Performance Optimization for the Origin 2000

http://www.cs.utk.edu/~mucci/MPPopt.html

Philip Mucci (mucci@cs.utk.edu)
Kevin London (london@cs.utk.edu)

University of Tennessee, Knoxville

Army Research Laboratory, Aug. 31 - Sep. 2

Outline

- Introduction to Performance Optimization
- Origin Architecture
- Performance Issues and Metrics
- Performance Tools
- Numerical Libraries
- Compiler Technology

Performance

- What is performance?
  - Latency
  - Bandwidth
  - Efficiency
  - Scalability
  - Execution time
- At what cost?

Performance Examples

- Operation Weather Forecasting Model
  - Scalability
- Database search engine
  - Latency
- Image processing system
  - Throughput

What is Optimization?

- Finding **hot spots** *&* **bottlenecks (profiling)**
  - Code in the program that uses a *disproportional* amount of *time*
  - Code in the program that uses system resources *inefficiently*
- Reducing **wall clock** time
- Reducing resource requirements

Types of Optimization

- Hand-tuning
- Preprocessor
- Compiler
- Parallelization

Steps of Optimization

- Profile
- Integrate libraries
- Optimize compiler switches
- Optimize blocks of code that dominate execution time
- Always examine correctness at every stage!

Performance Strategies

- Always use optimal or near optimal algorithms.
  - Be careful of resource requirements and problem sizes.
- Maintain realistic and consistent input data sets/sizes during optimization.
- Know when to stop.

Performance Strategies

- Make the Common Case Fast (Hennessy)

PROCEDURE	TIME
`main()`	13%
`procedure1()`	17%
`procedure2()`	20%
`procedure3()`	50%

- A 20% decrease of `procedure3()` results in 10% increase in performance.
- A 20% decrease of `main()` results in 2.6% increase in performance

Considerations when Optimizing

- Machine configuration, libraries and tools
- Hardware and software overheads
- Alternate algorithms
- CPU/Resource requirements
- Amdahl's Law
- Communication pattern, load balance and granularity

How high is up?

- Profiling reveals percentages of time spent in CPU and I/O bound functions.
- Correlation with representative low-level, kernel and application benchmarks.
- Literature search.
- Peak speed means nothing.
- Example: ISIS solver package

Origin 2000 Architecture

- Up to 64 nodes
- Each node has 2 R10000's running at 195 Mhz
- Each R10000 has on chip 32K instruction, 32K data caches
- Each R1000 has a 4MB off-chip unified cache.

Origin 2000 Architecture

- Each node is connected with a 624MB/sec CrayLink
- Shared memory support in hardware
- Supports message passing as well
- Variable page size (*dplace)*

R10000 Architecture

- 5 independent, pipelined, execution units
- 1 non-blocking load/store unit
- 2 asymmetric integer units (both add, sub, log)
- 2 asymmetric floating point units

R10000 Architecture

- Dynamic and speculative execution

Locality

Spatial - If location X is being accessed, it is likely that a location *near* X will be accessed *soon*.

Temporal - If location X is being accessed, it is likely that X will be accessed again *soon*.

