

Optimisation Tools



- Timers
- Profilers
- HPM Toolkit

- Wide range of timers available on the HPCx system
- Varying
 - precision
 - portability
 - language
 - ease of use

Timer	Usage	Wallclock/C PU	Resolution	Language	Portable
time	shell	both	centisecond	F, C/C++	yes
timex	shell	both	centisecond	F, C/C++	yes
gettimeofday	subroutine	wallclock	microsecond	C/C++	yes
read_real_time	subroutine	wallclock	nanosecond	C/C++	no
rtc	subroutine	wallclock	microsecond	F	no
irtc	subroutine	wallclock	nanosecond	F	no
dtime_	subroutine	CPU	centisecond	F	no
etime_	subroutine	CPU	centisecond	F	no
mclock	subroutine	CPU	centisecond	F	no
timef	subroutine	wallclock	millisecond	F	no
MPI_Wtime	subroutine	wallclock	microsecond	F, C/C++	yes
system_clock	subroutine	wallclock	centisecond	F	yes

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

struct timeval start_time, end_time;
main() {
    int total;
    gettimeofday(&start_time, (struct timeval*)0);

    /* code to be timed */

    gettimeofday(&end_time, (struct timeval*)0);

    total = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
            (end_time.tv_usec-start_time.tv_usec);

    printf("Total time %d microseconds \n", total);
}
```

```
integer :: clock0,clock1,clockmax,clockrate,ticks
real    :: secs
call system_clock(count_max=clockmax,
                  count_rate=clockrate)

call system_clock(clock0)
! code to be timed
call system_clock(clock1)

ticks = clock1-clock0
! reset negative numbers
ticks = mod(ticks+clockmax, clockmax)
secs = float(ticks)/float(clockrate)

print*, 'Total time ', secs, ' seconds'
```

- May have used irtc previously on the Cray

