DTU Compute
Department of Applied Mathematics and Computer Science

High-Performance Computing

Getting good performance from your application

Tuning techniques for serial programs on cache-based computer systems

---

Application Tuning

Overview

- ❏ Introduction
- ❏ Memory Hierarchy
- ❏ General Optimization Techniques
- ❏
- ❏ Compilers
- ❏ Analysis Tools
- ❏ Tuning Guide

De-vectorization

❑ Cache space and bandwidth are scarce resources

❑ Compilers know this – but sometimes they have to store data that does not need to be stored.

❑ This impacts:

  ❑ bandwidth

  ❑ cache capacity

  ❑ instruction scheduling

De-vectorization

❑ A typical problem with scratch data stored in vectors

❑ Difficult/impossible for the compiler to detect

❑ Depends on coding style

De-vectorization – Example

```
COMMON /SCRATCH/TMP(N)

DO I = 1, N
    TMP(I) = ...
           :
    ... = TMP(I)
END DO
          ▪
          ▪
          ▪
DO I = 1, N
    TMP(I) = ..
           :
END DO
```

❑ *Because TMP() is global, the compiler has to store it in the first loop*

❑ *In the second loop, TMP() is overwritten, but the compiler will most likely not see this*

❑ *The programmer may know that TMP() is a scratch array only*

De-vectorization – Solutions

Array TMP needed later on:

```
REAL   T1,TMP(N)

DO I = 1, N
    T1 = ...
         :
    ... = T1
END DO
          ▪
          ▪
          ▪
DO I = 1, N
    TMP(I) = ..
         :
END DO
```

Array TMP not needed later on:

```
REAL   T1

DO I = 1, N
    T1 = ...
         :
    ... = T1
END DO
          ▪
          ▪
          ▪
DO I = 1, N
    T1 = ..
         :
END DO
```

Stripmining

- ❑ **Large loops are difficult to optimize**

- ❑ **Especially the register allocation in the compiler has a hard time and can get confused**

- ❑ **Splitting the loop into smaller loops may improve performance**

- ❑ **However, this may cause scalars (local to the loop) to be replaced by vectors**

- ❑ **On very large loops this will increase memory usage notably**

- ❑ **Through stripmining memory usage can be kept under control**

```
DO I = 1, LONG
    X(I) = ...
    A    = ...
    Y(I) = A + ...
END DO
```

Split loop in two parts

```
DO I = 1, LONG
    X(I)  = ...
    VA(I) = ...
END DO

DO I = 1, LONG
    Y(I) = VA(I) + ...
END DO
```

Stripmining – Code structure

Application Tuning

```
DO i = is, ie
    ......
END DO
```

Stripmine this loop with length NS

```
DO i1 = is, ie, ns
    ivl = min(ie-i1+1,ns)

    DO i2 = 0, ivl-1
        i = i1+i2
        ........
    END DO
END DO
```

Split inner loop

```
DO i1 = is, ie, ns
    ivl = min(ie-i1+1,ns)

    DO i2 = 0, ivl-1
        i = i1+i2
        ........
    END DO

    DO i2 = 0, ivl-1
        i = i1+i2
        ........
    END DO

    DO i2 = 0, ivl-1
        i = i1+i2
        ........
    END DO
END DO
```

Best practice

It is up to you to write code such that the compiler can find opportunities for optimization:

❏ Write efficient, but clear code

❏ Avoid very "fat" (bulky) loops

❏ Design your data structures carefully

❏ Minimize global data

Best practice

❏ Branches:

    ❏ simplify where possible

    ❏ try to split the branch part out of the loop

❏ Avoid function calls in loops (use inlining)

❏ Leave the low level details to the compiler

Summary

❑ Most tuning techniques presented here are generic, i.e. they (probably/hopefully) improve your code on all cache based systems.

❑ The *tuning parameters* may be different, though, since they depend on the underlying hardware:

  ❑ cache sizes and levels

  ❑ prefetch

  and your problem's *memory footprint*

❑ Use the *best* compiler available on your platform.

Application Tuning

The Oracle Solaris Studio Compilers
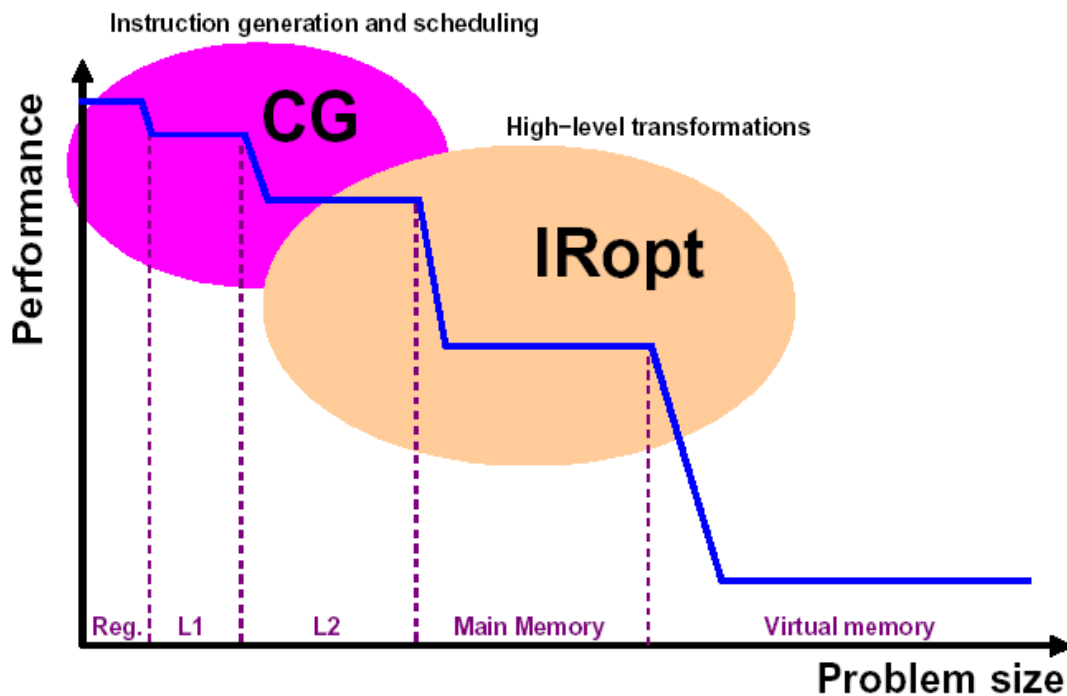## (a.k.a. Sun Studio)
–
an example how compilers work

