Blue Gene Compilers and Optimization

Allan Martin
IBM Toronto Lab

Mark Mendell
IBM Toronto Lab

3rd BG/L Systems Software & Applications Workshop, Tokyo   April 19-20, 2006

Outline

- **IBM Blue Gene compilers update**
- Performance results from V8.0/10.1
- Generating parallel (SIMD) code for Blue Gene
- What benefit can you expect from 440d?
- Future directions

Terminology & TLAs/FLAs

- SIMD = Single Instruction Multiple Data
- SIMDization is the automatic generation of SIMD instructions by the compiler
- 440d processor includes “parallel” instructions (aka “double hummer”, which act on 2 data (M=2 in SIMD)
- 440d has 32 primary + 32 secondary registers addressed with one 5-bit field
- SLP = Superword Level Parallelism (2 to 8-way)
- CPI = Cycles Per Iteration
- PTF = IBM’s term for a “fixpack”

IBM Compilers for Blue Gene/L

- First compiler release supporting PPC 440/440d (General Availability 10/2004):
  - XL C/C++ V7.0 Advanced Edition for Linux
  - XL Fortran V9.1 Advanced Edition for Linux
- Blue Gene support was separated from the Linux compilers and made available in the next release as a PRPQ March 17, 2006:
  - XL C/C++ V8.0 Advanced Edition for Blue Gene
  - XL Fortran V10.1 Advanced Edition for Blue Gene
  - PTF1 (coming soon) will support version 3 of toolchain

XL C/C++ V8.0 Advanced Edition for Blue Gene

- General performance improvements
- More GCC compatibility
- Perform subset of loop transformations at –O3 optimization level
- Improved performance of quad precision floating point
- Improved support for auto-simdization

XL Fortran V10.1 Advanced Edition for Blue Gene

- **General performance improvements**
- **Continued rollout of Fortran 2003**
- **Perform subset of loop transformations at –O3 optimization level**
- **Improved performance of quad precision floating point**
- **Improved support for auto-simdization**

Blue Gene Specific Work Items in V8.0/10.1

- Focus on correctness for SPEC2000 FP
- Performance tuning for SPEC2000, SPPM, ddcmd kernels, NAS 3.2 Serial
- Tuning of complex arithmetic for 440d (without –qhot)
- Tuning of high performance math library “MASS” for 440d

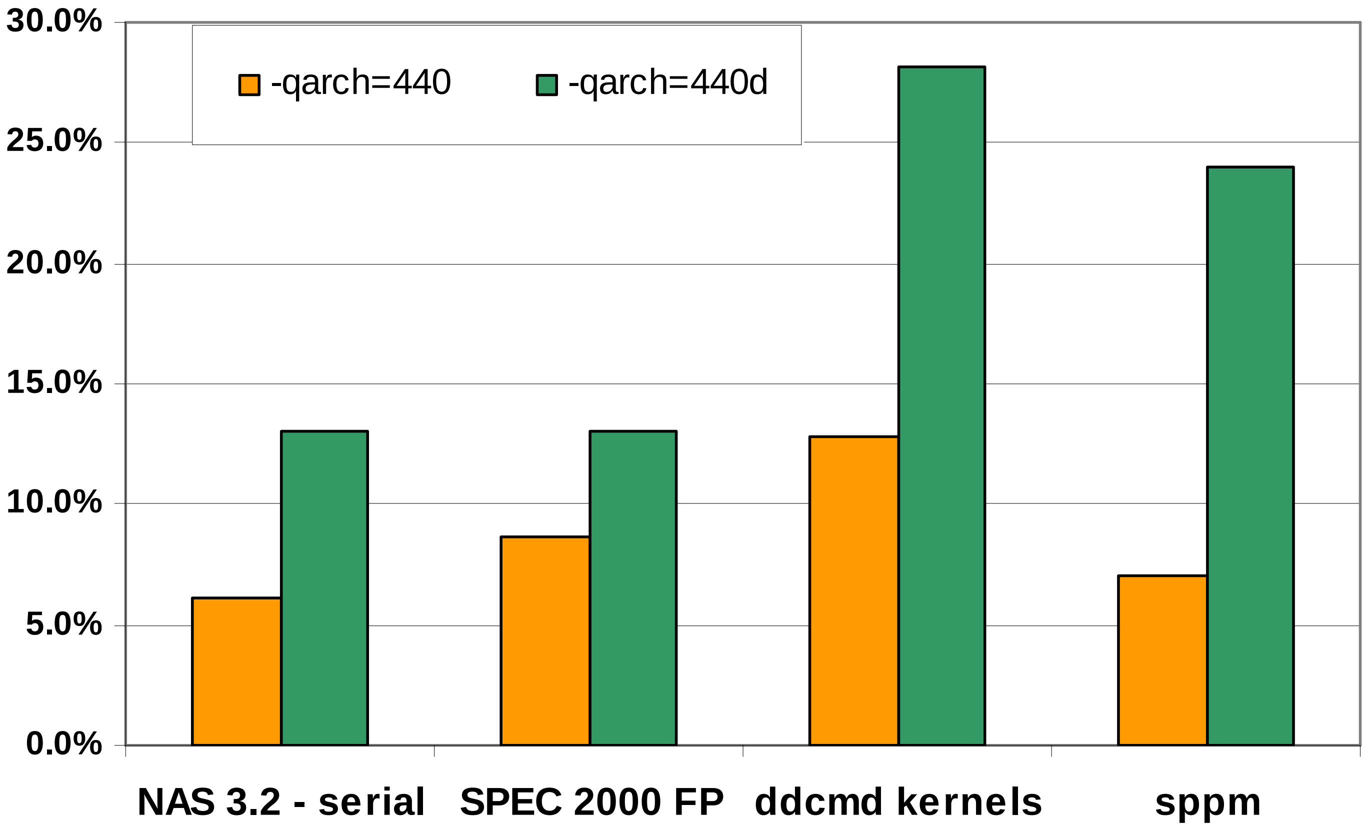
Blue Gene Specific Work Items in V8.0/10.1

- **New SIMDization features**
  - Mixedmode SIMDization: SIMDizing part of a loop without distributing the loop
- **SIMDization tuning items**
  - Enhanced interprocedural alignment analysis to track 16-byte compile-time alignment
  - Better alignment code generation to maximize load reuse across statements and across iterations
  - More reuse conscious loop distribution for SIMDization purposes

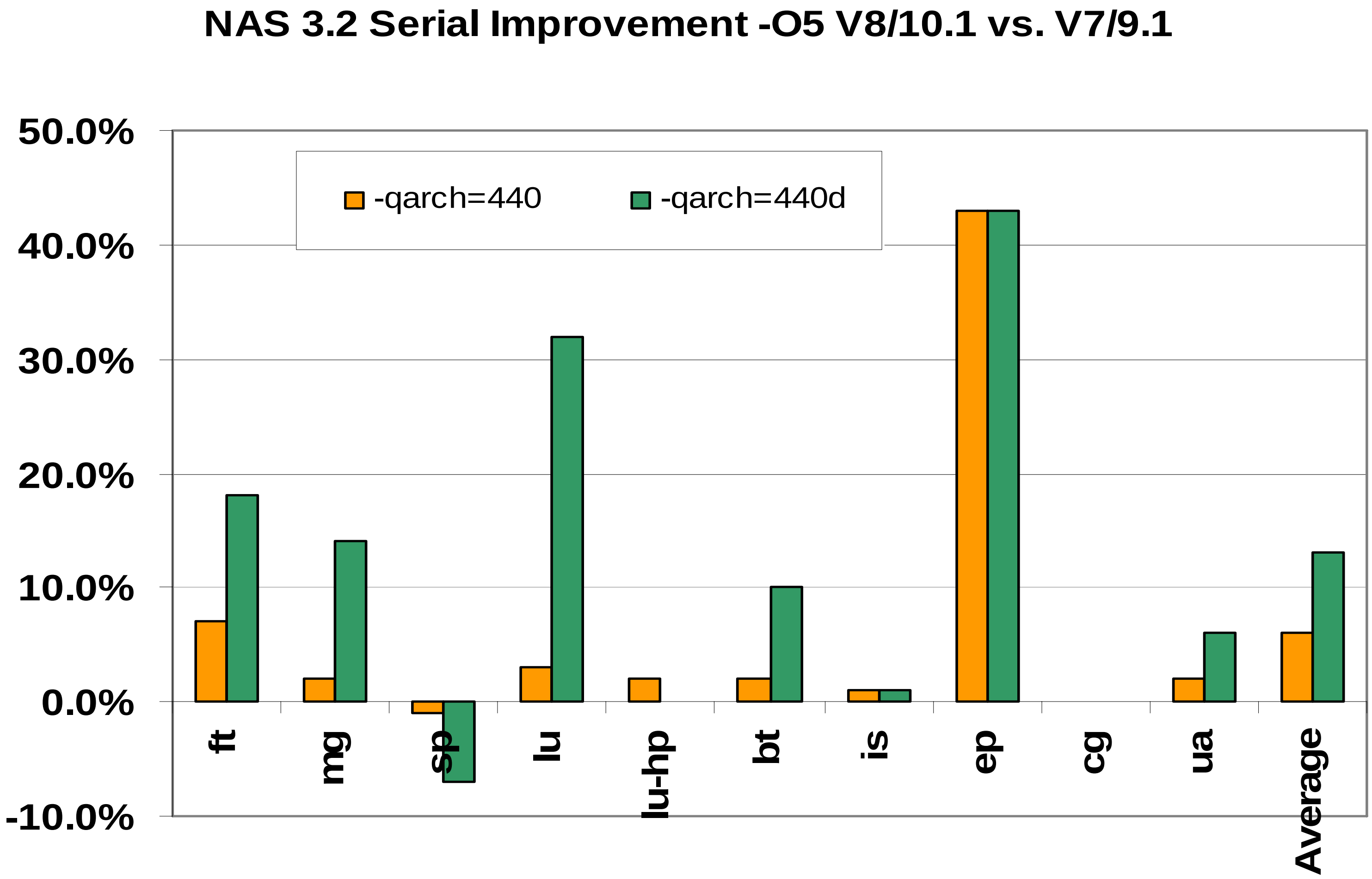
Outline

- IBM Blue Gene compilers update
- ➢**Performance results from V8.0/10.1**
- Generating parallel (SIMD) Code For Blue Gene
- What benefit can you expect from 440d?
- Future directions

