Optimizing Applications for Performance on the Pentium 4 Architecture

Arrian Mehis, Performance Engineer, Workstations
Ramesh Radhakrishnan, Performance Engineer, Servers
Dell Computer Corporation

Agenda

- Pentium 4 Architecture Breakdown
  - Key differences from the PIII
  - Using the P4's performance enhancing features
- Advanced Compiler Optimizations for the P4
- Evaluating P4 Optimization Techniques
- Conclusion

Agenda

- Pentium 4 Architecture Breakdown
  - Key differences from the PIII
  - Using the P4's performance enhancing features
- Advanced Compiler Optimizations for the P4
- Evaluating P4 Optimization Techniques
- Conclusion

Key Features of the Pentium 4

- Twenty stage pipeline
- Execution trace cache
- Hyper-Threading technology
- Faster system bus
- Faster execution units
- Enhanced floating-point / multimedia unit
- Streaming SIMD Extensions 2 (SSE2)

Using the P4's Performance Enhancing Features

- Loop structuring
- Branch predictability
- Store forwarding
- Code and data proximity
- SSE2 instruction set
- Data access patterns

Loop structuring

- Loop unrolling (unroll to 16 or fewer iterations)
- Innermost nesting level is free of inter-iteration dependencies
- Keep induction (loop) variable expressions simple
- Use the `pause` instruction in spin-wait and idle loops

Branch Predictability

- Generate code that is consistent with static branch prediction algorithms (backward taken, not forward taken)
- Keep code and data on separate pages
- Eliminate branches
  - Make basic code blocks contiguous
  - Unroll loops
  - Use the `cmov` instruction (conditional move)
- Inline where appropriate

Store Forwarding

- Sequence
  - Data to be forwarded to the load has been generated by an earlier store (executed)
- Size
  - Bytes loaded must be a subset of bytes stored
- Alignment
  - Cannot wrap around cache line boundary
  - Address of load is aligned with respect to address of store

Code & Data Proximity

- Avoid mixing code & data
  - Pad 1024 bytes apart (one cache line)
- Self-modifying code
  - Pipeline purged
  - Instructions re-fetched

SSE2 Instruction Set

- 144 total instructions
  - 128-bit registers `xmm0-xmm7`
  - Easily changed from 64-bit MMX `mm0-mm7`
- Improves performance for apps:
  - Inherently parallel
  - Recurring memory access patterns
  - Localized recurring ops performed on data
  - Data-independent control flow
- Handle floating-point exceptions without penalty

Data Access Patterns

- Effective when working with large matrices
  - Transposes
  - Inverses
  - Etc.
- “Block” data into several smaller chunks
  - Eliminate cache misses
  - Improve bus efficiency

Agenda

- Pentium 4 Architecture Breakdown
  - Key differences from the PIII
  - Using the P4's performance enhancing features
- Advanced Compiler Optimizations for the P4
- Evaluating P4 Optimization Techniques
- Conclusion

Advanced Compiler Optimizations for the P4

- P4 specific optimizations (Win32 / Linux)
  - -G7 / -tppt7
  - -QxW / -xW (-QaxW / -axW)
- Additional optimizations (Win32 / Linux)
  - -O3 / -O3
  - -Qipo / -ipo
  - -Qprof_gen, -Qprof_use / -prof_gen, -prof_use

P4 Specific Optimizations

- -G7 / -tppt7
  - –Generates code optimized for the P4 processor, through optimal instruction scheduling and cache management
- -QxW / -xW (-QaxW / -axW)
  - –Generates SSE2 instructions specifically supported by the P4 processor using vectorization.
  - –Can generate SSE instructions as well as generic IA-32 instructions (larger code size)

Additional Optimizations

- -O3 / -O3
  - –Enable –O2 (default) plus more aggressive optimizations.
- -Qipo / -ipo
  - –Interprocedural optimization (IPO)
  - –Optimizes multiple files, can reduce code size
  - –Optimizes function ordering, reduces overhead
- -Qprof_gen, -Qprof_use / -prof_gen, -prof_use
  - –Profile-guided optimization (PGO)
  - –More accurate branch prediction
  - –Improved register allocation, IPO inlining
  - –Basic block movement, improves I-cache behavior

Agenda

- Pentium 4 Architecture Breakdown
  - Key differences from the PIII
  - Using the P4's performance enhancing features
- Advanced Compiler Optimizations for the P4
- Evaluating P4 Optimization Techniques
- Conclusion