Sun Studio: Overview

❏ Compiler Components

❏ Compiler Options

❏ Compiler Commentary

❏ Best practice

❏ Note: to use the Studio tools, you have to load the module first:
`module load studio`

Sun Studio: Compiler Components

CG : Code Generator and
Instruction Level Optimizer
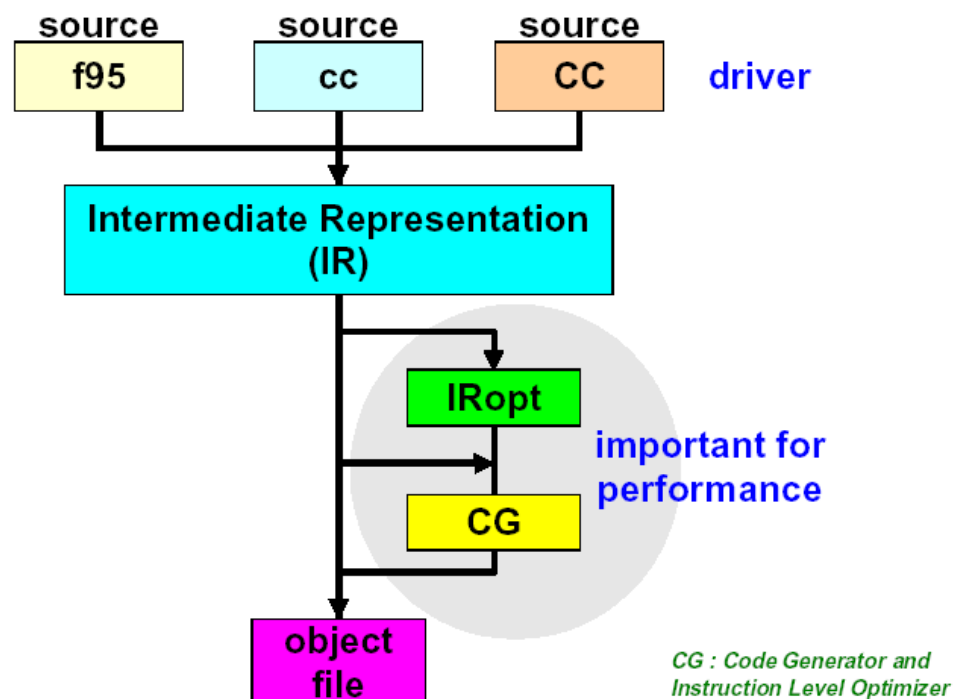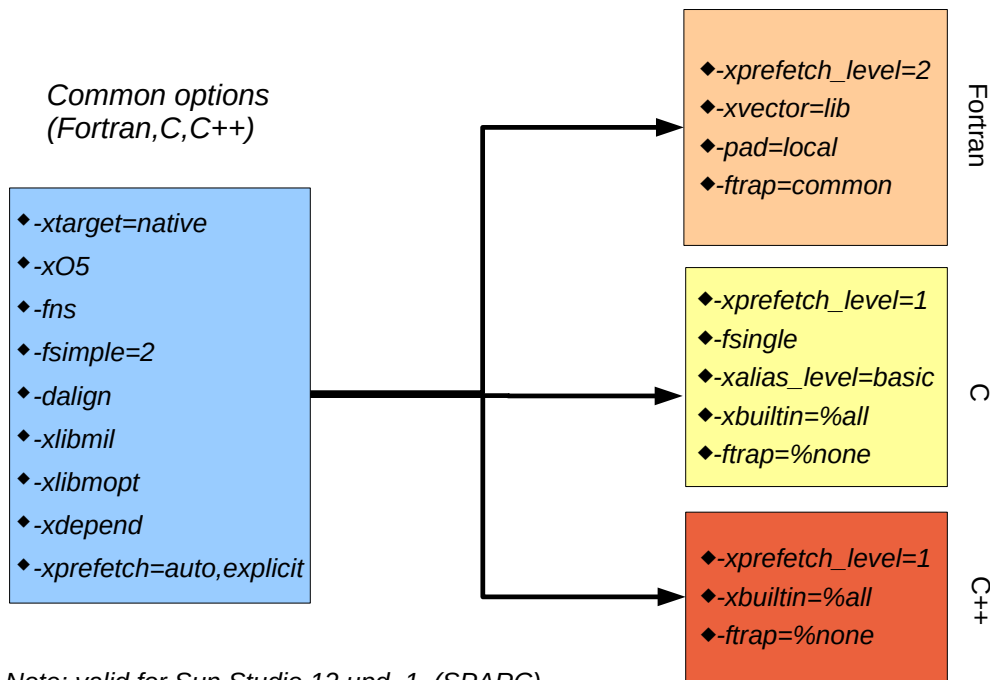
Sun Studio: Who does what?