Memory Hierarchy

Speed

Registers

Cache / TLB

<Level 2 Cache>

<Level 3 Cache>

Memory

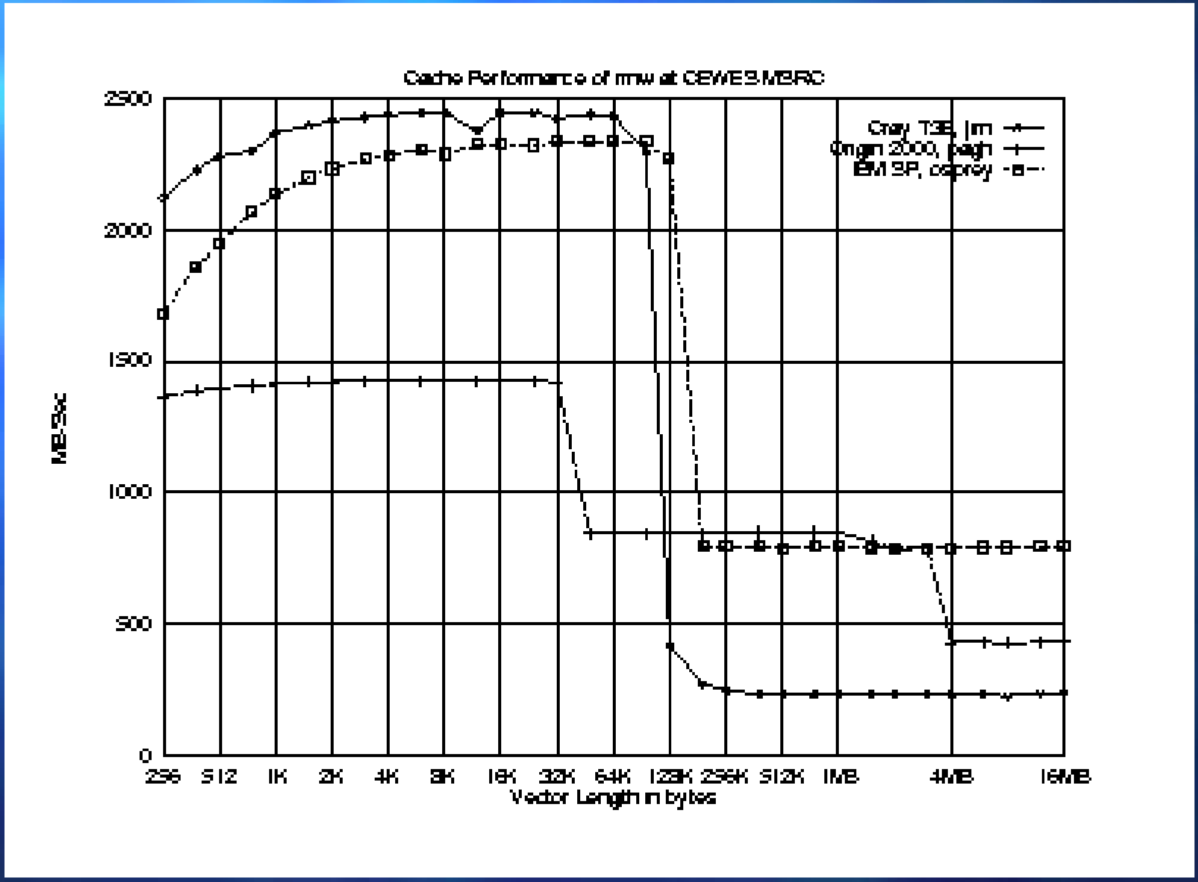
Disk

Size

SP2 Access Times

- Register: 0 cycles
- Cache Hit: 1 cycle
- Cache Miss: 8-12 cycles
- TLB Miss: 36-56 cycles
- Page Fault: ~100,000 cycles

Cache Performance



Cache Architecture

- Divided into smaller units of transfer called lines. (32-256B, 4-32 doubles)
- Each memory address is separated into:
  - Page number
  - Cache line
  - Byte offset

Cache Mapping

- Two major types of mapping
  - Direct Mapped

    Each memory address resides in only one cache line. (constant hit time)
  - N-way Set Associative

    Each memory address resides in one of N cache lines. (variable hit time)

2 way set associative cache

*distinct lines = size / line size * associativity*

Line0 Class0	Line1 Class0	Line2 Class0	Line3 Class0
Line0 Class1	Line1 Class1	Line2 Class1	Line3 Class1

Every data item can live in any class
but in only 1 line (computed from its address)

Memory Access

- Programs should be designed for maximal cache benefit.
  - Stride 1 access patterns
  - Using entire cache lines
  - Avoiding re-use after first reference
- Minimize page faults and TLB misses.

Asymptotic Analysis

- Algorithm X requires O(N log N) time on O(N processors)
- This ignores constants and lower order terms!

10N > N log N for N < 1024

*10N*N < 1000N log N for N < 996*

Amdahl's Law

- The performance improvement is limited by the fraction of time the faster mode can be used.

Speedup = Perf. enhanced / Perf. standard

Speedup = Time sequential / Time parallel

Time parallel = Tser + Tpar

Amdahl's Law

- Be careful when using speedup as a metric. Ideally, use it only when the code is modified. Be sure to completely analyze and documengt your environment.
- Problem: This ignores the overhead of parallel reformulation.

Amdahl’s Law

- Problem? This ignores scaling of the problem size with number of nodes.
- Ok, what about *Scaled Speedup?*
  - Results will vary given the nature of the algorithm
  - Requires O() analysis of communication and run-time operations.

