Cache Optimization in Multicomponent Unstructured-Grid Implicit CFD Codes

William D. Gropp
MCS Division, Argonne National Laboratory

Dinesh K. Kaushik
CS Department, Old Dominion University & Argonne

David E. Keyes
CS Department, Old Dominion University & ICASE

Barry F. Smith
MCS Division, Argonne National Laboratory

http://www.mcs.anl.gov/petsc-fun3d

Organization of the Presentation

- Understanding the sources of poor per-processor performance
- Performance issues for unstructured grid solvers
- Cache and register optimizations
- Conclusions

Motivation

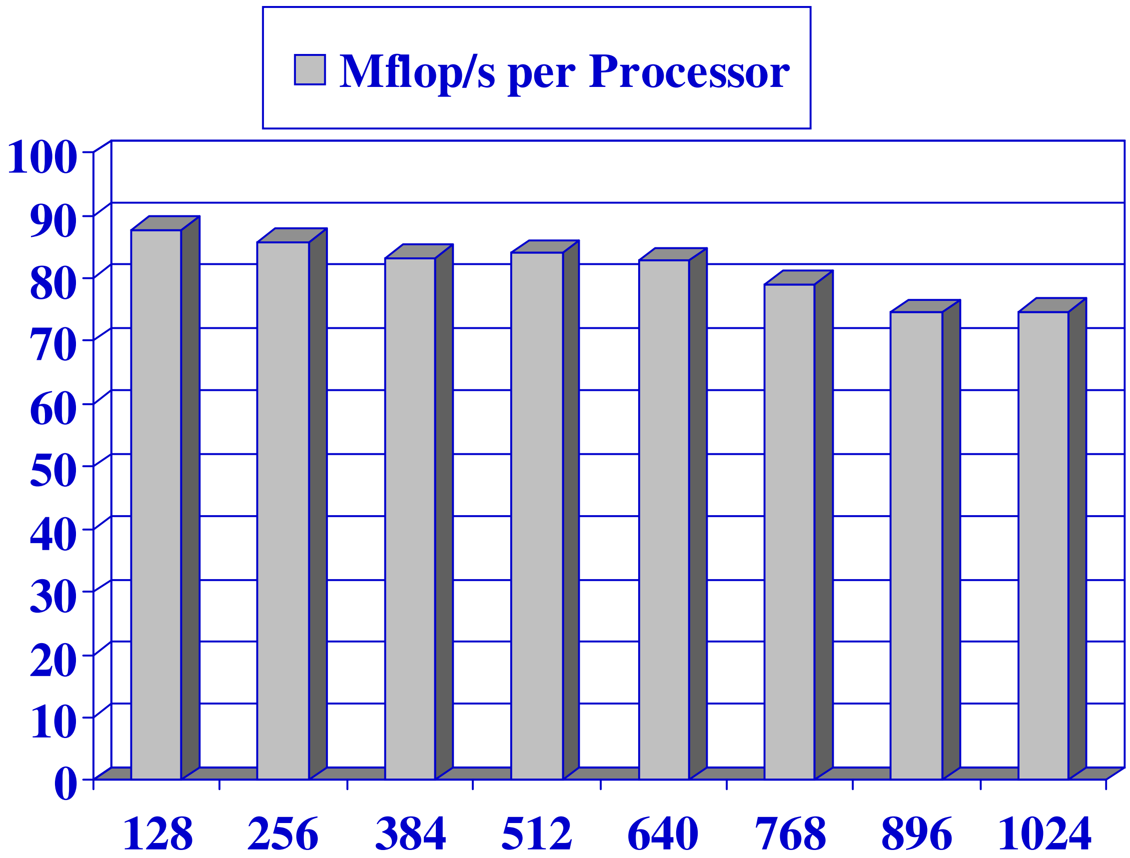
- Sequential performance on many machines is a low percentage of peak
- Per-processor performance on T3E stays fairly constant while going from 128 to 1024 processors
  - Parallel programming is easy! It is **uniprocessor** programming that is difficult
- Memory performance improvement rate (7% per year) is far behind the CPU performance growth (about 55% per year)
- The programmer can do a good job in expressing the coarse-grained concurrency but getting good cache locality is a big challenge (especially for unstructured PDE solvers)
- Getting good per processor performance is the key to achieving good parallel performance