Sun Studio: Minimal Compiler Options

In general, one gets very good performance by just using 3 options for compiling and linking:

```
-g -fast -m32
-g -fast -m64
```
(32-bit addressing)
(64-bit addressing)

For specific x86_64-processors (cross-compiling):

AMD Opteron:	`-g -fast -xchip=opteron`
Intel Nehalem:	`-g -fast -xchip=nehalem`
Intel Westmere:	`-g -fast -xchip=westmere`
Intel Sandy Bridge:	`-g -fast -xchip=sandybrigde`
Intel Ivy Bridge:	`-g -fast -xchip=ivybridge`
Intel Haswell:	`-g -fast -xchip=haswell`
Intel Broadwell:	`-g -fast -xchip=broadwell`

Sun Studio: The -fast macro

*Common options
(Fortran,C,C++)*

- *-xtarget=native*
- *-xO5*
- *-fns*
- *-fsimple=2*
- *-dalign*
- *-xlibmil*
- *-xlibmopt*
- *-xdepend*
- *-xprefetch=auto,explicit*

Fortran
- *-xprefetch_level=2*
- *-xvector=lib*
- *-pad=local*
- *-ftrap=common*

C
- *-xprefetch_level=1*
- *-fsingle*
- *-xalias_level=basic*
- *-xbuiltin=%all*
- *-ftrap=%none*

C++
- *-xprefetch_level=1*
- *-xbuiltin=%all*
- *-ftrap=%none*

Note: valid for Sun Studio 12 upd. 1 (SPARC)

Sun Studio: Recommendations

❏ Use a Makefile and `make` for compiling/linking:

```
OPT    = -g -fast
ISA    = -m64
CHIP   =
CFLAGS = $(OPT) $(ISA) $(CHIP)
```

❏ *Always start with* `-fast`*!*

❏ The compilers follow the 'rightmost option wins' rule, i.e. one can overrule options defined by the -fast macro.

Sun Studio: Recommendations

- ❏ `-fast` is a convenience macro that (in general) gives optimal performance with one single option

- ❏ `-fast` can change from one release to another!

- ❏ Use '`-fast -xdryrun`' to check what `-fast` expands to

What about other compilers? - I

- ❏ Other compilers have similar options, that combine many optimizations into a single option

- ❏ GCC: -O, -O3

- ❏ Intel: -O2 (default!), -O3, -fast

- ❏ look up in the manpages/documentation, what that corresponds to

- ❏ my experience: difficult to find out, not so easy to switch off 'unwanted options'

What about other compilers? - II

❏ GCC: show optimizer options

    ❏ gcc -Q –help=optimizers -O3 ....

❏ shows a list of all known '-f...' options and their status

❏ switch extra options on/off, e.g. loop unrolling

    ❏ -funroll-loops

    ❏ -fno-unroll-loops

Application Tuning

What about other compilers? - III

❏ Other compilers work in a similar way, but ...

❏ I've chosen Oracle Studio for various reasons:

    ❏ my own experience

    ❏ it is easy to use

    ❏ it makes it easy to test effects of certain options

    ❏ it is a nice tool for teaching

    ❏ it makes it easier for you to understand certain aspects in this class

❏ Downside: It seems that Oracle has stopped further development of the product in 2017!

Application Tuning

Sun Studio: Optimization levels

❏ The level of optmization can be specified with the -xO<u>n</u> option

❏ Each level includes the lower levels

❏ Choices for <u>n</u>:

- ❏ n = 1 – Basic block level optimizations
- ❏ n = 2 – Some additional optimizations
- ❏ n = 3 – Loop transformations and modulo scheduling
- ❏ n = 4 – Intra-file inlining and pointer tracking
- ❏ n = 5 – Aggressive optimizations

Inter Procedural Optimization

❏ With the -xipo option, the compiler stores additional information into the object files

❏ This information is used during the link phase to perform additional optimizations
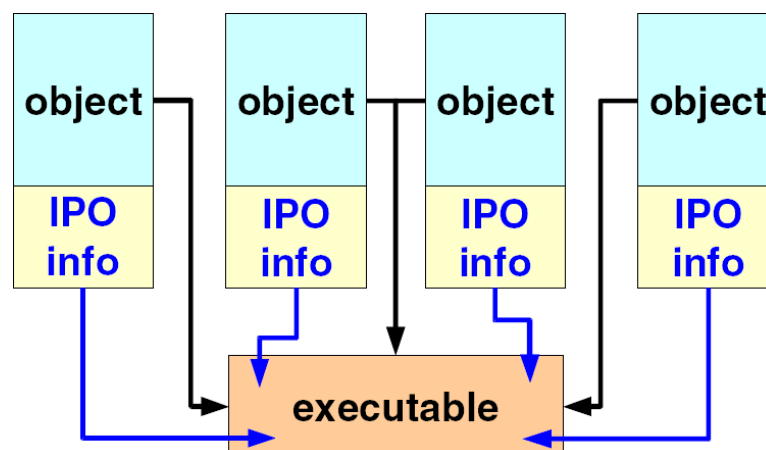
Prefetch: Hiding memory latency

❑ The number of clock cycles to access memory increases with the CPU clock speed.

❑ Prefetch is a way to overcome this:

   Fetch data ahead in time, anticipating future use.

❑ Special prefetch instructions must be available

| memory | compute | compute | compute |

| prefetch | prefetch | prefetch |

time
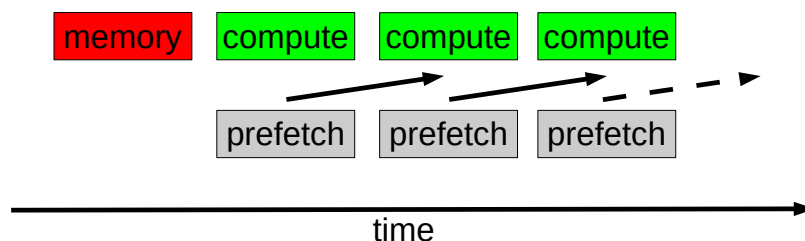
Prefetch Support

❑ Prefetch is a common feature in modern CPUs: both data and instruction prefetch.

❑ Implementation is system dependent!

