AMD Opteron™ & PGI: "Enabling the Worlds Fastest LS-DYNA Performance"

Tim Wilkens Ph.D.
Member of Technical Staff
tim.wilkens@amd.com

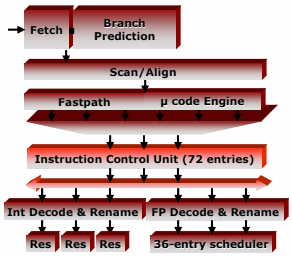
Agenda

- **Architecture – Opteron, Itanium, XeonEMT**
- **PGI Compiler – DYNA driven enhancements**
- **Performance – Neon and 3-Car Model**

Architecture Agenda

Opteron = Execution + Memory Access + IO

- **Customer Centric 64-bit Computing**
  - Instruction Decoding
  - Processors with Artificial Intelligence

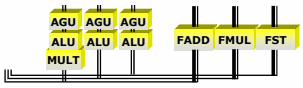


October 4, 2004 | Computation Products Group | 3

Architecture Agenda

Opteron = Execution + Memory Access + IO

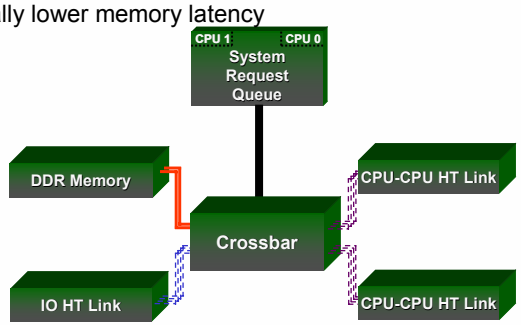
- **Not all X86 Processors are created =**
  - RISC Cores – scrupulous instruction preference



October 4, 2004 | Computation Products Group | 4

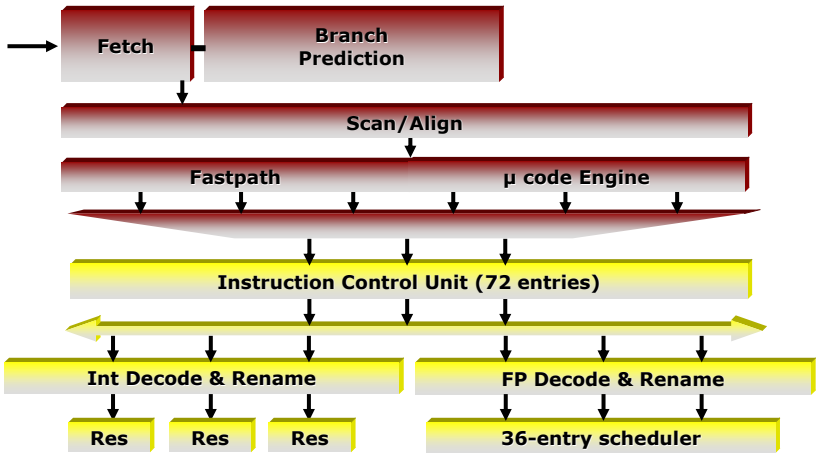
Architecture Agenda
Opteron = Execution + Memory Access + IO

- **Scalable Memory Bandwidth and IO**
  - physical memory scales with CPU #
  - memory bandwidth scales with CPU #
  - increased single threaded memory bandwidth
  - memory latency does not scale with CPU #
  - dramatically lower memory latency