Description of PETSc-FUN3D

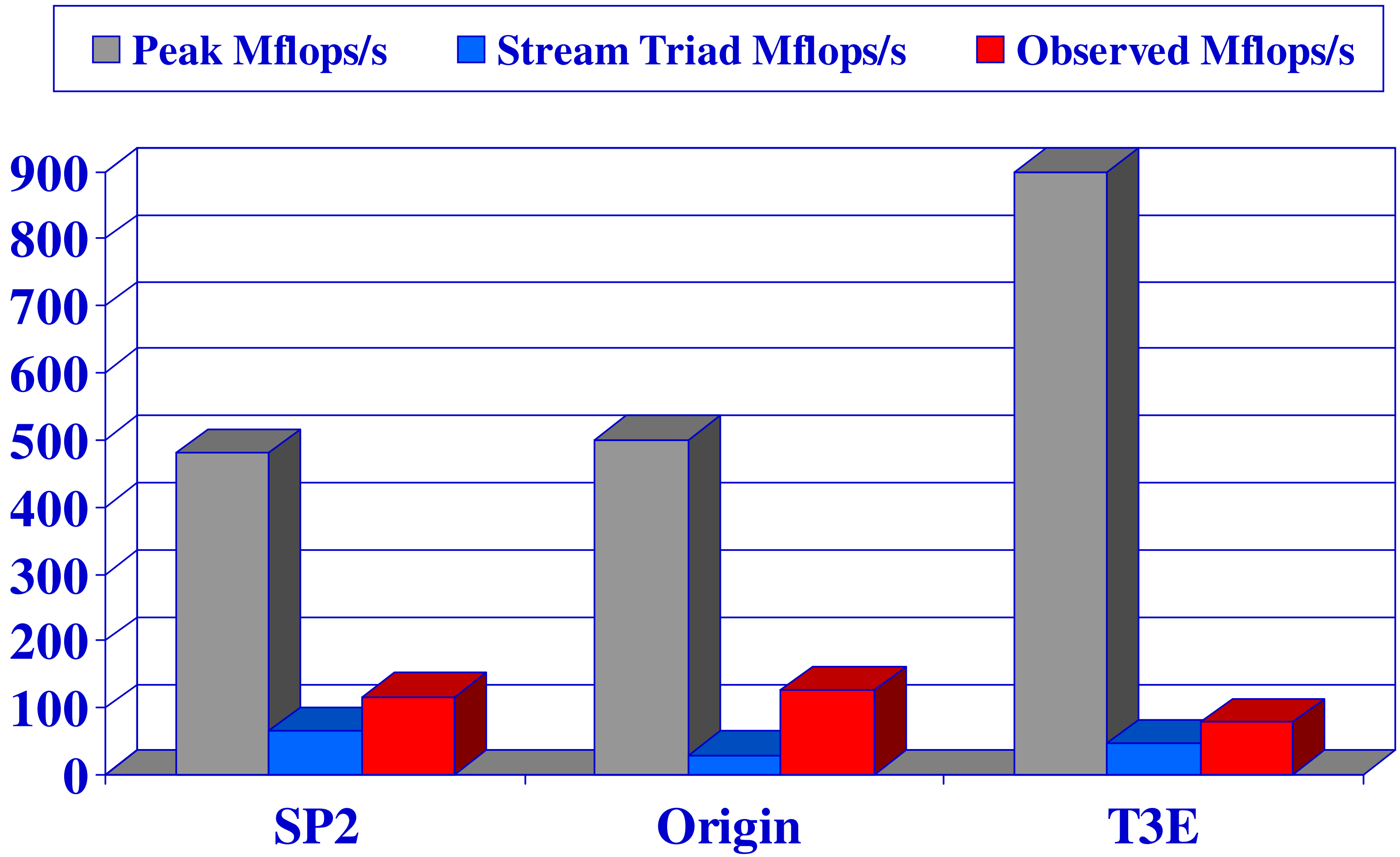
- PETSc-FUN3D is the result of porting FUN3D (developed by **W. K. Anderson, NASA Langley**) to **PETSc**
- Tetrahedral vertex-centered unstructured grid code for incompressible and compressible Euler and Navier-Stokes equations
- 1st- or 2nd-order Roe for convection and Galerkin for diffusion, and false time stepping with backward Euler for nonlinear continuation towards steady state
- Newton-Krylov-Schwarz (fully implicit, matrix free) solver; the timestep is advanced towards infinity by the switched evolution/relaxation (**SER**) of Van Leer and Mulder
- The preconditioner (incomplete LU with zero fill) in each domain is derived from from 1st-order accurate Jacobian

Per Processor Performance on T3E

Euler flow over an ONERA M6 Wing, on a tetrahedral grid of 2.8 M vertices, run up to 1024 processors of a 600 MHz T3E



Sequential Performance of PETSc-FUN3D



Three Fundamental Limiting Factors to Peak Performance

- Memory Bandwidth
  - Processor does not get data at the rate it requires
- Instruction Issue Rate
  - If the loops are load/store bound, we will not be able to do a floating point operation in every cycle even if the operands are available in primary cache
  - Several constraints (like primary cache latency, latency of floating point units etc.) are to be observed while coming up with an optimal schedule
- Fraction of Floating Point Operations
  - Every instruction is not floating point instruction

Analyzing A Simple Kernel: Sparse Matrix Product

- Sparse matrix vector product is important part of many iterative solvers
- Its performance modeling is easy
- We present simple analysis to predict better performance bounds (based on the three architectural limits) than the “marketing” peak of a processor

Performance Issues for Sparse Matrix Vector Product

- Little data reuse
- High ratio of load/store to instructions/floating-point ops
- Stalling of multiple load/store functional units on the same cache line
- Low available memory bandwidth

Sparse Matrix Vector Algorithm: A General Form

```
for every row, i {
  fetch ia(i+1)
  for j = ia(i) to ia(i + 1)  {    // loop over the non-zeros of the row
      fetch ja(j), a(j), x1(ja(j)), .....xN(ja(j))
      do N fmadd (floating multiply add)
   }
   Store y1(i) .....yN(i)
}
```

Estimating the Memory Bandwidth Limitation

Assumptions

- Perfect Cache (only compulsory misses; no overhead)
- No memory latency
- Unlimited number of loads and stores

Data Volume (AIJ Format)

m*sizeof(int) + N*(m+n)*sizeof(double))

// ia, N input (size n) and output (size m) vectors

+ nz* (sizeof(int) + sizeof(double))

// ja, and a arrays

= 4*(m+nz) + 8*(N*(m+n)+ nz)

Estimating the Memory Bandwidth Limitation II

- Number of Floating-Point Multiply Add (fmadd) Ops = N*nz
- For square matrices,

$$\text{Bytes transferred/fmadd} = \left(16 + \frac{4}{N}\right) * \frac{n}{nz} + \frac{12}{N}$$

(Since nz >> n, Bytes transferred / fmadd ~12/N)

- Similarly, for **Block AIJ** (**BAIJ**) format

$$\text{Bytes transferred/fmadd} = \left(16 + \frac{4}{N*b}\right) * \frac{n}{nz} + \left(\frac{4}{N*b} + \frac{8}{N}\right)$$

Performance Summary on 250 MHz R10000

- Matrix size, n = 90,708; number of nonzero entries, nz = 5,047,120
- Number of Vectors, N = 1, and 4

Format	Number of Vectors	Bytes / fmadd	Bandwidth		MFlops	
			Required	Achieved	Ideal	Achieved
AIJ	1	12.36	3090	276	58	45
AIJ	4	3.31	827	221	216	120
BAIJ	1	9.31	2327		84	55
BAIJ	4	2.54	635	229	305	175

Prefetching—Fully Use the Available Memory Bandwidth

- Many programs are not able to use the available memory bandwidth for various reasons
- Ideally a memory operation should be scheduled in each cycle since each cycle is a lost opportunity
- Compilers do not do enough prefetching
- Implementing and estimating the right amount of prefetching is hard

Estimating the Operation Issue Limitation

AT:address transln; **Br**: branch; **Iop**: integer op; **Fop**: floating point op; **Of**: offset calculation; **Ld**: load; **St**: store

```
for (i = 0, i < m; i++) {
        jrow = ia(i+1)                                   // 1Of, AT, Ld
        ncol =  ia(i+1) -ia(i)                           // 1 Iop
        Initialize, sum_1 …..sum_N                       //  N Ld
        for (j = 0; j < ncol; j++) {                     // 1 Ld
          fetch ja(jrow), a(jrow), x_1(ja(jrow)), …..x_N(ja(jrow))
                                                         // 1 Of, N+2 AT, and  Ld
          do N fmadd (floating multiply add)             // 2N Fop
        }                                                // 1 Iop, 1 Br
        Store sum_1…..sum_N in y_1(i) …..y_N(i)          // 1 Of, N AT, and St
    }                                                    // 1 Iop, 1 Br
```

