High-Performance Computing

Getting good performance from your application

Tuning techniques for serial programs on cache-based computer systems

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
```

Use of local variables in loops

```
double a[N];
double b[N];
double x;
for(int i=0; i<N; i++) {
    x = f(a[i]);
    b[i] = g(x);
}
```

```
double a[N];
double b[N];

for(int i=0; i<N; i++) {
    double x = f(a[i]);
    b[i] = g(x);
}
```

This version can “sometimes” give compilers better opportunities to optimize the loop!

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*

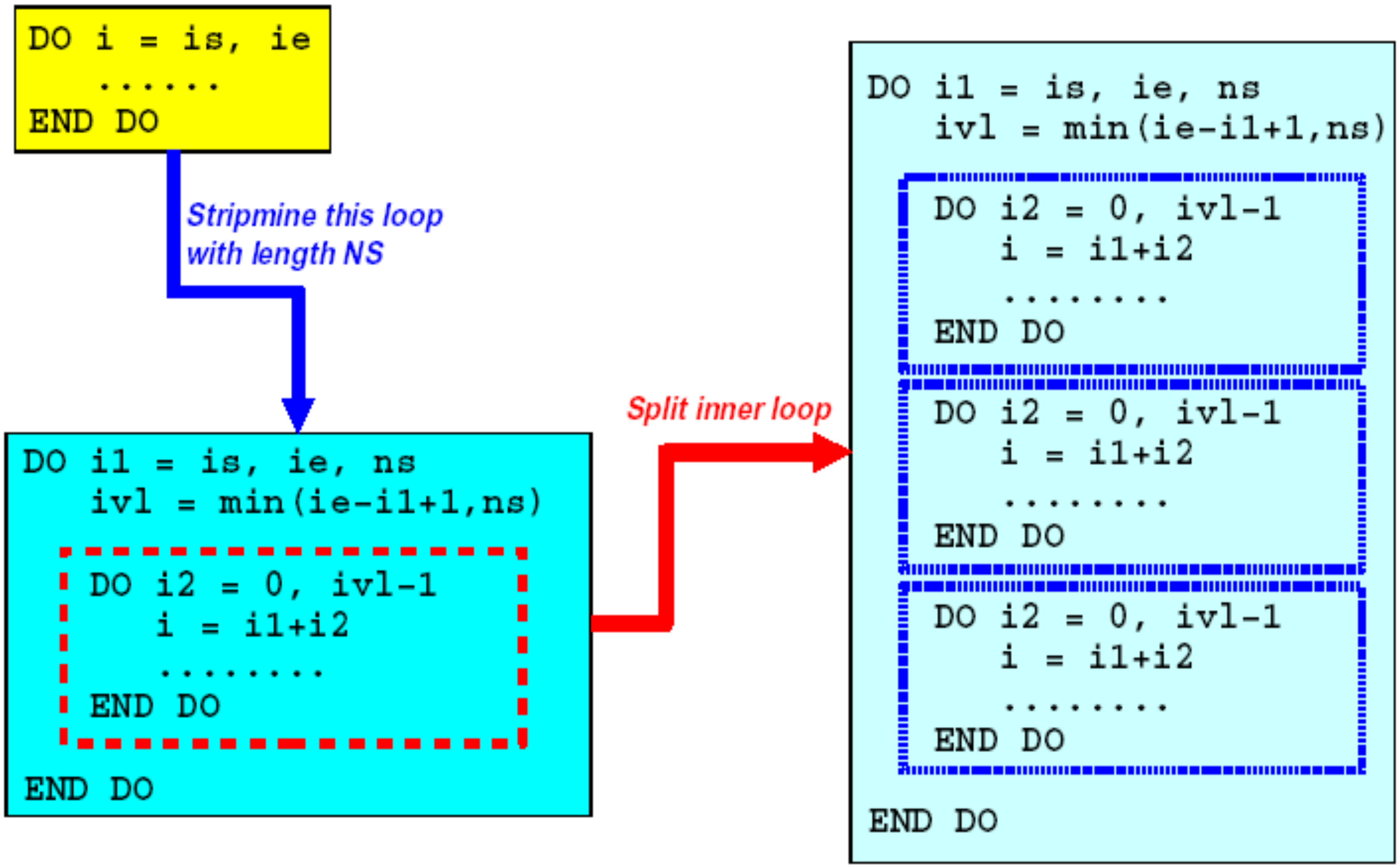
```
 DO I = 1, LONG
    X(I) = ...
    A    = ...
    Y(I) = A + ...
 END DO
```

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



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)
  - the compiler will typically inline code, if it has access to it at compile time (same source file)