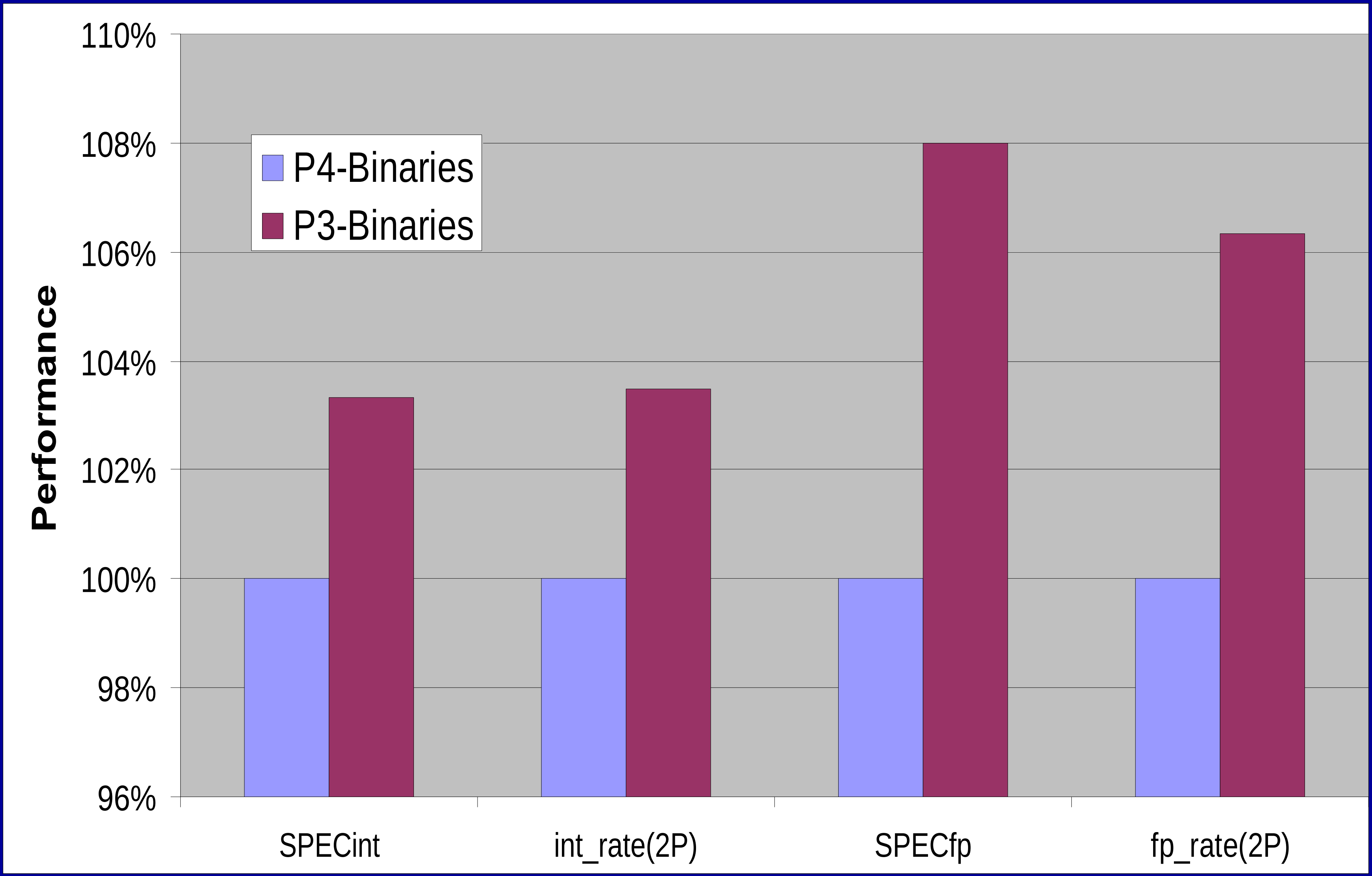
Evaluating P4 Optimization Techniques

- Common Benchmarks
  - SPEC CPU2000
  - LINPACK
  - HINT
  - SPEC Viewperf 6.1.2
- Coding Pitfalls
  - Using SSE2 Instructions
  - "Homemade" Benchmarks
- Data Access Patterns

SPEC CPU2000

- Windows platform
- Using the same Intel C++ & Fortran Compilers
  - PIII Binaries
    - Compiled with:
      - -QxK -Qipo -O3 PGO
  - P4 Binaries
    - Compiled with:
      - -QxW -Qipo -O3 PGO