Estimating the Operation Issue Limitation II

- Assumptions:
  - Data items are in cache
  - Each operation takes only one cycle to complete but multiple operation can graduate in one cycle
- If only one load or store can be issued in one cycle (as is the case on R10000 and many other processors), the best we can hope for is

$$\frac{\text{Number of floating point instructions}}{\text{Number of Loads and Stores}} * \text{Peak MFlops/s}$$

- Other restrictions (like primary cache latency, latency of floating point units etc.) need to be taken into account while creating the best schedule (especially on those processors where software pipelining is important)

Estimating the Fraction of Floating Point Operations

- Assumptions:
  - infinite number of functional units
  - data items are in primary cache
- Estimated number of floating point operations out of the total instructions:

$$\text{Total number of instructions completed } (I_t) = m * (3 * N + 8) + nz * (4 * N + 9)$$

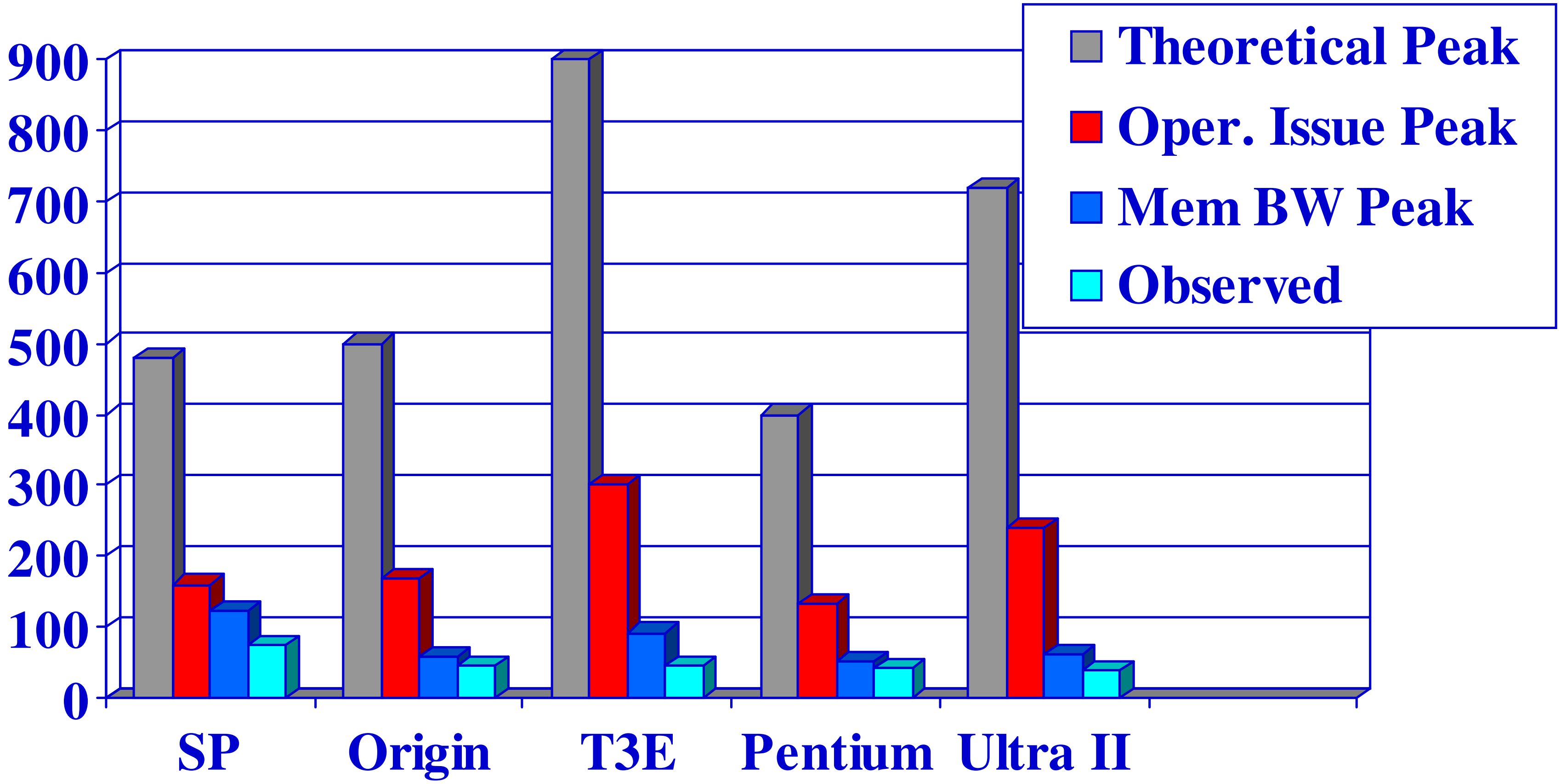
$$\text{Fraction spent on floating point work } (I_f) = \frac{2 * N * nz}{m * (3 * N + 8) + nz * (4 * N + 9)}$$