  - it is also possible to do this at the linking stage (will be covered later)
- 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

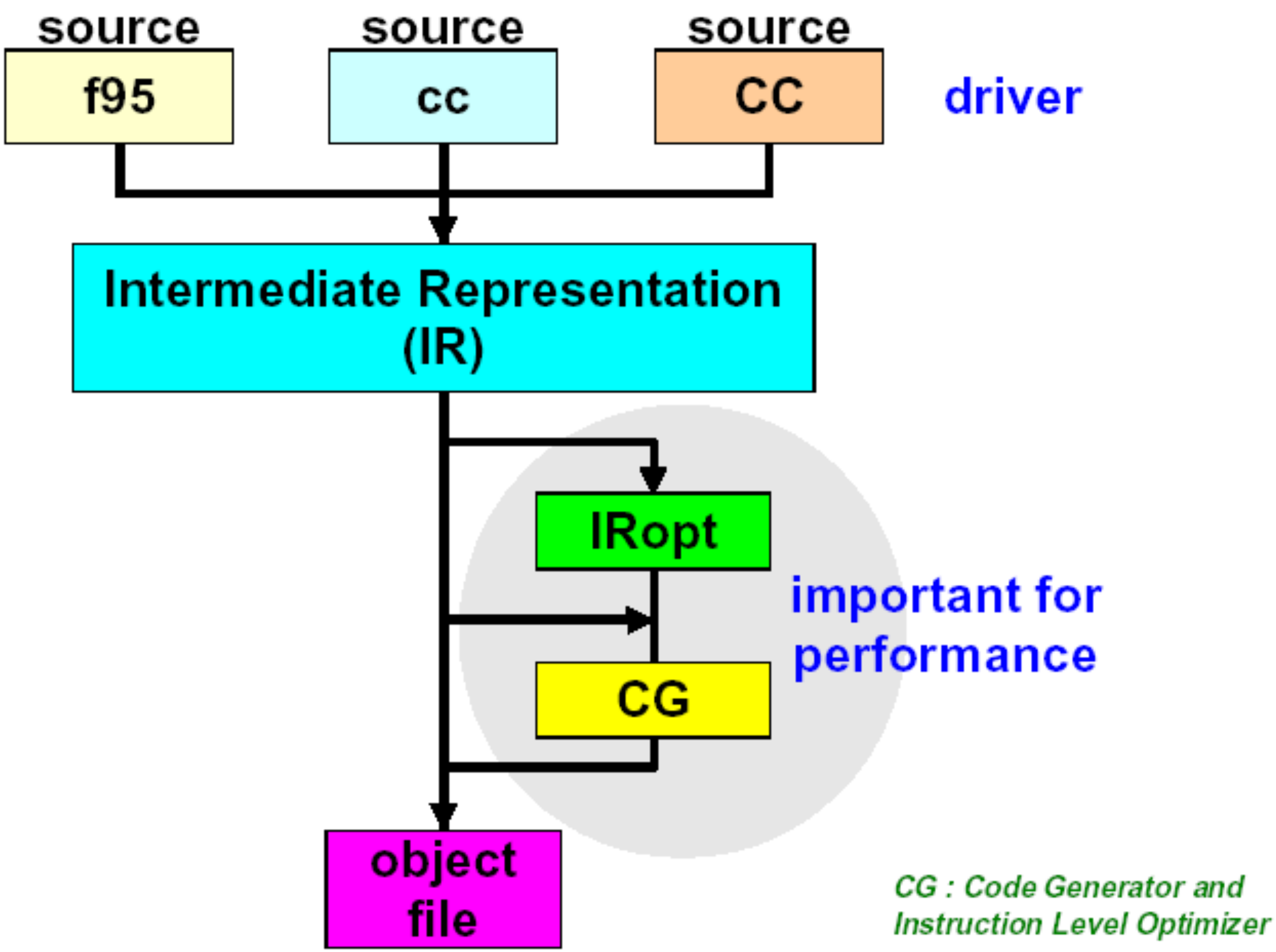
  and your problem's *memory footprint*
- Use the *best* compiler available on your platform.

How compilers work

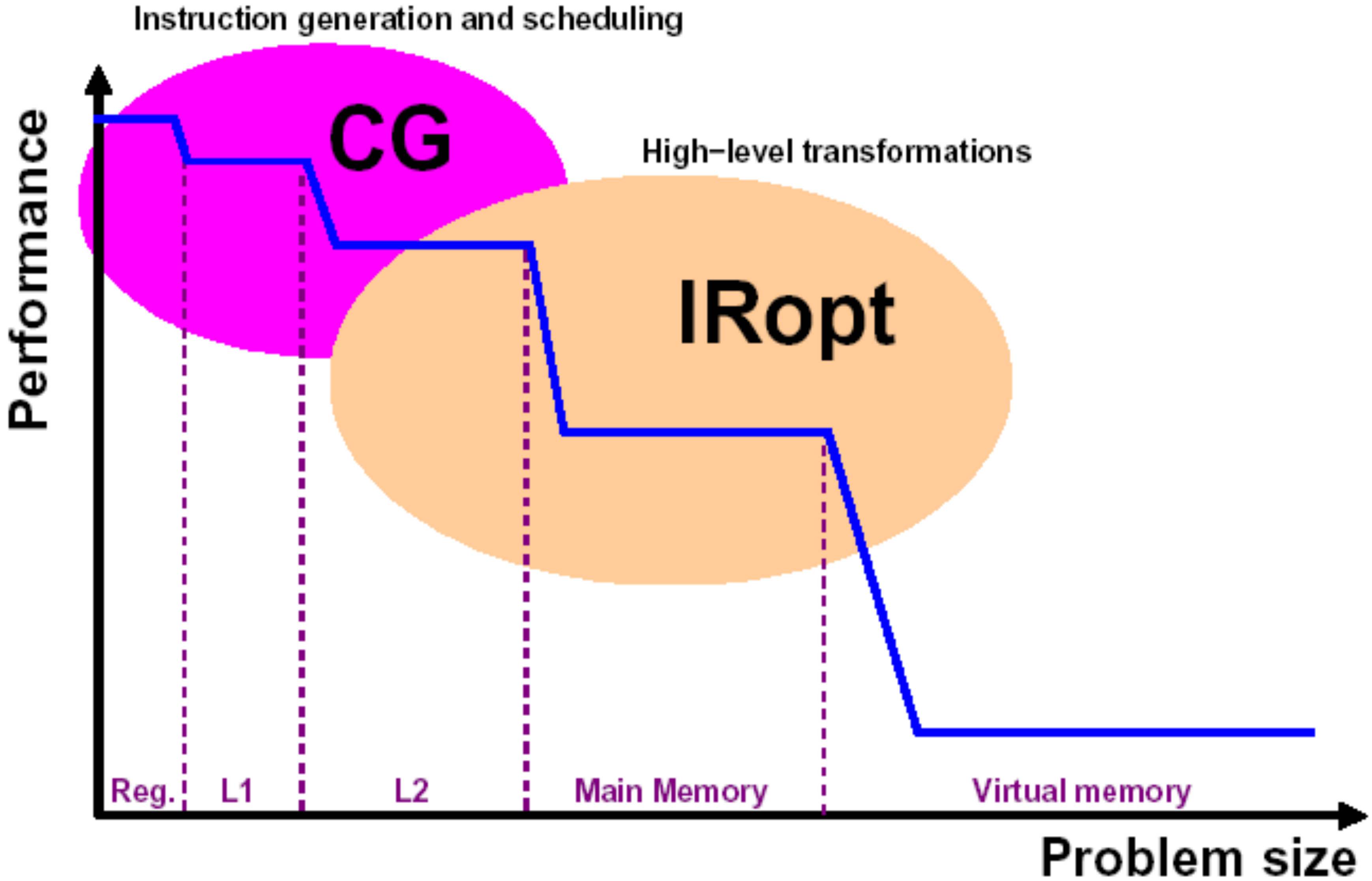
Compilers: overview

- Compiler Components
  - compilers are not single programs, but consist of a whole toolchain
  - using Oracle Studio as an example here, for illustration purposes
- Compiler Options
  - minimal set of optimization options ...
  - ... and the more detailed view
- Some specific examples

Oracle Studio: Compiler toolchain



Oracle Studio: Who does what?



Oracle Studio: Compiler Options

In general, one gets very good performance by just using 2 options for compiling and linking:

```
-g -fast
```

For specific x86_64-processors (cross-compiling):

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`
Intel Skylake:	`-g -fast -xchip=skylake`

Oracle 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)

What about other compilers?

- Most compilers have similar options, that combine many optimizations into a single option
  - but they do not mean always the same!
- GCC: -O, -O2, -O3, -Ofast
- Intel: -O0, -O2 (default!), -O3, -fast
- look up in the manpages/documentation, what that corresponds to
- some options have side effects
  - e.g. Intel and ‘-fast’ changes the linking

GCC: Recommendations

- a good start: -g -O3
- show optimizer options (“expand -O3”):
  - 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 or -fno-unroll-loops
- finding the differences: dump output of command above into files, and run diff on the files

GCC: Recommendations

- some differences between -O3 and -Ofast
  - e.g. math related options

```
Option                                    -O3              -Ofast
-----------------------------------------------------------------
-fassociative-math                        [disabled]       [enabled]
-ffinite-math-only                        [disabled]       [enabled]
-fmath-errno                              [enabled]        [disabled]
-freciprocal-math                         [disabled]       [enabled]
-ftrapping-math                           [enabled]        [disabled]
-funsafe-math-optimizations               [disabled]       [enabled]
```

GCC: Recommendations

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

```
OPT    = -g -O3
ISA    = -mavx2
CHIP   = -march=broadwell
CFLAGS = $(OPT) $(ISA) $(CHIP)
```

- *Always start with* `-O3` *!*
- Then add other options, to change/increase optimization

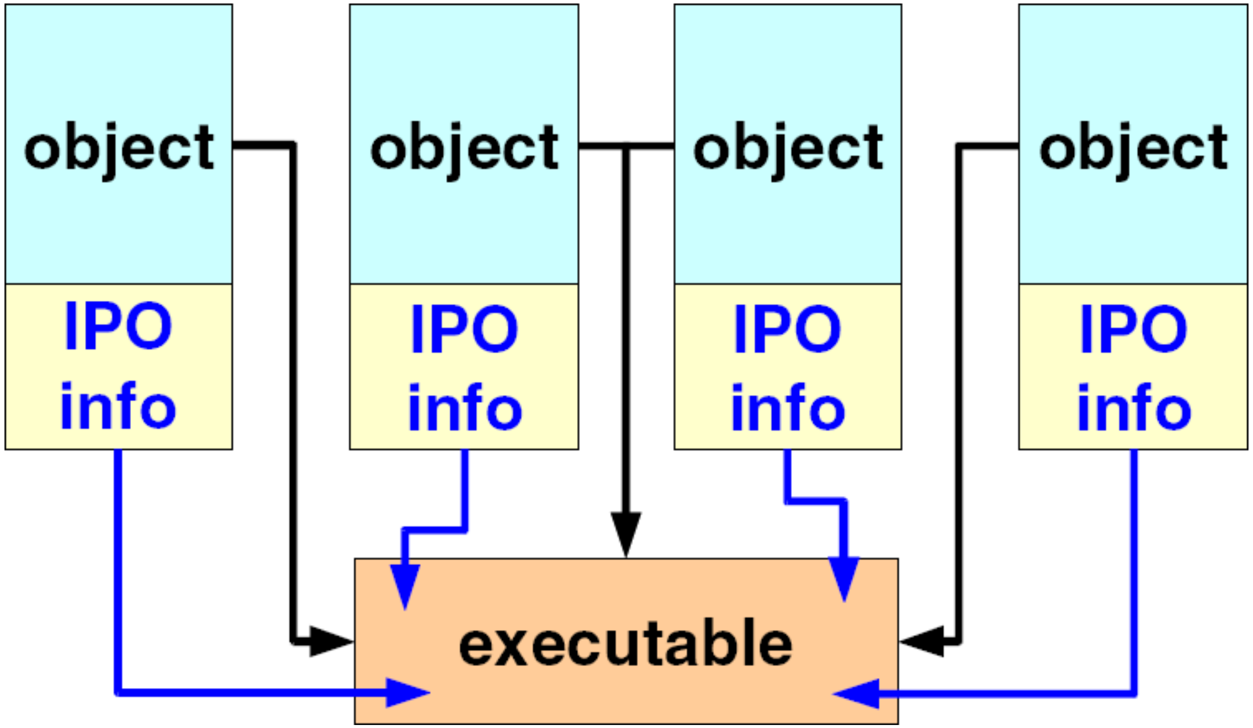
GCC: Useful options

- -Q --help=optimizers -On : displays the active optimizer settings for optimization level ‘On’ (long list)
- -Q --help=params : displays the internal parameters
- -fopt-info : show optimization info at compile time
- -v : displays the configured features
- --version : Shows the compiler version

