 2 loads and one store. The parallel loop executes approximately $\frac{1}{2}$ the number of instructions of the loop using the single FPU.

Notes:

- The test for alignment and size of n add extra overhead that would not be present when compiling with `-qarch=440`. This can reduce the benefit of the dual FPU unless the value of n is large enough, and is one cause of dual FPU code that is slower than the equivalent single FPU code.
- Whole program analysis using `-O5` at link time will try to propagate alignment information across the whole program. If *interprocedural analysis* (IPA) optimizer can find that all callers of subroutine **daxpy** always pass aligned parameters, then the alignment test may be omitted.

Program code that does not parallelize well

If we modify the **daxpy** routine above to handle non-stride one accesses, and add `-qreport` to the command line, we will find that the program is not parallelized.

```
subroutine daxpy1 (a,b,c,inca,incb,n)
  real*8 a(*),b(*)
  ia = 1
  ib = 1
  do 10 i = 1,n
    a(ia) = a(ia) + b(ib) * c
```

² `-qarch=440d` asks the compiler to use the dual FPU. `-qarch=440` generates code for a single FPU only.

```

        ia = ia + inca
        ib = ib + incb
10    continue
    end

```

The listing file contains:

```

>>>>> LOOP TRANSFORMATION SECTION <<<<<
1586-541 (I) <SIMD info> NON-SIMDIZABLE: other misc reasons. (Loop
        index 1 on line 5 with nest-level 0 and iteration count 100.)
1586-543 (I) <SIMD info> Total number of loops considered <"1">. Total
        number of loops simdized <"0">.

```

In this example, the loop is not parallelized because the SLP algorithm used to find parallelizable loads and stores doesn't handle non-stride 1 accesses.

If we add a **main** program to the above **daxpy1** routine, and compile *and* link with -O5, we can see how whole program analysis removes alignment testing:

```

program main
real*8 a(1000),b(1000)
call daxpy1 (a,b,5.0, 1, 1, 500)
end

```

The pseudo code generated for the **main** program and the call to **daxpy** is now:

```

if (a and b are disjoint) {
    Use parallel instructions to load/compute/store
} else {
    Load/compute/store using single FPU
}

```

Whole program analysis has enabled the compiler to discover that a and b are aligned on 16 byte boundaries, that the array is accessed using stride 1, and that the iteration count is large enough to be worth parallelizing. Unfortunately, it does not realize that a and b are already disjoint. We plan to fix this oversight shortly.

Unable to SIMDize messages from -qreport

When compiling with -qhot=simd and -qreport, the listing file may contain explanations of why the compiler was unable to generate parallel instructions:

```

NON-SIMDIZABLE: non-simdizable reductions.
NON-SIMDIZABLE: upper bound of loop too small.
NON-SIMDIZABLE: loop not innermost.
NON-SIMDIZABLE: data dependence due to aliasing.
NON-SIMDIZABLE: unknown alignment.
NON-SIMDIZABLE: invalid operation.
NON-SIMDIZABLE: invalid loop structure.
NON-SIMDIZABLE: loop with function calls.
NON-SIMDIZABLE: non stride one access.
NON-SIMDIZABLE: other misc reasons.

```

Knowing why a loop doesn't use the parallel instructions may lead to source code changes that will allow use of the dual FPU.

