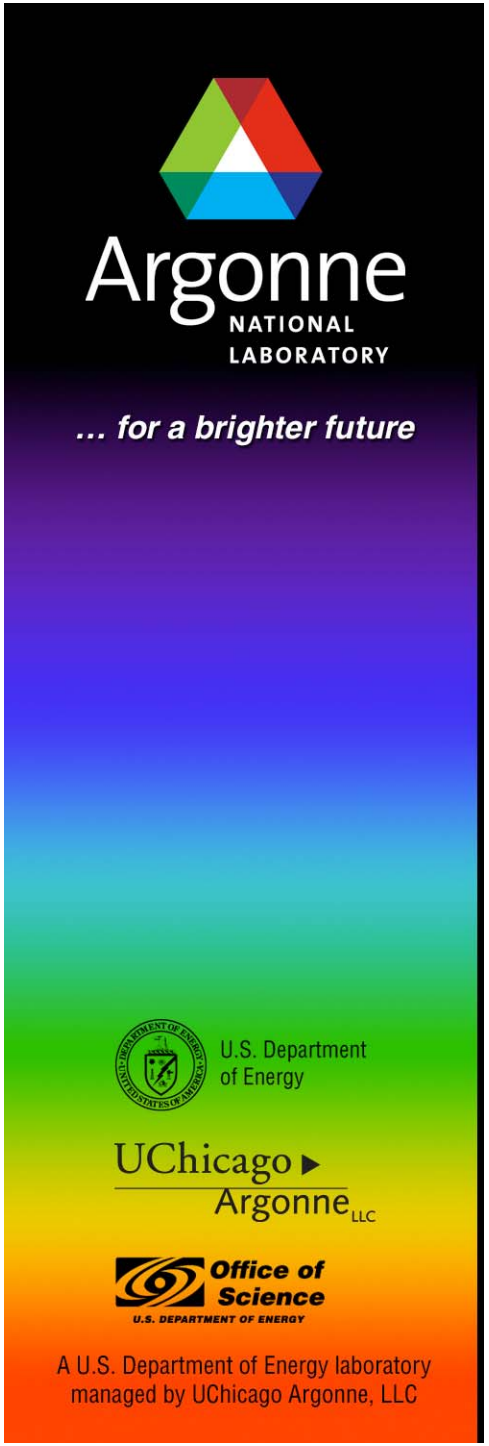
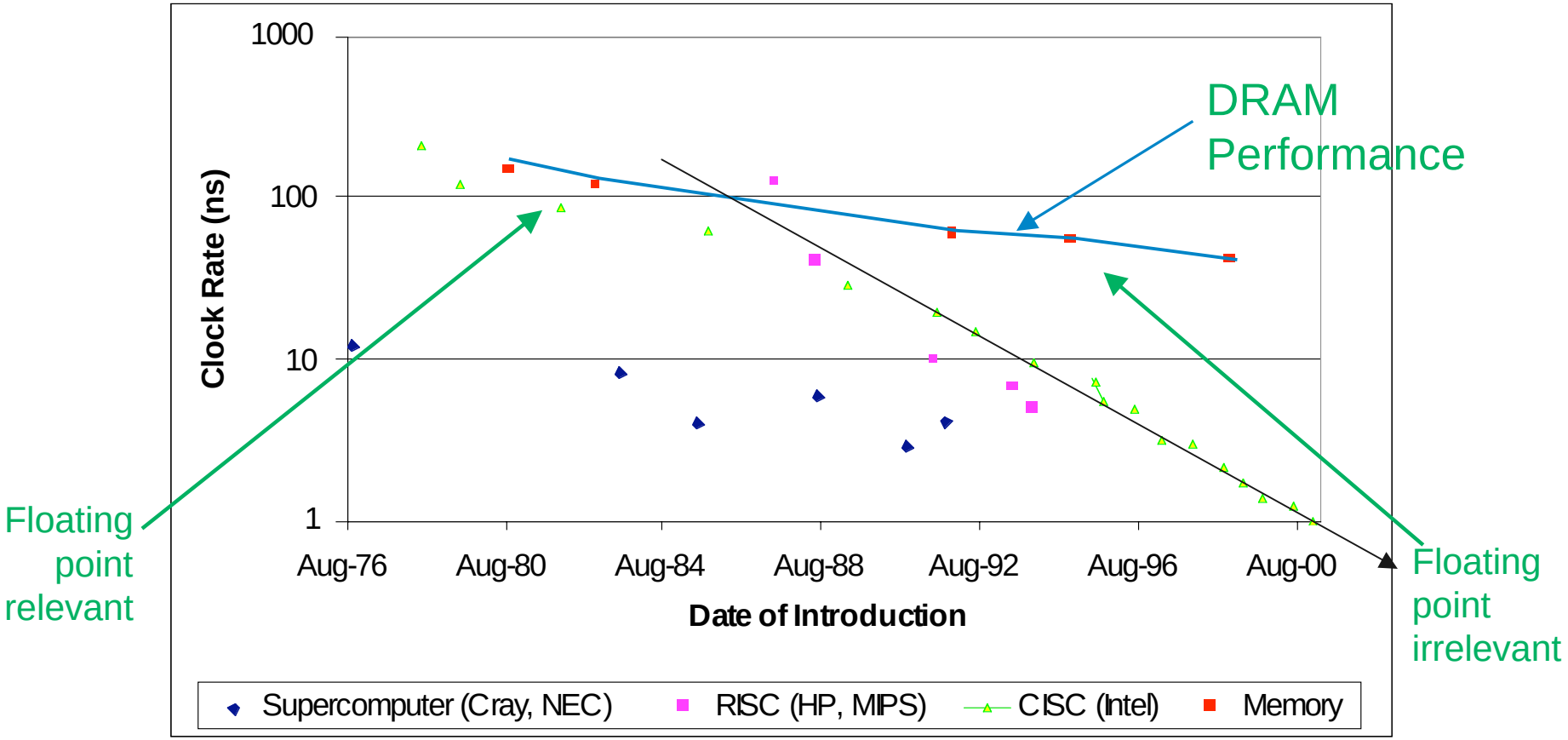