```
$ gcc --version
gcc (GCC) 6.3.0
Copyright (C) 2016 Free Software Foundation, Inc.
```

Inter Procedural Optimization

- When used, the compiler stores additional information into the object files
- This information is used during the link phase to perform additional optimizations



Inter Procedural Optimization

How to use with different compilers?

- GCC: here it is called ‘link time optimization’
  - -flto
  - check the documentation for more details
  - example on next slides
- clang: uses -flto as well
  - the internals might differ from GCC
- Intel:
  - -ipo

GCC: link time optimization

main.c

```
double init_array(int, float *);

int main(int argc, char *argv[]) {
    int len = atoi(argv[1]);
    float *arr = malloc(len * sizeof(arr));

    // put values into arr
    for(int i = 0; i < len; i++)
        init_array(i, &arr[i]);

    // print the first and last ten values of arr
    for(int i = 0; i < 10; i++)
        printf("a[%d] = %f  ...  a[%d] = %f\n",
               i, arr[i], len-10+i, arr[len-10+i]);

    return(0);
}
```

GCC: link time optimization

init.c

```
#include <math.h>

double
init_array(int n, float *val) {

    *val = (float)n;

    return sin(n);
}
```

We do not use the return value in main, i.e. all calls to sin() are wasted!

```
$ make
gcc -g -O3 -funroll-loops -c -o main.o main.c
gcc -g -O3 -funroll-loops -c -o init.o init.c
gcc -g -O3 -funroll-loops -o ipo_ex.gcc main.o init.o -lm

$ $ time ./ipo_ex.gcc 100000000 > /dev/null
real  0m4.618s
user  0m4.530s
sys   0m0.088s
```

GCC: link time optimization

```
$ make
gcc -g -O3 -funroll-loops -flto -c -o main.o main.c
gcc -g -O3 -funroll-loops -flto -c -o init.o init.c
gcc -g -O3 -funroll-loops -flto -o ipo_ex.gcc main.o init.o
-lm