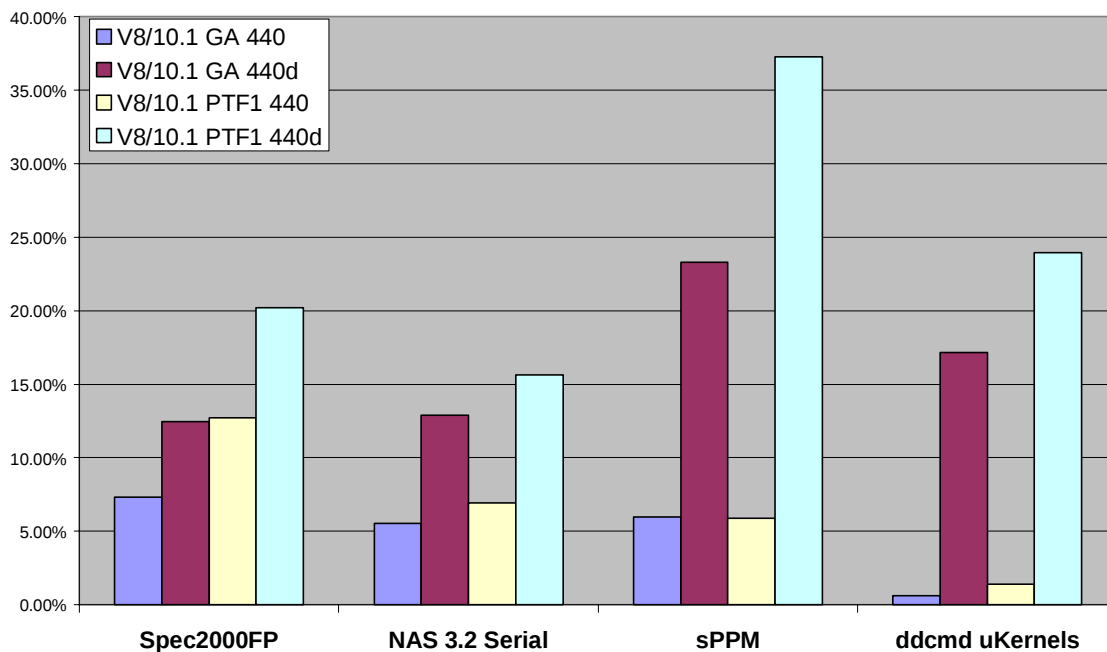
Performance improvements in the latest compilers

The newest versions of the IBM XL compilers (*IBM XL C/C++ Advanced Edition V8.0 for Blue Gene*, *IBM XL Fortran Advanced Edition V10.1 for Blue Gene*) have focused on increasing the quality of the compiler, as well as improving the performance of both -qarch=440 and -qarch=440d generated code. Performance improvements from the C/C++ V8.0 and Fortran V10.1 compilers for AIX and Linux have also improved the performance of BlueGene programs. In addition, significant effort has also been invested in improving the generation of SIMD instructions:

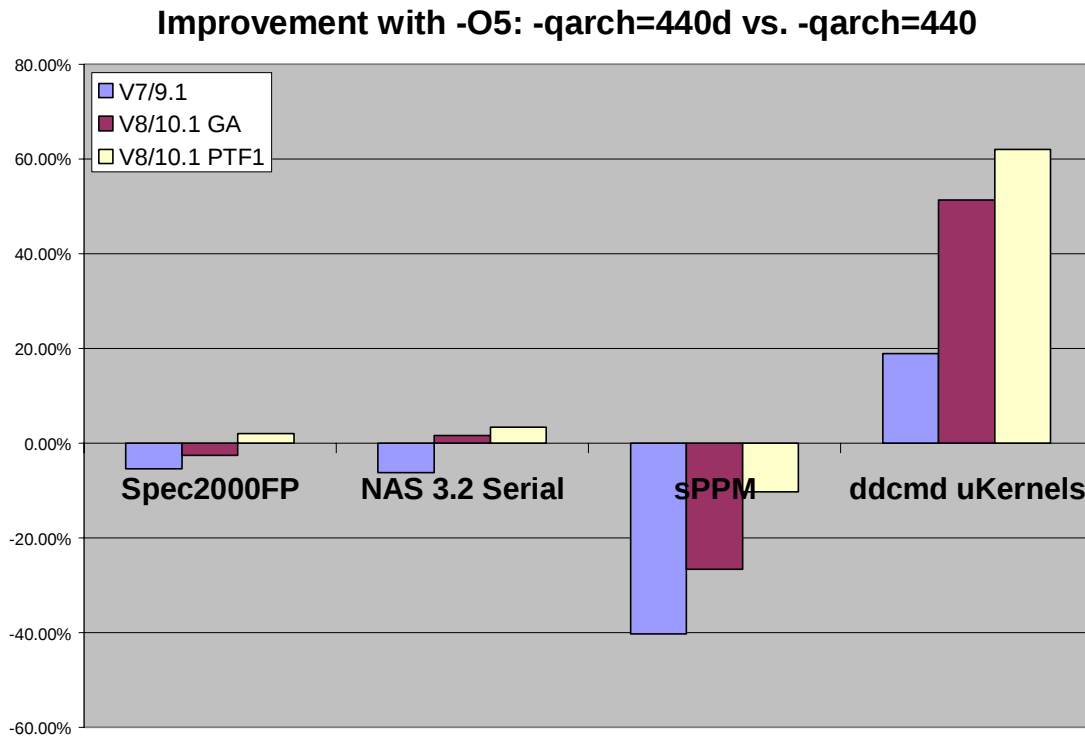
- Simdization of double complex with -qhot
- Simdizing part of a loop without distributing the loop
- Enhanced interprocedural alignment analysis to track 16-byte compile-time misalignment
- Better alignment code generation to maximize load reuse across statements and across iterations
- More reuse conscious loop distribution for simdization purposes