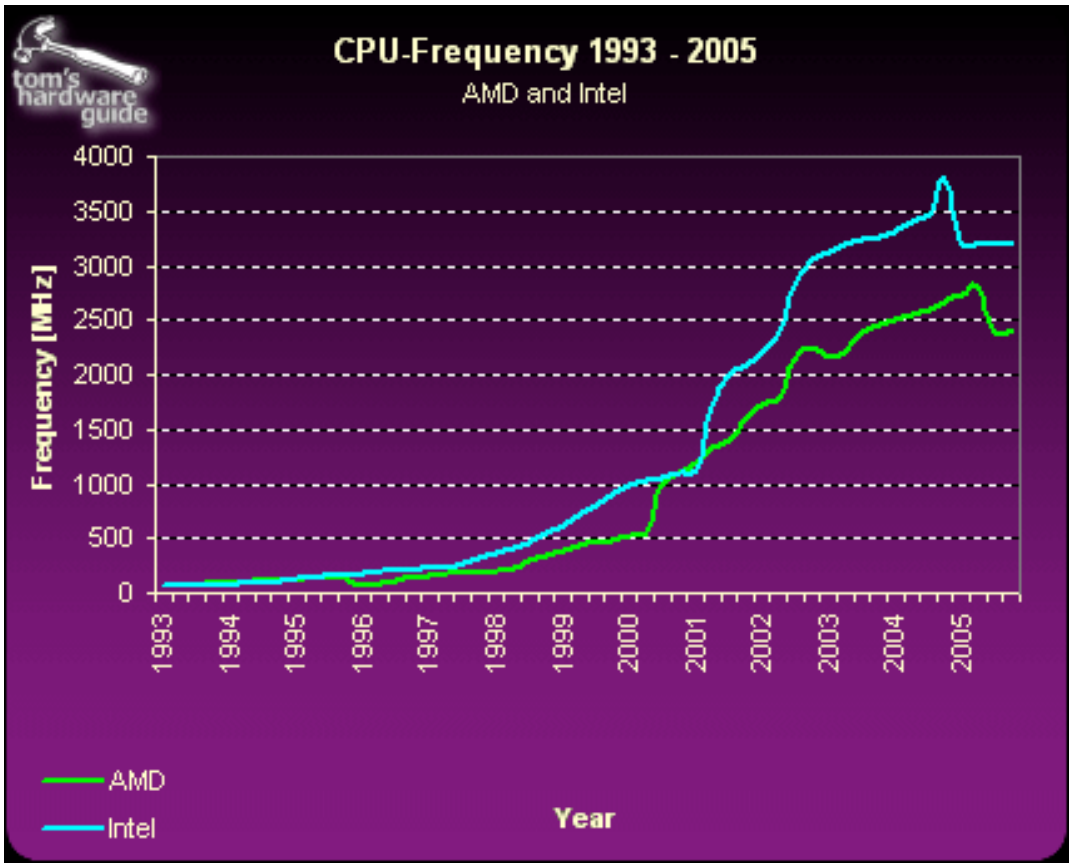
Overcoming the Barriers to Sustained Petaflop Performance

William D. Gropp
Mathematics and Computer Science
www.mcs.anl.gov/~gropp

Why is achieved performance on a single node so poor?



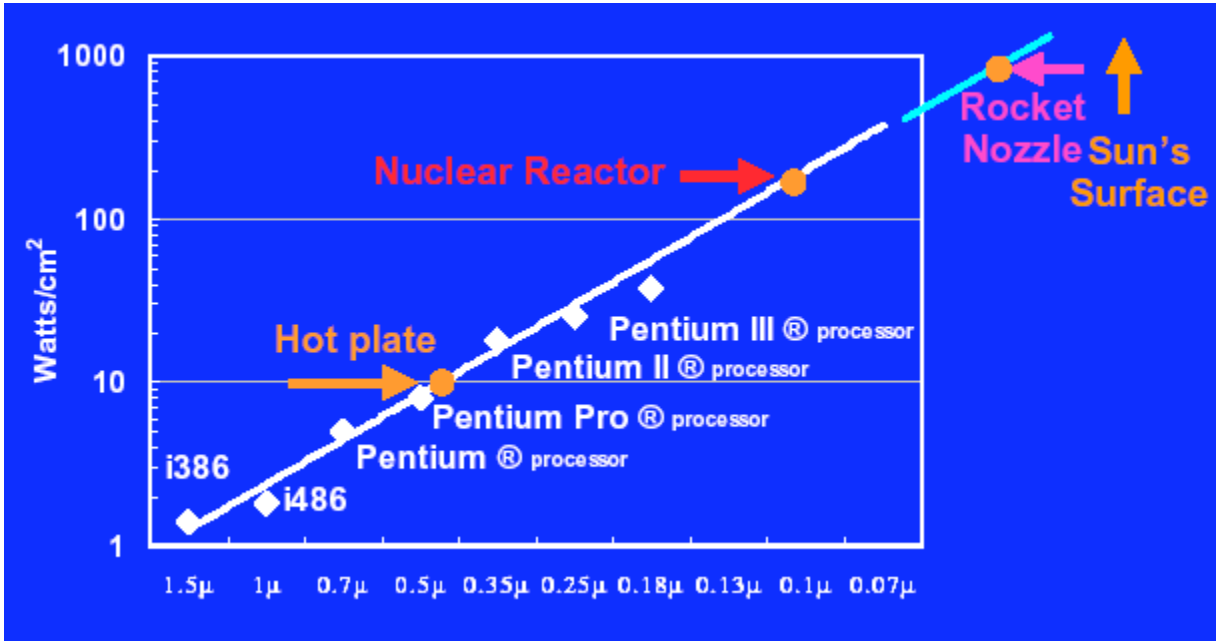
Peak CPU speeds are stable



- From http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/

Why are CPUs not getting faster?

- Power dissipation problems will force more changes
  - Current trends imply chips with energy densities greater than a nuclear reactor
  - Already a problem: Recalls of recent Mac laptops because they could overheat.
  - Will force new ways to get performance, such as extensive parallelism



Where will we get (Sustained) Performance?

- Algorithms that are a better match for the architectures
- Parallelism at all levels
  - Algorithms and Hardware
    - *Hardware includes multicore, GPU, FPGA,…*
- Concurrency at all levels
- A major challenge is to realize these approaches in code
  - Most compilers do poorly with important kernels in computational science
  - Three examples - sparse matrix vector product, dense matrix-matrix multiply, flux calculation

Sparse Matrix-Vector Product

- Common operation for optimal (in floating-point operations) solution of linear systems
- Sample code (in C):

```
for row=1,n
    m   = i[row] - i[row-1];
    sum = 0;
    for k=1,m
        sum += *a++ * x[*j++];
    y[i] = sum;
```

- Data structures are a[nnz], j[nnz], i[n], x[n], y[n]

Simple Performance Analysis

- Memory motion:
  - nnz (sizeof(double) + sizeof(int)) +
    n (2*sizeof(double) + sizeof(int))
  - Assume a perfect cache (never load same data twice; only compulsory loads)
- Computation
  - nnz multiply-add (MA)
- Roughly 12 bytes per MA
- Typical WS node can move 1-4 bytes/MA
  - Maximum performance is 8-33% of peak