Efficiency

- A measure of code quality?

$$E = \textit{Time sequential} / (P * \textit{Time parallel})$$

$$S = P * E$$

- Sequential time is not a good reference point. For Origin, 4 is good.

Performance Metrics

- **Wall Clock** time - Time from start to finish of our program.
  - Possibly ignore set-up cost.
- MFLOPS - Millions of floating point operations per second.
- MIPS - Millions of instructions per second.

Performance Metrics

- MFLOPS/MIPS are poor measures because
  - They are highly dependent on the instruction set.
  - They can vary inversely to performance!
    - What's most important?

EXECUTION TIME

Performance Metrics

- For purposes of optimization, we are interested in:
  - Execution time of our code
  - MFLOPS of our kernel code vs. peak in order to determine *EFFICIENCY*

Performance Metrics

- Fallacies
  - ***MIPS is an accurate measure for comparing performance among computers.***
  - ***MFLOPS is a consistent and useful measure of performance.***
  - ***Synthetic benchmarks predict performance for real programs.***
  - ***Peak performance tracks observed performance.***

(Hennessey and Patterson)

Performance Metrics

- Our analysis will be based upon:
  - Performance of a single machine
  - Performance of a single (*optimal*) algorithm
  - Execution time

Performance Metrics

For the purposes of comparing your codes performance among different architectures **base your comparison on time.**

…*Unless* you are completely aware of all the issues in performance analysis including architecture, instruction sets, compiler technology etc…

Issues in Performance

- Brute speed (MHz and bus width)
- Cycles per operation (startup + pipelined)
- Number of functional units on chip
- Access to Cache, RAM and storage (local & distributed)

Issues in Performance

- Cache utilization
- Register allocation
- Loop nest optimization
- Instruction scheduling and pipelining
- *Compiler Technology*
- Programming Model (Shared Memory, Message Passing)

Issues in Performance
Problem Size and Precision

- Necessity
- Density and Locality
- Memory, Communication and Disk I/O
- Numerical representation
  - INTEGER, REAL, REAL*8, REAL*16

Parallel Performance Issues

- *Single node performance*
- Compiler Parallelization
- I/O and Communication
- Mapping Problem - Load Balancing
- Message Passing or Data Parallel Optimizations

Performance Tools

Numerical Libraries

Compiler Technology

Serial Optimizations

- Use vendor libraries.
- Improve cache utilization.
- Improve loop structure.
- Use subroutine inlining.
- Use most aggressive compiler options.

Array Optimization

- Array Initialization
- Array Padding
- Stride Minimization
- Loop Fusion
- Floating IF’s
- Loop Defactorization
- Loop Peeling
- Loop Interchange
- Loop Collapse
- Loop Unrolling
- Loop Unrolling and Sum Reduction
- Outer Loop Unrolling

Array Allocation

- Array's are allocated differently in C and FORTRAN.

1	2	3
4	5	6
7	8	9

C: 1 2 3 4 5 6 7 8 9

Fortran: 1 4 7 2 5 8 3 6 9

Array Initialization

Which to choose?

- Static initialization requires:
  - Disk space and Compile time
  - Demand paging
  - Extra Cache and TLB misses.
  - Less run time
- Use only for small sizes with default initialization to 0.

Array Initialization

- Static initialization

```
REAL(8) A(100,100) /10000*1.0/
```

- Dynamic initialization

```
DO I=1, DIM1
   DO J=1, DIM2
      A(I,J) = 1.0
```

Array Padding

- Data in `COMMON` blocks is allocated contiguously
- Watch for powers of two and know the associativity of your cache.
- Example: Possible miss per element on T3E

```
common /xyz/ a(2048),b(2048)
```

Array Padding

*a = a + b * c*

	Tuned	Untuned	Tuned -O3	Untuned -O3
Origin 2000	1064.1	1094.7	800.9	900.3

Stride Minimization

- We must think about spatial locality.
- Effective usage of the cache provides us with the best possibility for a performance gain.
- *Recently* accessed data are likely to be faster to access.
- Tune your algorithm to minimize stride, *innermost index changes fastest*.

Stride Minimization

- Stride 1

```
do y = 1, 1000
   do x = 1, 1000
      c(x,y) = c(x,y) + a(x,y)*b(x,y)
```

- Stride 1000

```
do y = 1, 1000
   do x = 1, 1000
      c(y,x) = c(y,x) + a(y,x)*b(y,x)
```

Stride Minimization

Loop Fusion

- Loop overhead reduced
- Better instruction overlap
- Lower cache misses
- Be aware of associativity issues with array’s mapping to the same cache line.