```
integer(8) :: start, end, irtc
```

```
start = irtc()
```

```
! code to be timed
```

```
end = irtc()
```

```
print*, ' Total time ', end - start, 'nanoseconds'
```

- Investigate which parts of the program are responsible for most of the execution time
 - Program counter sampling
 - The program is interrupted at regular intervals (0.01s) and the location of the program counter (PC) is recorded
 - From these records the amount of time spent in each routine can be estimated
 - For stable results the runtime of the code has to be long compared to the sampling interval
 - Invocation counting
 - Compiler inserts a call to `mcount` each time a subroutine or function is invoked

- Compile and link your application with the option `-p`
`xlF90 -qsuffix=f=f90 -p -c my_prog.f90`
`xlF90 -p -o my_prog.x my_prog.o`
- Run your application on the backend, using `loadleveler`
 - This will generate a file: `mon.out`
- `prof` is available on most UNIX systems
- Examine the output
`prof my_prog.x |more`
- **Remark:** When compiling with `mpxlf` or `mpxlc`, the run will generate a file: `mon.out.0`, examine:
`prof my_prog.x -m mon.out.0 |more`

Sample output from prof

Relative time

Absolute time

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.__image_work_MOD_ma	64.5	36.02	36.02	2000	18.010
.__image_work_MOD_re	31.2	17.40	53.42	2000	8.700
.__hpf_random_modul	1.2	0.66	54.08		
.FormatControl	0.8	0.46	54.54		
.pgmwrite	0.4	0.24	54.78	1	240.
._xlfWriteFmt	0.4	0.24	55.02		
.WriteUnit	0.4	0.20	55.22		
.memmove	0.3	0.18	55.40		
.IOWrite	0.3	0.16	55.56		
.FmtLongIntToDec	0.1	0.06	55.62		
._fill	0.1	0.06	55.68		
.__image_work_MOD_ed	0.1	0.05	55.73	1	50.
.__image_mpi_comm_MO	0.1	0.05	55.78	1	50.
._xlfWriteFmtArray	0.1	0.04	55.82		
._mcount	0.0	0.01	55.83		
.__hpf_random_modul	0.0	0.01	55.84		
.image_main	0.0	0.00	55.84	1	
.__image_work_MOD_pr	0.0	0.00	55.84	2000	
.__image_work_MOD_up	0.0	0.00	55.84	6001	
.__time_tool_MOD_now	0.0	0.00	55.84	2	0.
.__locale_init	0.0	0.00	55.84		
.__time_init	0.0	0.00	55.84		
Standard input					

Average time/call

Number of Calls

Name of function or subroutine

- If `mcount` appears high on the list
 - Significant time spent in subroutine calls
 - Write longer routines or recompile with `-qinline`
- Profiling code with inlining needs care
 - Time spent in inlined routines will be attributed to mother routine
- Flat profile of `prof` provides no information on mother routine
 - Would be useful for routines called from different places
 - To obtain call tree use `gprof` or `xprofiler`, see below

- Compile and link your application with the option `-p`
`xlF90 -qsuffix=f=f90 -pg -c my_prog.f90`
`xlF90 -pg -o my_prog.x my_prog.o`
- **Note:** `-pg` is not `-p -g`
- Run your application on the backend, using loadleveler
 - This will generate a file: `gmon.out`
- Examine the output
`gprof my_prog.x gmon.out |more`

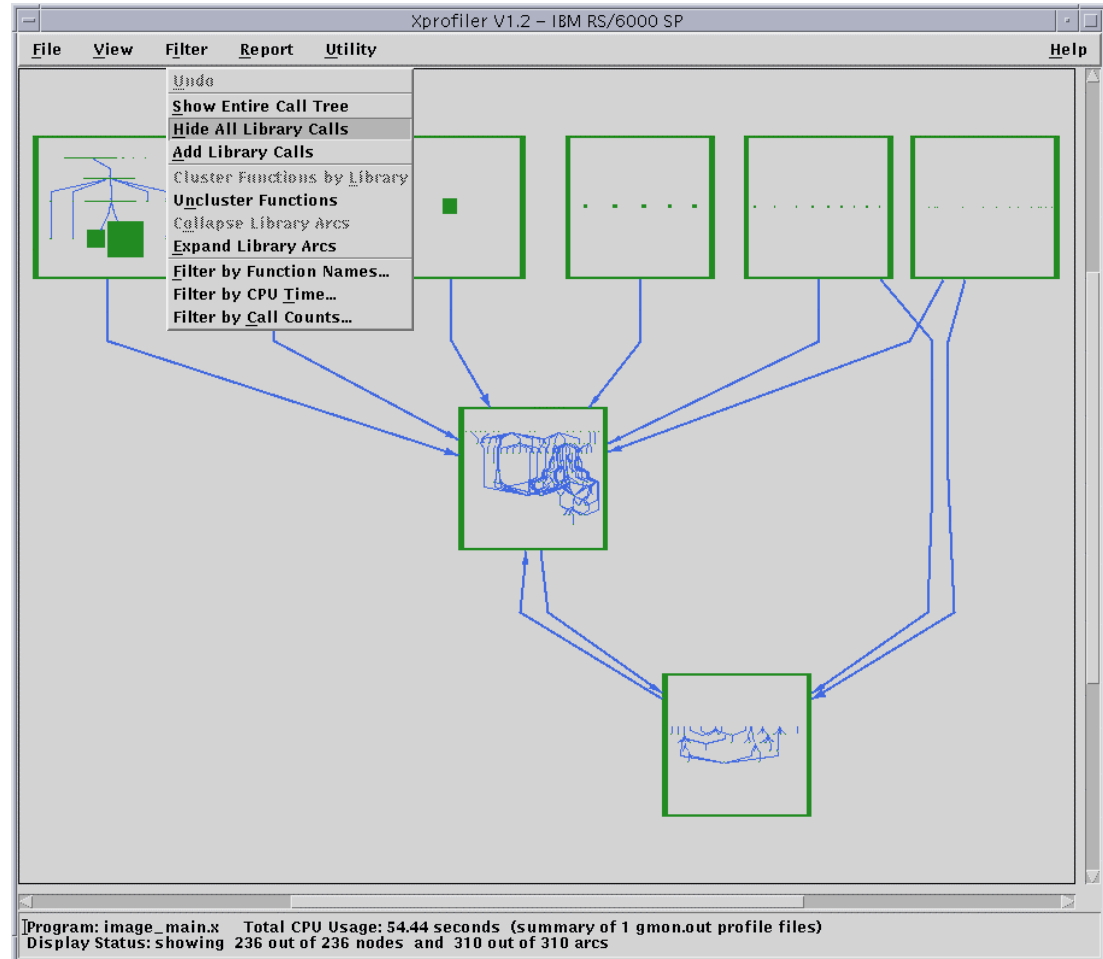
Sample output from gprof