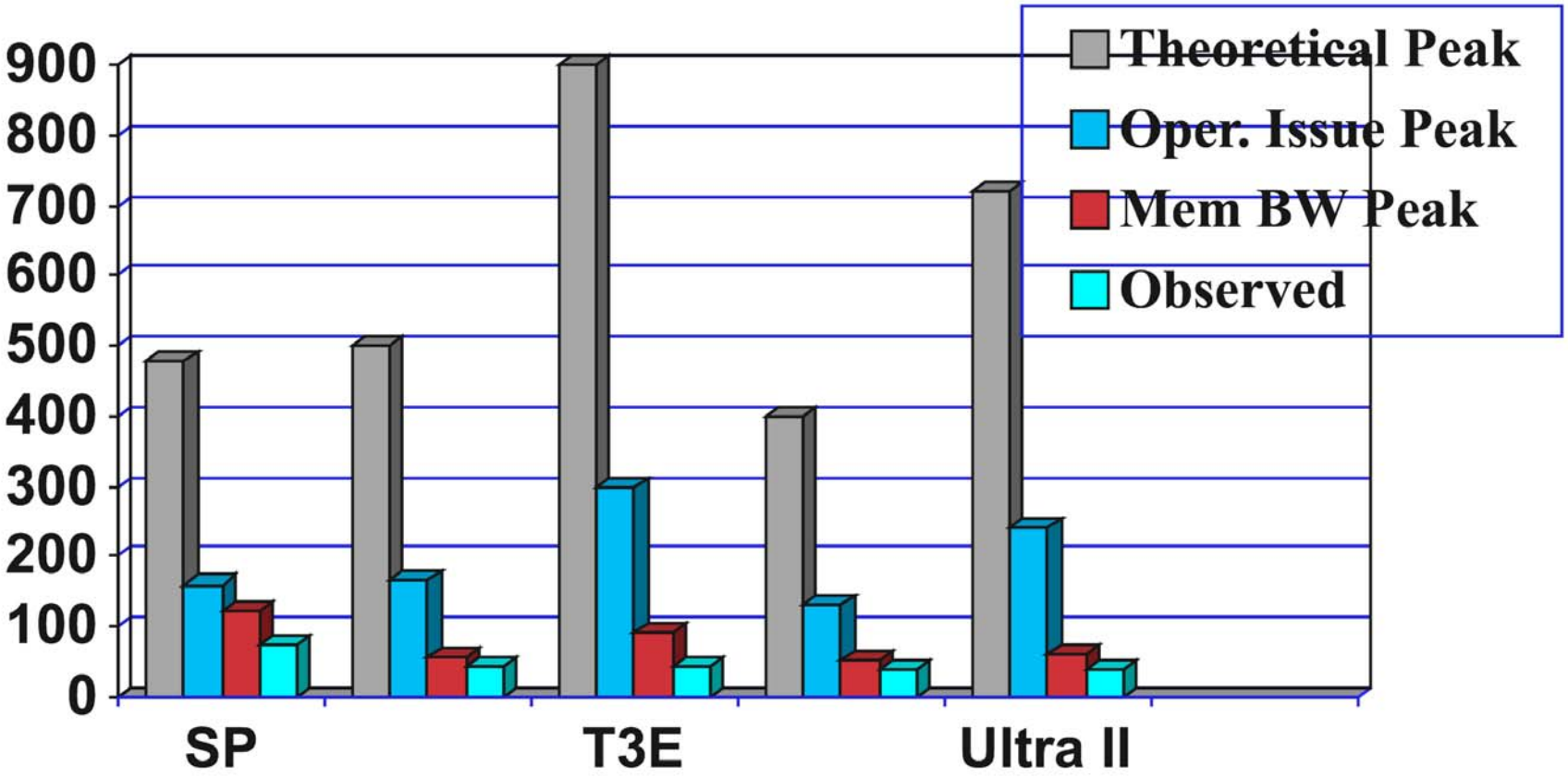
More Performance Analysis

- Instruction Counts:
  - nnz (2*load-double + load-int + mult-add) +
    n (load-int + store-double)
- Roughly 4 instructions per MA
- Maximum performance is 25% of peak (33% if MA overlaps one load/store)
  - (wide instruction words can help here)
- Changing matrix data structure (e.g., exploit small block structure) allows reuse of data in register, eliminating some loads (x and j)
- Implementation improvements (tricks) cannot improve on these limits

Realistic Measures of Peak Performance

Sparse Matrix Vector Product

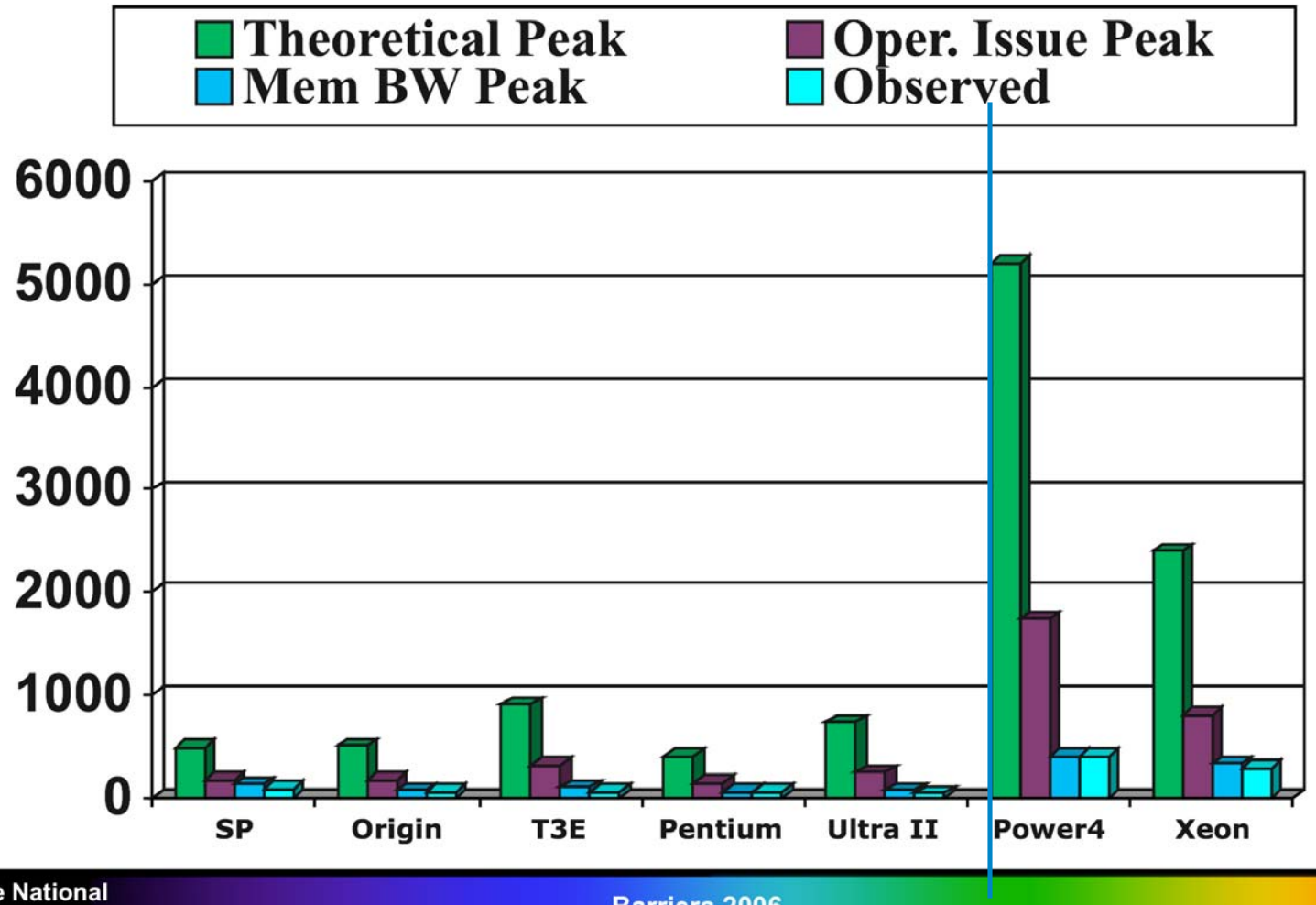
one vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