October 4, 2004 | Computation Products Group | 5

Customer Centric 64-bit Computing
Opteron vs Itanium



October 4, 2004 | Computation Products Group | 6

Customer Centric 64-bit Computing
Opteron vs Itanium

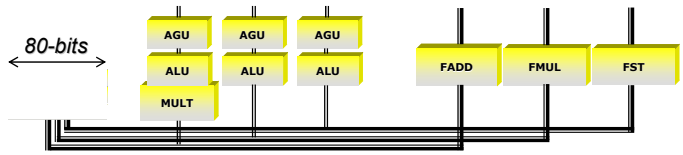
- **Progressive 64-bit approach: *32-bit instruction + prefix byte***
  - leverages x86 compiler technology – ***reliable compilers, port easily***
  - code size increase is minimal (~5%) – ***large caches not required***
- **x86 CPUs = RISC cores + CISC→RISC instruction decoders**
  - provides x86 processors high clock frequency and legacy compatibility
  - processor not compiler manages RISC core - ***recompile rarely***
  - Itanium is a slave to the compiler - ***recompile often***
- **out-of-order execution and register renaming**
  - Opteron manages it's registers intelligently – ***less compiler reliant***
  - Itanium requires the compiler to think for it – ***strong compiler reliance***

Both Opteron and Itanium are RISC, but Opteron doesn't require *reinventing compilers*, *large caches* & *a mint to purchace*

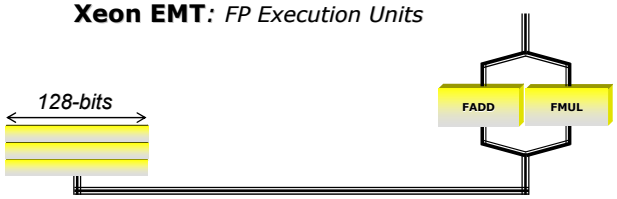
October 4, 2004 | Computation Products Group | 7

All X86 RISC Cores aren't created =
Opteron vs Xeon EMT

Opteron: *INT and FP Execution Units*



Xeon EMT: *FP Execution Units*



October 4, 2004 | Computation Products Group | 8

All X86 RISC Cores aren't created =
Opteron vs Xeon EMT

- **# of int pipes and pipeline depth impact integer throughput**
  - Opteron has 3 integer pipes – ***+50% reg,reg move thoughput***
  - Opteron has 3 **ALU**/**AGU** units – ***+50% +,-,logical, shift throughput***
  - # pipeline stages differs – ***shorter instruction execution latency***
- **Different Register File Sizes (Opteron 80-bit, Xeon 128-bit)**
  - size dictates # **RISC** ops in an x86 instruction – ***instruction preference***
  - dictates # bits written from **FPU** pipes – ***limits scalar SIMD throughput***
- **Design of FPU and issue bandwidth from FP scheduler**
  - Opteron: ADD/MUL/ST pipes eat and write 240 bits per clock
  - Xeon: ADD/MUL pipes eat and write 128 bits per clock

Though Xeon64 and Opteron are instruction compatible, Xeon64 delivers ½ the throughput per clock on SIMD scalar code

AMD Opteron™,Pentium®4 (FPU analysis)
Throughput of SSE, SSE2, x87 Operations

Operation	*SSE Scalar*	*SSE vector*	*SSE2 scalar*	*SSE2 vector*	*X87*
Add	1 / cycle	2 / cycle	1 / cycle	1 / cycle	1 / cycle
Multiply	1 / cycle	2 / cycle	1 / cycle	1 / cycle	1 / cycle
Add & Multiply	2 / cycle	4 / cycle	2 / cycle	2 / cycle	2 / cycle