The following chart shows the improvement at -O5 for both the V8/10.1 GA compilers and PTF1 compilers, compared to the latest update for the V7/9.1 compilers. Detailed breakdowns for each benchmark suite can be found in Appendix A.

Improvement with -O5: V8/10.1 GA, PTF1 vs. V7/9.1



The following chart shows how well the compiler uses the dual FPU. Examination of the detailed results shows that several benchmarks have seen a large penalty for using `-qarch=440d`. Investigation into some of these problems has led to increased performance, as can be seen by the improved results in the V8/10.1 GA and PTF1 versions.



Future directions

IBM plans to continue to address performance of dual FPU code in future updates and releases. Improvements in the SIMD framework will also benefit BlueGene. We expect that this will lead to better exploitation of the dual FPU.

Summary

The presence of a second FPU on the BG/L processors potentially allows double the performance on floating point algorithms over just using a single FPU. The ability of the IBM XL compilers to automatically use the dual FPU unit depends strongly on the properties of the source code. The more regular the accesses to floating point data, the more the compiler is able to exploit the dual FPU. Examples of regular access include *matrix multiplication* and *vector processing*. This paper has described the implementation of the dual FPU in BlueGene/L and some limitations of automatic compiler exploitation of this dual FPU. Our long term goal is to ensure that using the dual FPU will be no slower than single FPU code. This may not be achievable, due to the extra versioning necessary for alignment or aliasing checks, but the overhead should be minimized.

Recommended reading

The document “*Using the XL Compilers for Blue Gene*” (SC10-4310-00) comes with the IBM XL C/C++ Advanced Edition V8.0 for Blue Gene and IBM XL Fortran Advanced Edition V10.1 for Blue Gene compilers.

Contacting IBM

IBM welcomes your comments. You can send them to compinfo@ca.ibm.com or mail them to this address:

XL Compiler Development
Department 697
Application Development Technology Centre
Software Division Toronto Laboratory IBM Canada Ltd.
8200 Warden Avenue
Markham, Ontario
Canada – L6G 1C7