Loop Fusion

- Untuned

```
do i = 1, 100000
  x = x * a(i) + b(i)
  do i = 1, 100000
    x = x + a(i) / b(i)
  enddo
enddo
```

- Tuned

```
do i = 1, 100000
  x = x * a(i) + b(i)
  x = x + a(i) / b(i)
enddo
```

Loop Fusion

Loop Interchange

- Swapping the nested order of loops
  - Minimize stride
  - Reduce loop overhead where inner loop counts are small
  - Allows better compiler scheduling

Loop Interchange

- Untuned

```
real*8 a(2,40,2000)

do i=1, 2000
 do j=1, 40
   do k=1, 2
     a(k,j,i) = a(k,j,i)*1.01
   enddo
 enddo
enddo
```

- Tuned

```
real*8 a(2000,40,2)

do i=1, 2
 do j=1, 40
   do k=1, 2000
     a(k,j,i) = a(k,j,i)*1.01
   enddo
 enddo
enddo
```

Loop Interchange

Floating IF's

- IF statements that do not change from iteration to iteration may be moved out of the loop.
- Compilers can usually do this except when
  - Loops contain calls to procedures
  - Variable bounded loops
  - Complex loops

Floating IF's

- Untuned

```
do i = 1, lda
  do j = 1, lda
    if (a(i) .GT. 100) then
      b(i) = a(i) - 3.7
    endif
      x = x + a(j) + b(i)
  enddo
enddo
```

- Tuned

```
do i = 1, lda
  if (a(i) .GT. 100) then
    b(i) = a(i) - 3.7
  endif
  do j = 1, lda
     x = x + a(j) + b(i)
  enddo
enddo
```

Floating IF's

Loop Defactorization

- Loops involving multiplication by a *constant* in an array.
- Allows better instruction scheduling.
- Facilitates use of multiply-adds.

Loop Defactorization

- Note that floating point operations are not always associative.

$$(A + B) + C \;\; != A + (B + C)$$

- Be aware of your precision
- Always verify your results with unoptimized code first!

Loop Defactorization

- Untuned

```
do i = 1, lda
   A(i) = 0.0
   do j = 1, lda
     A(i)=A(i)+B(j)*D(j)*C(i)
   enddo
enddo
```

- Tuned

```
do i = 1, lda
  A(i) = 0.0
  do j = 1, lda
    A(i) = A(i) + B(j) * D(j)
  enddo
  A(i) = A(i) * C(i)
enddo
```

Loop Defactorization

Loop Peeling

- For loops which access previous elements in arrays.
- Compiler often cannot determine that an item doesn’t need to be loaded every iteration.

Loop Peeling

- Untuned

```
jwrap = lda
do i = 1, lda
  b(i) = (a(i)+a(jwrap))*0.5
  jwrap = i
enddo
```

- Tuned

```
b(1) = (a(1)+a(lda))*0.5
do i = 2, lda
   b(i) = (a(i)+a(i-1))*0.5
enddo
```

Loop Peeling

Loop Collapse

- For multi-nested loops in which the entire array is accessed.
- This can reduce loop overhead and improve compiler vectorization.

Loop Collapse

- Untuned

```
do i = 1, lda
   do j = 1, ldb
      do k = 1, ldc
         A(k,j,i) = A(k,j,i) + B(k,j,i) * C(k,j,i)
      enddo
   enddo
enddo
```

Loop Collapse

- Tuned

```
do i = 1, lda*ldb*ldc
   A(i,1,1) = A(i,1,1) + B(i,1,1) * C(i,1,1)
enddo
```

- More Tuned (declarations are 1D)

```
do i = 1, lda*ldb*ldc
   A(i) = A(i) + B(i) * C(i)
enddo
```

Loop Collapse

Loop Unrolling

- Data dependence delays can be reduced or eliminated.
- Reduce loop overhead.
- Might be performed well by the compiler or preprocessor. (Careful on the T3E)

Loop Unrolling

- Untuned

```
do i = 1, lda
   do j = 1, lda
      do k = 1, 4
         a(j,i) = a(j,i) + b(i,k) * c(j,k)
      enddo
   enddo
enddo
```

Loop Unrolling

- Tuned (4)

```
do i = 1, lda
   do j = 1, lda
      a(j,i) = a(j,i) +  b(i,1) * c(j,1)
      a(j,i) = a(j,i) +  b(i,2) * c(j,2)
      a(j,i) = a(j,i) +  b(i,3) * c(j,3)
      a(j,i) = a(j,i) +  b(i,4) * c(j,4)
   enddo
enddo
```

Loop Unrolling

Loop Unrolling and Sum Reductions

- When an operation requires as input the result of the last output.
- Called a Data Dependency.
- Frequently happens with multi-add instruction inside of loops.
- Introduce intermediate sums. Use your registers!