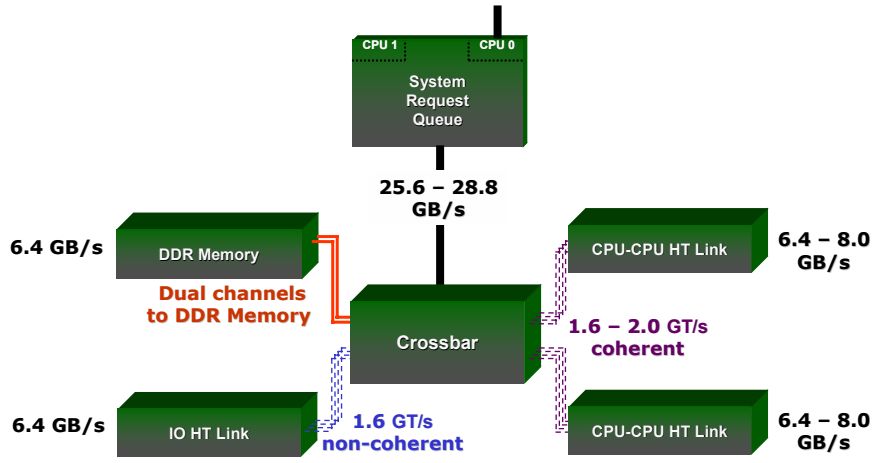
Operation	*SSE Scalar*	*SSE vector*	*SSE2 scalar*	*SSE2 vector*	*X87*
Add	1 / 2 cycles	2 / cycle	1 / 2 cycles	1 / cycle	1 / cycle
Multiply	1 / 2 cycles	2 / cycle	1 / 2 cycles	1 / cycle	1 / 2 cycles
Add & Multiply	1 / cycle	4 / cycle	1 / cycle	2 / cycle	1 / cycle

AMD Opteron™,Pentium®4 (ALU Analysis)
Throughput and Latency Comparison

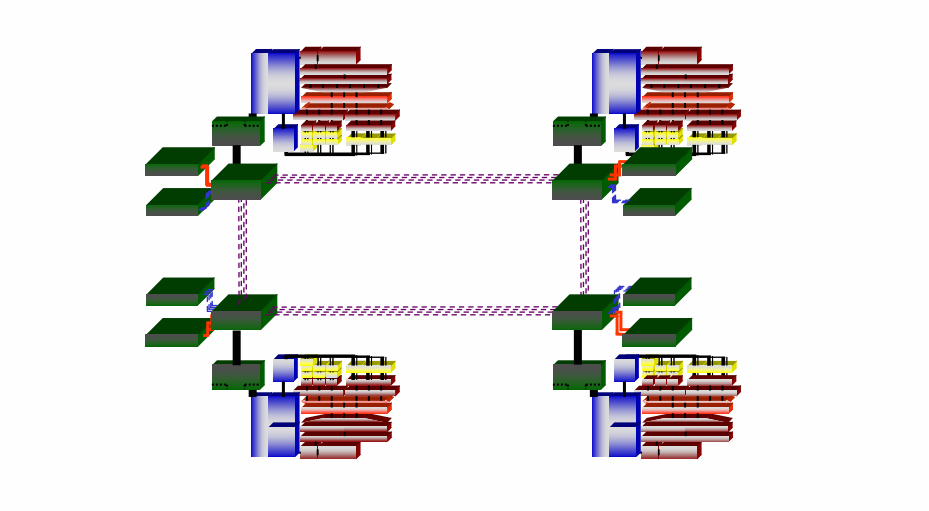
Operation	32-bit	64-bit
ADD/SUB	3 / cycle	3 / cycle
MUL_{signed}	1 / cycle 4 cycle latency	1 / 2 cycles
$MUL_{unsigned}$	1 / cycle 4 cycle latency	1 / 2 cycles
$MOV_{mem,reg}$	2 / cycle	2 / cycle
$MOV_{reg,reg}$	3 / cycle	3 / cycle
XOR/AND/OR	3 / cycle	3 / cycle
Shift/Rotate	3 / cycle	3 / cycle
DIV_{signed}	42 cycle latency	
$DIV_{unsigned}$	39 cycle latency	
LEA	3 / cycle	3 / cycle

Operation	32-bit	64-bit
ADD/SUB	2 / cycle	NA
MUL_{signed}	1 / cycle 18 cycle latency	NA
$MUL_{unsigned}$	1 / cycle 10 cycle latency	NA
$MOV_{mem,reg}$	2 / cycle	NA
$MOV_{reg,reg}$	2 / cycle	NA
XOR/AND/OR	2 / cycle	NA
Shift/Rotate	2 / cycle	NA
DIV_{signed}	80 cycle latency	NA
$DIV_{unsigned}$	80 cycle latency	NA
LEA	(2–0.5) / cycle	NA