Copyright Notice

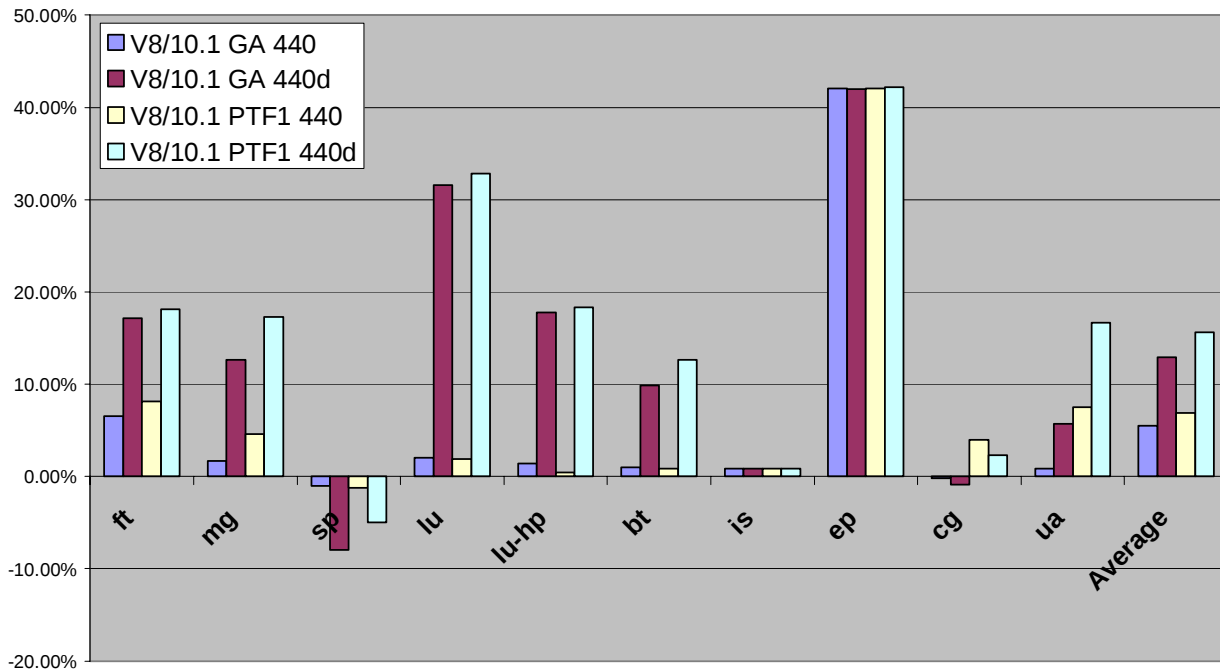
© Copyright IBM Corp. 2006. All Rights Reserved.

IBM is trademark or registered trademark of International Business Machines Corporation in the U.S., other countries or both.

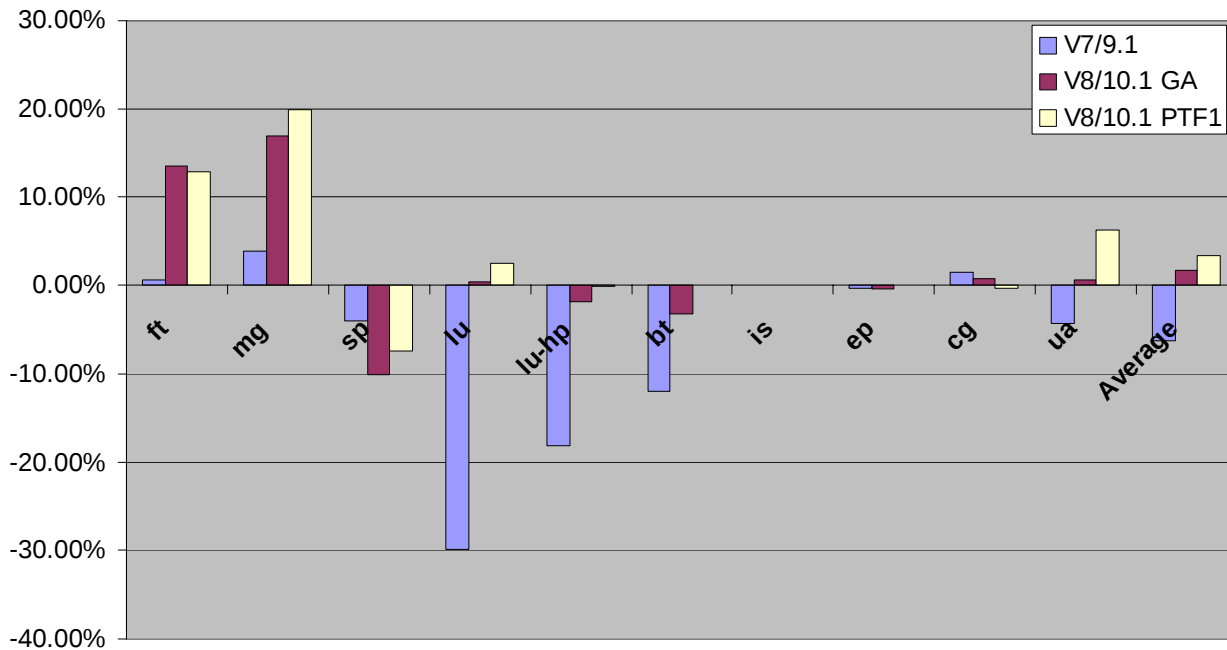
Appendix A: Detailed Compiler Results V8/10.1 vs. V7/9.1