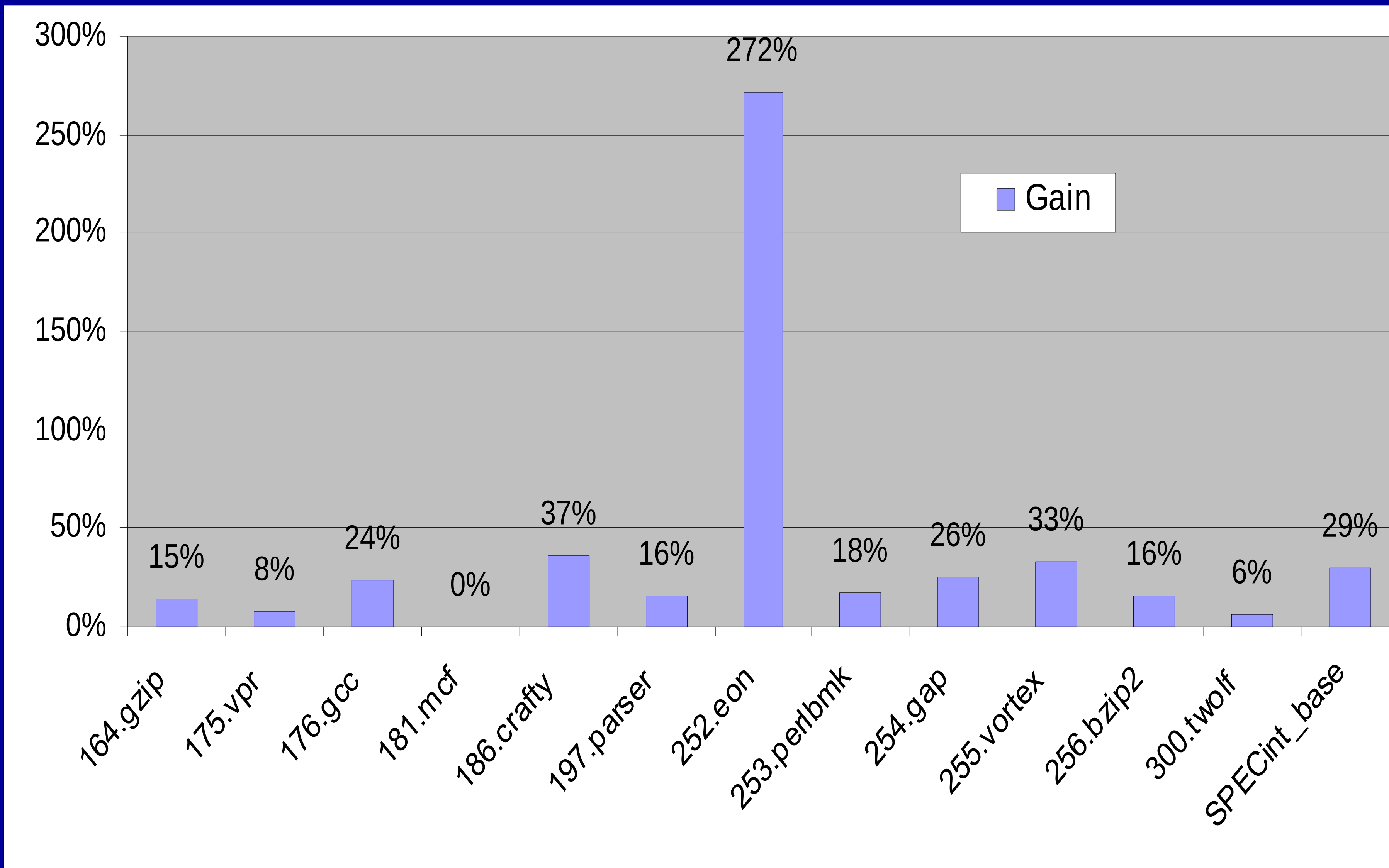
SPEC CPU2000



SPEC CPU2000 (INT)

- Linux platform
- Using Intel C++ Compiler vs. GCC (no GNU Fortran compiler)
  - ICC Binaries
    - Compiled with:
      - -xW -ipo -O3 PGO
  - GCC Binaries
    - Compiled with:
      - -O2