Scalable Memory Bandwidth and IO
Opteron's on die IO controller



Scalable Memory Bandwidth and IO
Opteron's on die IO controller



Scalable Memory Bandwidth and IO
Opteron's on die IO controller

- **Hypertransport**
  - asynchronous coherent communication – ***maintain* MP *cache coherency***
  - high rate of communication – ***low impact on* MP *memory latency***
- **Memory Bandwidth**
  - scales linearly with # of processors in system
  - greater % of theoretical peak delivered – ***low latency memory access***
- **Memory Latency**
  - memory requests retired rapidly – ***enhances memory bandwidth***
  - doesn't scale linearly with # CPUS – ***scalable SMP performance***

PGI 5.2 Agenda
Compiler Enhancements driven by DYNA

- **Overview of enhancements in PGI 5.2**
  - all vector code isn't created equal
  - addressing of common block variables
  - loop peeling & optimal vector code
  - packing scalars into vector format
  - shuffling data in loops with GPRs
  - excessive prefetching – caveats to using sw prefetch
  - tuning of the unrolling heuristic – less register pressure
  - expanded class of vectorizable loops
  - **F90** pointer addressing support for objects greater than 2 GB

All Vectorized Code isn't created =
Minimizing bubbles in the FPU pipeline

- consider the following loop:

```
DO i=1,N
    a(i) = a(i) + b(i)*[c(i)+d(i)]
ENDDO
```

PGI 5.1.3

```
movlps (d),%xmm0
movhps 8(d),%xmm0
movlps (c),%xmm1
movhps 8(c),%xmm1
addps %xmm0,%xmm1
movlps (b),%xmm2
movhps 8(b),%xmm2
mulps %xmm2,%xmm1
movlps (a),%xmm3
movhps 8(a),%xmm3
addps %xmm3,%xmm1
```

✓ uses 4 registers
✓ generates 8 bubbles

use register renaming →

```
movlps (d),%xmm0
movhps 8(d),%xmm0
movlps (c),%xmm1
movhps 8(c),%xmm1
addps %xmm0,%xmm1
movlps (b),%xmm0
movhps 8(b),%xmm0
mulps %xmm0,%xmm1
movlps (a),%xmm0
movhps 8(a),%xmm0
addps %xmm0,%xmm1
```

✓ uses 2 registers
✓ generates 8 bubbles

operate from memory → *on 16-byte aligned addresses*

PGI 5.2.*

```
movaps (d),%xmm0
addps (c),%xmm0
mulps (b),%xmm1
addps (a),%xmm1
```

✓ uses 1 register
✓ generates 2 bubbles

Addressing of Common Blocks
Minimize GPR stores and loads from stack

- consider the common blocks and loop below:

```
Common/test1/x1(N),x2(N),x3(N),y1(N),y2(N),y3(N)
Common/test2/px1(N),py1(N),px2(N),py2(N)
Common/test3/vx2(N),vx3(N),vx4(N),vy2(N),vy3(N),vy4(N),vz2(N),vz3(N),vz4(N)
Common/test3/g1(N),g2(N),g3(N),g4(N)
Common/test4/ax(N),ay(N),bz(N)

do i=1,N
      xi  =area(i)*(x3(i)-x2(i)-x4(i))
      yi  =area(i)*(y3(i)-y2(i)-y4(i))
      g1(i)= 1.-px1(i)*xi-py1(i)*htyi
      g2(i)=-1.-px2(i)*xi-py2(i)*htyi
      g3(i)= 2.-g1(i)
      g4(i)=-2.-g2(i)
      ax(i)=g2(i)*vx2(i)+g3(i)*vx3(i)+g4(i)*vx4(i)
      ay(i)=g2(i)*vy2(i)+g3(i)*vy3(i)+g4(i)*vy4(i)
      bz(i)=g2(i)*vz2(i)+g3(i)*vz3(i)+g4(i)*vz4(i)
enddo
```