Loop Unrolling and Sum Reductions

- Untuned

```
do i = 1, lda
   do j = 1, lda
      a = a + (b(j) * c(i))
   enddo
enddo
```

Loop Unrolling and Sum Reductions

- Tuned (4)

```
do i = 1, lda
   do j = 1, lda, 4
      a1 = a1 + b(j) * c(i)
      a2 = a2 + b(j+1) * c(i)
      a3 = a3 + b(j+2) * c(i)
      a4 = a4 + b(j+3) * c(i)
   enddo
enddo
aa = a1 + a2 +a3 + a4
```

Loop Unrolling and Sum Reductions

Outer Loop Unrolling

- For nested loops, unrolling outer loop may reduce loads and stores in the inner loop.
- Compiler may perform this optimization.

Outer Loop Unrolling

- Untuned
  - Each multiply requires two loads and one store.

```
do i = 1, lda
   do j = 1, ldb
      A(i,j) = B(i,j) * C(j) + D(j)
   enddo
enddo
```

Outer Loop Unrolling

- Tuned
  - Each multiply requires 5/4 loads and one store.

```
do i = 1, lda, 4
   do j = 1, ldb
      A(i,j)   = B(i,j) * C(j) + D(j)
      A(i+1,j) = B(i+1,j) * C(j) + D(j)
      A(i+2,j) = B(i+2,j) * C(j) + D(j)
      A(i+3,j) = B(i+3,j) * C(j) + D(j)
   enddo
enddo
```

Outer Loop Unrolling

Loop structure

- IF/GOTO and WHILE loops inhibit some compiler optimizations.
- Some optimizers and preprocessors can perform transforms.
- DO and for() loops are the most highly tuned.

Strength Reduction

- Reduce cost of mathematical operation with no loss in precision, compiler might do it.
  - Integer multiplication/division by a constant with shift/adds
  - Exponentiation by multiplication
  - Factorization and Horner’s Rule
  - Floating point division by inverse multiplication

Strength Reduction
Horner's Rule

- Polynomial expression can be rewritten as a nested factorization.

```
Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F =
((((Ax + B) * x + C) * x + D) * x + E) * x + F.
```

- Also uses multiply-add instructions
- Eases dependency analysis

Strength Reduction
Horner's Rule

Strength Reduction
Integer Division by a Power of 2

- Shift requires less cycles than division.
- Both dividend and divisor must both be unsigned or positive integers.

Strength Reduction
Integer division by a Power of 2

- Untuned

```
IL = 0
DO I=1,ARRAY_SIZE
  DO J=1,ARRAY_SIZE
    IL = IL + A(J)/2
  ENDDO
  ILL(I) = IL
ENDDO
```

- Tuned

```
IL = 0
ILL = 0
DO I=1,ARRAY_SIZE
  DO J=1,ARRAY_SIZE
    IL = IL + ISHFT(A(J),-1)
  ENDDO
  ILL(I) = IL
ENDDO
```

Strength Reduction

Integer division by a Power of 2

Strength Reduction
Factorization

- Allows for better instruction scheduling.
- Compiler can interleave loads and ALU operations.
- Especially benefits compilers able to do software pipelining.

Strength Reduction
Factorization

- Untuned

```
XX = X*A(I) + X*B(I) + X*C(I) + X*D(I)
```

- Tuned

```
XX = X*(A(I) + B(I) + C(I) + D(I))
```

Strength Reduction
Factorization

Subexpression Elimination
Parenthesis

- Parenthesis can help the compiler recognize repeated expressions.
- Some preprocessors and aggressive compilers will do it.