These measurements were made on a 700Mhz DD2 BG/L system at Watson Research Lab. The V7/9.1 compilers used update 3. The V8/10.1 GA compiler used was the version available March 17, 2006, and the PTF1 compiler is the version available June 23, 2006.

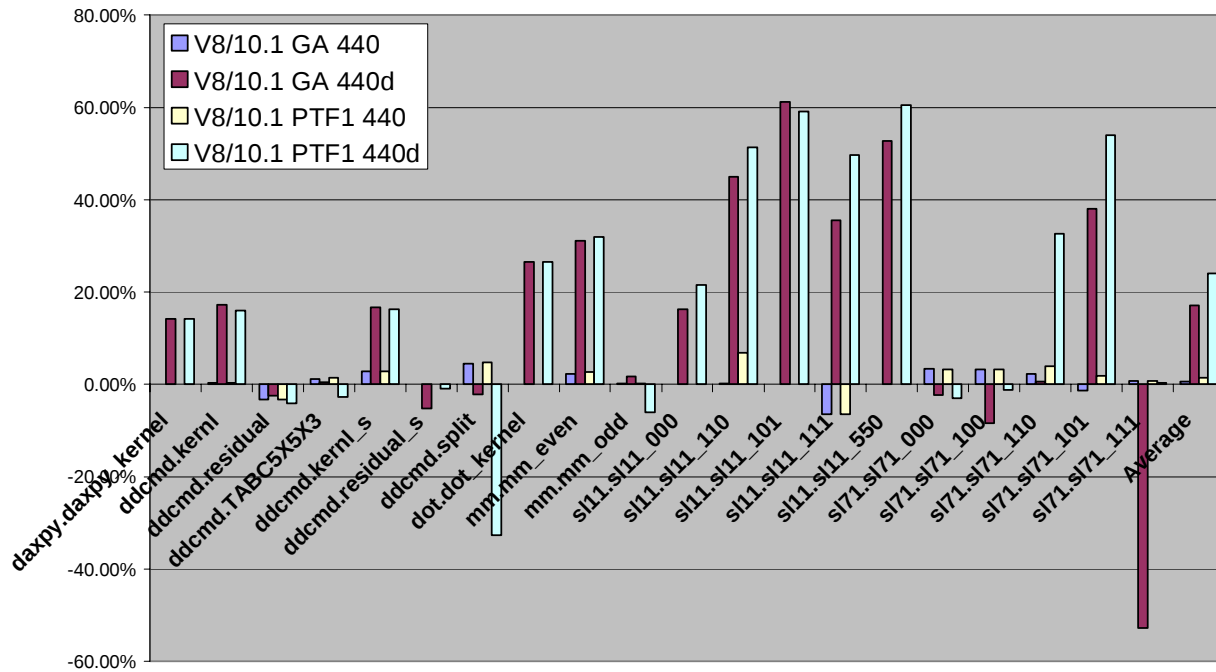
NAS Serial Improvement with -O5: V8/10.1 GA, PTF1 vs. V7/9.1 Compilers