Addressing of Common Blocks
Minimize GPR stores and loads from stack

- consider the common blocks and loop below:

```
Common/test1/x1(N),x2(N),x3(N),y1(N),y2(N),y3(N)
Common/test2/px1(N),py1(N),px2(N),py2(N)
Common/test3/vx2(N),vx3(N),vx4(N),vy2(N),vy3(N),vy4(N),vz2(N),vz3(N),vz4(N)
Common/test3/g1(N),g2(N),g3(N),g4(N)
Common/test4/ax(N),ay(N),bz(N)
```

```
movq    -120(%rsp), %r15
movlps  (%r15,%rcx), %xmm4
movhps  8(%r15,%rcx), %xmm4
movq    -112(%rsp), %r15
movlps  (%r15,%rcx), %xmm5
movhps  8(%r15,%rcx), %xmm5
subps   %xmm4, %xmm5
movq    -104(%rsp), %r15
movlps  (%r15,%rcx), %xmm4
movhps  8(%r15,%rcx), %xmm4
subps   %xmm4, %xmm5
movq    -96(%rsp), %r15
```

- *PGI 5.1.5 uses sepatate GPRs to address each array, even for →'s in the same common block*
- *Accentuates register pressure in loops. One LS-DYNA loop had 54 GPR mov's to and from stack in PGI 5.1.5, Intel 7.1 had 0 occuracnces of GPR movs*

Addressing of Common Blocks
Minimize GPR stores and loads from stack

- consider the common blocks and loop below:

```
Common/test1/x1(N),x2(N),x3(N),y1(N),y2(N),y3(N)
Common/test2/px1(N),py1(N),px2(N),py2(N)
Common/test3/vx2(N),vx3(N),vx4(N),vy2(N),vy3(N),vy4(N),vz2(N),vz3(N),vz4(N)
Common/test3/g1(N),g2(N),g3(N),g4(N)
Common/test4/ax(N),ay(N),bz(N)
```

```
movaps   -12000(%r9,%rdx),%xmm4
movaps   -8000(%r9,%rdx),%xmm9
movaps   (%r8,%rdx),%xmm5
movaps   4000(%r8,%rdx),%xmm6
movaps   %xmm3,%xmm7
subl     $8,%eax
subps    -24000(%r9,%rdx),%xmm4
addl     $8,%ecx
subps    -20000(%r9,%rdx),%xmm9
movaps   %xmm7,%xmm8
subps    (%r9,%rdx),%xmm4
mulps    %xmm5,%xmm4
```

- ***PGI 5.2.* accesses all entities in the same common block with 1 GPR***
- ***GPR register pressure in PGI 5.2.* is greatly reduced. No excess rops, in comparison to Intel, are generated***
- ***Executable Operations from memory are now performed – less bubbles in FPU pipeline***

Loop Peeling & Optimal Vector Code
Common Block Illustration

- Efficient code vectorization requires:
  - *uniform relative alignment of pointers in loops*
    - Can be achieved via use of common blocks
    - Span of arrays covered in each loop iteration should be a multiple of 4 or 2 in single or double precision
  - *loop peeling to adjust common pointers to 16-byte aligned locations*
  - *performing *,+,- from memory (requires 16-byte alignment)*
- PGI 5.2.* implements peeling of code in **CB** loops

```
Common/test1/vx2(N),vx3(N),vx4(N),vy2(N),vy3(N),vy4(N),vz2(N),vz3(N),vz4(N)
Common/test2/g1(N),g2(N),g3(N),g4(N)
Common/test3/ax(N),ay(N),bz(N)

do i=1,N
      ax(i)=g2(i)*vx2(i)+g3(i)*vx3(i)+g4(i)*vx4(i)
      ay(i)=g2(i)*vy2(i)+g3(i)*vy3(i)+g4(i)*vy4(i)
      bz(i)=g2(i)*vz2(i)+g3(i)*vz3(i)+g4(i)*vz4(i)
enddo
```