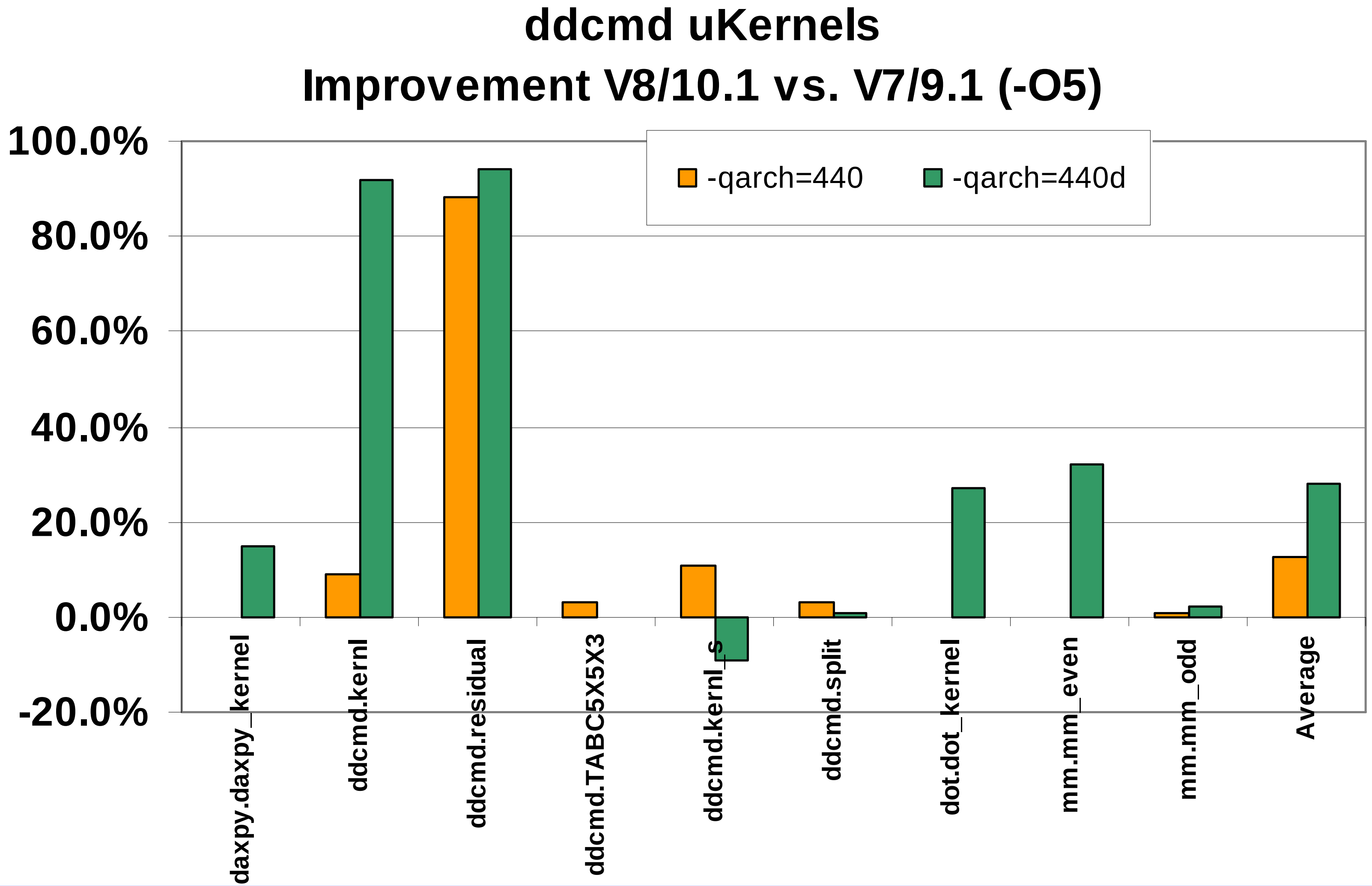
Improvement -O5 V8/10.1 vs. V7/9.1



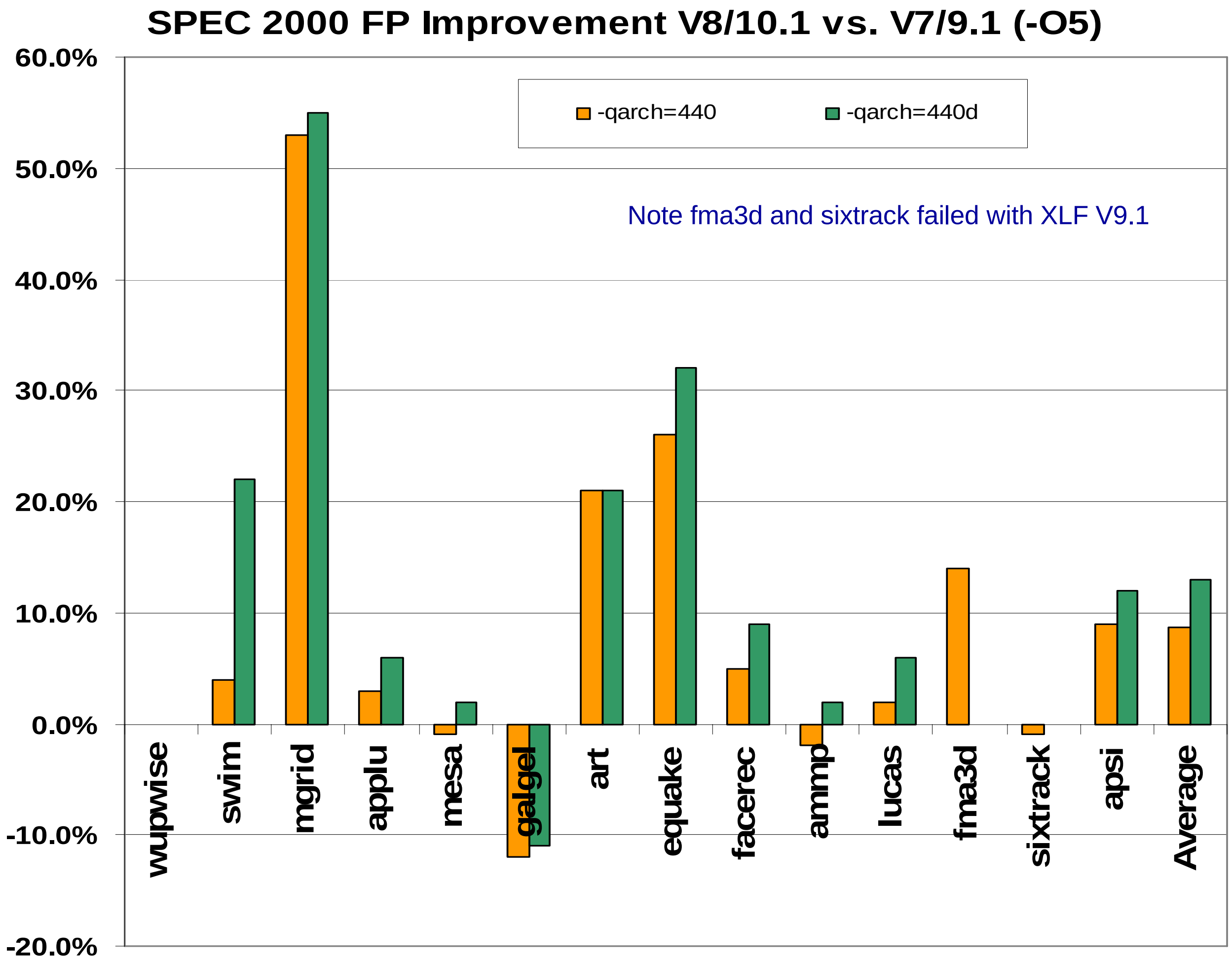
Improvement -O5 V8/10.1 vs. V7/9.1



Improvement -O5 V8/10.1 vs. V7/9.1

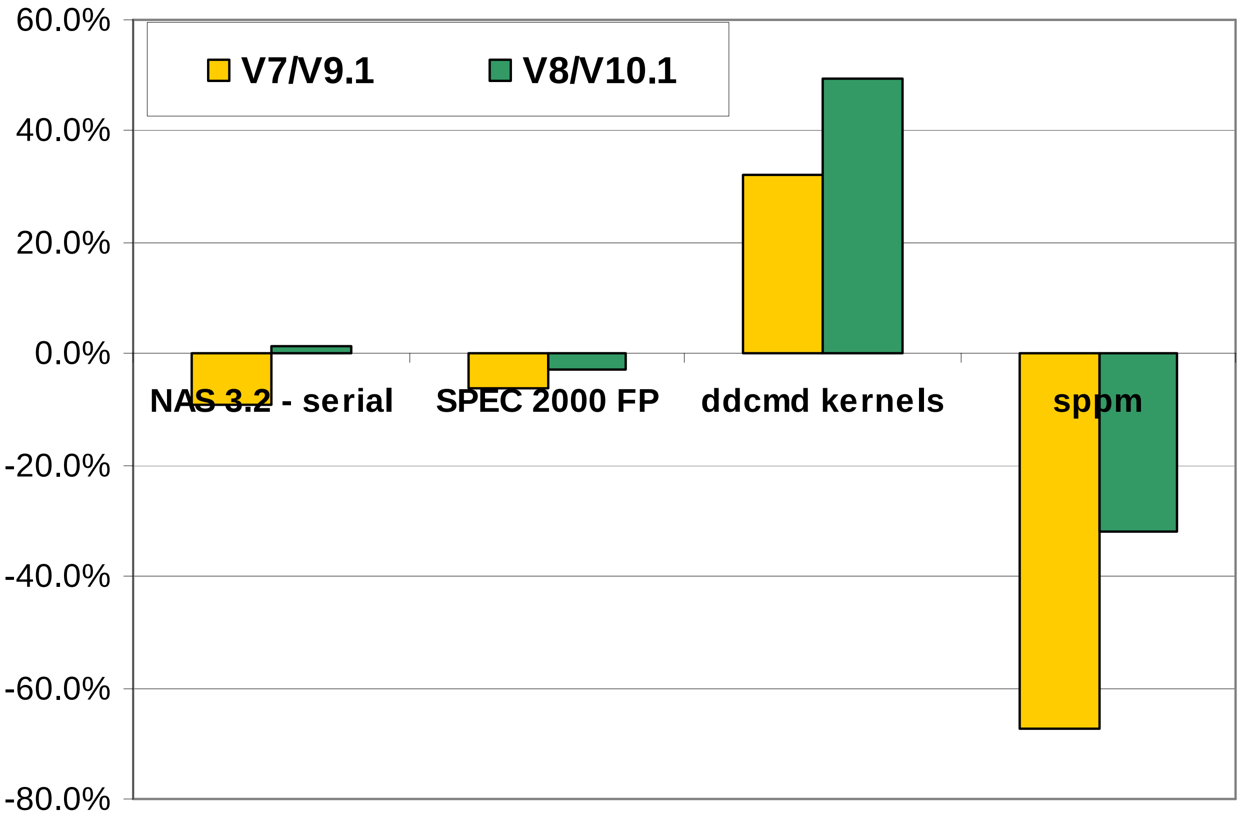


Improvement -O5 V8/10.1 vs. V7/9.1