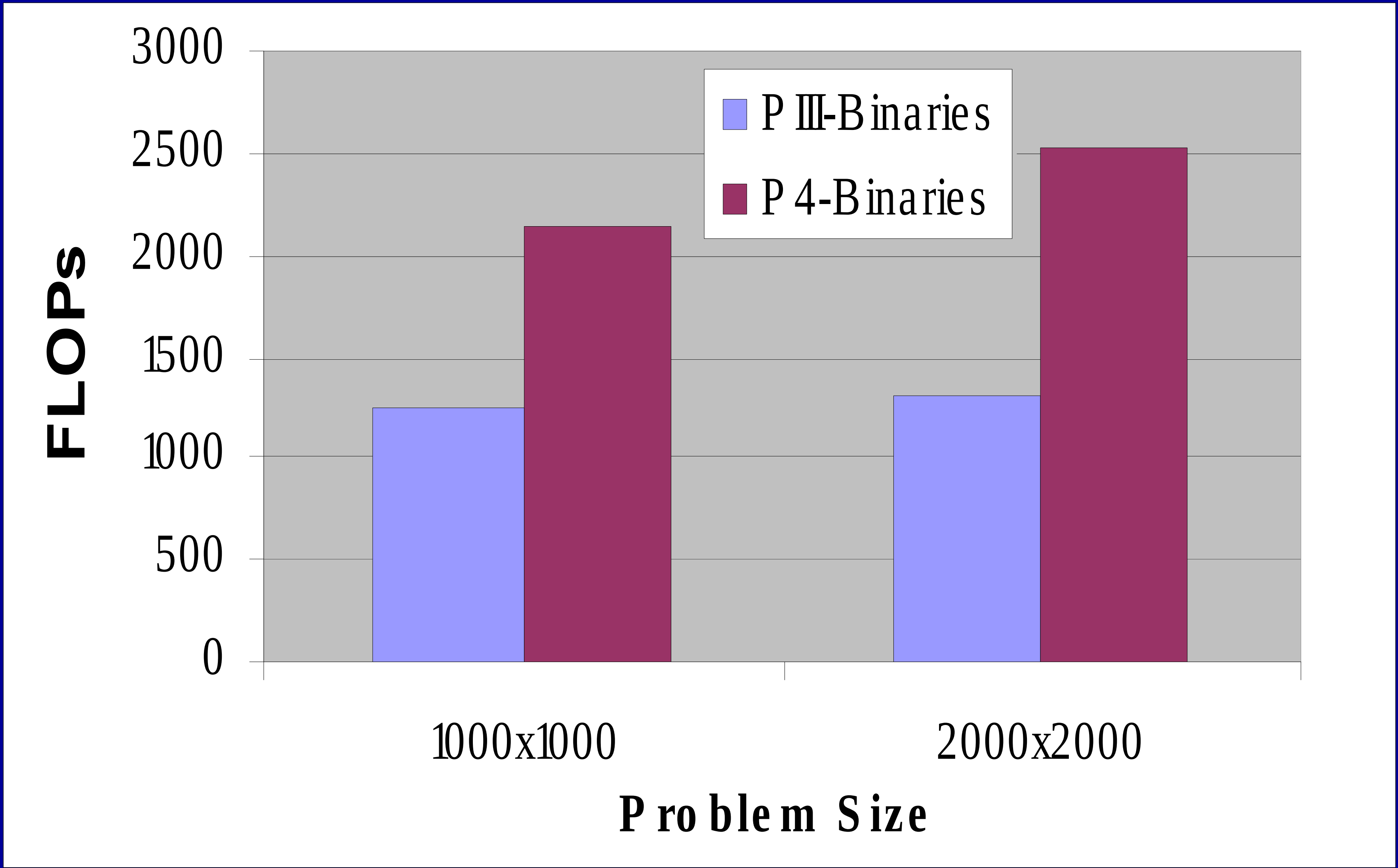
SPEC CPU2000 (INT)



LINPACK

- Windows platform
- Using the same Intel C++ & Fortran Compilers
  - PIII Binaries
    - Compiled with:
      - -QxK -Qipo -O3 PGO
  - P4 Binaries
    - Compiled with:
      - -QxW -Qipo -O3 PGO