❑ Examples of CPUs that support prefetch:

  ❑ AMD Opteron

  ❑ Intel (since Pentium 3, 4)

  ❑ UltraSPARC IIIi, IIICu, IV, IV+

❑ Compilers have to support prefetch.

❑ There is hardware prefetch, too (Xeon, Opteron)

Sun Studio: Prefetch options

❏ Automatic – done by the compiler:

  ❏ -xprefetch=yes

  ❏ Control level and type:

    ❏ -xprefetch_level=n (n = 1, 2 or 3)

    ❏ -xprefetch_auto_type=[no%]indirect_array_access

❏ Explicit – controlled by the user:

  ❏ Either through functions calls (C/C++) or directives (Fortran).

  ❏ Can be combined with automatic prefetch.

  ❏ For more information see the compiler manual.

Prefetch: example

Example: Matrix times vector in C (row version)
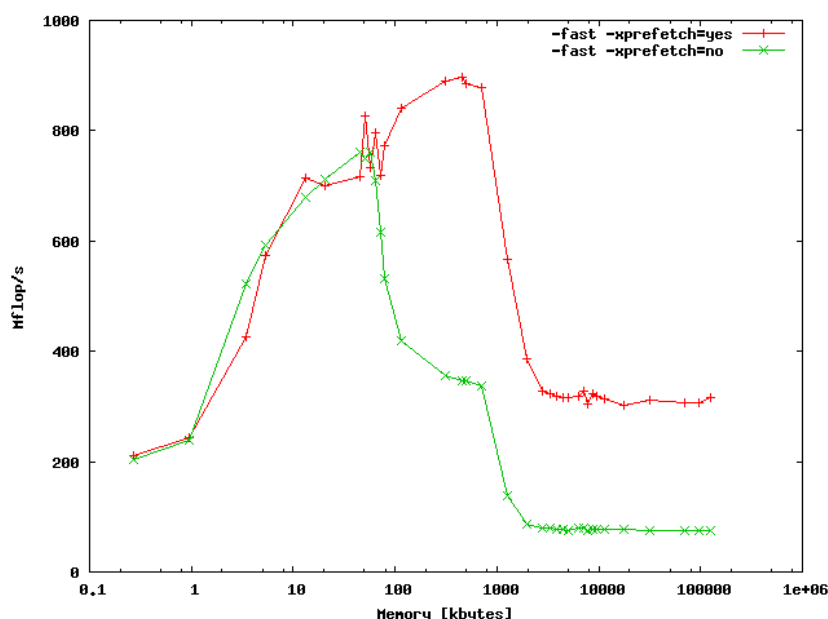
US-IIIi @ 1062 MHz
L1   : 64 kB
L2   :   1 MB
Peak   : 2.1 Gflop/s

Pointer overlap – or "aliasing"

```
void vecadd(int n, double *a, double *b, double *c)
{
    for(int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

```
vecadd(n, &a[0], &b[0], &a[1]);
```

```
void vecadd(n,   &a[0], &b[0], &a[1])
{
    for(int i = 0; i < n; i++)
        a[i+1] = a[i] + b[i];
}
```
Data dependency!

Sun Studio: C code and -xrestrict

❏ Pointer aliasing problem: The C compiler <u>has to assume</u> that different pointers may overlap:

   ❏ Correct – but non-optimal – code will be generated

   ❏ Only the programmer might know, that there is no overlap.

❏ You can tell the compiler that there is no overlap, with the `-xrestrict` option.

❏ Note: It is then <u>your responsibility</u> that this assumption will not be violated!

Sun Studio: C code and -xrestrict

- ❏ Syntax: -xrestrict=n, where n is one of
  - ❏ %none : all pointers may overlap (default)
  - ❏ %all : no pointers overlap
  - ❏ f1[,f2] : a comma separated list of function names
- ❏ -xrestrict is the same as -xrestrict=%all
- ❏ Warning: Use -xrestrict only if you can make sure that the pointers <u>never</u> overlap!

Sun Studio: C code and -xrestrict

Vector addition example: non-overlapping vectors!
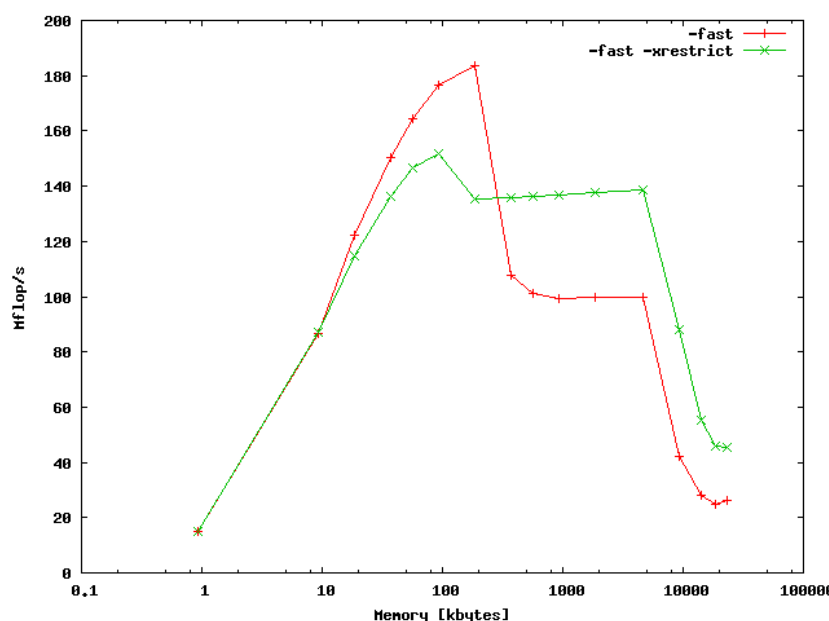
US-III @ 900 MHz
L1   : 64 kB
L2   :   8 MB
Peak   : 1.8 Gflop/s
Sun Studio 10

Pointer overlap – fix your code

❑ If you can assure that the pointers don't overlap, you can fix your code using the C99 'restrict' keyword:

```
void
vecadd(int n, double * restrict a,
       double * restrict b, double * restrict c)
{
    for(int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

❑ Advantage: local change, no global assumption

❑ Needs a C99 compliant compiler to be portable!