```
joachim@bronzite:~  
Window Edit Options Help  
-----  
index %time self descendent called/total parents  
called+self name children index  
called/total  
-----  
[1] 96.0 0.00 53.41 1/1  
0.00 53.41 1  
0.04 0.22 1/1  
0.05 0.00 1/1  
0.00 0.00 6001/6001  
0.00 0.00 1/1  
-----  
6.6s  
[2] 96.0 0.00 53.41 <spontaneous>  
0.00 53.41 1/1  
-----  
[3] 95.5 0.00 53.10 2000/2000  
0.00 53.10 2000  
36.06 0.00 2000/2000  
17.04 0.00 2000/2000  
-----  
[4] 64.8 36.06 0.00 2000/2000  
36.06 0.00 2000  
-----  
Standard input
```

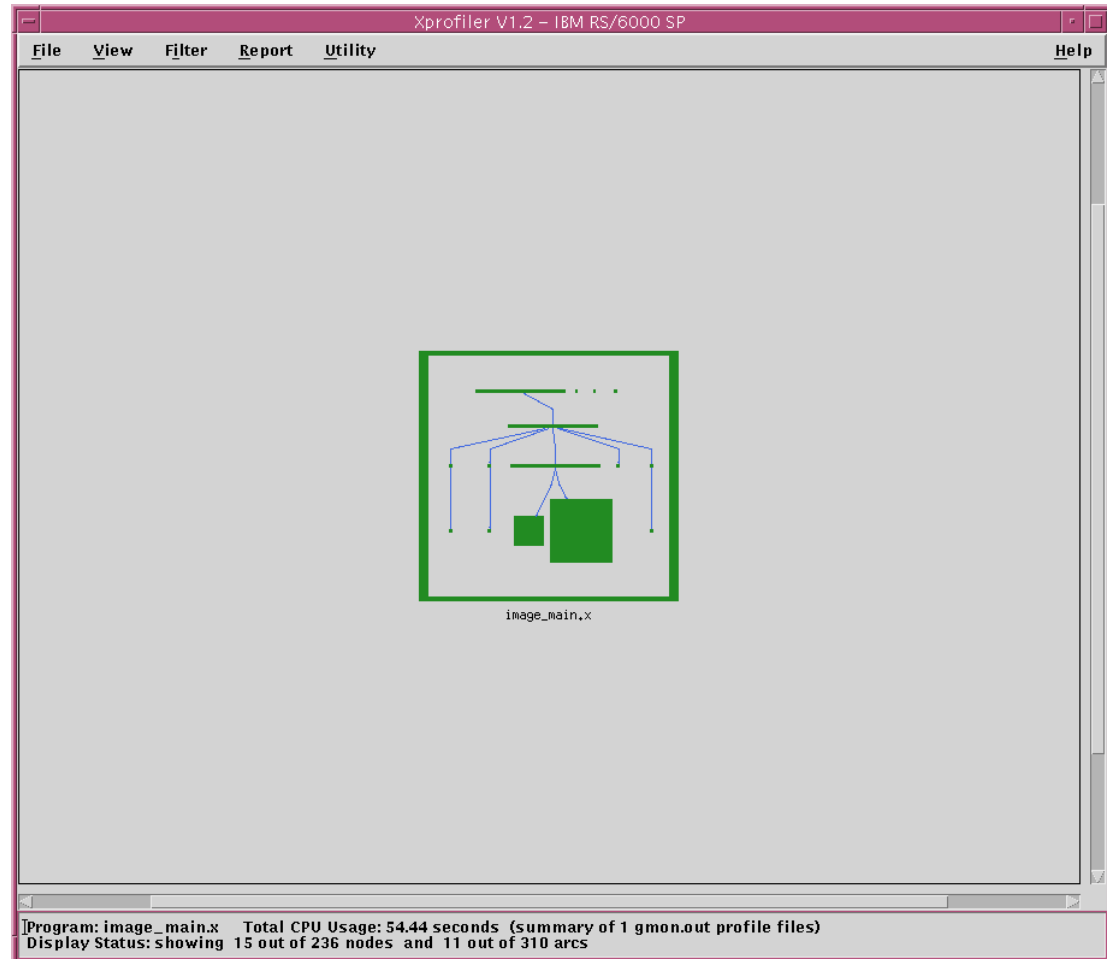
- **xprofiler** reads the same input data as **gprof**
 - Compile and link with `-pg` and run to obtain `gmon.out`