LINPACK

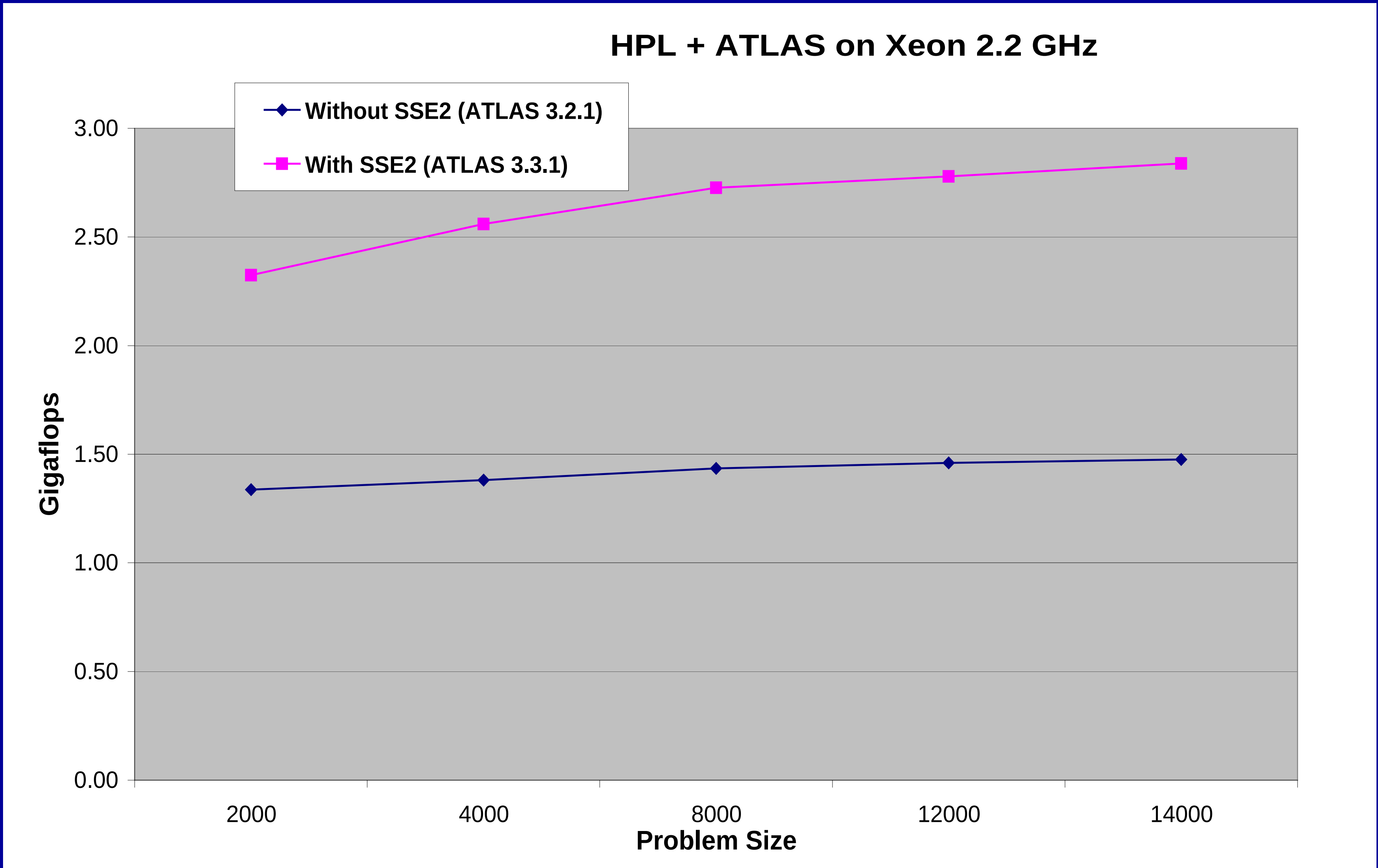


FLOPs
3000
2500
2000
1500
1000
500
0
PIII-Binaries
P4-Binaries
1000x1000
2000x2000
Problem Size

HIGH PERFORMANCE LINPACK

- Linux Platform
- Using the same Intel C++ Compiler
  - SSE2 binaries
    - Compiled with:
      - -QxW -Qipo -O3 PGO
  - Normal Binaries