Subexpression Elimination
Parenthesis

- Untuned

```
XX = XX + X(I)*Y(I)+Z(I) + X(I)*Y(I)-Z(I) + X(I)*Y
   (I) + Z(I)
```

- Tuned

```
XX = XX + (X(I)*Y(I)+Z(I)) + X(I)*Y(I)-Z(I) + (X
   (I)*Y(I) + Z(I))
```

Subexpression Elimination

Parenthesis

Subexpression Elimination
Type Considerations

- Changes the type or precision of data.
  - Reduces resource requirements.
  - Avoid type conversions.
  - Processor specific performance.
- Do you really need 8 or 16 bytes of precision?

Subexpression Elimination
Type Considerations

- Consider which elements are used together?
  - Should you be merging your arrays?
  - Should you be splitting your loops for better locality?
  - For C, are your structures packed tightly in terms of storage and reference pattern?

Subroutine Inlining

- Replaces a subroutine call with the function itself.
- Useful in loops that have a large iteration count and functions that don't do a lot of work.
- Allows better loop optimizations.
- Most compilers can do inlining but not that well on large applications.

Optimized Arithmetic Libraries

- (P)BLAS: Basic Linear Algebra Subroutines
- (Sca)LAPACK: Linear Algebra Package
- ESSL: Engineering and Scientific Subroutine Library
- NAG: Numerical Algorithms Group
- IMSL: International Mathematical and Statistical Lib.

Optimized Arithmetic Libraries

- Advantages:
  - Subroutines are quick to code and understand.
  - Routines provide *portability.*
  - Routines perform well.
  - Comprehensive set of routines.

SGI Origin 2000

- MIPS R10000, 195Mhz, 5.1ns
- 64 Integer, 64 Floating Point Registers
- 4 Instructions per cycle
- Up to 2 Integer, 2 Floating Point, 1 Load/Store per cycle
- 4 outstanding cache misses
- Out of order execution

SGI Origin 2000

- 64 Entry TLB, variable page size
- 32K Data, 32K Instruction, 4MB unified.
- Data is 2-way set associative, 2-way interleaved.
- 32B/128B line size.

SP2

- IBM Power 2 SC, 135Mhz
- 32 Integer, 32 Floating Point Registers
- 6(8) Instructions per cycle
- 2 Integer, 2 Floating Point, 1 Branch,. 1 Conditional
- Zero cycle branches, dual FMA

SP2

- 256 Entry TLB
- Primary Cache 128K Data, 32K Instruction
- 4 way set associative
- 256B line size

T3E

- Alpha 21164, 450Mhz
- Primary Cache 8K Data, 8K Instruction
- 96KB on-chip 3-way associative secondary cache
- 2 FP / 2 Int / cycle

T3E

- Scheduling very important
- 64 bit divides take 22-60 CP
- Ind Mult/Add takes 4 cp, but issued every cycle

T3E

- Latency hiding features
  - Cache bypass
  - Streams
  - E-registers
- 6 queued Dcache misses/WBs to Scache
- Load/store merging
- 32/64 byte line Dcache/Scache
- 2 cycle/8-10 cycle hit Dcache/Scache

T3E Streams

- Designed to provide automatic prefetching for densely strided data.
- 6 stream buffers, two 64 byte lines each
- Starts when 2 contiguous misses
- Look at difference in loads
  - 875MB/sec with streams
  - 296MB/sec without

T3E Streams

- Count references to memory in your loops, make sure no more than six.
- May need to split loops to reduce streams.
- To use them

```
setenv SCACHE_D_STREAMS 1
man intro_streams
man streams_guide
```

T3E E-registers

- 512 64-bit off-chip registers for transferring data to/from remote/local memory
- SHMEM library
  - Local, shared, distributed, memory access routines that use the E-registers.

  `man intro_shmem`
- Block copy
  - 775 MB/sec vs 401 MB/sec

T3E Cache Bypass

- Reduces memory traffic requirements.
- Fortran

```
!DIR$ CACHE_BYPASS var1, var2
```

- C

```
#pragma _CRI cache_bypass var1,
  var2
```

- Block copy
  - 593 MB/sec vs 401 MB/sec

T3E E-registers

- Benchlib library
  - One sided data transfers from memory to E-registers bypassing cache
  - Scatter / Gather in hardware
  - Nonblocking
  - More complicated to use than SHMEM
  - Not supported by Cray
  - 592 MB/sec vs 401 MB/sec for copy

O2K Flags and Libraries

- `-O,-O2` - Optimize
- `-O3` – Maximal generic optimization, may alter semantics.
- `-Ofast=ip27` - SGI compiler group’s best set of flags.
- `-IPA=on` – Enable interprocedural analysis.
- `-n32` – 32-bit object, best performer.
- `-INLINE:<func1>,<func2>` – Inline all calls to func1 and func2.
- `-LNO` – Enable the loop nest optimizer.
- `-feedback` – Record information about the programs execution behavior to be used by `IPA, LNO`.
- `-lcomplib.sgimath -lfastm` – Include BLAS, FFTs, Convolutions, EISPACK, LINPACK, LAPACK, Sparse Solvers and the fast math library.
- `dplace` – program to change the page size of your executable. This reduces TLB, page faults and increase MPI performance.