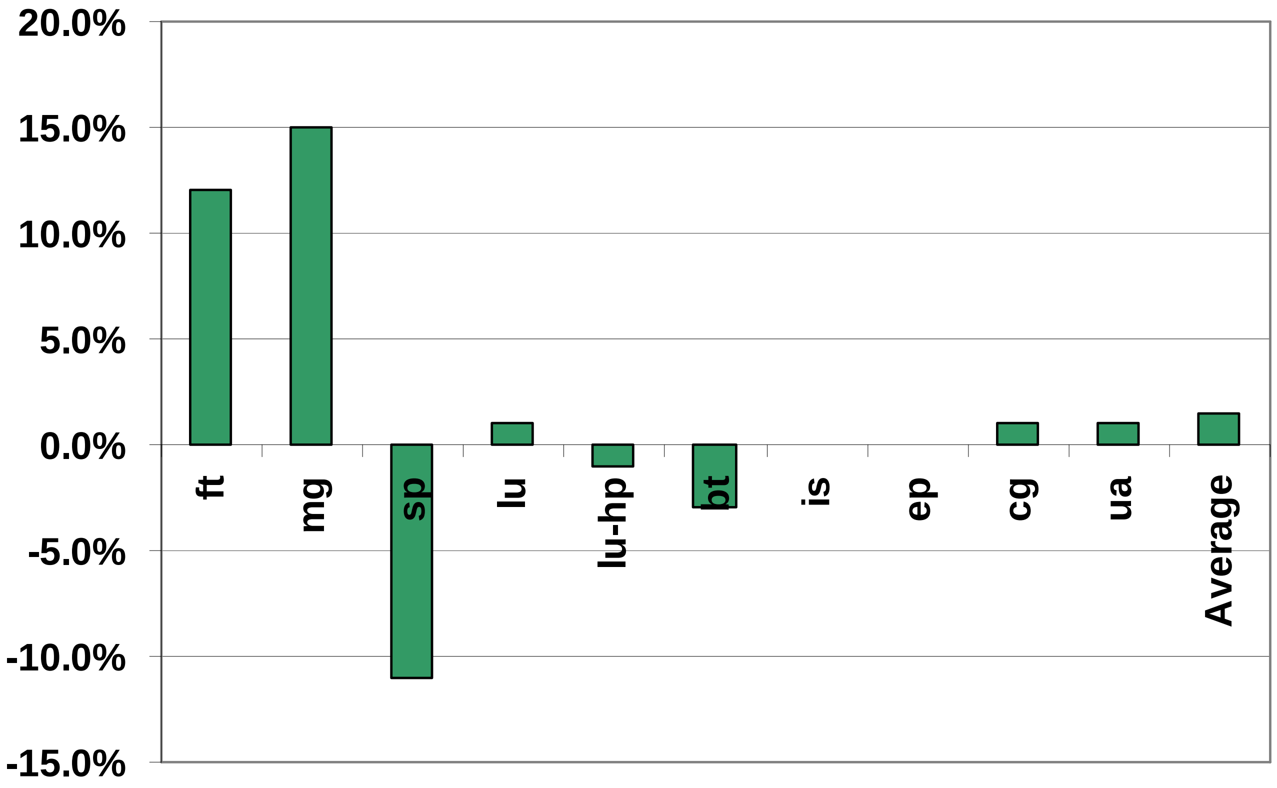
Note fma3d and sixtrack failed with XLF V9.1

Improvement -O5 -qarch=440d vs. -qarch=440

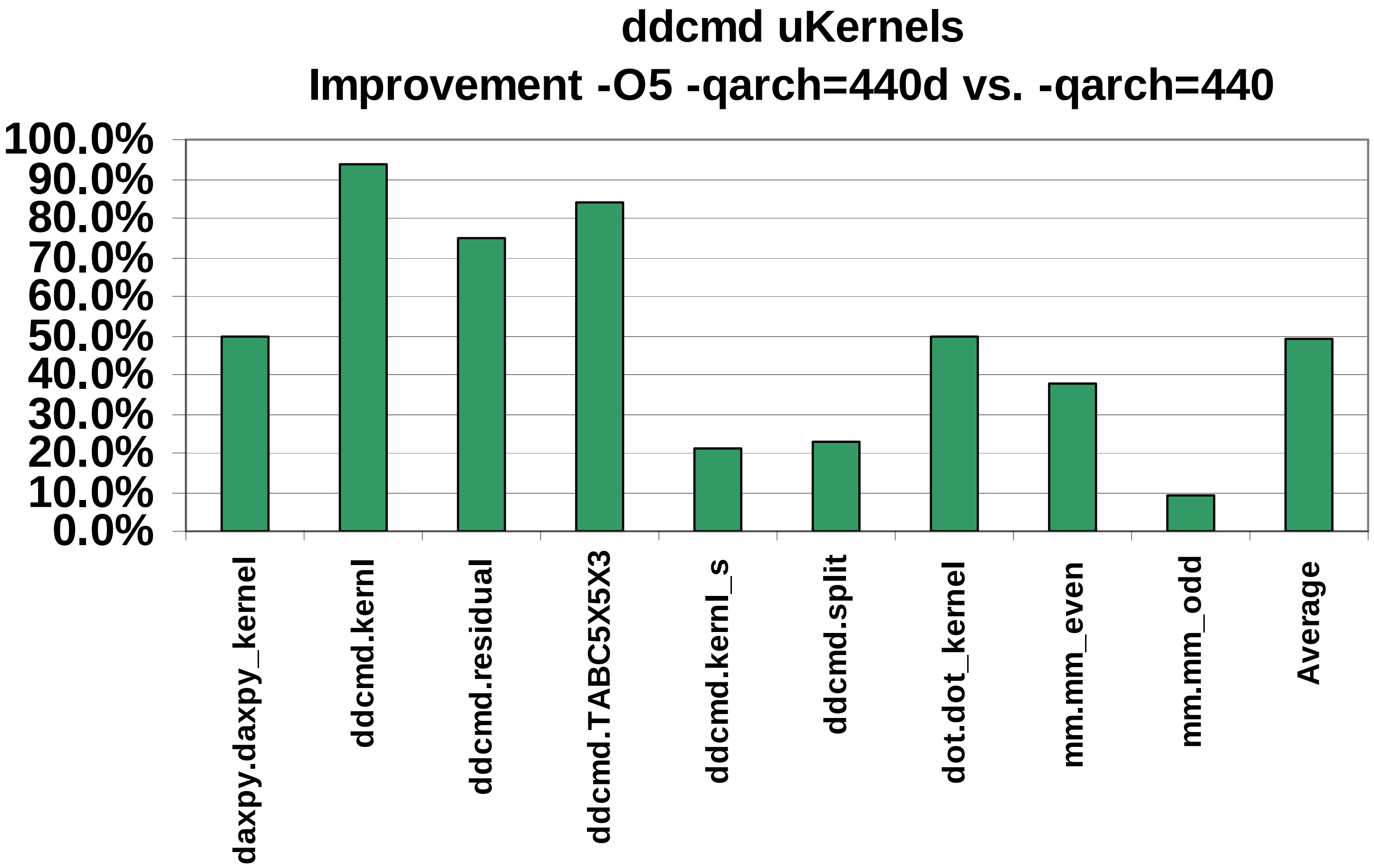


Improvement -O5 -qarch=440d vs. -qarch=440

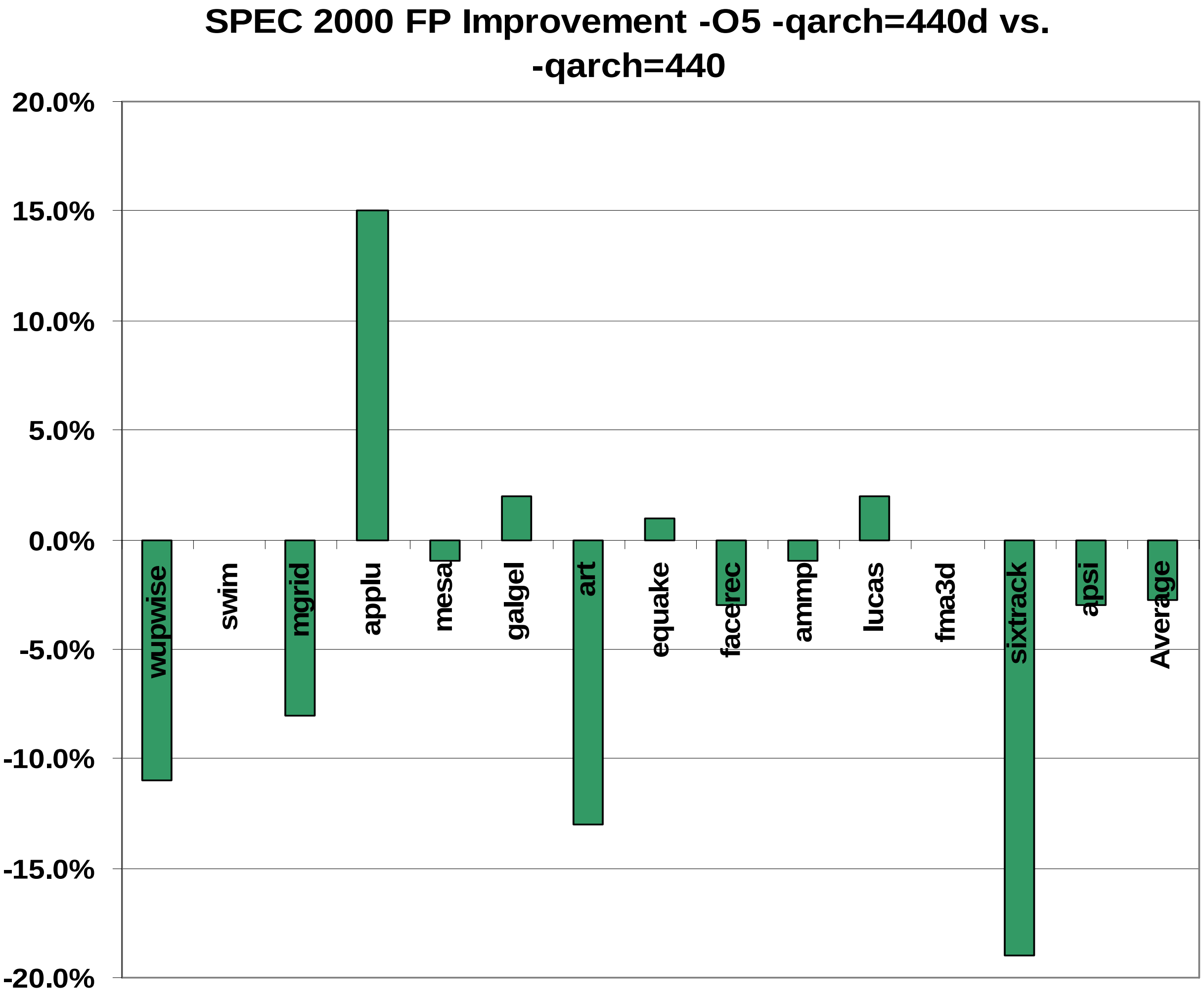
NAS 3.2 Serial Improvement -qarch=440d vs. -qarch=440