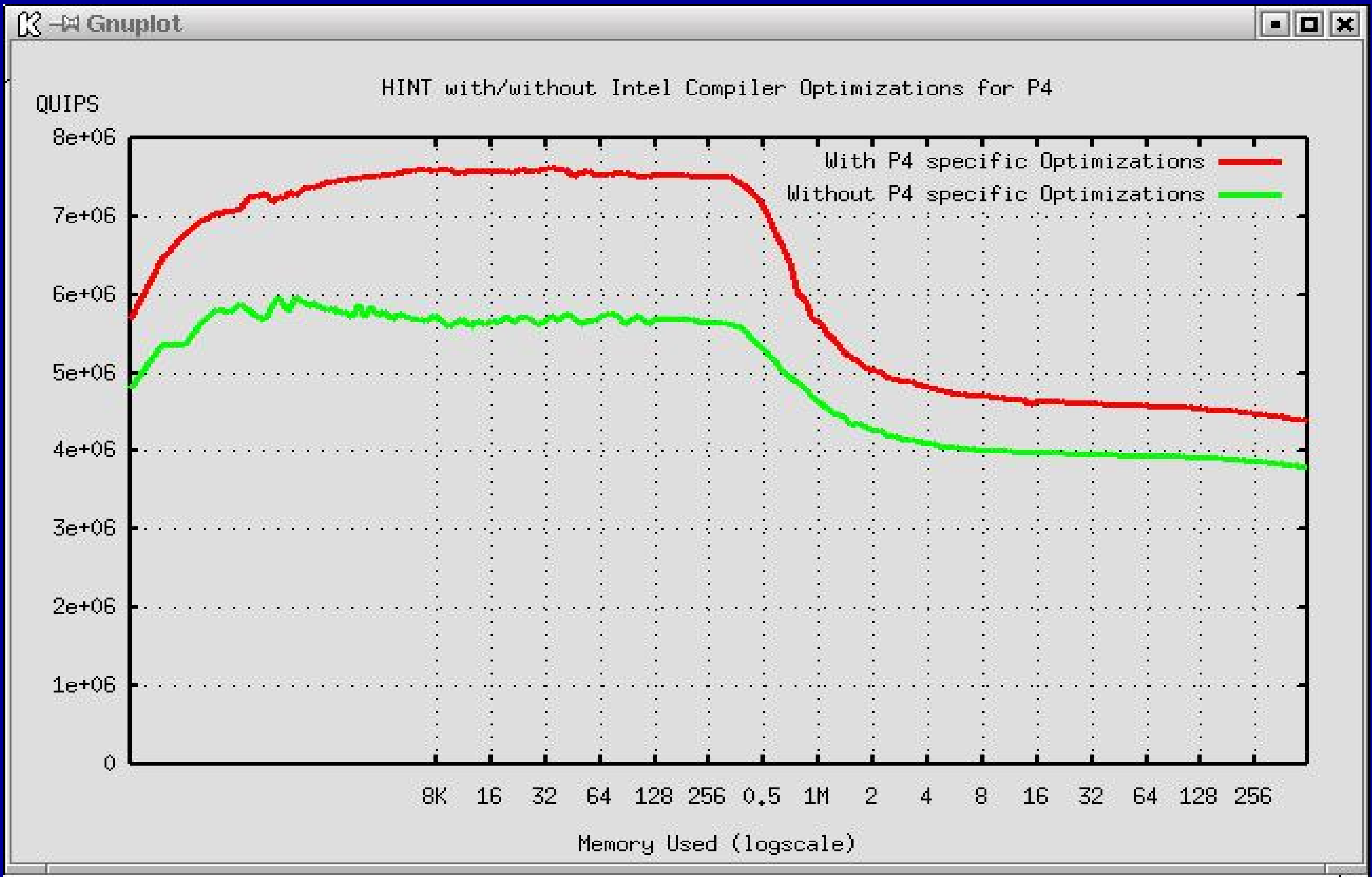
HIGH PERFORMANCE LINPACK



HINT

- Linux Platform
- Using the same Intel C++ Compiler
  - SSE2 binaries
    - Compiled with:
      - -xW
  - Normal Binaries