- For **N=1, I_f = 0.18 and N = 4, I_f = 0.34.**

Realistic Measures of Peak Performance

Sparse Matrix Vector Product

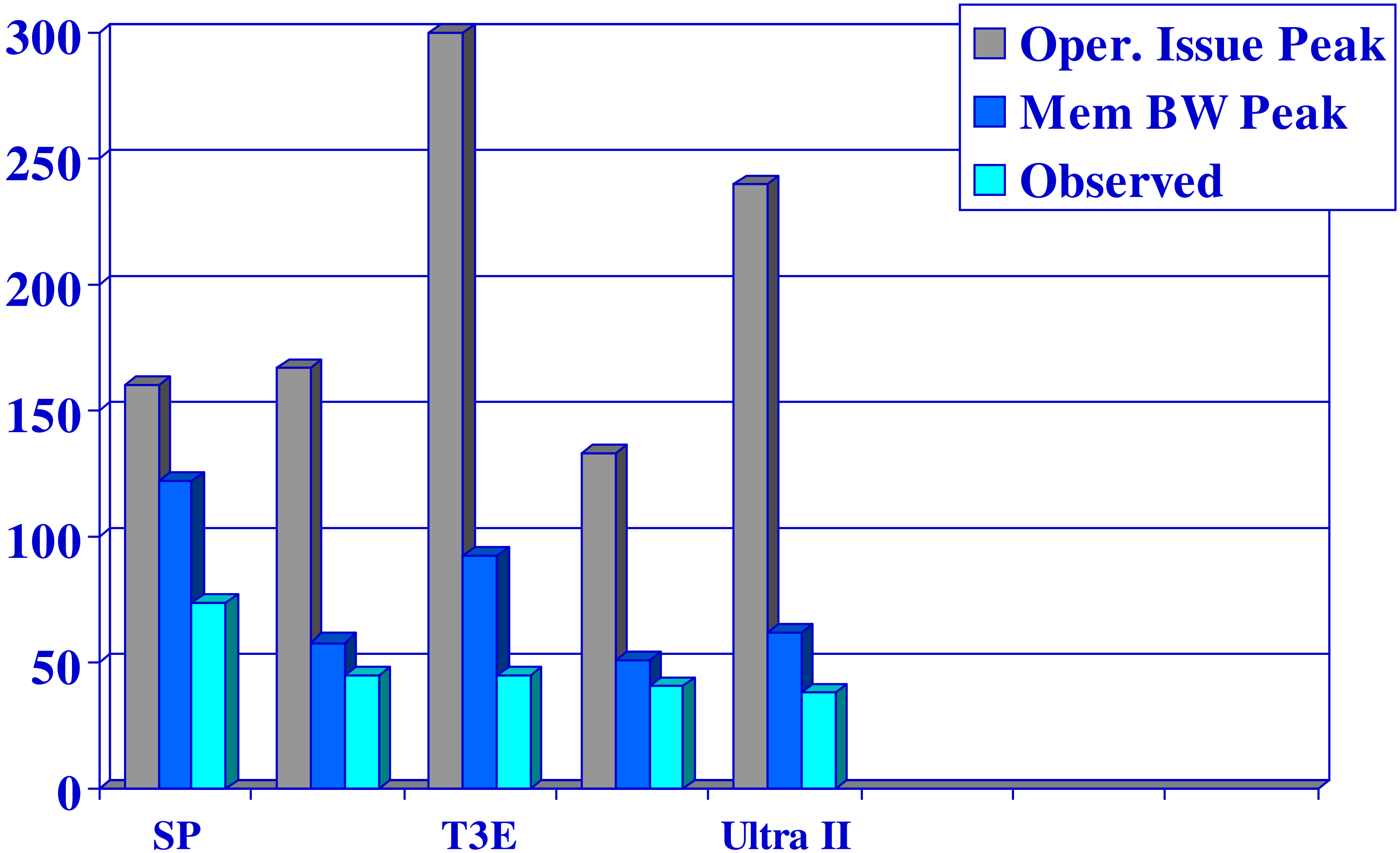
one vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



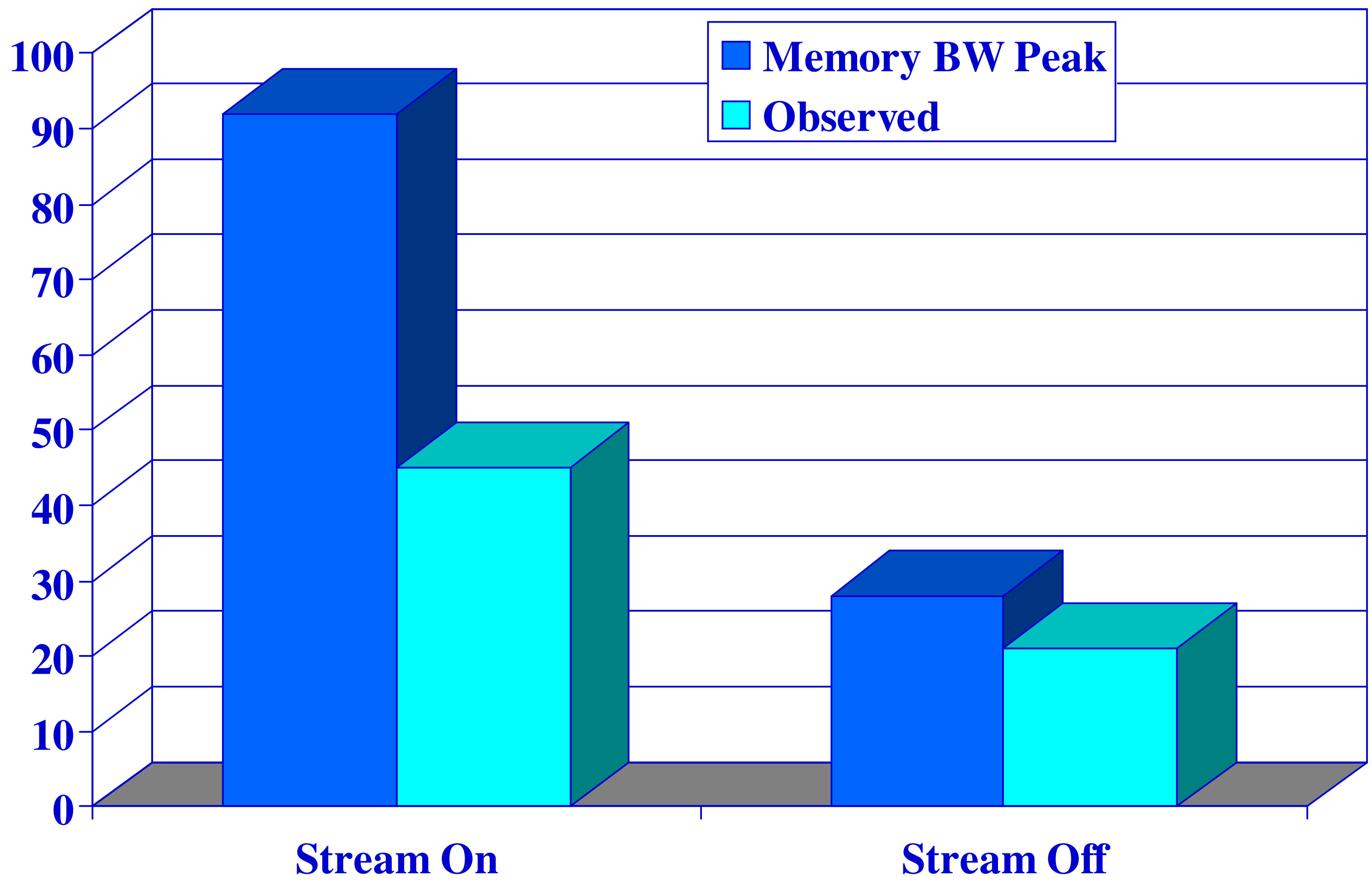
Experimental Performance

Sparse Matrix Vector Product

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



T3E Performance—A Closer Look



Implications

- Reducing memory use is critical
  - Reuse data items
  - Reuse items in cache
  - Other memory effects also important (see TLB, ahead)
- Reducing the number of non-floating-point instructions is also important
  - Reuse items in registers (reduce loads, address computation)