Realistic Measures of Peak Performance

Sparse Matrix Vector Product

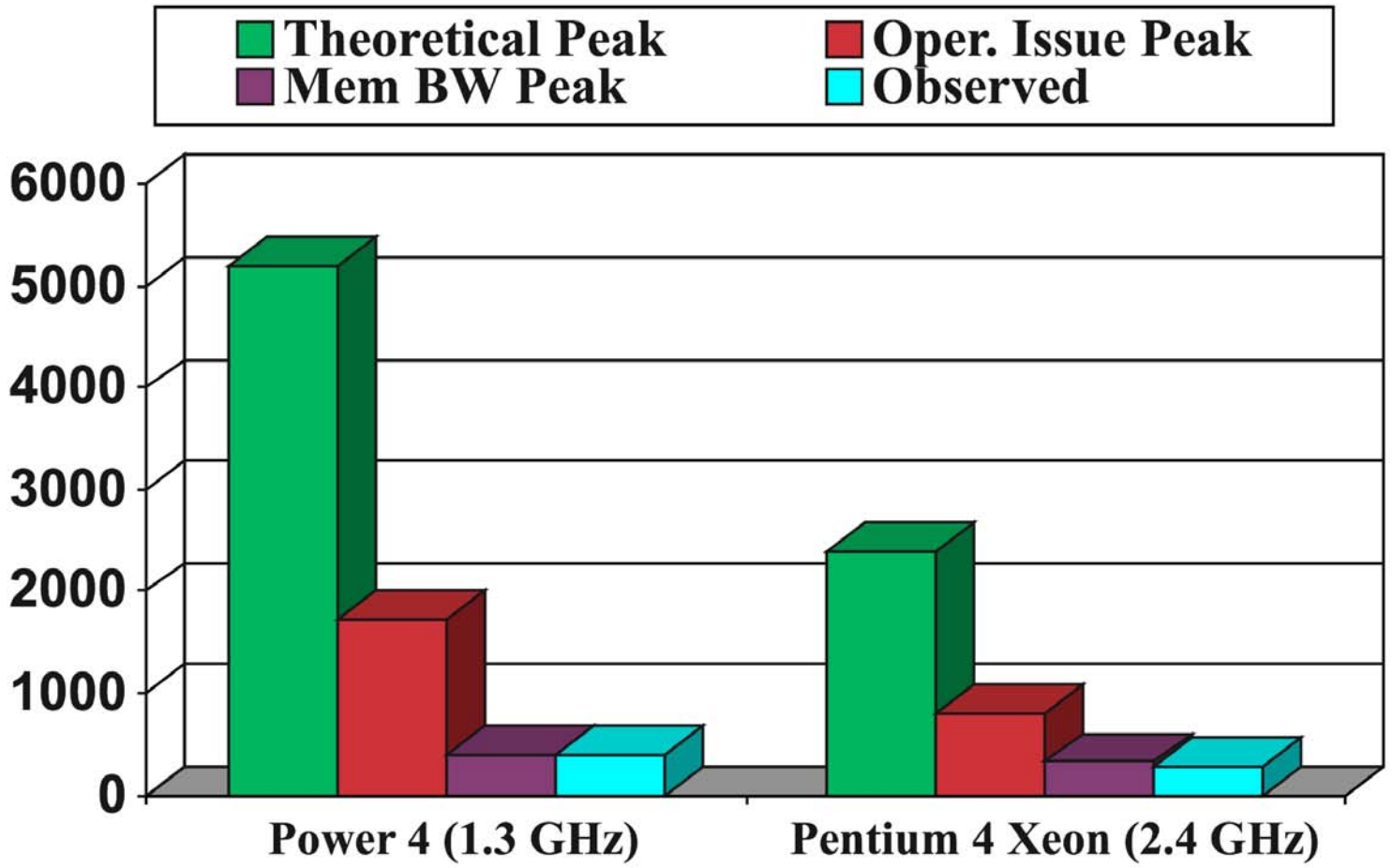
one vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



Realistic Measures of Peak Performance

Sparse Matrix Vector Product

One vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



Thanks to Dinesh Kaushik;
ORNL and ANL for compute time

What About CPU-Bound Operations?

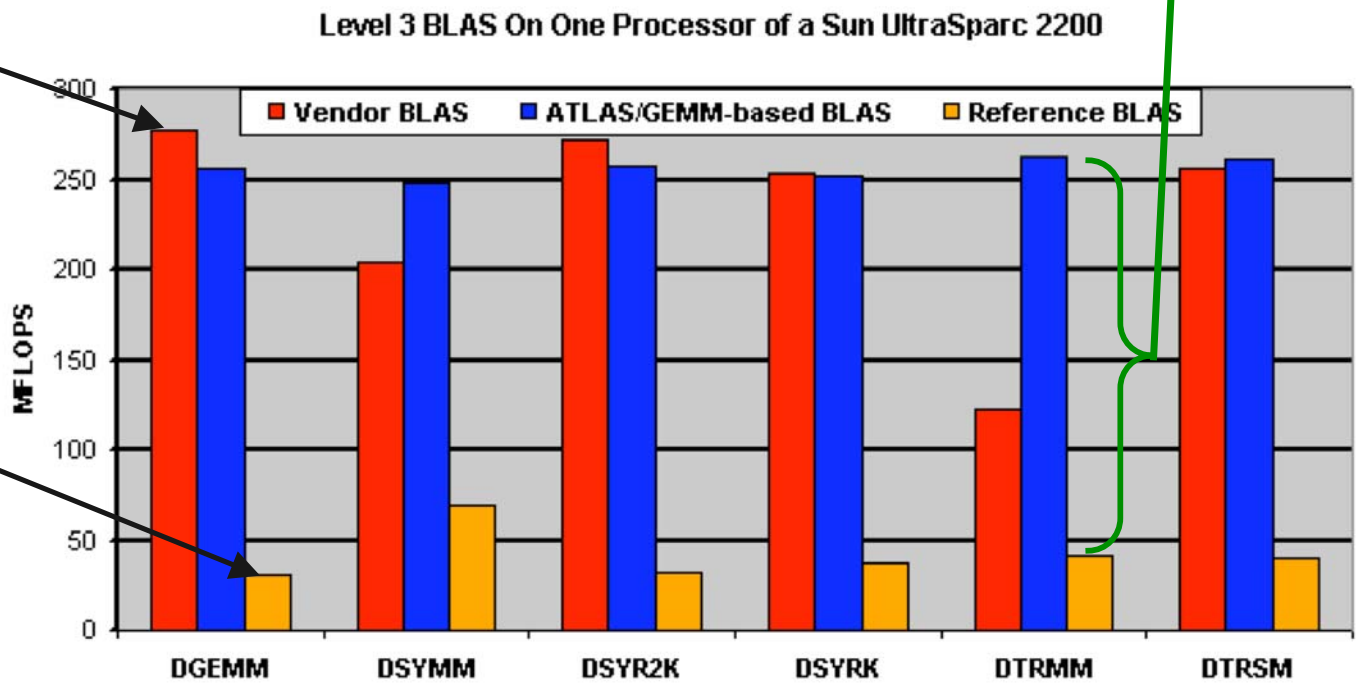
- Dense Matrix-Matrix Product
  - Probably the numerical program most studied by compiler writers
  - Core of some important applications
  - More importantly, the core operation in High Performance Linpack (HPL)
  - Should give optimal performance…

How Successful are Compilers with CPU Intensive Code?