Sun Studio: Useful options – 1

❑ -flags : Lists all the available compiler flags on the screen (long list)

❑ -xhelp=readme : Displays the README file (release notes) on the screen

❑ -xdryrun : see what the compiler would do (macro expansion, no compilation!)

❑ -V : Shows the compiler version

```
% f90 -V
f90: Sun Fortran 95 8.4 SunOS_sparc 2009/06/03
Usage: f90 [ options ] files.  Use 'f90 -flags' for details
```

Sun Studio: Useful options – 2

- -g : Generates debugging information and adds compiler commentary to the object file. Necessary if the performance analysis tools should be used.

- **Note**: -g can be used in connection with optimization!

- -xlibmopt : Optimized version of libm

- -xlibmil: Use fast assembly language inline templates for standard functions

- -xvector : vectorized intrinsics, e.g. sqrt()

Sun Studio: Useful options – 3

- -xprefetch_level=<u>n</u> : Try different levels of prefetching (n = 1,2,3)

- -xrestrict : (C/C++) tell the compiler that your pointers do point to 'restricted areas' in memory, i.e. no overlap of arrays

- -stackvar : (Fortran) keep local data on stack

- -xlic_lib=sunperf : Link with the Sun Performance Library (LAPACK, BLAS, Sparse-BLAS (Netlib and NIST), FFT, Sparse Solvers)

Sun Studio: Useful options – 4

❏ Options that are useful under development – but should be avoided in production:

  ❏ -C : array boundary checking (Fortran)

  ❏ -pg/-p : Unix profilers – you don't really need them, use the Performance Analyzer instead.

  ❏ -xcheck=... : implements runtime checks.

    ❏ -xcheck=init_local : initialize local variables with a value that is likely to cause an arithmetic exception (Fortran)

    ❏ -xcheck=stkovf : check for stack overflow.

 ❏ -Xlist : check your Fortran code for inconsistencies (commented source in .lst file)

Application Tuning

Sun Studio: Where to get it?

❏ Go to Oracle TechNetwork

❏ Download and install it – it is for free!

❏ Best practice: It is always good to have more than one compiler (or compiler version) at hand!

  ❏ to test compatibility / portability

  ❏ to exclude compiler bugs
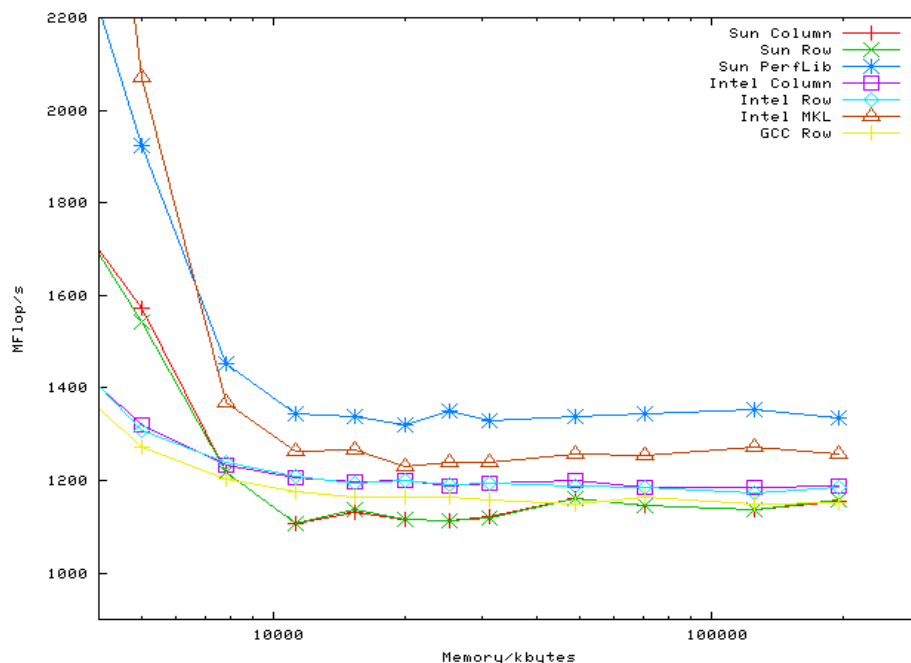
Application Tuning

Matrix times vector

- ❏ 2 code versions: Column and Row
- ❏ 3 compilers under Linux:
  - ❏ GCC
  - ❏ Intel C
  - ❏ Sun Studio C
- ❏ 2 libraries:
  - ❏ Intel MKL
  - ❏ Sun Performance Library
  - ❏ Note: this is a quick comparison! (June 2008)