Enhancing Locality

- Choose data layouts that enhance locality at every level of memory hierarchy
- Storage/use patterns should follow memory hierarchy
  - **Blocks for Registers**
    - block storage format for multicomponent systems—saves CPU cycles
  - **Interlaced Data Structures for Cache**
    - Choose

      $u1,v1,w1,p1,u2,v2,w2,p2,\ldots$

      in place of

      $u1,u2,\ldots,v1,v2,\ldots,w1,w2,\ldots,p1,p2,\ldots$
  - **Subdomains for Distributed Memory**
    - "Chunky" domain decomposition for optimal surface-to-volume (communication-to-computation) ratio
  - This hierarchy is concerned with different issues than the **algorithmic efficiency** issues associated with the hierarchies of grids

Data Layouts and Reorderings

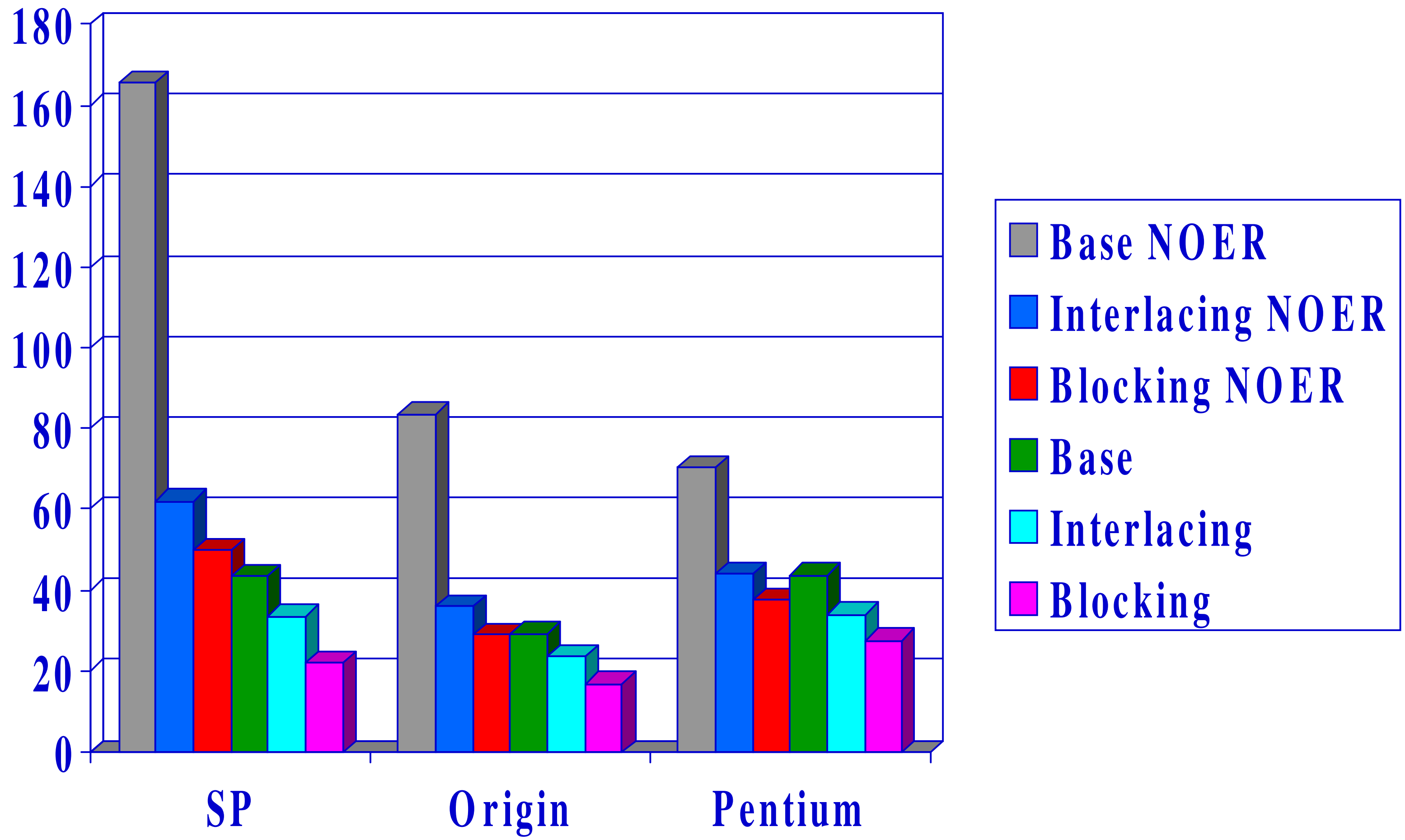
- Edge Reordering
  - Sort the nodes at either ends of the edges
  - Effectively transforms an edge based loop into a node based loop
  - Enhances temporal locality
- Node Reordering
  - Bandwidth reducing orderings will reduce the TLB and cache misses by referring to data items that are close in memory.
  - Our experience is with RCM and Sloan

Locality Enhancing Strategies in PETSc-FUN3D

- Flow over M6 wing with a grid of 22,677 vertices (90,708 DOFs incompressible; 113,385 compressible)
- Turn on each optimization one by one to isolate the effect of each
- Employed the “best” optimization flags
- Five Architectures considered: Cray T3E, IBM SP, Origin 2000, Intel Pentium, and Sun Ultra
- Impact of these techniques vary on different architectures—improvement ranges from **2.5 on Pentium to 7.5 on SP**