Large gap between natural code and specialized code

Hand-tuned

Compiler



From Atlas

Enormous effort required to get good performance

Consequences of Memory/CPU Performance Gap

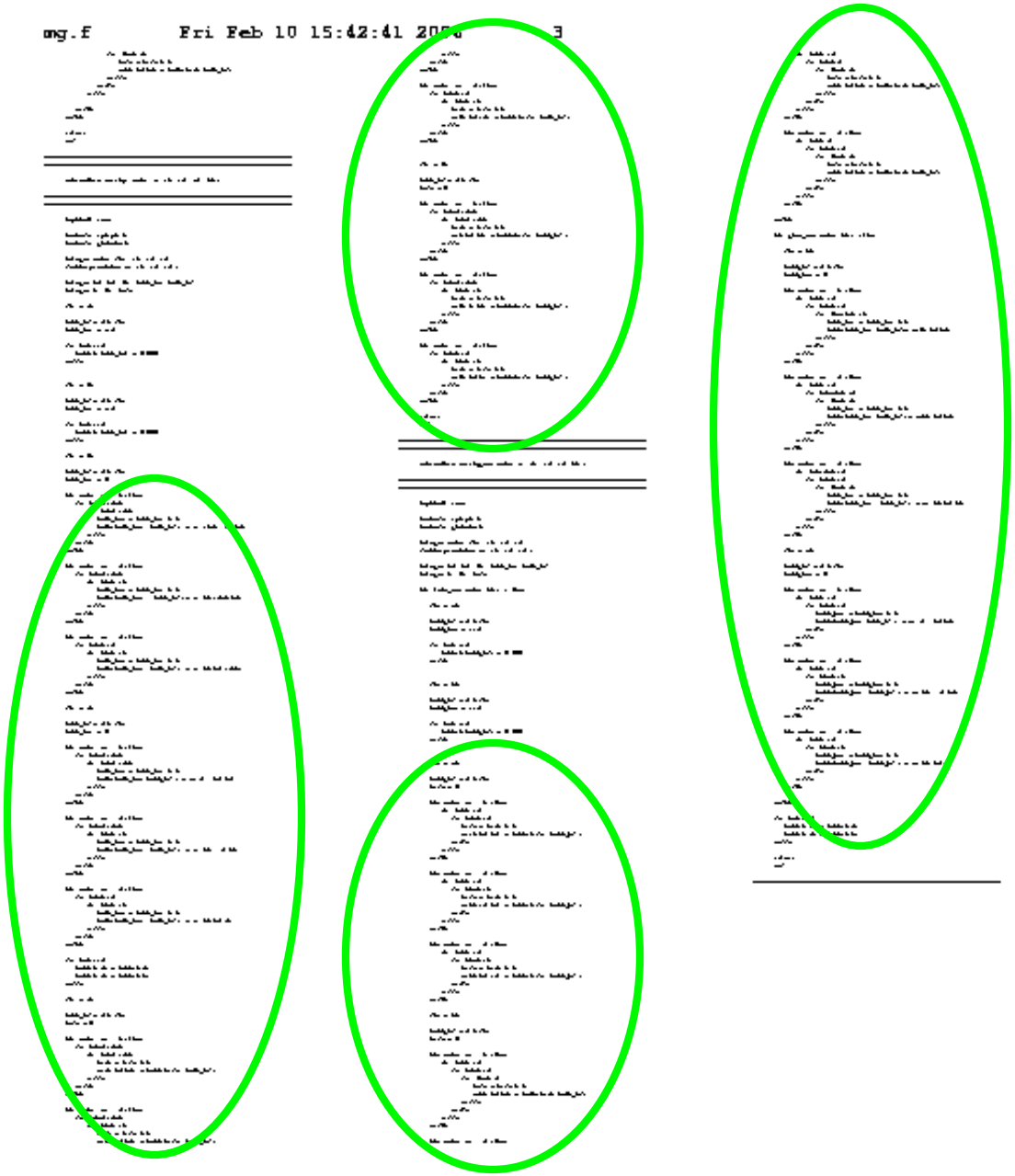
- Performance of an application may be (and often is) limited by memory bandwidth or latency rather than CPU clock
- "Peak" performance determined by the resource that is operating at full speed for the algorithm
  - Often memory system (e.g., see STREAM results)
  - Sometimes instruction rate/mix (including integer ops)
- For example, sparse matrix-vector operation performance is best estimated by using STREAM performance
  - Note that STREAM performance is delivered performance to a Fortran or C program, not memory bus rate time width
  - High latency of memory and low number of outstanding loads can significantly reduce sustained memory bandwidth

Performance for Real Applications

- Dense matrix-matrix example shows that even for well-studied, compute-bound kernels, compiler-generated code achieves only a small fraction of available performance
  - "Fortran" code uses "natural" loops, i.e., what a user would write for most code
  - Others use multi-level blocking, careful instruction scheduling etc.
- Algorithms design also needs to take into account the capabilities of the system, not just the hardware
  - Example: Cache-Oblivious Algorithms (http://supertech.lcs.mit.edu/cilk/papers/abstracts/abstract4.html)
- Adding concurrency (whether multicore or multiple processors) just adds to the problems…

Distributed Memory code

- Single node performance is clearly a problem.
- What about parallel performance?
  - Many successes at scale (e.g., Gordon Bell Prizes for >100TF on 64K BG nodes
  - Some difficulties with load-balancing, designing code and algorithms for latency, but skilled programmers and applications scientists have been remarkably successful
- Is there a problem?
  - There is the issue of **productivity**. Consider the NAS parallel benchmark code for Multigrid (mg.f):



What is the problem?
The user is responsible for all steps in the decomposition of the data structures across the processors

Note that this does give the user (or someone) a great deal of flexibility, as the data structure can be distributed in arbitrary ways across arbitrary sets of processors

Another example…

Manual Decomposition of Data Structures

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31
32	33	36	37	40	41	44	45
34	35	38	39	42	43	46	47
48	49	52	53	56	57	60	61
50	51	54	55	58	59	62	63

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

- Trick!
  - This is from a paper on dense matrix tiling for uniprocessors!
- This suggests that managing data decompositions is a common problem for real machines, whether they are parallel or not
  - *Not just an artifact of MPI-style programming*
  - Aiding programmers in data structure decomposition is an important part of solving the productivity puzzle

Possible solutions

- Single, integrated system
  - Best choice when it works
    - *Matlab*
- Current Terascale systems and many proposed petascale systems exploit hierarchy
  - Successful at many levels
    - *Cluster hardware*
    - *OS scalability*
  - We should apply this to productivity software
    - *The problem is hard*
    - *Apply classic and very successful Computer Science strategies to address the complexity of generating fast code for a wide range of user-defined data structures.*
- How can we apply hierarchies?
  - One approach is to use libraries
    - *Limited by the operations envisioned by the library designer*
  - Another is to enhance the users ability to express the problem in source code

Annotations

- Aid in the introduction of hierarchy into the software
  - Its going to happen anyway, so make a virtue of it
- Create a “market” or ecosystem in transformation tools
- Longer term issues
  - Integrate annotation language into “host” language to ensure type safety, ensure consistency (both syntactic and semantic), closer debugger integration, additional optimization opportunities through information sharing, …

Examples of the Challenges

- Fast code for DGEMM (dense matrix-matrix multiply)
  - Code generated by ATLAS omitted to avoid blindness ☺
  - Example code from “Superscalar GEMM-based Level 3 BLAS”, Gustavson et al on the next slide
- PETSc code for sparse matrix operations
  - Includes unrolling and use of registers
  - Code for diagonal format is fast on cache-based systems but slow on vector systems.
    - *Too much code to rewrite by hand for new architectures*
- MPI implementation of collective communication and computation
  - Complex algorithms for such simple operations as broadcast and reduce are far beyond a compiler’s ability to create from simple code