Application Tuning

Matrix times vector

Application Tuning
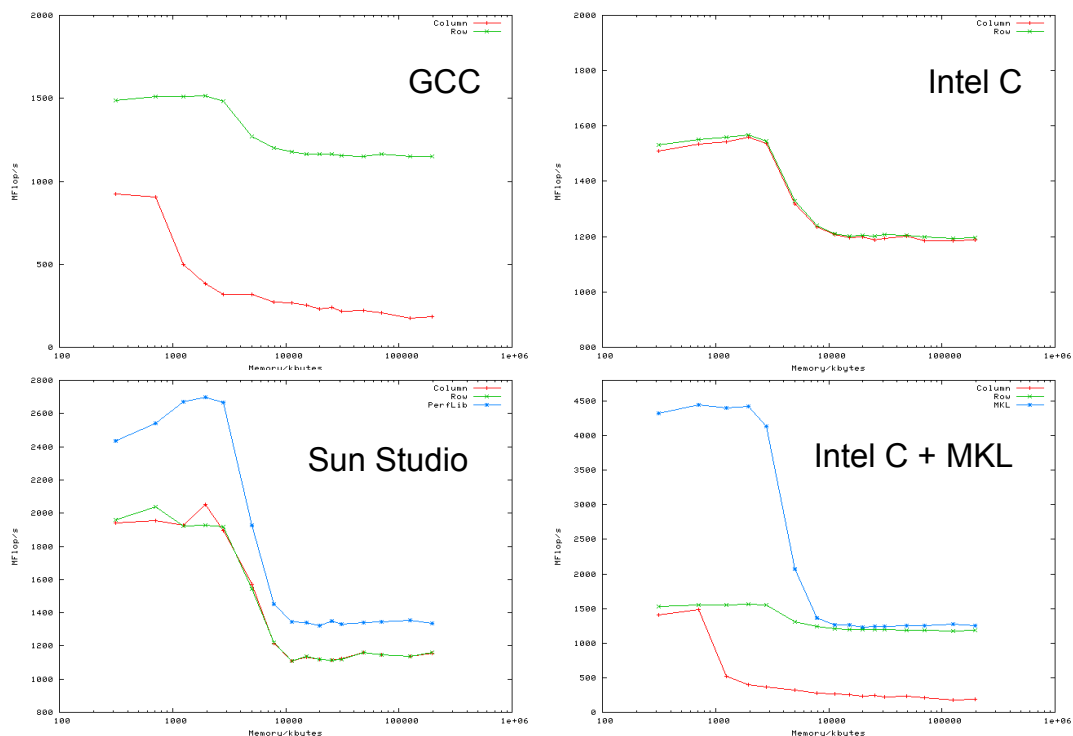
Matrix times vector

Comparison for large data sets:

Ease of use – vendor library

❑ Intel/GCC with Intel MKL:

- ❑ several compiler options
- ❑ several linker options
- ❑ depends on platform: IA32 or EMT64
- ❑ set up of run-time environment
- ❑ different downloads, installations and licenses

❑ Sun Studio:

- ❑ one linker option: -xlic_lib=sunperf
- ❑ no need to set up run-time environment
- ❑ one download, installation – and no license

Compiler commentary

Sun Studio: Compiler Commentary

How do I know what the compiler did with my code?

❏ Compile with -g (C++: -g0) and use the er_src command on the object files:

  ❏ This generates a code listing with comments from the compiler (also used in the Performance Analyzer GUI).

❏ Command: `er_src -src foo.c foo.o`

❏ Note: the examples on the next slides are from the Solaris SPARC version of Sun Studio – the x64 versions can differ

Sun Studio: Compiler Commentary

```c
// ex.c
#define A(i,j) a[(i)*n + (j)]
#define B(i,j) b[(i)*n + (j)]
#define C(i,j) c[(i)*n + (j)]

void
mysub(int m, int n, double *a, double *b,
      double *c) {

    int i, j;

    for(j = 0; j < n; j++)
      for(i = 0; i < m; i++)
        C(i,j) = A(i,j) + B(i,j);

}
```

Sun Studio: Compiler Commentary

```
5. void
6. mysub(int m, int n, double *a,
      double *b, double *c){
   <Function: mysub>
7.
8.      int i, j;
9.
```

UltraSPARC

```
Source loop below has tag L1
 10.      for(j = 0; j < n; j++)

Source loop below has tag L2
L2 scheduled with steady-state cycle count = 5
L2 unrolled 2 times
L2 has 2 loads, 1 stores, 6 prefetches, 1 FPadds, 0
FPmuls, and 0 FPdivs per iteration

 11.          for(i = 0; i < m; i++)
 12.              C(i,j) = A(i,j) + B(i,j);
 13. }
```

Sun Studio: Compiler Commentary

```
5.  void
6.  mysub(int m, int n, double *a,
         double *b, double *c){
    <Function: mysub>
7.
8.      int i, j;
9.
```

UltraSPARC

Source loop below has tag L1
L1 interchanged with L2
L1 scheduled with steady-state cycle count = 2
L1 unrolled 4 times
L1 has 2 loads, 1 stores, 3 prefetches, 1 FPadds, 0
FPmuls, and 0 FPdivs per iteration

```
10.      for(j = 0; j < n; j++)
```
Source loop below has tag L2
L2 interchanged with L1
```
11.         for(i = 0; i < m; i++)
12.            C(i,j) = A(i,j) + B(i,j);
13. }
```

Application Tuning

Sun Studio: Compiler Commentary

```
5.  void
6.  mysub(int m, int n, double *a,
         double *b, double *c){
    <Function: mysub>
7.
8.      int i, j;
9.
```

x86_64

Source loop below has tag L1
L1 interchanged with L2
L1 cloned for microvectorizing-epilog. Clone is L6
L1 is micro-vectorized

```
10.      for(j = 0; j < n; j++)
```
Source loop below has tag L2
L2 interchanged with L1
```
11.         for(i = 0; i < m; i++)
12.            C(i,j) = A(i,j) + B(i,j);
13. }
```

Application Tuning

Sun Studio: Compiler Commentary

```
% cc -g -fast -xrestrict -xvector=no%simd -c ex.c
% er_src -src ex.c ex.o
```

x86_64

```
   5. void
   6. mysub(int m, int n, double *a,
          double *b, double *c){
     <Function: mysub>
   7.
   8.      int i, j;
   9.

 Source loop below has tag L1
 L1 interchanged with L2
  10.      for(j = 0; j < n; j++)

 Source loop below has tag L2
 L2 interchanged with L1
  11.         for(i = 0; i < m; i++)
  12.            C(i,j) = A(i,j) + B(i,j);
  13. }
```

Sun Studio: Compiler Commentary

```
% cc -g -fast -xrestrict -c ex2.c
% er_src -src ex2.c ex2.o
```

```
   7. void
   8. mysub(int m, int n, double *a,
          double *b, double *c){
     <Function: mysub>
   9.
  10.      int i, j;
  11.
 Source loop below has tag L1
  12.      for(j = 0; j < n; j++)
 Source loop below has tag L2
  13.         for(i = 0; i < m; i++) {
  14.            C(i,j) = A(i,j) + B(i,j);

 Function mean not inlined because the compiler has
not seen the body of the routine
  15.            mean(C(i,j));
  16.         }
  17. }
```

Compiler Commentary – I

Why is this useful?