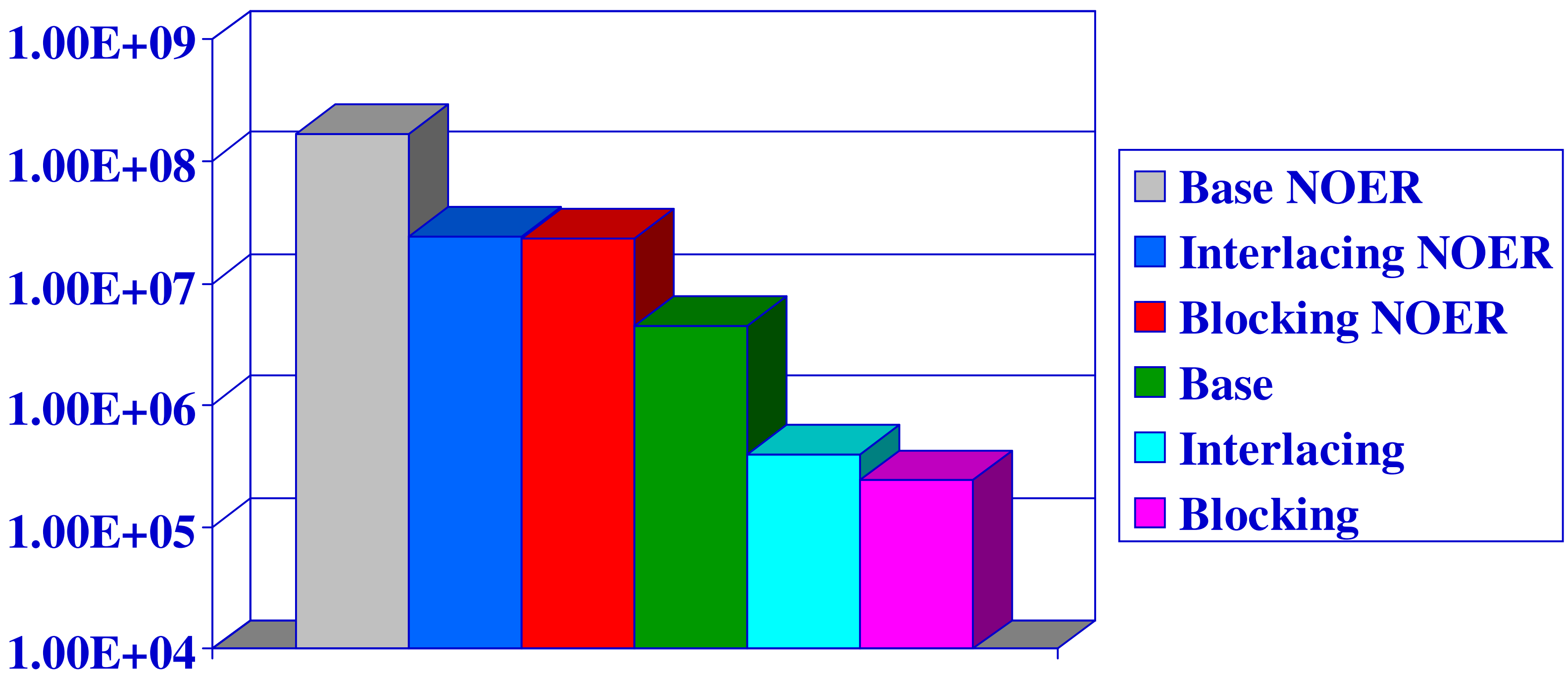
Sequential Performance—Time/iteration

SP: IBM P2SC ("thin"), 120 MHz, cache: 128 KB data and 32 KB instr
Origin: MIPS R10000, 250 MHz, cache 32 KB data/32KB instr/4MB L2
Pentium: Intel Pentium II, 400 MHz, cache: 16KBdata/16KB instr/512 KB L2