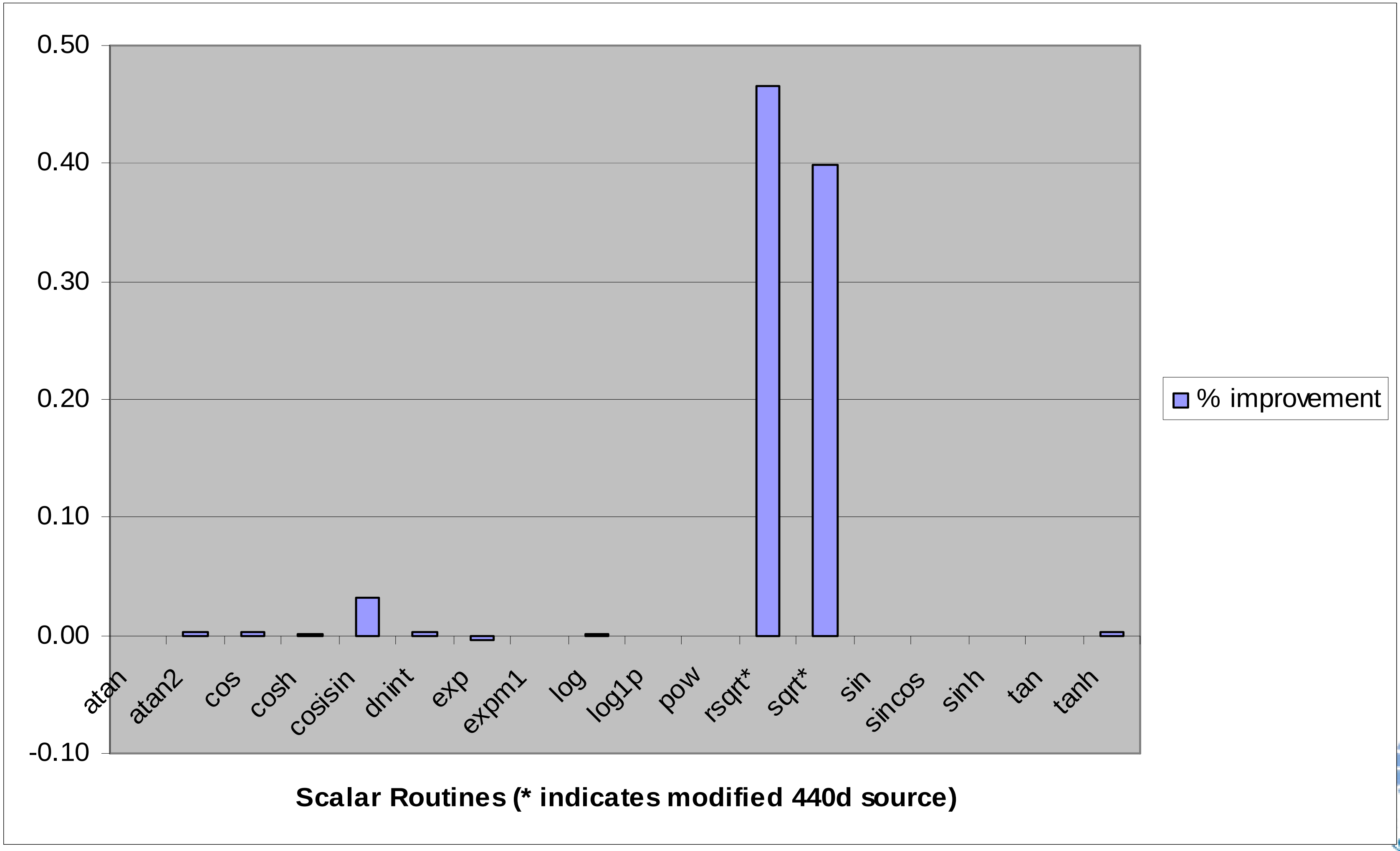
Improvement -O5 -qarch=440d vs. -qarch=440



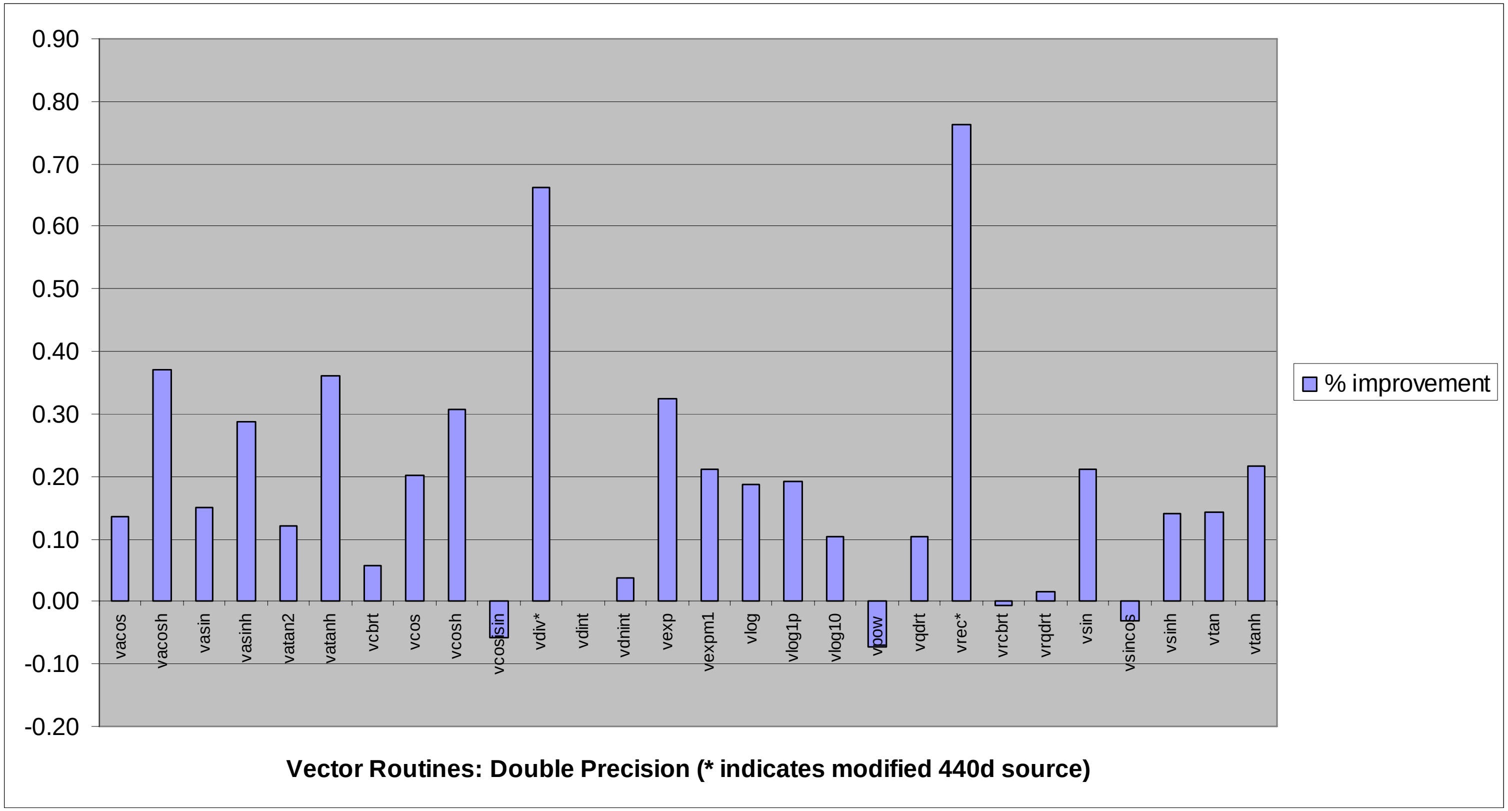
Improvement -O5 -qarch=440d vs. -qarch=440