$ time ./ipo_ex.gcc 100000000 > /dev/null
real    0m0.115s
user    0m0.020s
sys     0m0.093s
```

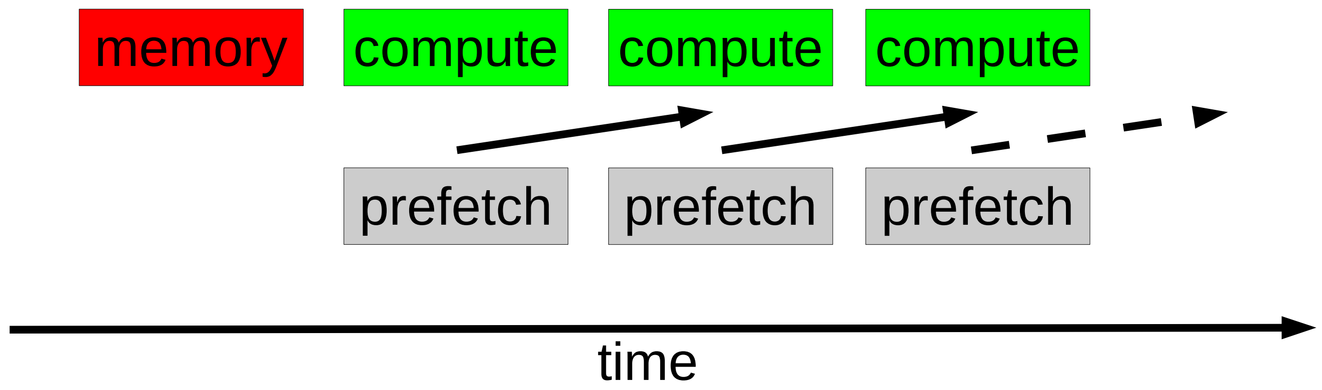
- in the compile phase, extra information is generated
- the linker can use this information to optimize further
- here, the calls to sin() are removed, as we do not use the result
    - no reference to sin() in the executable

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

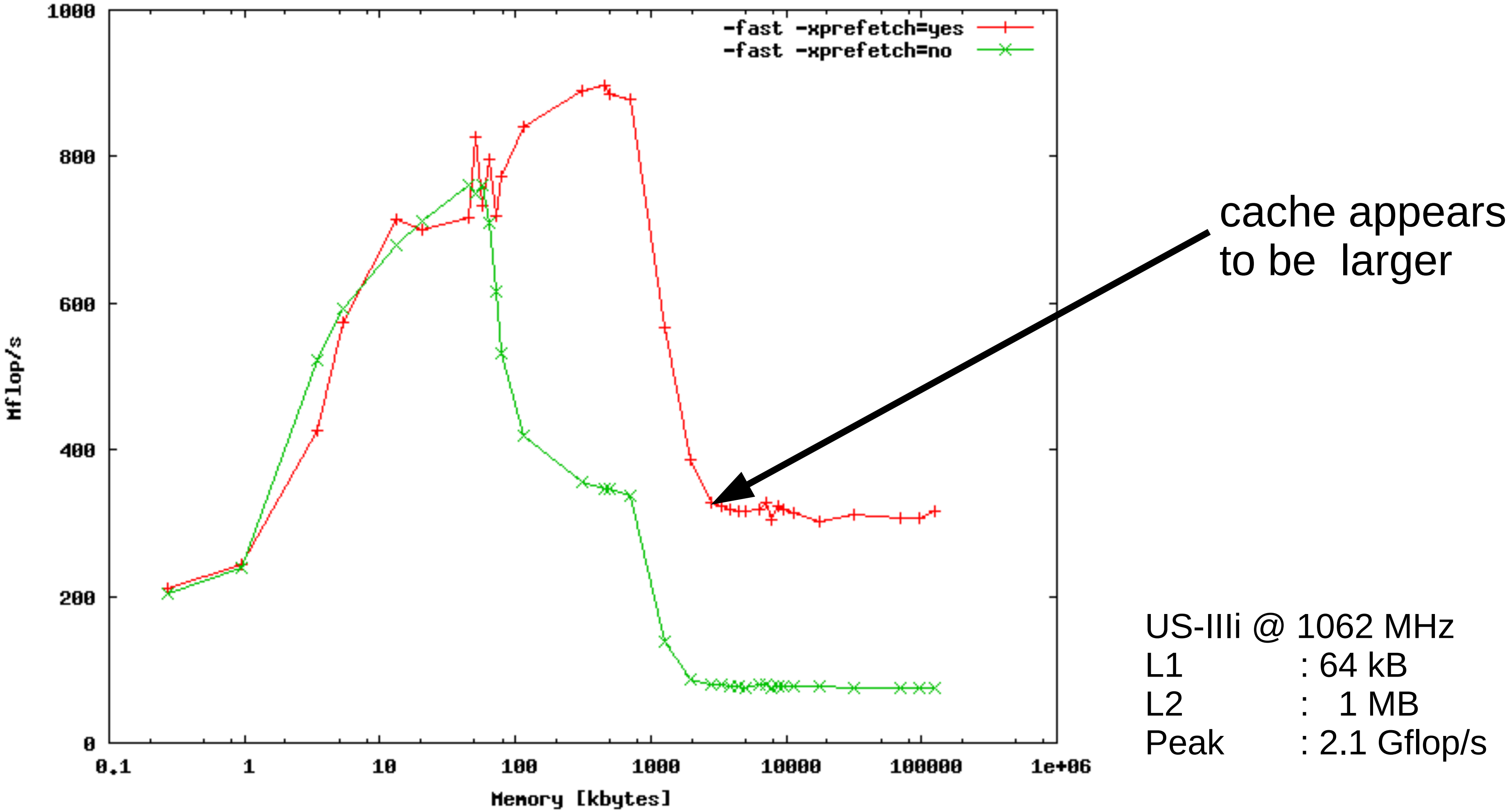


Prefetch Support

- Prefetch is a common feature in modern CPUs: both data and instruction prefetch.
- Implementation is system dependent!
- There is both
  - software prefetch (compiled into the program)
  - hardware prefetch – which often cannot be disabled
- x86_64 CPUs have HW prefetch
- next slide: effect of prefetch on a CPU w/o hardware prefetch

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