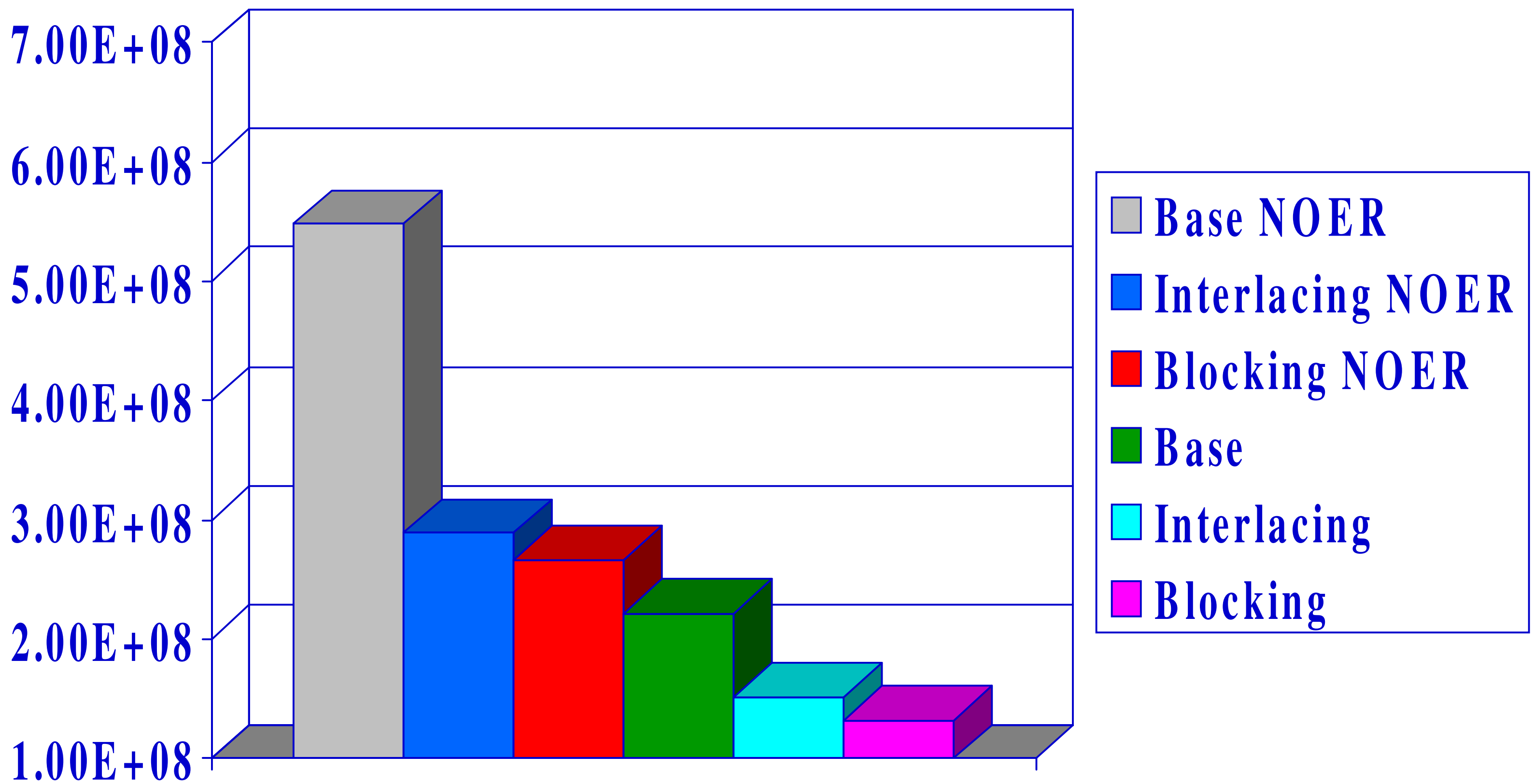


TLB Misses:
Measured Values on Origin

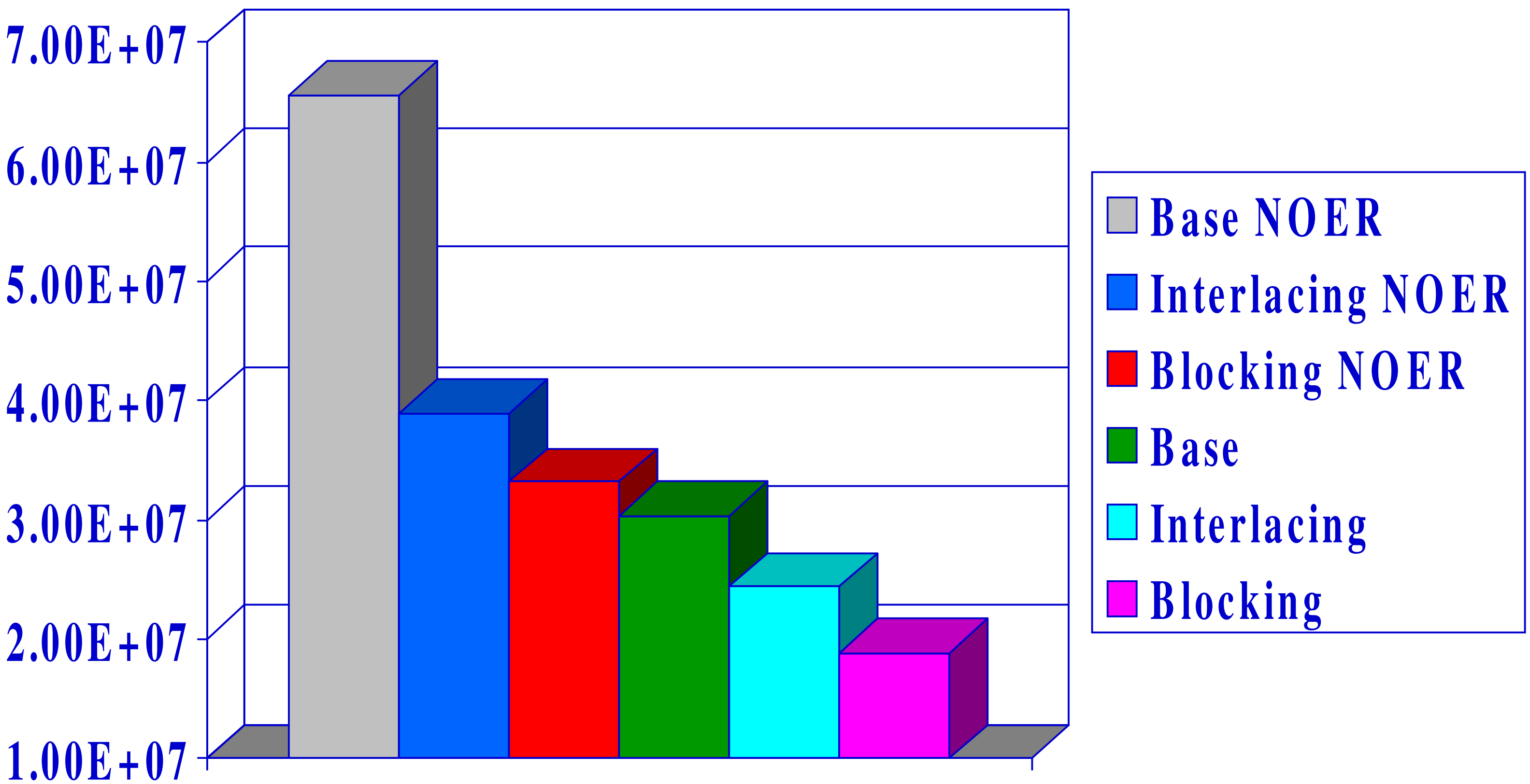
Log scale!



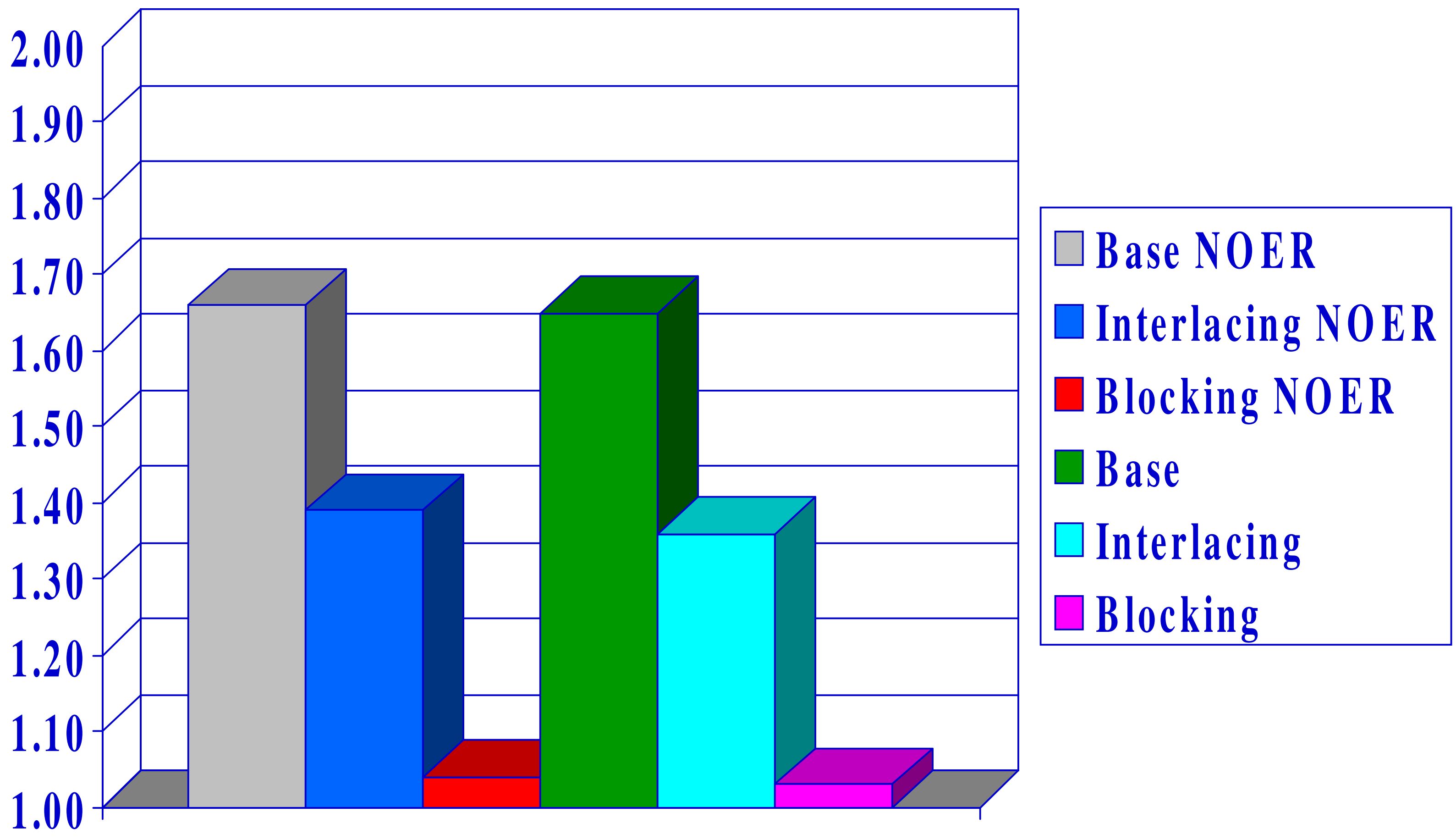
Primary Cache Misses:
Measured Values on Origin