A fast DGEMM (sample)

```
      SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
     $                   BETA, C, LDC )
...
               UISEC = ISEC-MOD( ISEC, 4 )
               DO 390 J = JJ, JJ+UJSEC-1, 4
                  DO 360 I = II, II+UISEC-1, 4
                     F11 = DELTA*C( I,J )
                     F21 = DELTA*C( I+1,J )
                     F12 = DELTA*C( I,J+1 )
                     F22 = DELTA*C( I+1,J+1 )
                     F13 = DELTA*C( I,J+2 )
                     F23 = DELTA*C( I+1,J+2 )
                     F14 = DELTA*C( I,J+3 )
                     F24 = DELTA*C( I+1,J+3 )
                     F31 = DELTA*C( I+2,J )
                     F41 = DELTA*C( I+3,J )
                     F32 = DELTA*C( I+2,J+1 )
                     F42 = DELTA*C( I+3,J+1 )
                     F33 = DELTA*C( I+2,J+2 )
                     F43 = DELTA*C( I+3,J+2 )
                     F34 = DELTA*C( I+2,J+3 )
                     F44 = DELTA*C( I+3,J+3 )
                     DO 350 L = LL, LL+LSEC-1
                        F11 = F11 + T1( L-LL+1, I-II+1 )*
     $                                T2( L-LL+1, J-JJ+1 )
                        F21 = F21 + T1( L-LL+1, I-II+2 )*
     $                                T2( L-LL+1, J-JJ+1 )
                        F12 = F12 + T1( L-LL+1, I-II+1 )*
     $                                T2( L-LL+1, J-JJ+2 )
                        F22 = F22 + T1( L-LL+1, I-II+2 )*
     $                                T2( L-LL+1, J-JJ+2 )
                        F13 = F13 + T1( L-LL+1, I-II+1 )*
     $                                T2( L-LL+1, J-JJ+3 )
                        F23 = F23 + T1( L-LL+1, I-II+2 )*
     $                                T2( L-LL+1, J-JJ+3 )
                        F14 = F14 + T1( L-LL+1, I-II+1 )*
     $                                T2( L-LL+1, J-JJ+4 )
                        F24 = F24 + T1( L-LL+1, I-II+2 )*
     $                                T2( L-LL+1, J-JJ+4 )
                        F31 = F31 + T1( L-LL+1, I-II+3 )*
     $                                T2( L-LL+1, J-JJ+1 )
                        F41 = F41 + T1( L-LL+1, I-II+4 )*
     $                                T2( L-LL+1, J-JJ+1 )
                        F32 = F32 + T1( L-LL+1, I-II+3 )*
     $                                T2( L-LL+1, J-JJ+2 )
                        F42 = F42 + T1( L-LL+1, I-II+4 )*
     $                                T2( L-LL+1, J-JJ+2 )
                        F33 = F33 + T1( L-LL+1, I-II+3 )*
     $                                T2( L-LL+1, J-JJ+3 )
                        F43 = F43 + T1( L-LL+1, I-II+4 )*
     $                                T2( L-LL+1, J-JJ+3 )
                        F34 = F34 + T1( L-LL+1, I-II+3 )*
     $                                T2( L-LL+1, J-JJ+4 )
                        F44 = F44 + T1( L-LL+1, I-II+4 )*
     $                                T2( L-LL+1, J-JJ+4 )
                     CONTINUE
...
*     End of DGEMM.
*
      END
```

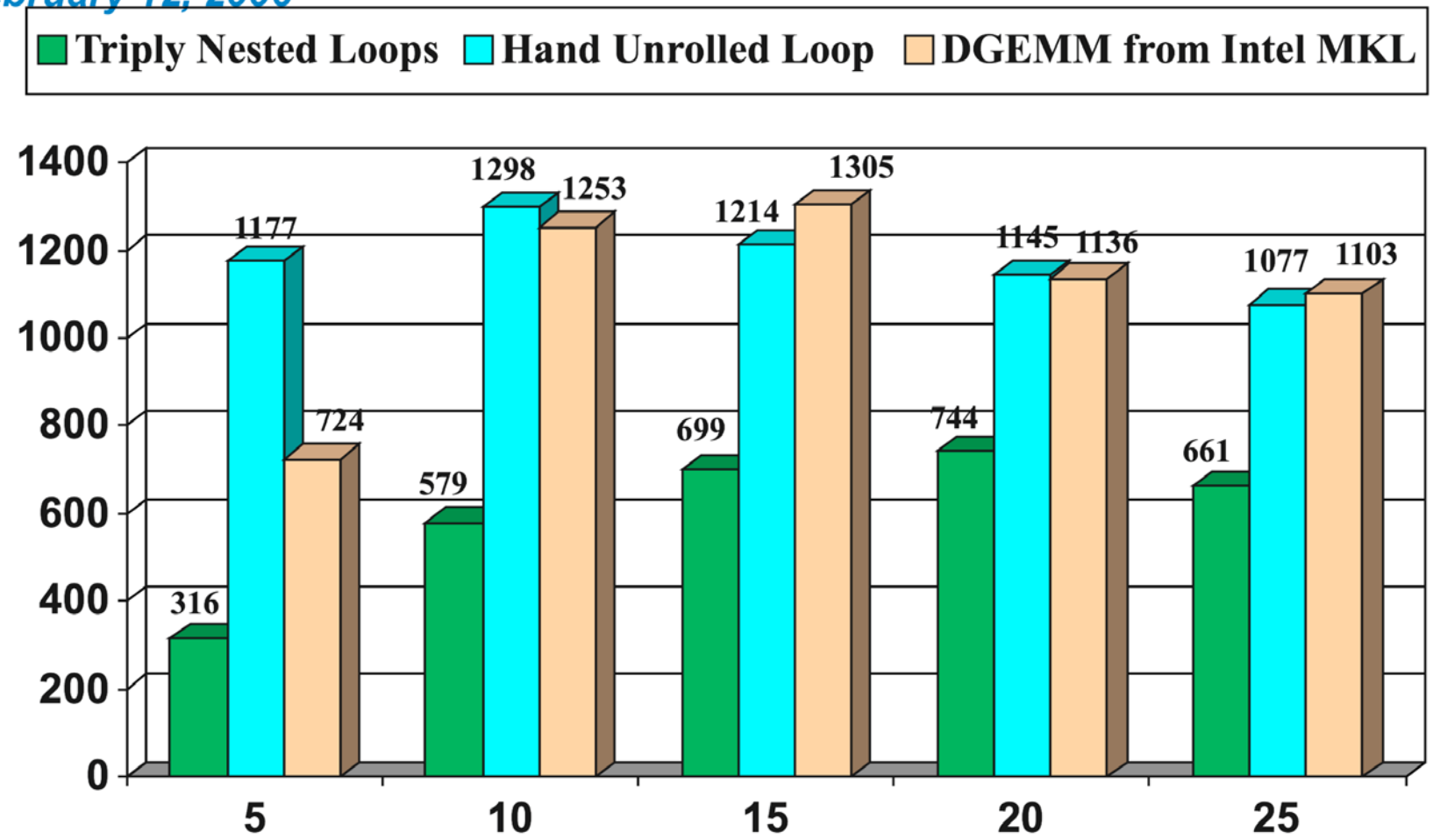
Why not just

```
do i=1,n
   do j=1,m
      c(i,j) = 0
      do k=1,p
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
   enddo
enddo
```

Note: This is just part of DGEMM!