GCC: Prefetch options

- enabled with -O2 and higher
  - -fprefetch-loop-arrays
  - there is a number of parameters, that can be tuned
    - simultaneous-prefetches
    - min-insn-to-prefetch-ratio
    - prefetch-min-insn-to-mem-ratio
  - check with ‘gcc -Q –help=params’
  - usually it is not necessary to change the defaults
  - has hardly any effect, due to HW prefetch

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!

Pointer overlap – or "aliasing"

- 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, using the `restrict` keyword
- Note: It is then <u>your responsibility</u> that this assumption will not be violated!

No pointer overlap – use of ‘restrict’

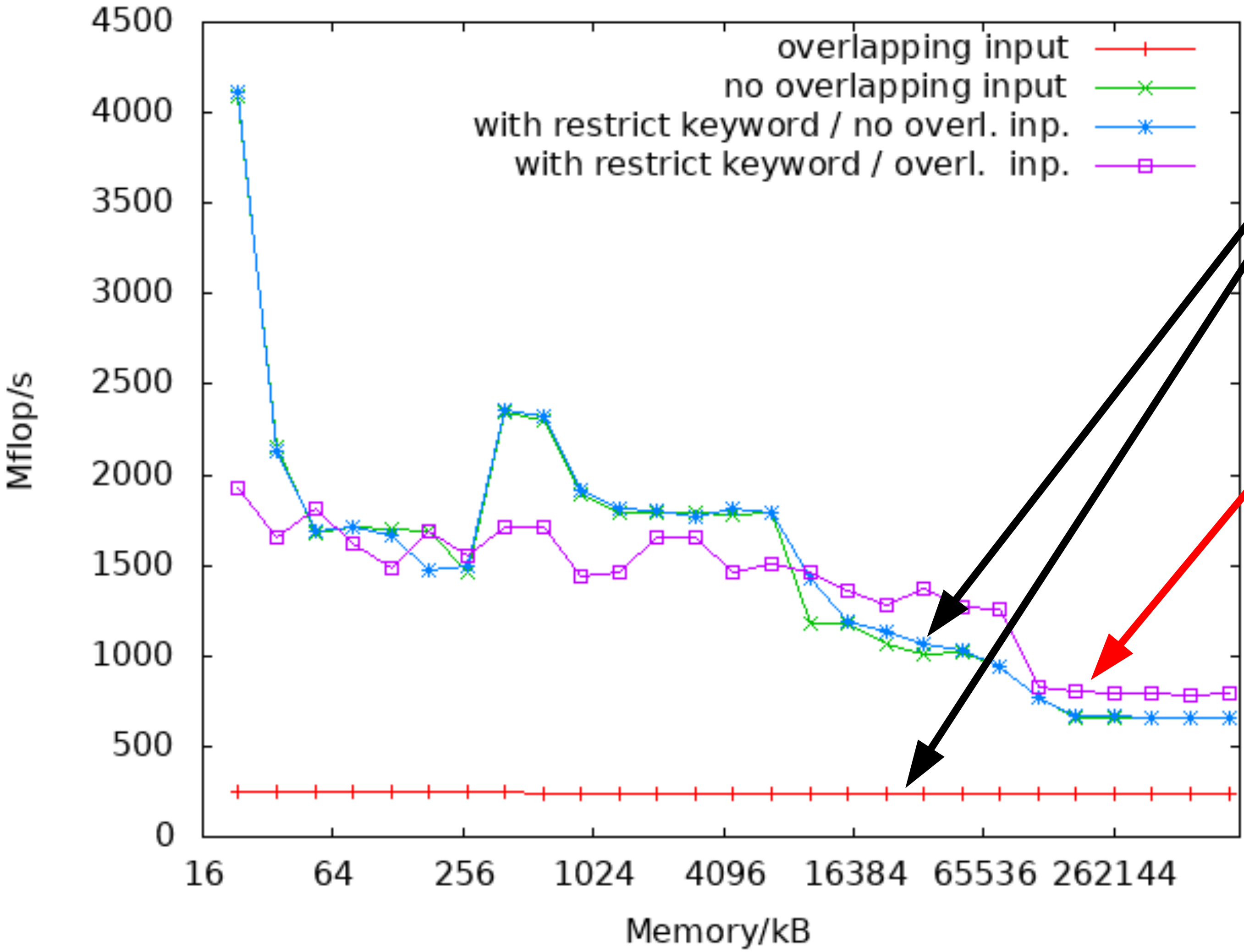
- If you can assure that the pointers don't overlap, you can fix your code using the C99 'restrict' keyword:

```
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];
}
```

- Needs a C99 compliant compiler to be portable!
- Wrong results, if called with overlapping pointers!

Compiler optimization vs ‘restrict’

Vector addition example



compiler generated multiple versions

Faster – but wrong results!

XeonE5-2650v4 @ 2.2GHz
L1 : 32 kB
L2 : 256 kB
L3 : 30 MB
GCC 10.2
-O3 -funroll-loops

Compiler optimization vs ‘restrict’

- compilers can create multiple versions of loops
- runtime decision, which path to take
  - no overlap in data: use optimized version
  - overlap in data: use the “slow” – but correct – version
- with ‘`restrict`’ keyword:
  - only the optimized version is generated
  - gives wrong results, if not called correctly
- for simple codes like here, leave it to the compiler!

Other tricks

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

- dwarfdump file.o

- and look for keywords
  - command_line or producer
- Very useful to check what has been done to compile the code.

```
$ gcc -g -O3 -funroll-loops -c -o init.o init.c
$ dwarfdump init.o | grep producer
   DW_AT_producer  GNU C17 10.2.0 -mtune=generic
                                 -march=x86-64 -g -O3 -funroll-loops
```

Analysis tools

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

- gprofng
  - Linux – open source
  - part of GNU ‘binutils’ since version 2.39
  - formerly “Oracle Studio Performance Analyzer”
- 'perf' command line tool (Linux)
- Intel: Vtune Performance Analyzer
  - Windows/Linux
  - part of the OneAPI tools (free)
- Mac OS X: Instruments (part of Xcode)

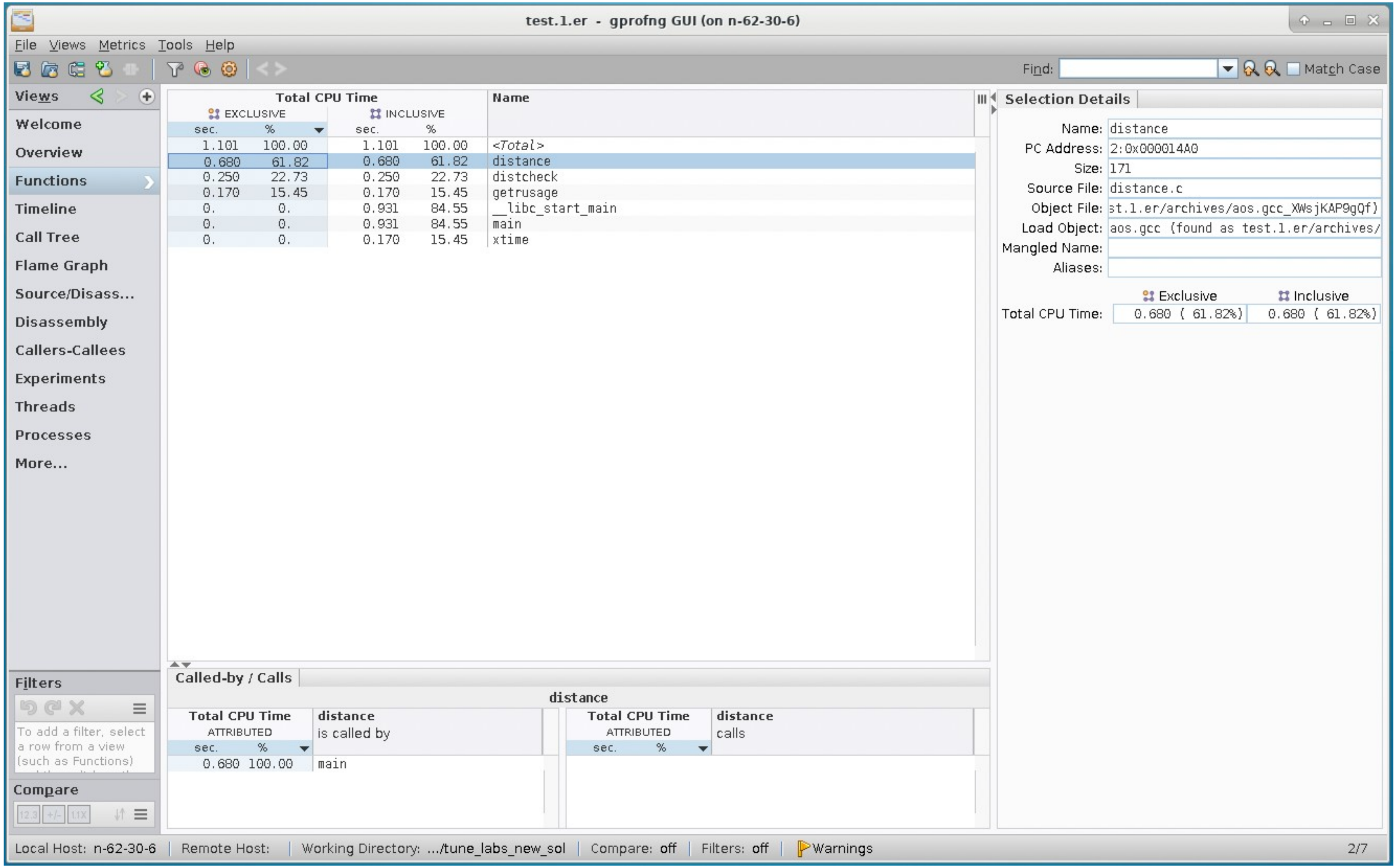