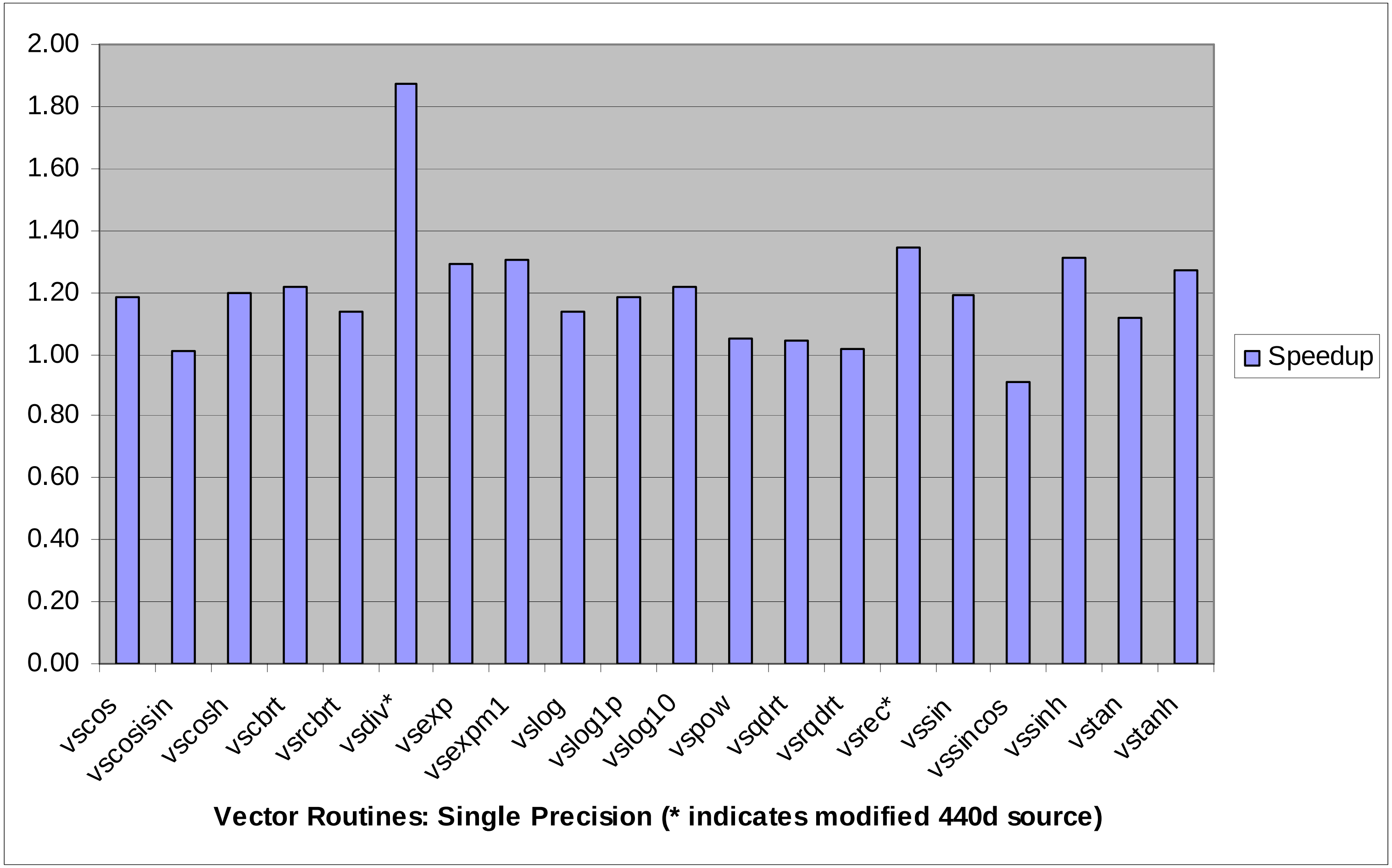
Improvement of MASS: 440d vs. 440



Improvement of MASS: 440d vs. 440



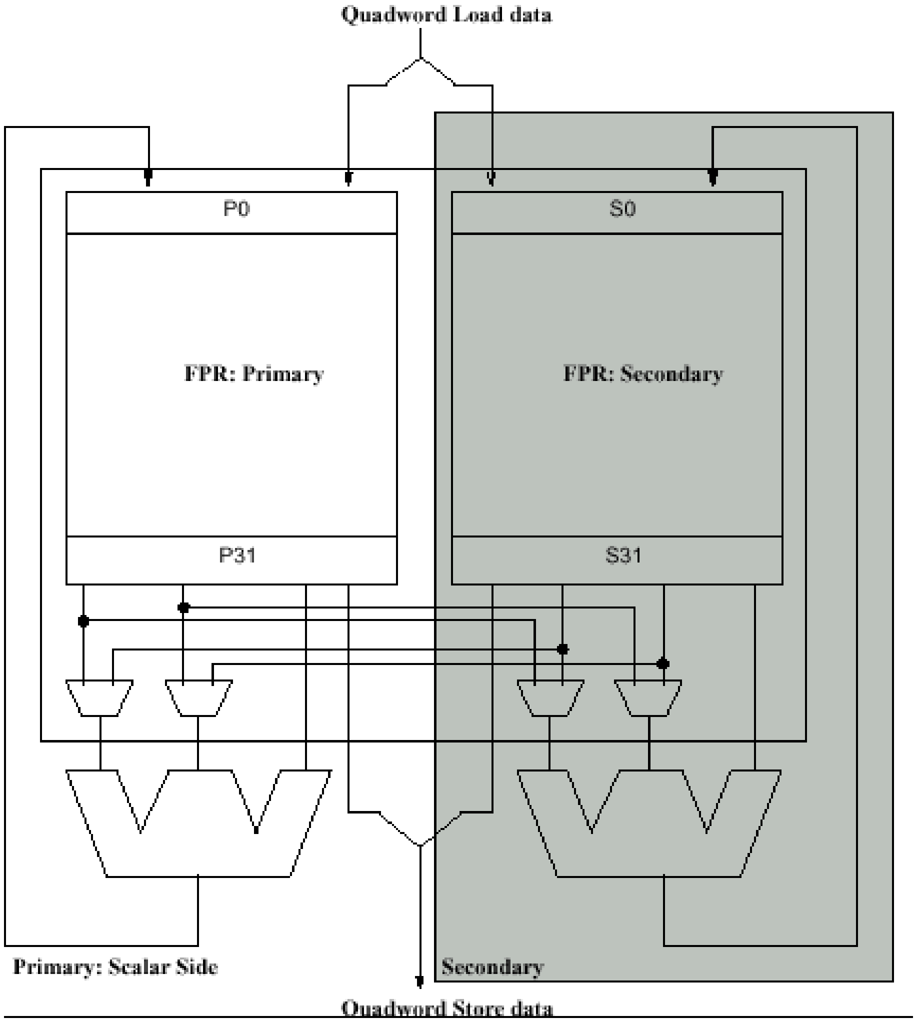
Improvement of MASS: 440d vs. 440



Outline

- IBM Blue Gene compilers update
- Performance results from V8.0/10.1
- ➢ Generating parallel (SIMD) Code For Blue Gene
- What benefit can you expect from 440d?
- Future directions

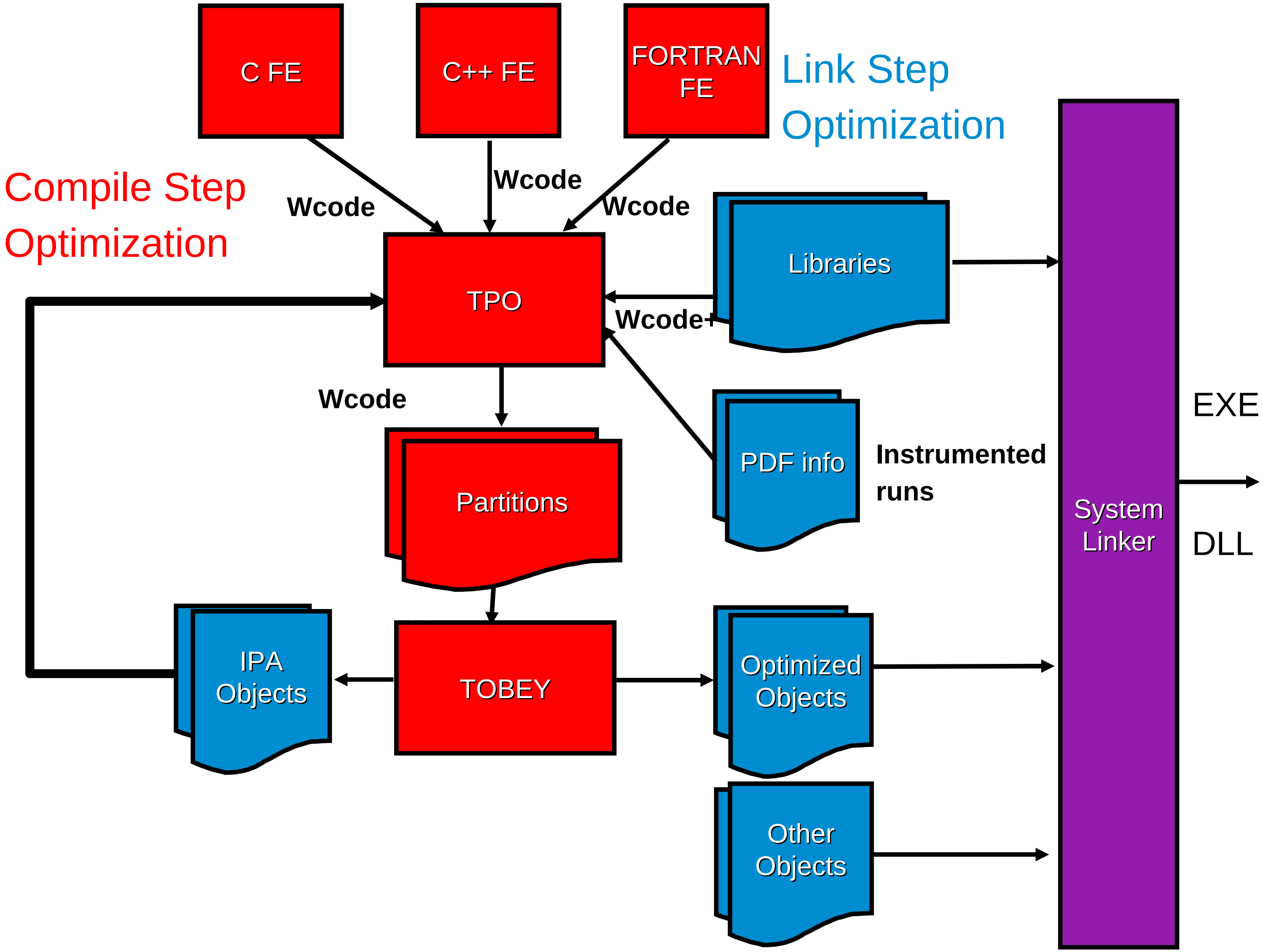
Blue Gene/L Dual Floating Point Unit “Double Hummer”



SIMDization Algorithm

- **S. Larsen and S. Amarasinghe. *Exploiting Superword Level Parallelism with Multimedia Instruction Sets.* In Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, B.C., June 2000**
- **Designed for MMX and AltiVec type instruction sets (symmetrical)**
- **Basic Algorithm:**
  - Find aligned loads/stores in a basic block and pair them up (require 16-byte alignment to be known)
  - Follow use/def chains to find pairs of isomorphic instructions that can be parallelized
  - Combine pairs of instructions to form larger groups
  - Estimate cost/benefit of SIMD code sequence
  - If positive, generate and schedule the instructions

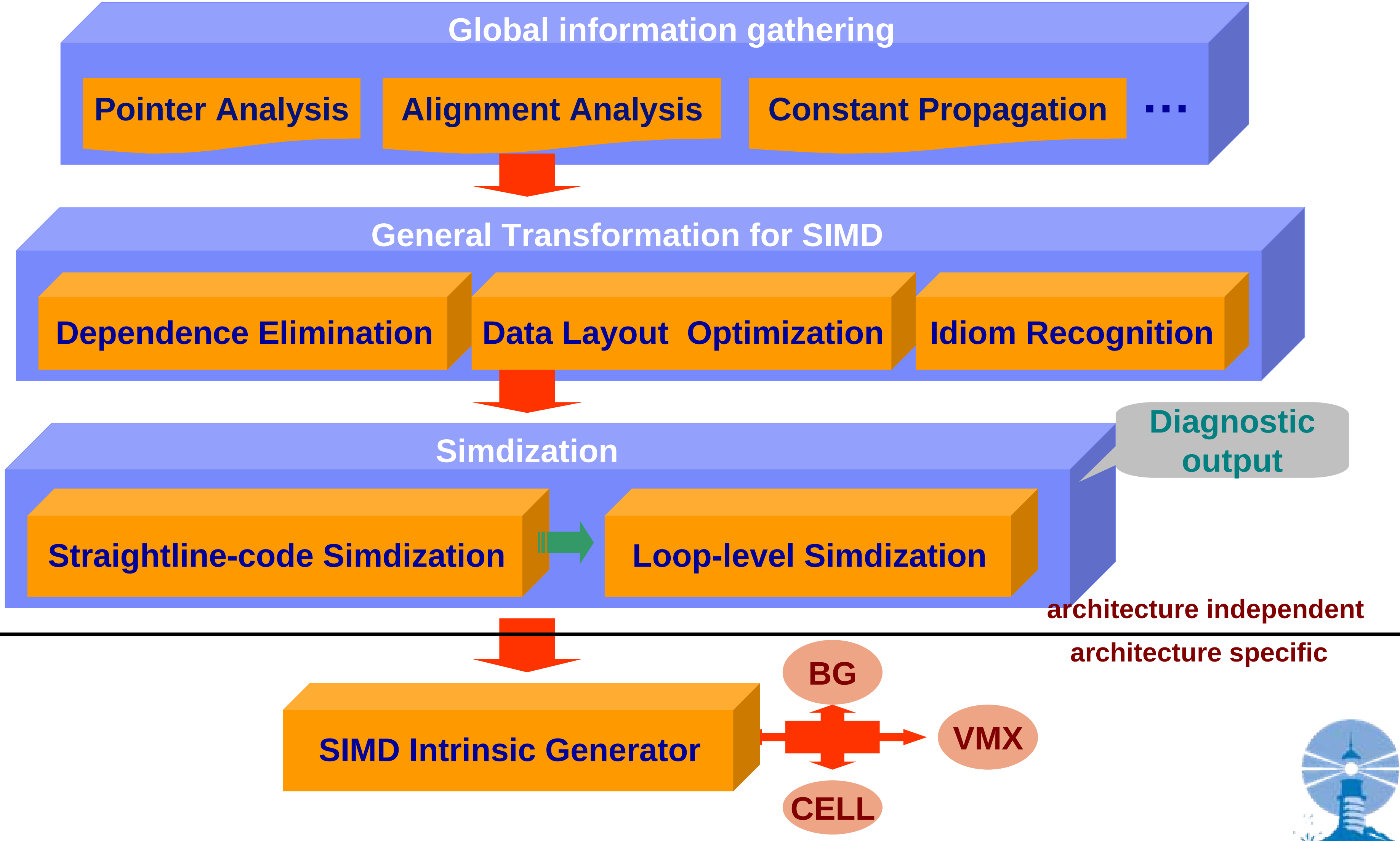
IBM Compiler Architecture



Where does SIMDization Occur?