Loop Peeling & Optimal Vector Code
Common Block Illustration

- Efficient code vectorization requires:
  - *uniform relative alignment of pointers in loops*
    - Can be achieved via use of common blocks
    - Span of arrays covered in each loop iteration should be a multiple of 4 or 2 in single or double precision
  - *loop peeling to adjust common pointers to 16-byte aligned locations*
  - *performing *,+,- from memory (requires 16-byte alignment)*
- PGI 5.2.* implements peeling of code in **CB** loops

```
Common/test1/vx2(N),vx3(N),vx4(N),vy2(N),vy3(N),vy4(N),vz2(N),vz3(N),vz4(N)
…
· Check relative alignment of test1,test2 and test3 common block pointers
· If pointers are aligned to 16-byte boundaries – JUMP TO VECTORSSE LOOP
· Scalar SSE loop +6*9 – used to align CB pointers on a 16-byte boundary
· Vector SSE loop +6*9 – used to perform most of computation
· Scalar SSE loop +6*9 – final iterations not covered by vector SSE loop
```


Optimized Scalar→Vector Transforms
Packing 4 scalars in a vector loop

- Some vector loops require performing vector operations of scalar data upon vector quantities:

```
a(i) = a(i) + b(j,i)*c(i) + d(j,i)*e(i)
```

  - *PGI 5.1.5 does this via reading 4 floats, storing them to stack and then reading them in a 128-bit load:*
    - Create load / store dependencies
    - Excessive # of rops required to perform this function
    - Requires 8 x 32-bit movss loads / stores, 1 movaps read (14 rops)
    - Creates 8 bubbles down FPU pipes
  - *PGI 5.2.* does this via interleaving floats*
    - 4 x movss reads, 2 x Unpcklps, 1 x movlhps (11 rops)
    - Creates 4 bubbles down FPU pipes
    - Much shorter latency

Use GPRs to shuffle data
Not an absolute statement but almost

- GPRs have the following advantages in loops that "only" shuffle data around:
  - *movss decodes to 2 rops, a GPR mov decodes to 1 rop*
  - the FPU pipe can only perform 1 32-bit or 64-bit store per cycle while the ALU unit can perform 2 of either
  - pseudo vector copies of floats can be performed using 64-bit GPRs to perform 2 at a time, this utilizes the full throughput of the ALU and load store unit
  - double precision moves should still be more efficient because the ALU unit can perform 2 x 64-bit stores per cycle whereas the FPU can only perform 1 x 64-bit store per cycle
  - caution must be taken into consideration to not generate excessive register pressure
  - ALU throughput can be affected if there are many ALU ops in addition to loads and stores occurring (add, sub, lea, etc.)

Excessive Prefetching
Caveats about software prefetch

- Prefetching can preemptively bring data into the cache in advance of it's use, but:
  - Opteron has a very robust HW prefetcher for sequential data accesses
    - HW prefetches move into L2 (12 vs 3 cycle latency compared to L1
    - does not consume execution dispatch bandwidth / sw prefetches do
  - SW prefetches across 4KB page boundaries are dropped and suffer a 90 cycle latency penalty
  - SW prefetch of non-sequential data accesses offers little benefit
    - only 4-8 bytes of every 64 bytes fetched is useful
    - Rate of cache evictions is very high, useful data now has to be fetched from L2
    - MAB units in processor consumed quickly and prevents loads from occurring
  - SW prefetches consume 1 of the 3 execution dispatch slots per clock cycle, thus limiting throughput through the FPU and IPC