*Performance of Matrix-Matrix Multiplication (MFlops/s vs. n2; n1 = n2; n3 = n2*n2)*

Intel Xeon 2.4 GHz, 512 KB L2 Cache, Intel Compilers at –O3 (Version 8.1), February 12, 2006



Observations About Performance Programming

- Much use of mechanical transformations of code to achieve better performance
  - Compilers do not do this well
    - *Too many other demands on the compiler*
- Use of carefully crafted algorithms for specific operations such as allreduce, matrix-matrix multiply
  - Far more challenging than the performance transformations
- Increasing acceptance of some degree of automation in creating code
  - ATLAS, PhiPAC, TCE
  - Source-to-source transformation systems
    - *E.g., ROSE, Aspect Oriented Programming (asod.net)*

Potential challenges faced by languages

1. Time to develop the language.
2. Divergence from mainstream compiler and language development.
3. Mismatch with application needs.
4. Performance.
5. Performance portability.
6. Concern of application developers about the success of the language.

- Understanding these provides insights into potential solutions
- Annotations can *complement* language research by using the principle of *separation of concerns*
- The annotation approach can be applied to *new* languages, as well

Key Observations

- 90/10 rule
  - current languages adequate for 90% of code
  - 10% of code causes 90% of trouble
- Memory hierarchy issues a major source of problems
  - Significant effort is put into relatively mechanical transformations of code
  - Other transformations are avoided because of their negative impact on the readability and maintainability of the code.
    - *Example is loop fusion for routines that sweep over a mesh to apply different physics. Fusion, needed to reduce memory bandwidth requirements, breaks modularity of routines written by different groups.*
- Coordination of distributed data structures another major source of problems
  - But the need for performance encourages a global/local separation
    - *Reflected in PGAS languages*
- New languages may help, but not anytime soon
  - New languages will never be the entire solution
  - Applications need help now

One Possible Approach

- Use annotations to augment existing languages
  - Not a new approach; used in HPF, OpenMP, others
  - Some applications already use this approach for performance portability
    - *WRF weather code*
- Annotations do have limitations
  - Fits best when most of the code is independent of the parts affected by the annotations
  - Limits optimizations that are available to approaches that augment the language (e.g., telescoping languages)
- But they also have many advantages…

Annotations example: STREAM triad.c for BG/L

```
void triad(double *a, double *b, d
{
  int i;
  double ss = 1.2;
  /* --Align;;var:a,b,c;; */
  for (i=0; i<n; i++)
    a[i] = b[i] + ss*c[i];
 /* --end Align */
}
```

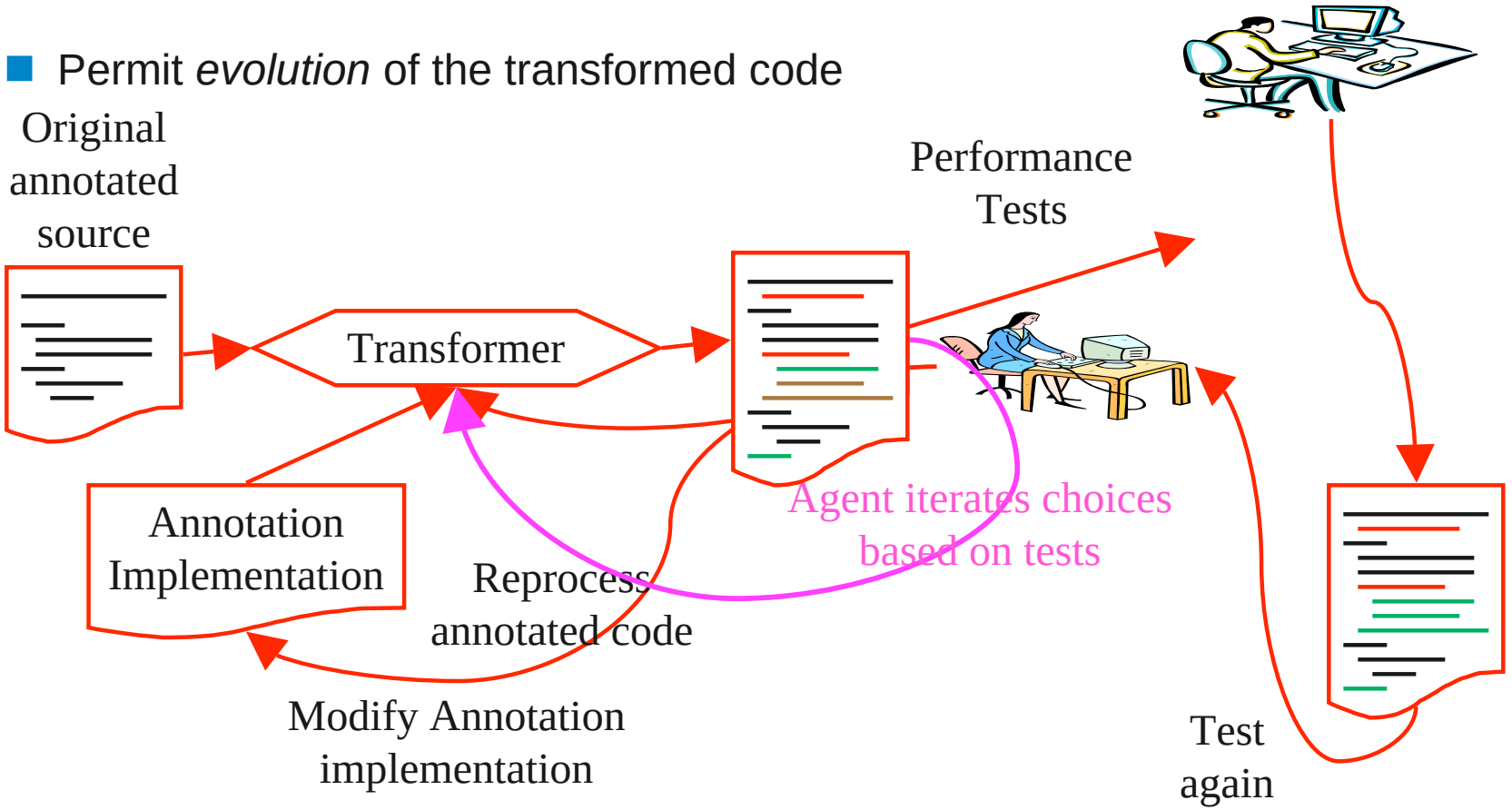
```
void triad(double *a, double *b, double *c, int n)
{
#pragma disjoint (*c,*a,*b)
  int i;
  double ss = 1.2;
  /* --Align;;var:a,b,c;; */
 if ( ((int)(a) | (int)(b) | (int)(c)) & 0xf == 0) {
   __alignx(16,a);
   __alignx(16,b);
   __alignx(16,c);
  for (i=0;i<n;i++) {
    a[i] = b[i] + ss*c[i];
 }
 }
 else {
    for (i=0;i<n;i++) {
        a[i]=b[i] + ss*c[i];
 }
  /* --end Align */
}
```

Simple annotation example: STREAM triad.c on BG/L

Size	No Annotations (MB/s)	Annotations (MB/s)	
10	1920.00	2424.24	
100	3037.97	6299.21	
1000	3341.22	8275.86	2.5X
10000	1290.81	3717.88	
50000	1291.52	3725.48	
100000	1291.77	3727.21	2.9X
500000	1291.81	1830.89	
1000000	1282.12	1442.17	
2000000	1282.92	1415.52	
5000000	1290.81	1446.48	1.12X

Code Development Cycle

- Permit *evolution* of the transformed code



Advantages of annotations

- These parallel the challenges for languages

1. Speeds development and deployment by using source-to-source transformations.
   - Higher-quality systems can preserve the readability of the source code, avoiding one of the classic drawbacks of preprocessor and source-to-source systems.
2. Leverages mainstream language developments by building on top of those languages, not replacing them.
3. Provides a simpler method to match application needs by allowing experts to develop abstractions tuned to the needs of a class (or even a single important) application.
   - Also enables the evaluation of new features and data structures

Advantages of annotations (con't)

4. Provides an effective approach for addressing performance issues by permitting (but not requiring) access by the programmer to low-level details.
   - Abstractions that allow domain or algorithm-specific approaches to performance can be used because they can be tuned to smaller user communities than is possible in a full language.
5. Improves performance portability by abstracting platform-specific low-level optimization code.
6. Preserves application investment in current languages.
   - Allows use of existing development tools (debuggers) and allows maintenance and development of code independent of the tools the process the annotations.