SP2 Flags and Libraries

- -O,-O2 - Optimize
- -O3 - Maximum optimization, may alter semantics.
- -qarch=pwr2, -qtune=pwr2 - Tune for Power2.
- -qcache=size=128k,line=256 - Tune Cache for Power2SC.
- -qstrict - Turn off semantic altering optimizations.
- -qhot - Turn on addition loop and memory optimization, Fortran only.
- -Pv,-Pv! - Invoke the VAST preprocessor before compiling. (C)
- -Pk,-Pk! - Invoke the KAP preprocessor before compiling. (C)
- -qhsflt - Don't round floating floating point numbers and don't range check floating point to integer conversions.
- -inline=<func1>,<func2> - Inline all calls to func1 and func2.
- -qalign=4k - Align large arrays and structures to a 4k boundary.
- -lesslp2 - Link in the Engineering and Scientific Subroutine Library.

T3E Flags and Libraries

- `-O,-O2` - Optimize
- `-O3` - Maximum optimization, may alter semantics.
- `-apad` - Pad arrays to avoid cache line conflicts
- `-unroll2` - Apply aggressive unrolling
- `-pipeline2` - Software pipelining
- `-split2` - Apply loop splitting.
- `-Wl"-Dallocate(alignsz)=64b"` Align common blocks on cache line boundary
- `-lmfastv` - Fastest vectorized intrinsics library
- `-lsci` - Include library with BLAS, LAPACK and ESSL routines
- `-inlinefrom=<>` - Specifies source file or directory of functions to inline
- `-inline2` - Aggressively inline function calls.

Timers

- `time <command>` returns 3 kinds.
  - Real time: Time from start to finish
  - User: CPU time spent executing your code
  - System: CPU time spent executing system calls
- `timex` on the SGI.
- Warning! The definition of CPU time is different on different machines.

Timers

- Sample output for csh users:

```
   1          2          3           4      5             6          7
1.150u 0.020s 0:01.76 66.4 15+3981k 24+10io 0pf+0w
```

1) User (ksh)
2) System (ksh)
3) Real (ksh)
4) Percent of time spent on behalf of this process, not including waiting.
5) 15K shared, 3981K unshared
6) 24 input, 10 output operations
7) No page faults, no swaps.

Timers

- Latency is not the same as resolution.
  - Many calls to this function will affect your wall clock time.

prof

- Profiles program execution at the procedure level
- Available on most Unix systems, not T3E
- Displays the following:
  - Name, percentage of CPU time
  - Cumulative and average execution time
  - Number of time procedure was called

prof

- Compile your code with `-p`
- After execution the CWD will contain mon.out.(x)
- Type `prof`, it will look for mon.out in the CWD. Otherwise give it name(s) with the `-m option`
- Format of output is:

```
Name %Time Seconds Sumsecs #Calls msec/call
```

prof

- All procedures called by the object code, many will be foreign to the programmer.
- Statistics are created by sampling and then looking up the PC and correlating it with the address space information.
- Phase problems may cause erroneous results and reporting.

SpeedShop on the SGI

- `ssusage` collects information about your program's use of machine time and resources.
- `ssrun` allows you to run *experiments* on a program to collect performance data.
- `prof` analyzes the performance data you have recorded using `ssrun` and provides formatted reports.

SpeedShop on the SGI

- Collects hardware statistics at the subroutine level
  - Build the application
  - Run *experiments* to collect data
    - `ssrun -exp <exp> <exe>`
  - Examine the data
    - `prof <exe> <exe.ssrunfiles>`
  - Optimize

SpeedShop Usage

- Usage: ssrun [-exp expt] [-mo marching-orders] [-purify] [-v | -verbose] [-hang] [-x display window] [-name target-name] a.out [a.out-arguments]

- Defined experiments are:
  usertime, pcsamp, fpcsamp, pcsampx, fpcsampx, ideal, prof_hwc, gi_hwc, cy_hwc, ic_hwc, isc_hwc, dc_hwc, dsc_hwc, tlb_hwc, gfp_hwc, fgi_hwc, fcy_hwc, fic_hwc, fisc_hwc, fdc_hwc, fdsc_hwc, ftlb_hwc, fgfp_hwc, heap, fpe.