- **In TPO (high-level inter-procedural optimizer)**
  - Active with –qhot, -O4, or –O5
  - TPO does most loop level/inlining/cloning optimizations
  - TPO will *version* loops for alignment or trip count
  - Some basic block SIMDization
- **In Tobey (low-level backend optimizer)**
  - complex arithmetic on double floats is an ideal target
  - other non-regular double floats are also packed
  - Tobey does most code motion/scheduling/machine specific optimizations
  - Tobey will try to generate SIMD code for all basic blocks

A Unified Simdization Framework (TPO)



TOBEY (Backend) Blue Gene SIMDization

- **SIMDization applied to all Basic Blocks**
  - May generate asymmetrical BG instructions
- **Structure copies using parallel load/store**
- **Reductions applied to TPO generated intrinsics**
  - __fpadd, __fpmadd, etc.
- **Local scheduling**
  - Primary/Secondary register interlock handling
- **Register allocation**
  - Spills using parallel load/stores
- **Loop unroller tries to be smart about BG load/store, parallelizable instructions**

Enabling SIMDization

- **Compile for the right machine**
  - -qarch=440d –qtune=440
- **Turn on the right optimizations**
  - -O/-O2: Basic Tobey (Backend) SIMDization
  - -O3: adds some loop optimizations, modulo scheduling
  - -O3 –qhot: adds TPO SIMDization, more loop opts, versioning
  - -O4,-O5 (compile and link): whole-program analysis & SIMDization
- **Tune your programs**
  - help the compiler with extra info (directive/pragmas)
  - use TPO compiler feedback (-qreport and -qxflag=diagnostic)
  - modify algorithms (stride-one memory accesses)

Disabling SIMDization

- **Compile for 440**
  - to completely disable SIMDization: -qarch=440 –qtune=440
- **Turn off the right optimizations**
  - compile for –qarch=440d –qtune=440
  - disable TPO simdization (keep Tobey simdization, with at least –O3)
    - for a loop: #pragma nosimd | !IBM* NOSIMD
    - completely: -qhot=nosimd
  - disable Tobey simdization (keep TPO simdization)
    - **not supported, may not work, try at your own risk**
    - completely: -qxflag=nhummer:ncmplx

When is Alignment Handling Needed?

- **All Aligned -> NONE**
  - for (i=0; i<100; i++)   a[i] = b[i] + c[i];

action: none

- **Misaligned, All Relatively Aligned -> NONE**
  - for (i=0; i<100; i++)   a[i+1] = b[i+1] + c[i+1];

action: peel first iter

- **Misaligned, Compile Time -> YES**
  - for (i=0; i<100; i++)   a[i+1] = b[i+1] + c[i];
  - a[i+1], b[i+1] relatively aligned, c[i] not relatively aligned

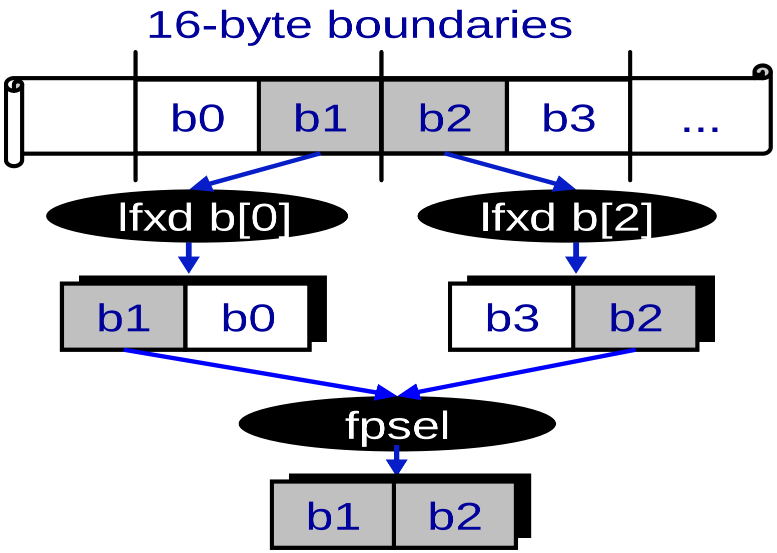
action: peel first iter, realign c[i]

- **Misaligned, Runtime -> NONE**
  - for (i=0; i<100; i++)   a[i+1] = b[i+1] + c[i+k];

action: versioning, peel first iter,
realign c[i+k] depending on versioning

Dealing with Misaligned Data on Blue Gene

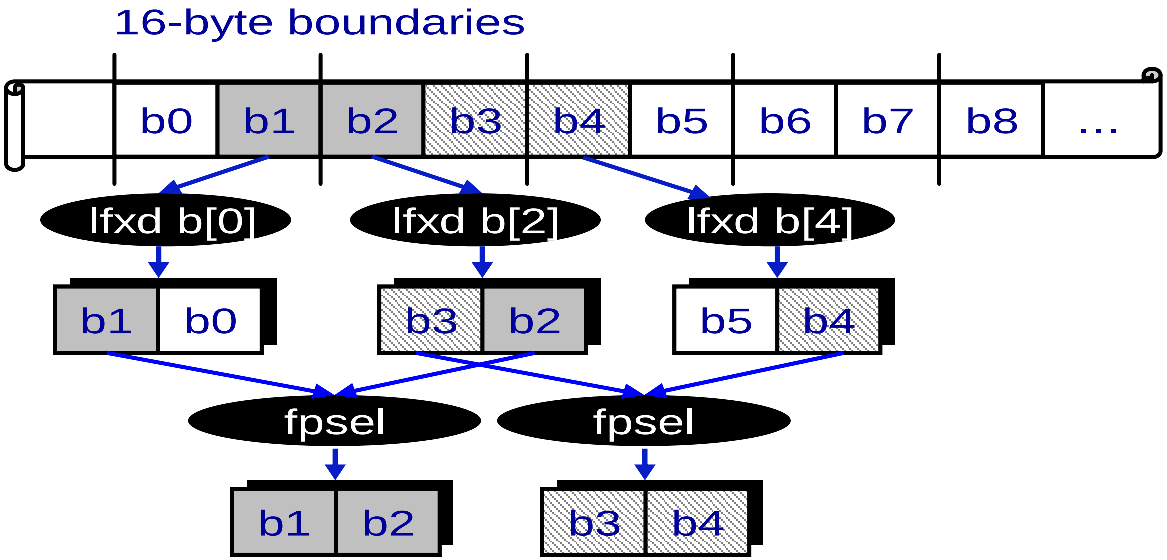
- **Load one misaligned quad:**



1 misaligned-quad load costs 2 aligned-quad cross-loads + 1 select

Dealing with Misaligned Data on Blue Gene (continued)

- **Load multiple consecutive misaligned quad data:**
  - reuse quad load from previous iteration



1 misaligned-quad load costs 1 aligned-quad cross-load + 1 select

Outline

- IBM Blue Gene compilers update
- Performance results from V10.1/8.0
- Generating SIMD code for Blue Gene

➢ **What benefit can you expect from 440d?**

- Future directions

What code benefits from double hummer?

- **According to John D. McCalpin, most “real” FP applications and Spec2000 FP have:**
  - Only 20% of instructions are floating point
  - 40% of instructions are load/store
  - Having 2 independent load/store and FP units allows 42% more instructions to issue per cycle
    - No data restrictions, but roughly represents upper bound
- **Code must have significant Instruction Level Parallelism (ILP)**
- **Code must comply with other restrictions of 440d**
- **Benefits are in terms of throughput, not latency**

Instruction Level Parallelism

- **Codes with significant ILP:**
  - Complex arithmetic
  - Vector operations
  - Matrix multiply
  - memcpy
- **Codes without significant ILP:**
  - Recurrence-relations (loop carried dependence)
  - Dependent calculations (non-loop or non-pipelinable)
  - Aliased loads & stores (pointers can be bad – use disjoint)

Example with ILP

```
subroutine addme (a,b,c,n)
real*8 a(n),b(n),c(n)
call alignx(16,a)
call alignx(16,b)
call alignx(16,c)
do 10 i = 1,n
a(i) = b(i) + c(i)
10       continue
end
```

Code generated at –O3

- **Unrolled by 4x, fully SIMDized, executes in 6 cycles (1.5 CPI)**

```
7|                               CL.57:
7| 000058 lfpdx    7C84339C   1      LFPL      fp4,fp36=b(gr4,gr6,0,trap=8)
7| 00005C fpadd    00601018   1      FPADD     fp3,fp35=fp0,fp32,fp2,fp34,fcr
7| 000060 lfpdx    7CA5339C   1      LFPL      fp5,fp37=c(gr5,gr6,0,trap=8)
7| 000064 stfpdx   7C23379C   1      SFPL      a(gr3,gr6,0,trap=8)=fp1,fp33
7| 000068 addi     38630020   1      AI        gr3=gr3,32
7| 00006C lfpdx    7C043B9C   1      LFPL      fp0,fp32=b(gr4,gr7,0,trap=24)
7| 000070 addi     38840020   1      AI        gr4=gr4,32
7| 000074 fpadd    00242818   1      FPADD     fp1,fp33=fp4,fp36,fp5,fp37,fcr
7| 000078 lfpdx    7C453B9C   1      LFPL      fp2,fp34=c(gr5,gr7,0,trap=24)
7| 00007C stfpdx   7C63479C   1      SFPL      a(gr3,gr8,0,trap=-8)=fp3,fp35
7| 000080 addi     38A50020   1      AI        gr5=gr5,32
0| 000084 bc       4320FFD4   0      BCT       ctr=CL.57,taken=100%(100,0)
```

Example without ILP (due to recurrence)

```
void p(double *p, const double *q) {
  #pragma disjoint (*p, *q)
  __alignx (16, p);  __alignx (16, q);
  for (int i=1; i<N; i++)
    p[i] = q[i] + p[i-1];
}
```