Is This Ugly?

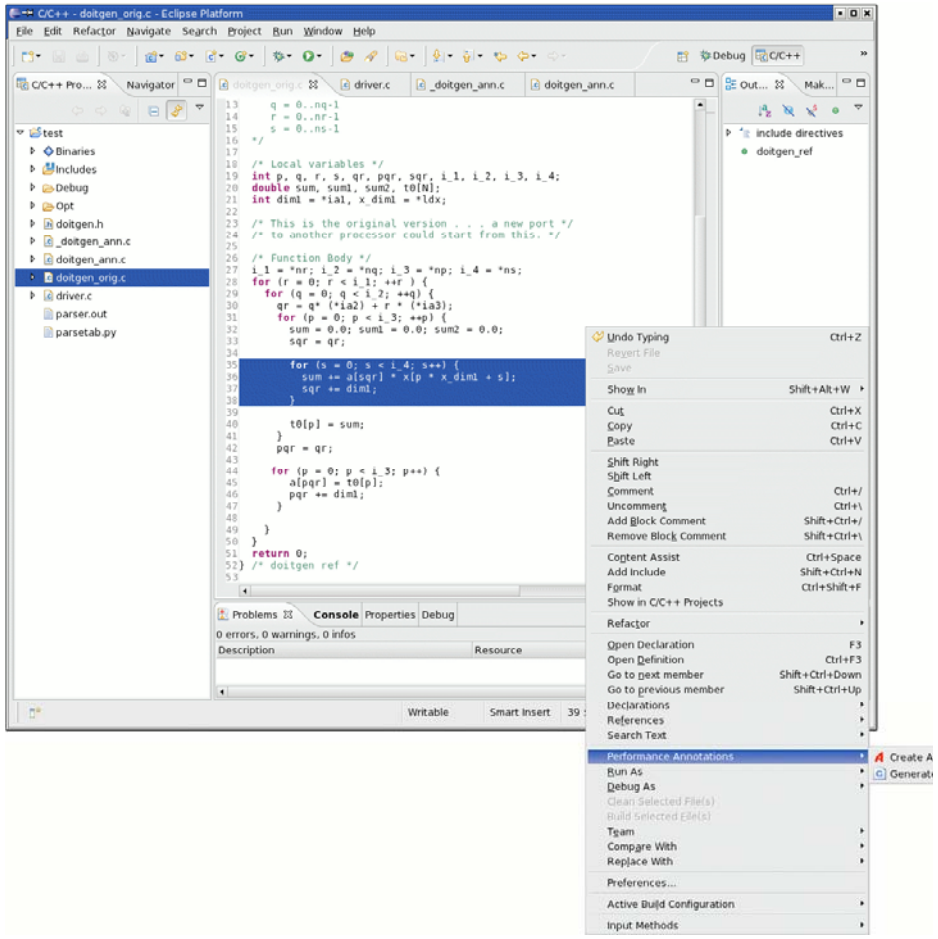
- You bet!
  - But it starts the process of considering the code generation process as consisting of a hierarchy of solutions
  - Separates the integration of the tools as seen by the user from the integration as seen by “the code”
- It can evolve toward a cleaner approach, with well-defined interfaces between hierarchies, and with a compilation-based approach to provide better syntax and semantic analysis
- But only if we accept the need for a hierarchical, compositional approach.
- This complements rather than replaces advances in languages, such as global view approaches

- In the near term, how do these ideas apply to multicore processors. Here are my top three …

Three Ways to Make Multicore Work

- **Number 3:**
- Software Engineering: Better ways to restructure codes
  - E.g. Loop fusion (vs the more maintainable and understandable to the computational scientist approach of using separate loops). Need to present the computational scientist with the best code to maintain and change, while efficiently managing the creation of more memory-bandwidth-friendly codes. Must manage the issues mentioned by Ken
  - Library routine fusion (telescoping languages)
    - *While libraries provide good abstractions and often better implementations, those very abstractions can introduce extra memory motion*
  - Tools to manage locality
    - *Compile time (local/global?) and Runtime (memory views, perhaps similar to file views in parallel file systems)*

Source code transformation tool for performance annotations, thanks to Boyanna Norris



Three Ways to Make Multicore Work

- **Number 2:**
- Programming Models: Work with the system to coordinate data motion
  - Vectors, Streams, Scatter/Gather, …
  - Provide better compile and runtime abstractions about reuse and locality of data
  - Stop pretending that we can provide an efficient, single-clock-cycle-to-memory, programming model and help programmers express what really happens (but maintaining an abstraction so that codes are not instance-specific)
  - I didn't say programming languages
  - I didn't say threads
    - *See, e.g., Edward A. Lee, "The Problem with Threads," Computer, vol. 39, no. 5, pp. 33-42, May, 2006.*
    - *"Night of the Living Threads", http://weblogs.mozillazine.org/roc/archives/2005/12/night_of_the_living_threads.html, 2005*
    - *"Why Threads Are A Bad Idea (for most purposes)" John Ousterhout (~2004)*
    - *"If I were king: A proposal for fixing the Java programming language's threading problems" http://www-128.ibm.com/developerworks/library/j-king.html, 2000*

Three Ways to Make Multicore Work

- **Number 1:**
- Mathematics: Do more computational work with less data motion
  - E.g., Higher-order methods
    - *Trades memory motion for more operations per word, producing an accurate answer in less elapsed time than lower-order methods*
  - Different problem decompositions (no stratified solvers)
    - *The mathematical equivalent of loop fusion*
    - *E.g., nonlinear Schwarz methods*
  - Ensemble calculations
    - *Compute ensemble values directly*
  - It is time (really past time) to rethink algorithms for memory locality and latency tolerance

Conclusions

- It's the memory hierarchy
- A pure, compiler based approach is not credible until
  1. $\frac{\min(\text{performance of compiler on MM})}{\max(\text{performance of hand-tuned MM})} > 0.9$
  2. The "condition number" of that ratio is small (less than 2)
  3. Your favorite performance challenge
- Compilation is *hard*!
- At the node, the memory hierarchy limits performance
  - Architectural changes can help (e.g., prefetch, more pending loads/stores) but will always need algorithmic and programming help
  - Algorithms must adapt to the realities of modern architectures
- Between nodes, complexity of managing distributed data structures limits productivity and the ability to adopt new algorithms
  - Domain (or better, data-structure) specific nano-languages, used as part of a hierarchical language approach, can help