❏ Compilers are no longer a black box!

❏ What the compiler has done – or hasn't done/couldn't do – to the code is made visible to the programmer.

❏ Useful information is provided, so a programmer can take action, e.g.

  ❏ code changes

  ❏ different set of compiler options

Compiler Commentary – II

More reasons, why it is really useful:

❏ annotations are where they belong: in the code – and not on the screen during compilation (or in a log file)

❏ review at any time – even a long time after the code has been compiled

❏ visible in the analyzer output as well, together with runtime profile

Other tricks

Reconstruct the compiler options from the object files and/or executable:

- ❏ dwarfdump file.o (C/C++/Fortran)
- ❏ dumpstabs file.o (Fortran, before Studio 12u1)

❏ and look for

- ❏ command_line (dwarfdump) or
- ❏ CMDLINE (dumpstabs) in the output.

❏ Very useful to check what has been done.

❏ Note: dwarfdump works for GCC .o files, too

Application Tuning

DTU

Application Tuning

Analysis tools

DTU

Analysis tools

❑ analysis tools are useful to detect bottlenecks in codes

❑ modern analysis tools (unlike "old" profilers) work even on 'non-instrumented' code: no need to recompile (in principle)

❑ runtime profiles down to the source level (profilers usually work on function/subroutine level)

Analysis tools

❑ Oracle: Solaris Studio Performance Analyzer

  ❑ Linux (x64)

❑ Intel: Vtune Performance Analyzer (Windows/Linux)

❑ Mac OS X: Instruments (part of Xcode)

❑ 'perf' command line tool (Linux)

❑ Google Performance Tools (Linux/Windows):

  ❑ collection of runtime libraries and command line tools

# Sun Studio: Performance Analyzer

❑ Sun Studio provides a powerful toolset for runtime analysis

❑ Both GUI and command line tools

  ❑ analyzer – GUI for collecting and analyzing performance data

  ❑ collect – Command to collect performance data

  ❑ er_print – Command to analyze performance data in ASCII format (good for scripting)

Sun Studio: Performance Analyzer
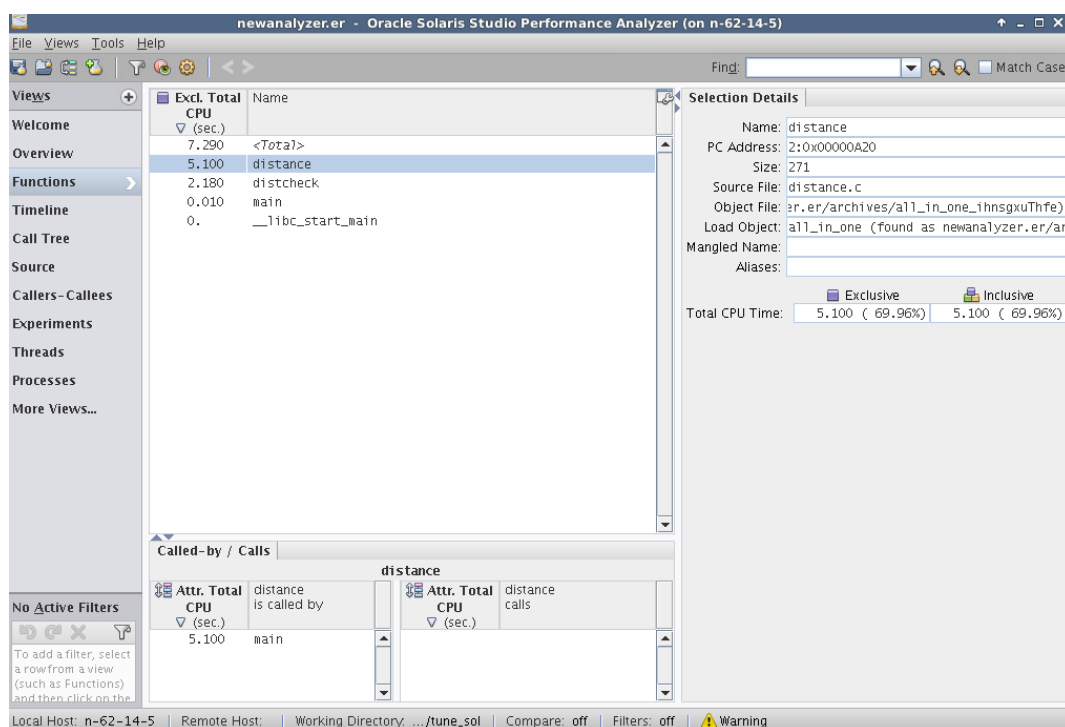
Sun Studio: Performance Analyzer
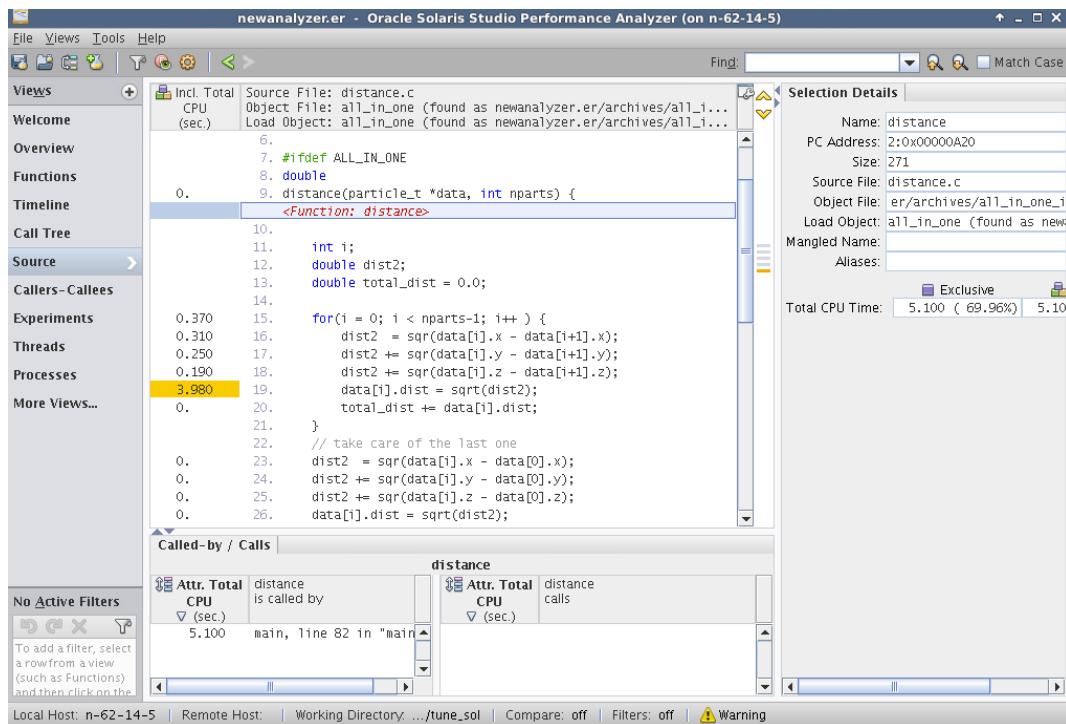
Hardware Performance Counters