gprofng – application profiling tool

- gprofng provides a powerful toolset for runtime application performance analysis
- both command line tools – and a GUI
- gprofng works with sub-commands
  - ... collect app – Command to collect performance data
  - ... display text – Command to analyze performance data in ASCII format (good for scripting)
  - ... display gui – GUI for collecting and analyzing performance data

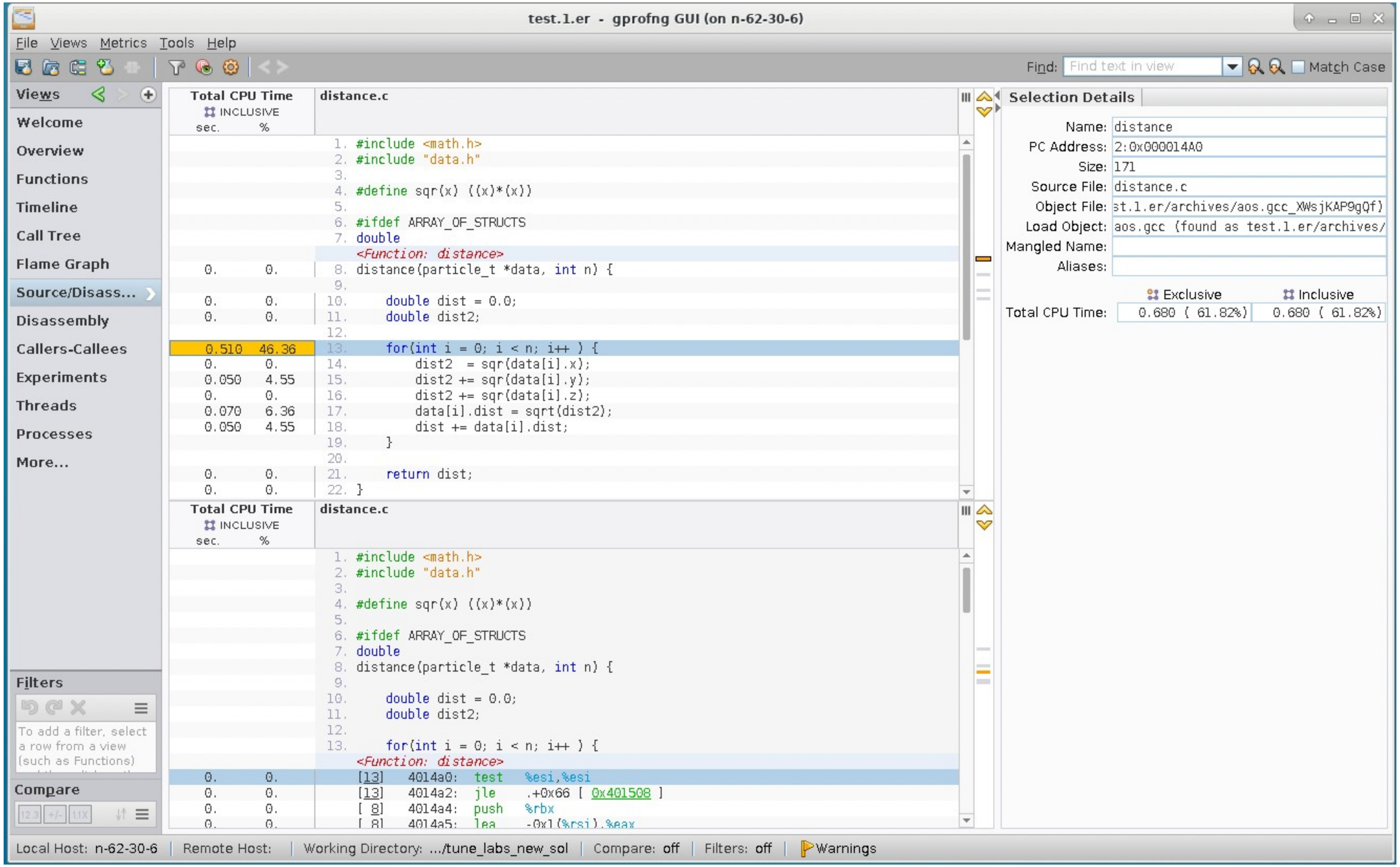
gprofng – application profiling tool

- Useful on-line documentation for gprofng
  - Blog:
    gprofng: The Next Generation GNU Profiling Tool
  - Wiki page: The gprofng Application Profiling Tool
  - Video:
    https://www.youtube.com/watch?v=JvnWIv2THTg
  - gprofng GUI - link to the GUI homepage (the GUI is not part of the standard binutils, and might need to be installed as an add-on)

gprofng – application profiling tool



gprofng – application profiling tool



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

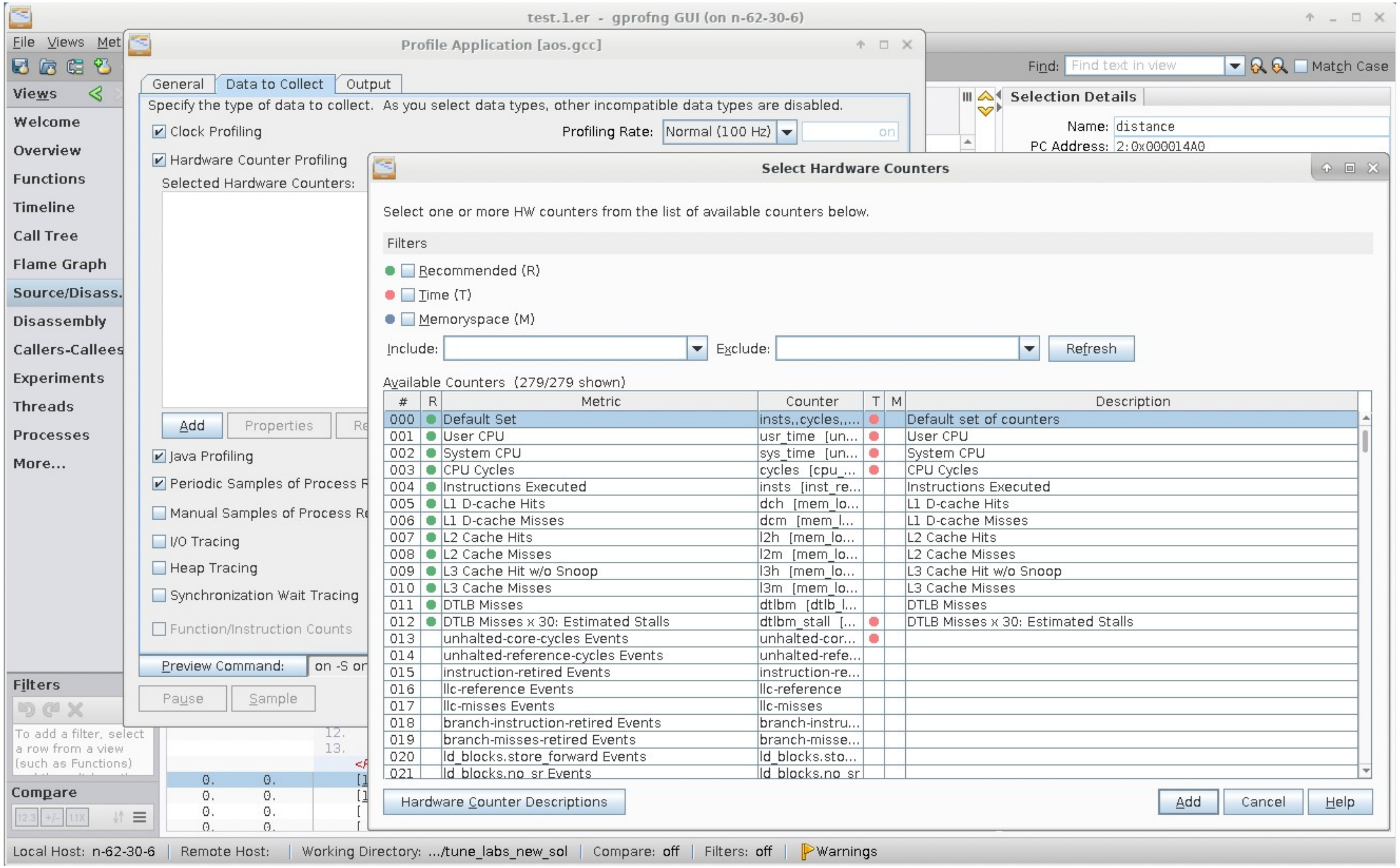
Using the Performance Counters

- Available performance counters:
  - system and CPU dependent
  - get a list:
  - `% perf list`
  - `% gprofng collect app -h`
  - example: no. of available performance counters on
    - AMD Opteron: 169
    - Xeon E5-... v3: 266
    - Xeon E5-... v4: 311
    - Xeon Gold... : 246

Application Tuning

Using the Performance Counters

Activating performance counters in gprofng GUI:



gprofng – application profiling tool

gprofng 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 changes: code and/or compiler options
- Repeat the last two steps until you are satisfied

End of lecture 2