Tuning of Unrolling Heuristic
Less is sometimes more

- Excessive unrolling of some classes of loops increases register pressure:
  - Loops that do not benefit from compiler unrolling:
    - multi-dimensional arrays in which (***i***, j,…) ***i*** isn't the fastest moving index
    - arrays whose index needs to be loaded to be determined, ***x(BIN(i))***
    - loops large in size that exceed the # of floating-point registers
  - GPR and FP registers are spilled to memory causing:
    - excess RISC operation counts compared – more work required more execution time
    - address generation held up by register load dependencies
    - out of order execution is limited via not being able to load data to process
- **PGI 5.2** *unrolls less aggressively, allowing out of order execution within the processor to mask latency rather than compiler unrolling*

Expanded Class of Vector loops
Vector Code generation enhancements

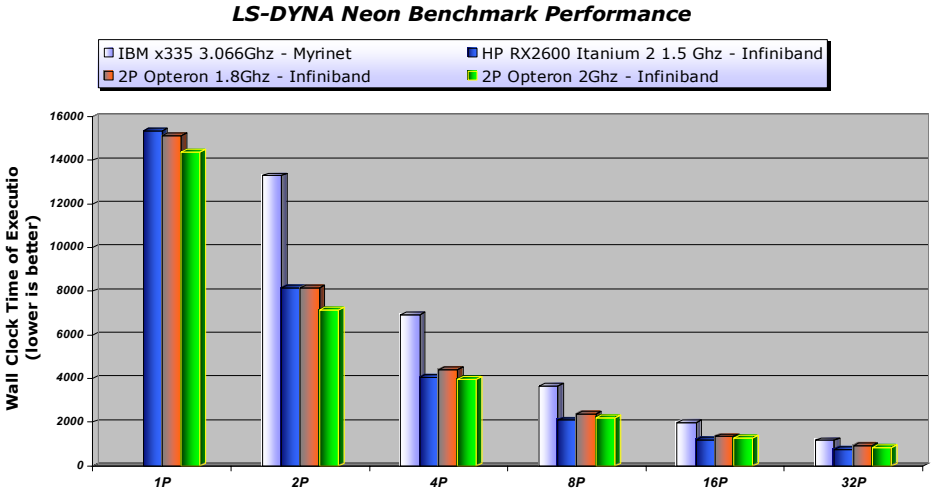
- Loops with the following constructs now vectorize in **PGI 5.2** :
  - loops containing **SIGN** or **MERGE** intrinsics
  - large loops containing more than a preset limit of instructions
  - Loop-carry Reduction Elimination (**LRE**) interfered with some loops vectorization
  - invariant IF / ELSE transformations that hoist IF / ELSE constructs not dependent upon loop variables outside of loop replicating loop with all cases of IF / ELSE statement
  - Loops that operate upon data objects > 2 GB
  - Loops in programs compiled with the –i8 switch