Ideal Experiment

```
Prof run at: Fri Jan 30 01:59:32 1998
Command line: prof nn0.ideal.21088
-----------------------------------------------------------
      3954782081: Total number of cycles
       20.28093s: Total execution time
      2730104514: Total number of instructions executed
           1.449: Ratio of cycles / instruction
             195: Clock rate in MHz
          R10000: Target processor modeled
------------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------
       cycles(%)  cum %     secs    instrns calls procedure(dso:file)
3951360680(99.91)  99.91    20.26 2726084981     1 main(nn0.pixie:nn0.c)
1617034( 0.04)     99.95     0.01    1850963  5001 doprnt
```

Pcsamp Experiment

```
-------------------------------------------------------------------
Profile listing generated Fri Jan 30 02:06:07 1998
    with:       prof nn0.pcsamp.21081
-------------------------------------------------------------------
samples   time    CPU    FPU    Clock   N-cpu  S-interval Countsize
   1270    13s R10000 R10010 195.0MHz   1     10.0ms      2(bytes)
Each sample covers 4 bytes for every 10.0ms ( 0.08% of 12.7000s)
-------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (dso:file)
   1268    13s( 99.8)   13s( 99.8)           main (nn0:nn0.c)
      1  0.01s(  0.1)   13s( 99.9)        _doprnt
```

Example of UserTime

```
-------------------------------------------------------------------------
Profile listing generated Fri Jan 30 02:11:45 1998
    with:       prof nn0.usertime.21077
-------------------------------------------------------------------------
        Total Time (secs)      : 3.81
        Total Samples          : 127
        Stack backtrace failed: 0
        Sample interval (ms)   : 30
        CPU                    : R10000
        FPU                    : R10010
        Clock                  : 195.0MHz
        Number of CPUs         : 1
-------------------------------------------------------------------------
index  %Samples  self    descendents  total         name
(1)    100.0%    3.78        0.03     127           main
(2)      0.8%    0.00        0.03       1           _gettimeofday

(3)      0.8%    0.03        0.00       1           _BSD_getime
```

tprof for the SP2

- Reports CPU usage for programs and system. i.e.
  - All other processes while your program was executing
  - Each subroutine of the program
  - Kernel and Idle time
  - Each line of the program
- We are interested in source statement profiling.

tprof for the SP2

- Also based on sampling, which may cause erroneous reports.
- Compile using -qlist and -g
- tprof <program> <args>
- Leaves a number of files in the CWD preceded by __.

  __h.<file>.c - Hot line profile

  __t.<subroutine>_<file>.c - Subroutine profile

  __t.main_<file>.c - Executable profile

PAT for the T3E

- Performance analysis tool is a low-overhead method for
  - Estimating time in functions
  - Determining load balance
  - Generating and viewing trace files
  - Timing individual calls
  - Displaying hardware performance counter information

PAT for the T3E

- Uses the UNICOS/mk `profil()` system call to gather information by periodically sampling and examining the program counter.
- Works on C, C++ and Fortran executables
- No recompiling necessary
- Just link with `-lpat`

Apprentice for the T3E

- Graphical interface for identifying bottlenecks.

```
% f90 -eA <file>.f -lapp
% cc -happrentice <file>.c -lapp
% a.out
% apprentice app.rif
```

apprenti

Additional Material

`http://www.cs.utk.edu/~mucci/MPPopt.html`

- Slides
- Optimization Guides
- Papers
- Pointers
- Compiler Benchmarks

References

http://www.nersc.gov

http://www.mhpcc.gov

http://www-jics.cs.utk.edu

http://www.tc.cornell.edu

http://www.netlib.org

http://www.ncsa.uiuc.edu

http://www.cray.com

http://www.psc.edu

References

Hennessey and Patterson: *Computer Architecture, A Quantitative Approach*

Dongarra et al: MPI, *The Complete Reference*

Dongarra et al: PVM, *Parallel Virtual Machine*

Vipin Kumer et al*: Introduction to Parallel Computing*

IBM: *Optimization and Tuning Guide for Fortran, C, C++*