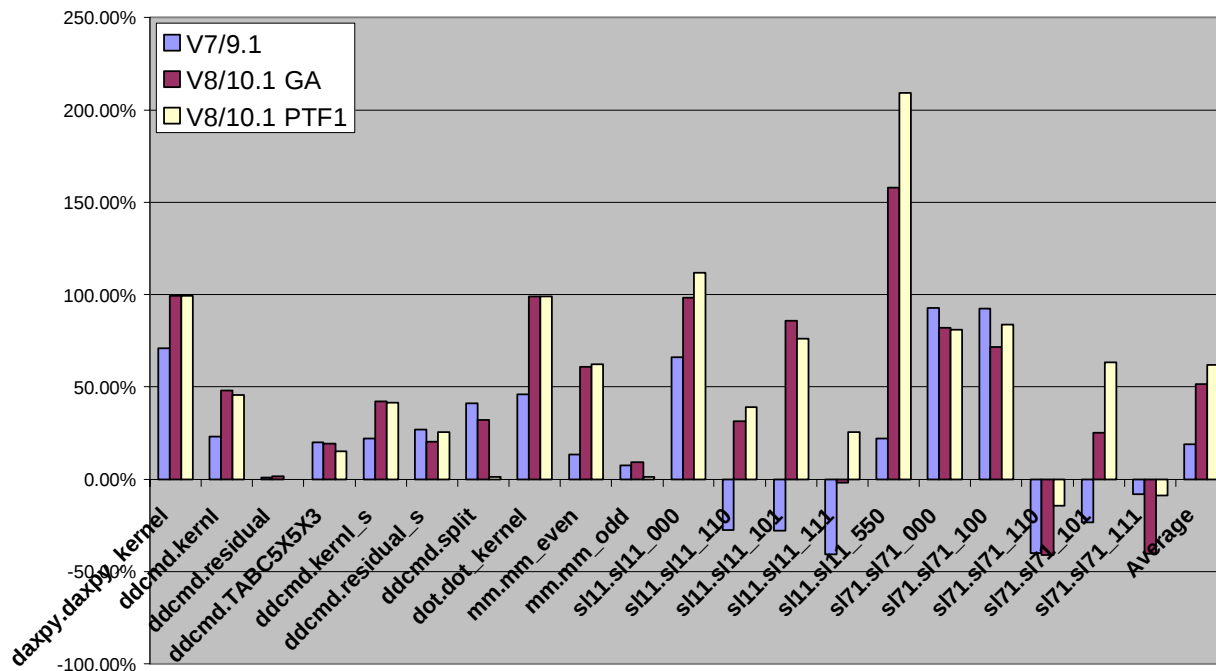
NAS Serial Improvement with -O5: -qarch=440d vs. -qarch=440



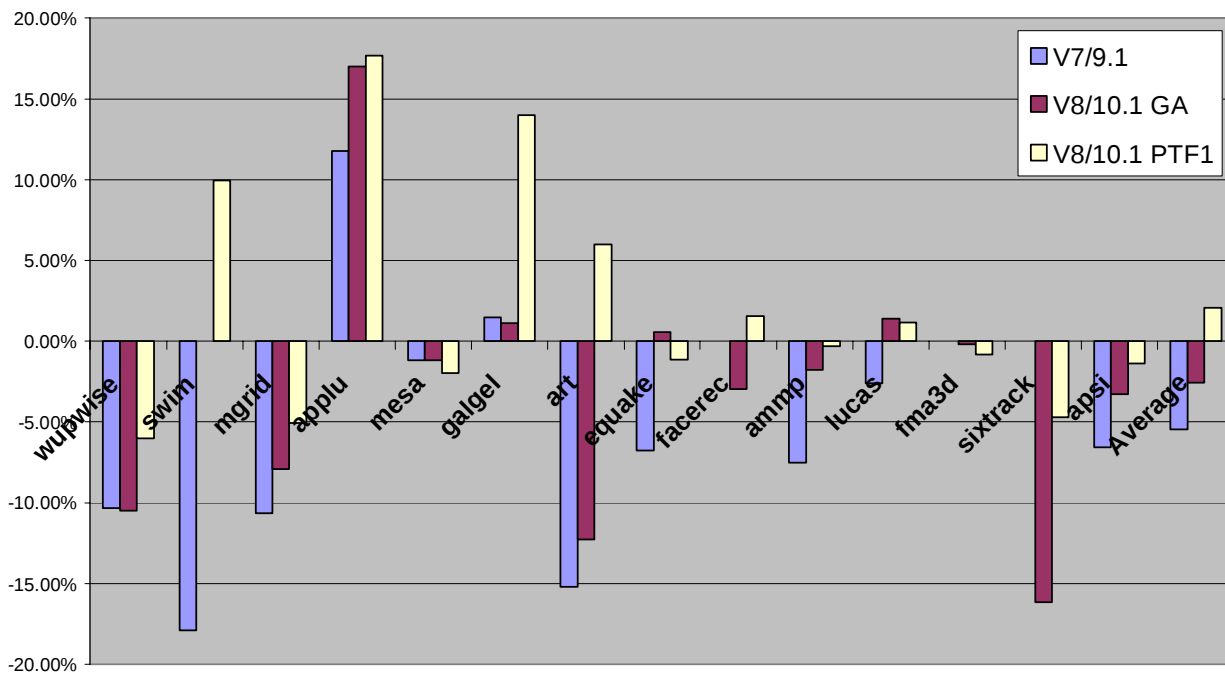
ddc cmd uKernels Improvement with -O5: V8/10.1 GA, PTF1 vs. V7/9.1 Compilers