LS-DYNA Performance

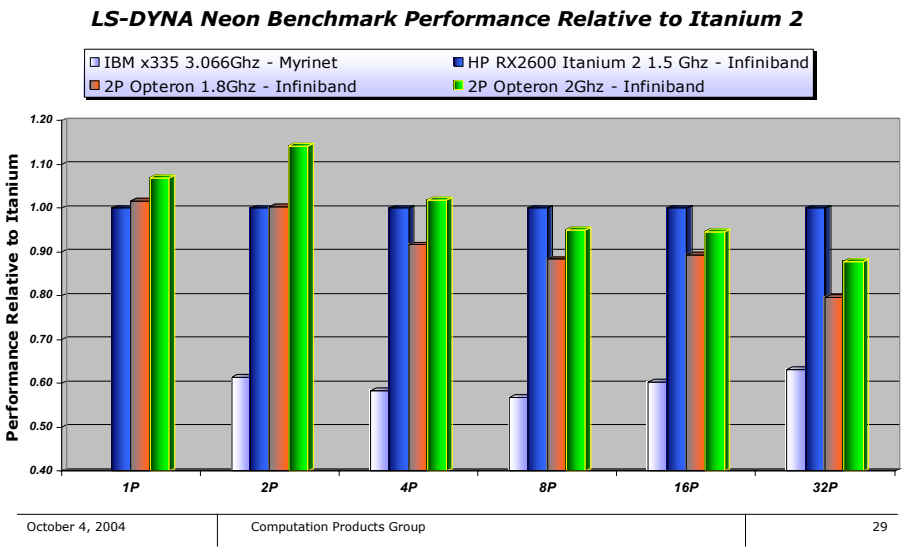
Neon and 3-Car Models

64-bit LS-DYNA v5434

Neon Model Performance

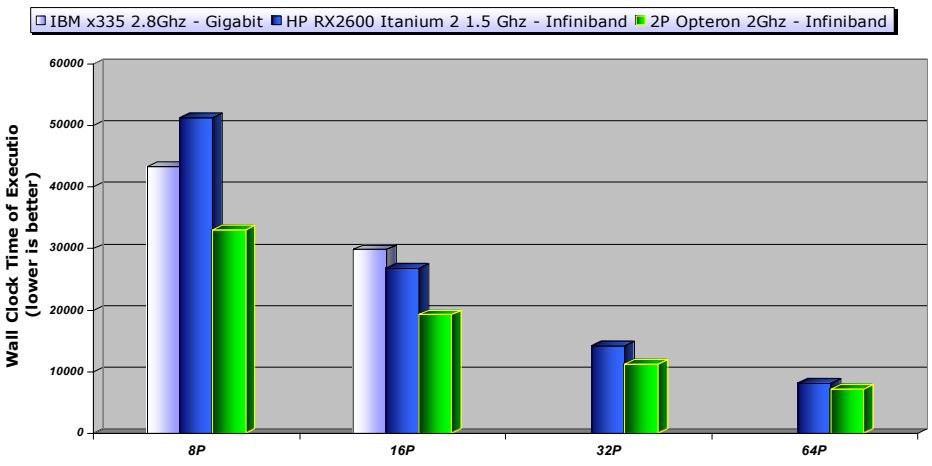


64-bit LS-DYNA v5434
Neon Model Performance



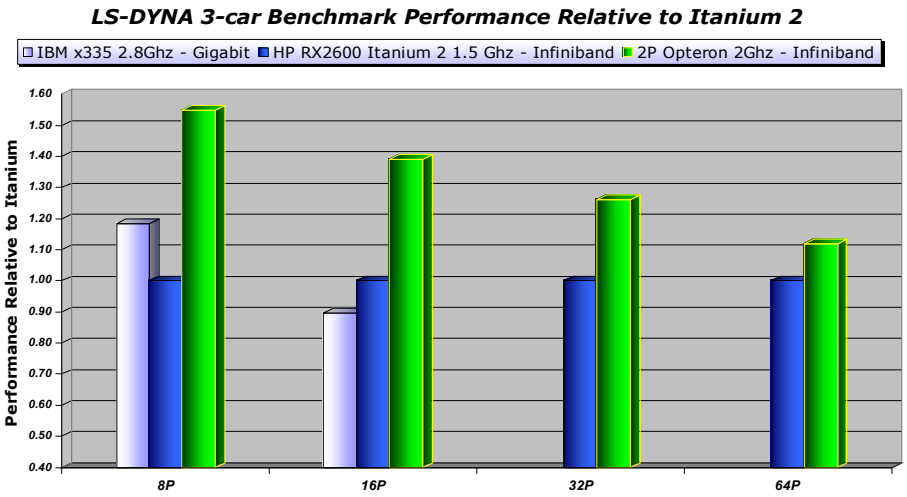
October 4, 2004 | Computation Products Group | 29

64-bit LS-DYNA v5434
3-Car Model Performance



October 4, 2004 | Computation Products Group | 30

64-bit LS-DYNA v5434
3-Car Model Performance



Trademark Attribution

AMD, the AMD Arrow Logo, AMD Opteron and combinations thereof are trademarks of Advanced Micro Devices, Inc. HyperTransport is a licensed trademark of the HyperTransport Technology Consortium. Other product names used in this presentation are for identification purposes only and may be trademarks of their respective companies.