Code generated at –O3 -qhot

- **Unrolled by 4x, predictively commoned, not SIMDized, executes in 20 cycles (5 CPI)**

```
7|                              CL.64:
7| 000040 lfd      C8260008   1     LFL       fp1=q[]0.rns4.(gr6,8)
7| 000044 fadd     FC41002A   1     AFL       fp2=fp1,fp0,fcr
7| 000048 stfdu    DC050020   1     STFDU     gr5,p[]0.rns3.(gr5,32)=fp0
7| 00004C lfd      C8860010   1     LFL       fp4=q[]0.rns4.(gr6,16)
7| 000050 stfd     D8450008   1     STFL      p[]0.rns3.(gr5,8)=fp2
7| 000054 fadd     FC22202A   1     AFL       fp1=fp2,fp4,fcr
7| 000058 lfd      C8660018   1     LFL       fp3=q[]0.rns4.(gr6,24)
7| 00005C stfd     D8250010   1     STFL      p[]0.rns3.(gr5,16)=fp1
7| 000060 fadd     FC43082A   2     AFL       fp2=fp3,fp1,fcr
7| 000064 lfdu     CC060020   0     LFDU      fp0,gr6=q[]0.rns4.(gr6,32)
7| 000068 stfd     D8450018   1     STFL      p[]0.rns3.(gr5,24)=fp2
7| 00006C fadd     FC02002A   3     AFL       fp0=fp2,fp0,fcr
0| 000070 bc       4320FFD0   0     BCT       ctr=CL.64,taken=100%(100,0)
```

Alignment Restrictions

- **Parallel loads/stores important, most common resource limitation**
- **16-byte alignment needed for parallel loads/stores**
  - Loads crossing cache line boundary cause alignment traps: cost 1000s of cycles
  - 16-byte load from 8-byte boundary: 25% have alignment trap
  - Works well with complex? Sometimes (O4/O5 helps)
  - Real*8/double arrays must have:
    - 16-byte alignment (use alignx or O4/O5)
    - Stride-1 access patterns
    - Matching index's in loops

Example with Misalignment

```
void p(double *p, const double *q, const double *r) {
  #pragma disjoint (*p, *q, *r)
__alignx (16, p);  __alignx (16, q); __alignx (16, r);
  for (int i=1; i<N; i++)
    p[i] = q[i] + r[i-1];
}
```

Code generated at –O3 -qhot

- **Unrolled by 16x, partially SIMDized, did not use fpsel (2 CPI)**

```
7|                             CL.74:
7| 0000C4 lfpdx     7CC84B9C   1     LFPL      fp6,fp38=$.V.q[]0.rns5.1(gr8,gr9,0,trap=16)
7| 0000C8 lfpdx     7CE8EB9C   1     LFPL      fp7,fp39=$.V.q[]0.rns5.1(gr8,gr29,0,trap=-64)
7| 0000CC lfsdx     7CA6E99C   1     LFL       fp37=$.V.r[]0.rns4.0(gr6,gr29,0,trap=-64)
7| 0000D0 stfpdx    7C4B579C   1     SFPL      $.V.p[]0.rns3.2(gr11,gr10,0,trap=32)=fp2,fp34
7| 0000D4 fpadd     00844018   1     FPADD     fp4,fp36=fp4,fp36,fp8,fp40,fcr
7| 0000D8 stfpdux   7C2B3FDC   1     SFPLU     gr11,$.V.p[]0.rns3.2(gr11,gr7,0,trap=128)=fp1,fp33
7| 0000DC lfsdx     7C66499C   1     LFL       fp35=$.V.r[]0.rns4.0(gr6,gr9,0,trap=16)
7| 0000E0 lfd       C846FFC8   1     LFL       fp2=$.V.r[]0.rns4.0(gr6,-56)
7| 0000E4 stfpdx    7C0BE79C   1     SFPL      $.V.p[]0.rns3.2(gr11,gr28,0,trap=-112)=fp0,fp32
7| 0000E8 stfpdx    7C8BF79C   1     SFPL      $.V.p[]0.rns3.2(gr11,gr30,0,trap=-80)=fp4,fp36
7| 0000EC fpadd     01072818   1     FPADD     fp8,fp40=fp7,fp39,fp5,fp37,fcr
7| 0000F0 lfpdx     7D28DB9C   1     LFPL      fp9,fp41=$.V.q[]0.rns5.1(gr8,gr27,0,trap=-48)
7| 0000F4 lfsdx     7C46D99C   1     LFL       fp34=$.V.r[]0.rns4.0(gr6,gr27,0,trap=-48)
6| 0000F8 lfd       C806FFD8   1     LFL       fp0=$.V.r[]0.rns4.0(gr6,-40)
7| 0000FC lfpdx     7C28539C   1     LFPL      fp1,fp33=$.V.q[]0.rns5.1(gr8,gr10,0,trap=32)
7| 000100 stfpdx    7D0BEF9C   1     SFPL      $.V.p[]0.rns3.2(gr11,gr29,0,trap=-64)=fp8,fp40
7| 000104 fpadd     00E91018   1     FPADD     fp7,fp39=fp9,fp41,fp2,fp34,fcr
... (code omitted) ...
0| 000164 bc        4320FF60   0     BCT       ctr=CL.74,taken=100%(100,0)
```

Example without Alignx

```
void p(double *p, const double *q, const double *r) {
  #pragma disjoint (*p, *q, *r)
  //no alignment information available
  for (int i=1; i<N; i++)
    p[i] = q[i] + r[i];
}
```

Code generated at –O3 -qhot

```
if (p,q,r aligned) {
  Execute SIMDized version (with parallel loads/stores)
} else { // 3 CPI
    6|                              CL.218:
    6| 0002A0 lfd       C8840008   1     LFL       fp4=q[]0.rns2.(gr4,8)
    6| 0002A4 lfd       C8650008   1     LFL       fp3=r[]0.rns1.(gr5,8)
    6| 0002A8 stfd      D8430008   1     STFL      p[]0.rns0.(gr3,8)=fp2
    6| 0002AC stfsdx    7C433D9C   1     STFL      p[]0.rns0.(gr3,gr7,0,trap=16)=fp34
    6| 0002B0 fpadd     00A10018   1     FPADD     fp5,fp37=fp1,fp33,fp0,fp32,fcr
    6| 0002B4 lfsdx     7C84399C   1     LFL       fp36=q[]0.rns2.(gr4,gr7,0,trap=16)
    6| 0002B8 lfsdx     7C65399C   1     LFL       fp35=r[]0.rns1.(gr5,gr7,0,trap=16)
    6| 0002BC lfd       C8240018   1     LFL       fp1=q[]0.rns2.(gr4,24)
    6| 0002C0 lfd       C8050018   1     LFL       fp0=r[]0.rns1.(gr5,24)
    6| 0002C4 stfd      D8A30018   1     STFL      p[]0.rns0.(gr3,24)=fp5
    6| 0002C8 fpadd     00441818   1     FPADD     fp2,fp34=fp4,fp36,fp3,fp35,fcr
    6| 0002CC stfsdux   7CA345DC   1     STFDU     gr3,p[]0.rns0.(gr3,gr8,0,trap=32)=fp37
    6| 0002D0 lfsdux    7C2441DC   1     LFDU      fp33,gr4=q[]0.rns2.(gr4,gr8,0,trap=32)
    6| 0002D4 lfsdux    7C0541DC   1     LFDU      fp32,gr5=r[]0.rns1.(gr5,gr8,0,trap=32)
    0| 0002D8 bc        4320FFC8   0     BCT       ctr=CL.218,taken=100%(100,0)
}
```

Other restrictions of 440d

- **Asymmetric instructions**
  - Good for complex arithmetic, hard to generate automatically
- **Double precision arithmetic only**
- **IEEE exceptions unavailable**
  - -qflttrap disables SIMDization
- **Parallel loads/stores are index-form**
  - Displacements must be held in index register
  - GPR register pressure increases
  - May require additional fixed-point arithmetic

Throughput vs. Latency

- **Parallel code improves execution throughput**
  - Can reduce execution unit usage by 1/2
- **Latency is unaffected**
  - Effectively overlaps 2 calculations, but no change in latency of calculation
- **SIMDization may show little or no benefit in latency-dependent code**

Example of Latency-Dependent Code

```
void foo(_Complex double* out, _Complex double* in1,
               _Complex double* in2) {
  #pragma disjoint (*out, *in1, *in2)
   __alignx(16, out); __alignx(16, in1); __alignx(16, in2);
  *out = (*in1) * (*in2);
}
```

Code generated at –O3 –qhot –qarch=440d

- **Fully SIMDized, executes in 17 cycles**

```
  | 000000                          PDEF      foo                                      Issue Cycle
 2|                                 PROC       out,in1,in2,gr3-gr5
 5| 000000 lfpdx    7C00239C    1   LFPL       fp0,fp32=20:21:in1[]0.rns5.#re(gr4,0)     0
 0| 000004 addis    3C800000    1   LA         gr4=.+CONSTANT_AREA%HI(gr2,0)             0
 5| 000008 lfpdx    7C402B9C    1   LFPL       fp2,fp34=22:23:in2[]0.rns4.#re(gr5,0)     1
 0| 00000C addi     38840000    1   LA         gr4=+CONSTANT_AREA%LO(gr4,0)              1
 5| 000010 lfpsx    7C20231C    1   LFPS       fp1,fp33=+CONSTANT_AREA(gr4,0)            2
 5| 000014 fxcxnpma 1022083A    1   FXCXNPMA   fp1,fp33=fp1,fp33,fp32,fp0,fp34,fp34,fcr  6
 5| 000018 fxcpmadd 00020824    1   FXPMADD    fp0,fp32=fp1,fp33,fp0,fp32,fp2,fp2,fcr    11
 5| 00001C stfpdx   7C001F9C    1   SFPL       24:25:out[]0.rns3.#re(gr3,0)=fp0,fp32     16
 6| 000020 bclr     4E800020    0   BA         lr                                        16
```

Code generated at –O3 –qhot –qarch=440

- **Un-SIMDized, executes in 17 cycles**

```
 | 000000                          PDEF     foo                              Issue Cycle
2|                                 PROC      out,in1,in2,gr3-gr5
5| 000000 lfd      C8040008   1    LFL       fp0=in1[]0.rns5.#im(gr4,8)      0
5| 000004 lfd      C8250000   1    LFL       fp1=in2[]0.rns4.#re(gr5,0)      1
5| 000008 lfd      C8450008   1    LFL       fp2=in2[]0.rns4.#im(gr5,8)      2
5| 00000C lfd      C8640000   1    LFL       fp3=in1[]0.rns5.#re(gr4,0)      3
5| 000010 fmul     FC800072   1    MFL       fp4=fp0,fp1,fcr                 5
5| 000014 fmul     FC0000B2   1    MFL       fp0=fp0,fp2,fcr                 6
5| 000018 fmadd    FC4320BA   3    FMA       fp2=fp4,fp3,fp2,fcr             10
5| 00001C fmsub    FC030078   1    FMS       fp0=fp0,fp3,fp1,fcr             11
5| 000020 stfd     D8430008   0    STFL      out[]0.rns3.#im(gr3,8)=fp2      15
5| 000024 stfd     D8030000   0    STFL      out[]0.rns3.#re(gr3,0)=fp0      16
6| 000028 bclr     4E800020   0    BA        lr                              16
```

Other Limitations

- **Benefit function:**
  - Difficult to estimate benefit of SIMDization in complicated code, without having carried out all steps of compilation
  - Asymmetrical instructions greatly complicate evaluation, may require primary/secondary moves
    - Cost of moving between primary and secondary registers is 5 cycles
  - Tuning of cost/benefit heuristics is actively ongoing

What Benefit Can You Expect From 440d?