❏ Almost all modern CPUs have built-in hardware performance counters:

  ❏ How many instructions were executed?

  ❏ How many clock cycles were used?

  ❏ How many L1 data cache misses occured?

❏ The supported counters are usually listed in the architecture reference manuals.

❏ Be aware: The counter names are not for beginners!

Using the Performance Counters

❑ Native OS tools, e.g. Linux:

    ❑ perf – Performance monitoring tool

    ❑ requires newer Linux kernel (> 2.6.31)

    ❑ examples:

```
% perf stat -e <event_name> -- command
% perf stat -e <event_name> -p PID

% perf record -e <event_name> -- command
% perf report
```

    ❑ 'perf top' - requires root priviliges
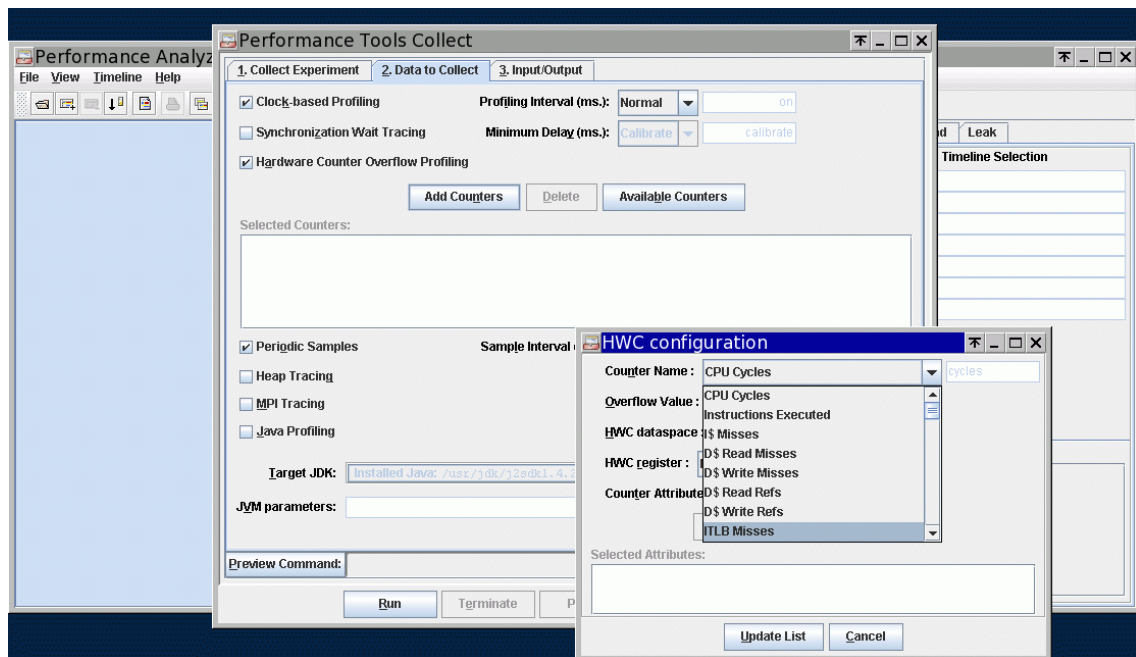

Using the Performance Counters

❑ Available performance counters:

    ❑ system and CPU dependent

    ❑ get a list:

    ❑ `% perf list`

    ❑ `% collect`   (no argument)

    ❑ example: no. of available performance counters on

        ❑ US-IV: 70

        ❑ US-IV+:   101

        ❑ AMD Opteron: 169

        ❑ Xeon E5-... v3:   269

Using the Performance Counters

Application Tuning

Activating performance counters in analyzer:

Sun Studio: Performance Analyzer

Application Tuning

Analyzer demo

Tuning Guide – compact version

❏ Make a 'baseline' version (with different data sets/memory requirements)

❏ Try to find the best compiler options

  ❏ with or w/o prefetching

  ❏ ...

❏ Use analysis tools to locate the 'hot spots'

❏ Introduce code changes

❏ Repeat the last two steps until you are satisfied

Application Tuning