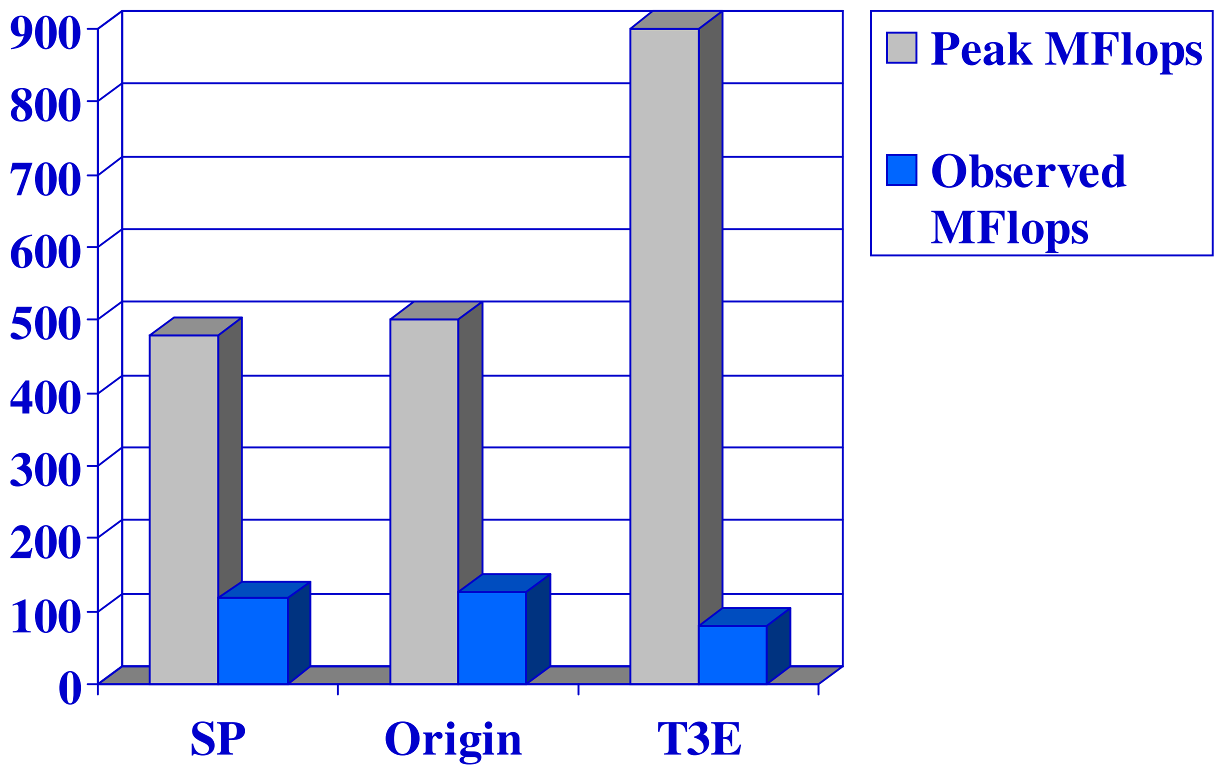
Secondary Cache Misses: Measured Values on Origin



Graduated Loads and Stores Per Floating Point Instruction



Sequential Performance of PETSc-FUN3D



Conclusions

- The per-processor performance is crucial to get good parallel performance
- Our models predict the performance of sparse matrix-vector operations on a variety of platforms, including the effects of memory bandwidth, and instruction issue rates
- The achievable "peak performance" for these operations is a small fraction of the stated peak, independent of code quality
  - compiler improvements can help but will not solve the problem
- Intelligent prefetching is required to fully utilize the available memory bandwidth
- Data structure transformations (like blocking, interlacing, and edge reordering) that enhance the temporal and spatial locality in the memory reference patterns have improved the performance by a large factor (2.5 on Pentium and 7.5 on SP2).

Future Directions

- Design better data structures and implementation strategies for sparse matrix vector and related operations
- Integrate our understanding of the performance issues with developments in block-structured algorithms to produce linear and nonlinear solvers that achieve a higher fraction of peak performance on a per-node basis
- Look at important special cases in hierarchical algorithms where our performance model recommends alternate data structures and library methods

References

- *On the interaction of Architecture and Algorithm in the Domain-Based Parallelization of an Unstructured Grid Incompressible Flow Code* (Kaushik, Keyes, and Smith), 1998, in "Proc. Of the 10th Intl. Conf. On Domain Decomposition Methods", J. Mandel et al., eds., AMS, pp. 311-319.
  - **Cache-aware focus**
- *Newton-Krylov-Schwarz Methods for Aerodynamic Problems: Compressible and Incompressible Flows on Unstructured Grids* (Kaushik, Keyes, and Smith), 1998, submitted to "Proc. of the 11th Intl. Conf. On Domain Decomposition Methods", C.-H Lai et al., eds.
  - **Multi-platform focus**
- These can be downloaded from **`http://www.cs.odu.edu/~keyes`**