- **Answer: It depends…**
  - Performance is application-specific
  - Should expect between 0% and 100% speedup
  - Unfortunately, experience shows some degradations:
    - Register spilling
    - Primary/secondary moves unexpectedly needed
    - Cost/benefit heuristics imperfect – often too much SIMDization

Outline

- IBM Blue Gene compilers update
- Performance results from V10.1/8.0
- Generating SIMD code for Blue Gene
- What benefit can you expect from 440d?

➢ **Future directions**

Future Directions

- Continuing to tune for performance in upcoming compiler fixpacks (PTFs)
  - Small, safe items in PTFs
  - Larger features in next release
- Tuning to ensure 440d >= 440
  - Cost/benefit heuristics
  - Register pressure heuristics
- Long list of opportunities for SIMD improvement (good input from IBM Japan)

Future Directions (continued)

– **Improved passing/returning of structures to remove redundant stores (from BlueMatter test cases)**

– **Optimization tuning:**

- **SIMD generation, folding**
- **Interprocedural propagation of alignment information**
- **Misaligned SIMD generation**
- **Software Prefetching (using dcbt, dcbz)**
- **Inlining**
- **Loop distribution**
- **Loop unrolling**
- **Register allocation**
- **Instruction scheduling (global, local, modulo)**

More Information

- **White Paper by Mark Mendell: “Exploiting the Dual FPU in Blue Gene/L” available at:**
  - http://www-1.ibm.com/support/docview.wss?uid=swg27007511
- **Mark Mendell’s presentation at 2nd Workshop:**
  - http://www.epcc.ed.ac.uk/BGworkshop/PROCEEDINGS/22_MarkMendell.pdf

Questions?

<following slides just for reference>

FORTRAN 2003 Support in XLF V10.1

- **Data manipulation enhancements**
  - ALLOCATABLE components (except resizing on assignment)
  - INTENT specifications of pointer arguments
  - PROTECTED attribute and statement
  - VALUE attribute and statement
  - procedure declaration statement (PROCEDURE statement)
  - **relaxed specification expression**
- **Support for IEC 60559 (IEEE 754) exceptions and arithmetic**
  - IEEE_EXCEPTIONS, IEEE_ARITHMETIC and IEEE_FEATURES intrinsic modules
- **Input/output enhancements**
  - stream access (allows access to a file without reference to any record structure)
  - the FLUSH statement
  - the NEW_LINE intrinsic
  - access to input/output error messages (IOMSG= specifier on data-transfer operations, file-positioning, FLUSH and file inquiry statements)
  - **BLANK= and PAD= specifiers on READ statement**
  - **DELIM= specifier on WRITE statement**
- Enumerations and enumerators
- Procedure pointers (except PASS attribute, declaring intrinsic procedure)

FORTRAN 2003 Support in XLF V10.1 (cont’d)

- **Derived-type enhancements**
  - mixed component accessibility (allow PRIVATE and PUBLIC attribute on derived type components)
- **Interoperability with C programming language**
  - ISO_C_BINDING intrinsic module (except C_F_PROCPOINTER)
  - BIND attribute and statement
- **The ASSOCIATE construct**
- **Scoping enhancement**
  - the ability to control host association into interface bodies (IMPORT statement)
- **Enhancement integration with the host operating system**
  - access to command line arguments (COMMAND_ARGUMENT_COUNT, GET_COMMAND_ARGUMENT, and GET_ENVIRONMENT_VARIABLE intrinsics)
  - access to the processor's error messages (IOMSG= specifier)
  - ISO_FORTRAN_ENV intrinsic module

GNU C/C++ Compatibility Enhancements

Full list of GNU C/C++ compatibility enhancements in XL C/C++ V8.0 can be found here:
http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=/com.ibm.xlcpp8a.doc/language/ref/gcc_cext.htm

- Labels as values / computed goto
- Nested functions (C only)
- Naming types
- Conditionals with omitted operands
- Zero length arrays

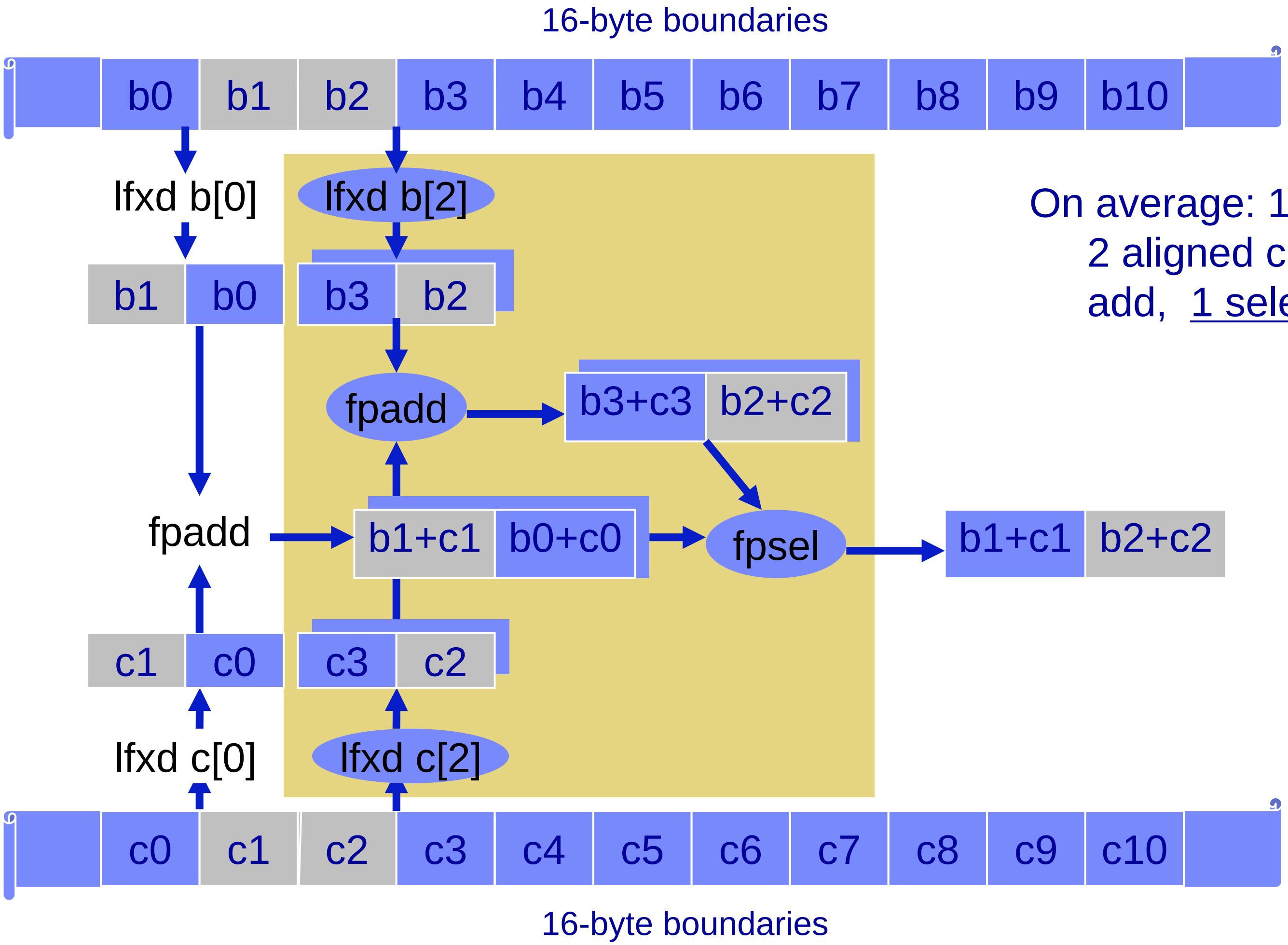
- Labeled elements (C only)
- Case ranges (C only)
- Cast to union (C only)
- Function Attributes
  - Support

    Noinline, always_inline, format, format_arg, section
  - Accept and ignore

    used

Minimizing Data Reorganization Overhead

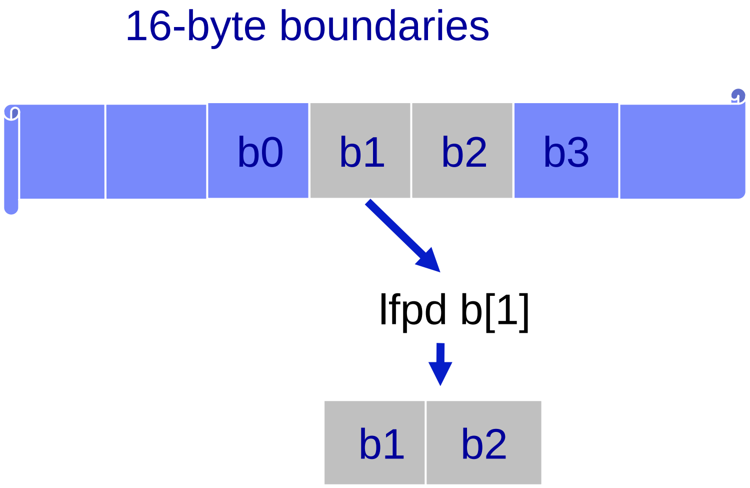
- **for (i=0; i<100; i++) a[i] = b[i+1] + c[i+1] ;**

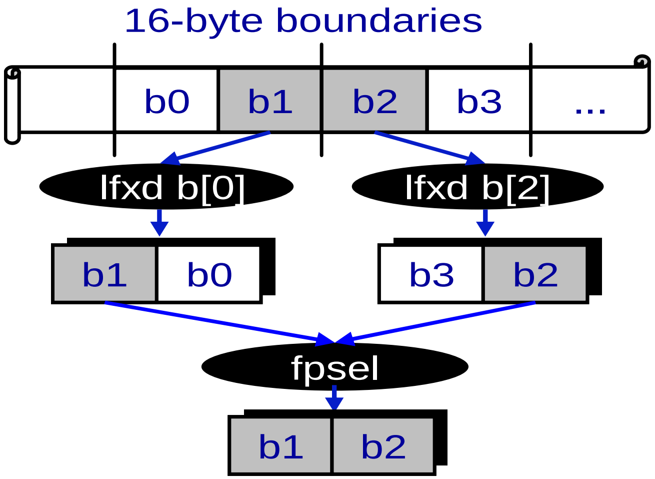


On average: 1 aligned store, 2 aligned cross-loads, 1 add, 1 select

Issues with Runtime Alignment

- **Depending on the alignment, different code sequences may be generated**
  - When alignment is runtime, the compiler does not know which code sequence to generate


16-byte boundaries
b0
b1
b2
b3
lfpd b[1]
b1
b2


16-byte boundaries
b0
b1
b2
b3
...
lfxd b[0]
lfxd b[2]
b1
b0
b3
b2
fpsel
b1
b2