ddc cmd uKernels Improvement with -O5: -qarch=440d vs. -qarch=440

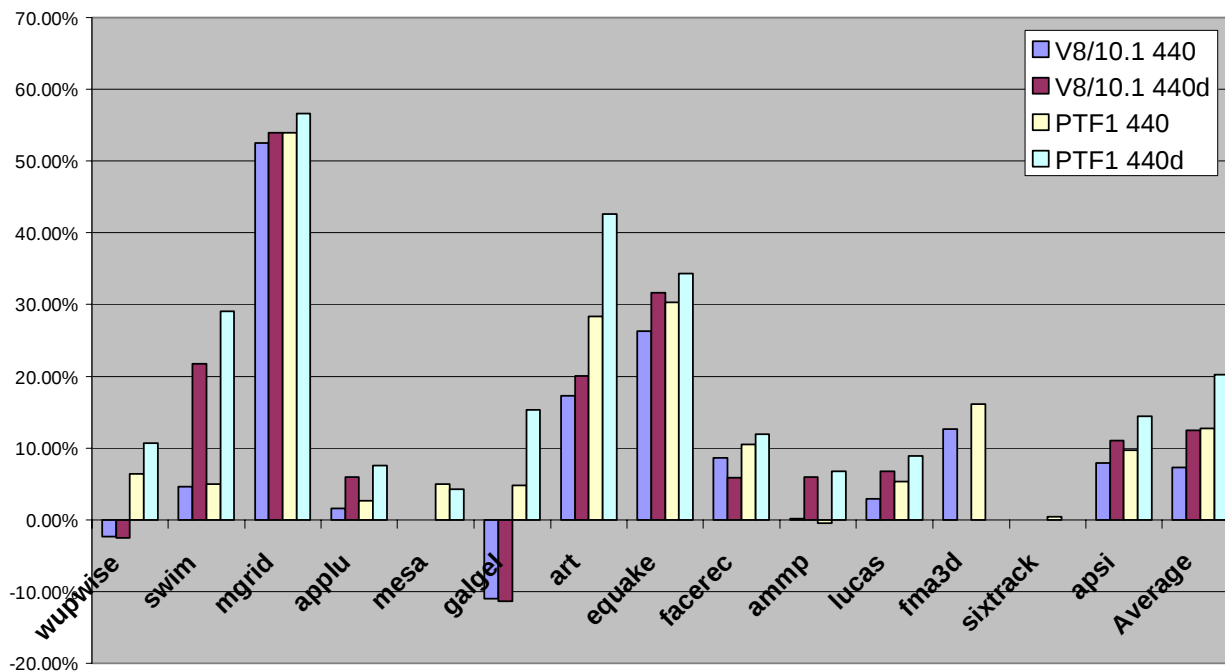


SPEC2000FP Improvement with -O5: -qarch=440d vs. -qarch=440



Note: sixtrack and fma3d failed with -qarch=440d -O5 with V7/9.1 compilers

SPEC2000FP Improvement with -O5: V8/10.1 GA, PTF1 vs. V7/9.1 Compilers



Note: sixtrack and fma3d failed with -qarch=440d -O5 with V7/9.1 compilers