HINT

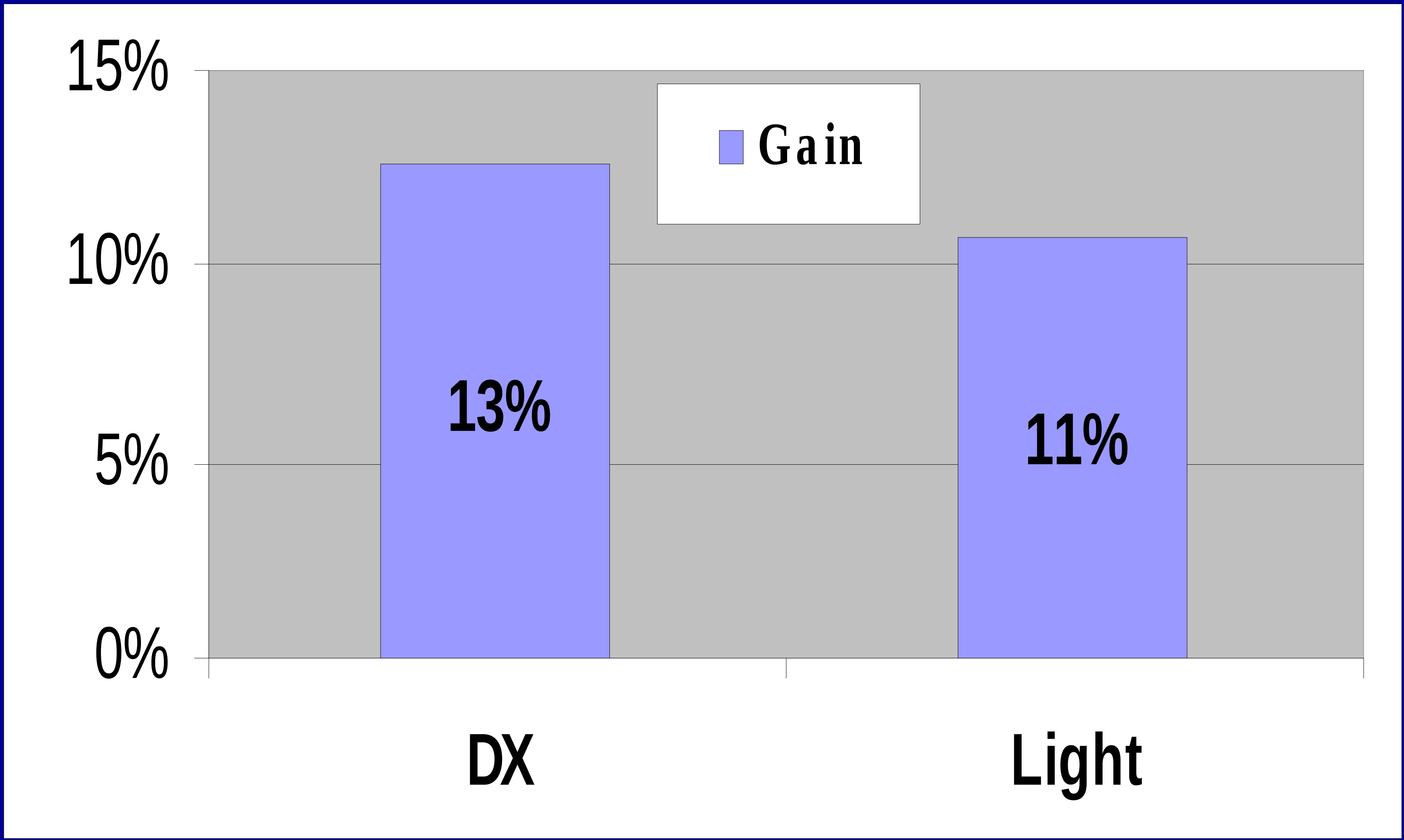


Gnuplot
HINT with/without Intel Compiler Optimizations for P4
QUIPS
8e+06
7e+06
6e+06
5e+06
4e+06
3e+06
2e+06
1e+06
0
With P4 specific Optimizations
Without P4 specific Optimizations
8K 16 32 64 128 256 0.5 1M 2 4 8 16 32 64 128 256
Memory Used (logscale)

SPEC Viewperf 6.1.2

- Windows platform
- Using Intel C++ Compiler vs. Microsoft Visual C++
  - ICL Binaries
    - Compiled with:
      - -QxW -O3
  - CL Binaries
    - Compiled with:
      - -O2

SPEC Viewperf 6.1.2



Using SSE2 Instructions

- Success Story – Computer Associates
- Windows platform
- Code snippet:

```
double f1 = 10.0, f2 = 2.33456
for(j=0;j<1000;j++) {
  for(i=0;i<1000000;i++) {
    f1 = f2*f1;
    f1 = f1/(f2+1.0);
    i1 = i1*i2;
    i1 = 10;
  }
}
```

Using SSE2 Instructions

- Problem:

```
f1 = f2*f1;
f1 = f1/(f2+1.0);
```

  – The variable `f1`, originally 10.0, is multiplied by a number that is close to 2/3 (2.33456/3.33456)
  – Eventually, because the loop count is really large, the result becomes really small
  – Traditional P4 optimizations can not resolve all coding pitfalls
- Result:
  – Masked floating point exceptions are generated
  – 1950 seconds to complete on 2.0GHz P4

Using SSE2 Instructions

- Solution:

```
__asm {
  movlpd xmm1, f1       // xmm1 = 10.1 (f1)
  movlpd xmm2, f2       // xmm2 = 2.3346 (f2)
  movlpd xmm3, f3       // xmm3 = 1.0 (f3)
}
for(j=0;j<1000;j++) {
  $A1:
  add eax, 1            // i++
  mulsd xmm1, xmm2      // f1 = f2*f1
  addsd xmm2, xmm3      // f2 = f2+1.0
  divsd xmm1, xmm2      // if i<1000000 then
  jle $A1               // jump and link to $A1 (loop)
  ALIGN 4               // align section by 4 bytes
}
```

  - Executes in less than a second

“Homemade” Benchmarks #1

- Success Story – University of Alberta
- Linux platform

```
double a, b, c
c=0; b=0.21;

for(i=1;i<N;i++)
  c=c+cos(b)*exp(-0.5*c);
```

“Homemade” Benchmarks #1

- Problem

```
c=c+cos(b)*exp(-0.5*c);
```

  - Transendental code (cos, sin, exp, etc.)
    - Only x87 floating-point code supports transcendental instructions alone
  - GCC compiler

- Result
  - PIII 1.OGHz outperforming P4 2.0GHz

“Homemade” Benchmarks #1

- Solution
  - –Recompile with SSE2 optimizations

	GCC	ICC
PIII	-O2	-xK –O3
P4	-O2	-xW –O3

	GCC	ICC
PIII	18.67 s	6.698 s
P4	21.43 s	4.203 s

  - –Over 5x gain on the P4!
  - –Over 2.75x gain on the PIII

“Homemade” Benchmarks #2

- Success Story – University of Alberta
- Linux platform

```
double *a, *b;
unsigned long i;
a = (double *)malloc(N*sizeof(double));
b = (double *)malloc(N*sizeof(double));
a[0]=0.0;  b[0]=0.0;

for(i=1;i<50000000;i++) {
  a[i]=(double)i + 2.7*b[i-1];
  b[i]=(double)i + 2.7*a[i-1];
}
```

“Homemade” Benchmarks #2

- Problem
  - Large loop count – 50 million
  - Pointer chasing
  - Type casting (can impact memory access)
  - GCC compiler

- Result
  - PIII 1.OGHz outperforming P4 2.0GHz by 4x!

“Homemade” Benchmarks #2

- Solution
  - –Recompile with SSE2 optimizations

	GCC	ICC
PIII	-O2	-xK –O3
P4	-O2	-xW –O3

	GCC	ICC
PIII	25.161 s	24.586 s
P4	111.94 s	2.826 s

- –Over 39x gain on the P4!
- –Negligible gain on PIII

Data Access Patterns

- Useful with large matrices, arrays, etc.
  - Inverse
  - Transposes
  - Etc.
- Traditional method
  - Traverse element by element
    - Entire memory domain
    - Inefficient cache usage (cache misses)
- Blocking method
  - Traverse “blocks” of smaller data
  - Fits into cache
    - Much more efficient
    - Conserves bus bandwidth

Data Access Patterns

- Traditional Method
  - Matrix transpose

```
#define N 8192    // matrix row/column size

for(i=0;i<N;i++) {
  for(j=0;j<N;j++)
    pDst[j*N+i] = pSrc[i*N+j];
  }
}
```

Data Access Patterns

- Blocking Method
  - –Matrix transpose

```
#define N 8192          // matrix row/column size
#define Q 32            // block row/column size

for(i=0;i<N/Q;i++)
  for(j=0;j<N/Q;j++)
    SrcStart = i*Q*N + j*Q;
    DstStart = j*Q*N + i*Q;
    for(ii=0;ii<Q;ii++) {
      SrcOffset = SrcStart + N*ii;
      DstOffset = DstStart + ii;
      for(jj=0;jj<Q;jj++) {
        pDst[DstOffset] = pSrc[SrcOffset++];
        DstOffset += N;
      }
    }
  }
```

Agenda

- Pentium 4 Architecture Breakdown
  - Key differences from the PIII
  - Using the P4's performance enhancing features
- Advanced Compiler Optimizations for the P4
- Case Studies
- Conclusion

Conclusion

- Difference between problem resolution and true performance
- Recompiling won't *always* guarantee a performance gain
  - May require recoding (want to avoid)
  - Dependent on each individual workload
  - Try recompiling at the very least!
- Beware of coding pitfalls
  - Use SSE2 when you can
  - Be wary of legacy code (PIII and earlier)
- Follow ***Intel's P4 Optimization Guide***