- **Key differences to gprof**
 - Graphical user interface (gui)
 - Profiling at source statement level
 - This needs compiler option `-g` in addition to `-pg`
 - Needs care when interpreting results
- **Starting xprofiler**
`xprofiler my_prog.x gmon.out &`

The xprofiler main window

- **xprofiler starts with its main window**
- **Often extremely busy**
 - Includes calls to libraries
- **For not too complex applications choose:**
 - **Filter**
 - **Hide all Library Calls**

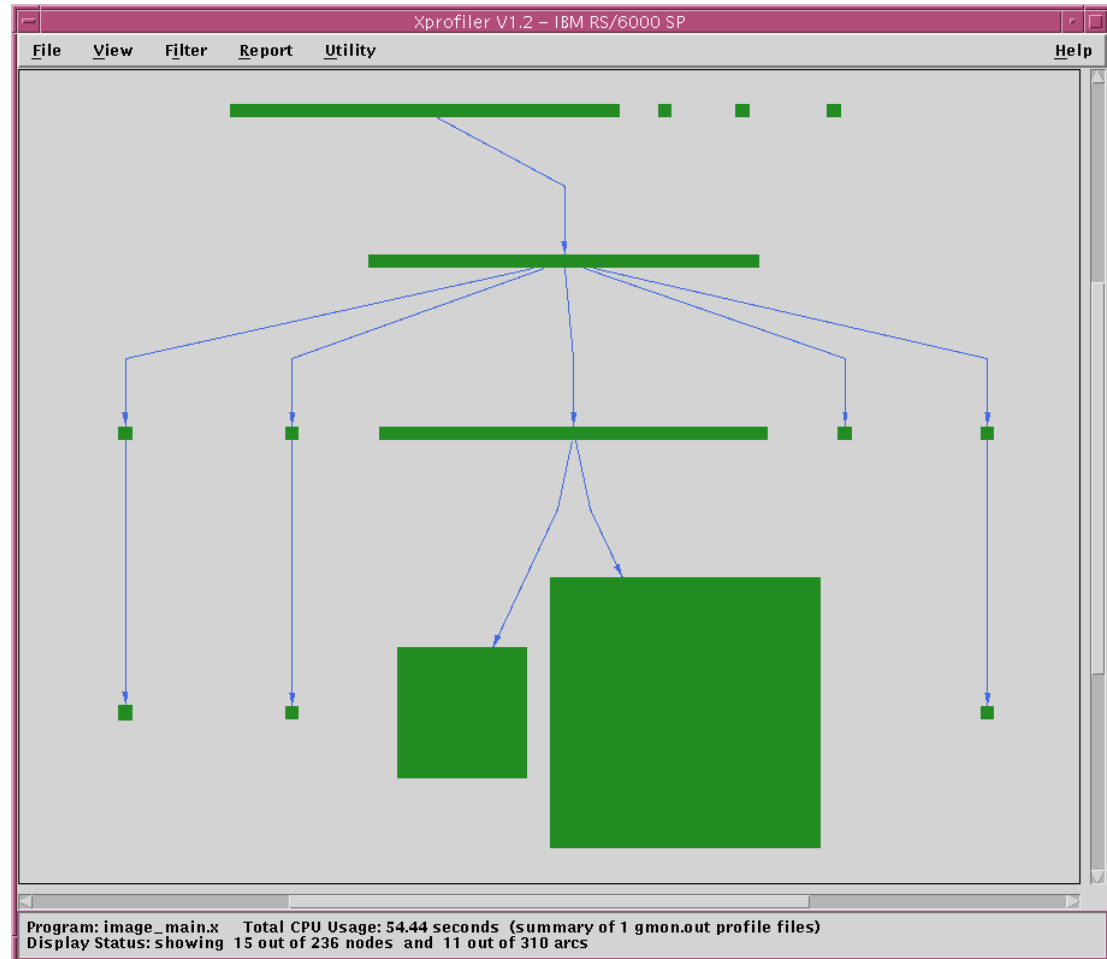


- After hiding the library calls, you are left with a cluster box, containing your executable
- To un-cluster:
 - **Filter**
 - **Uncluster Functions**

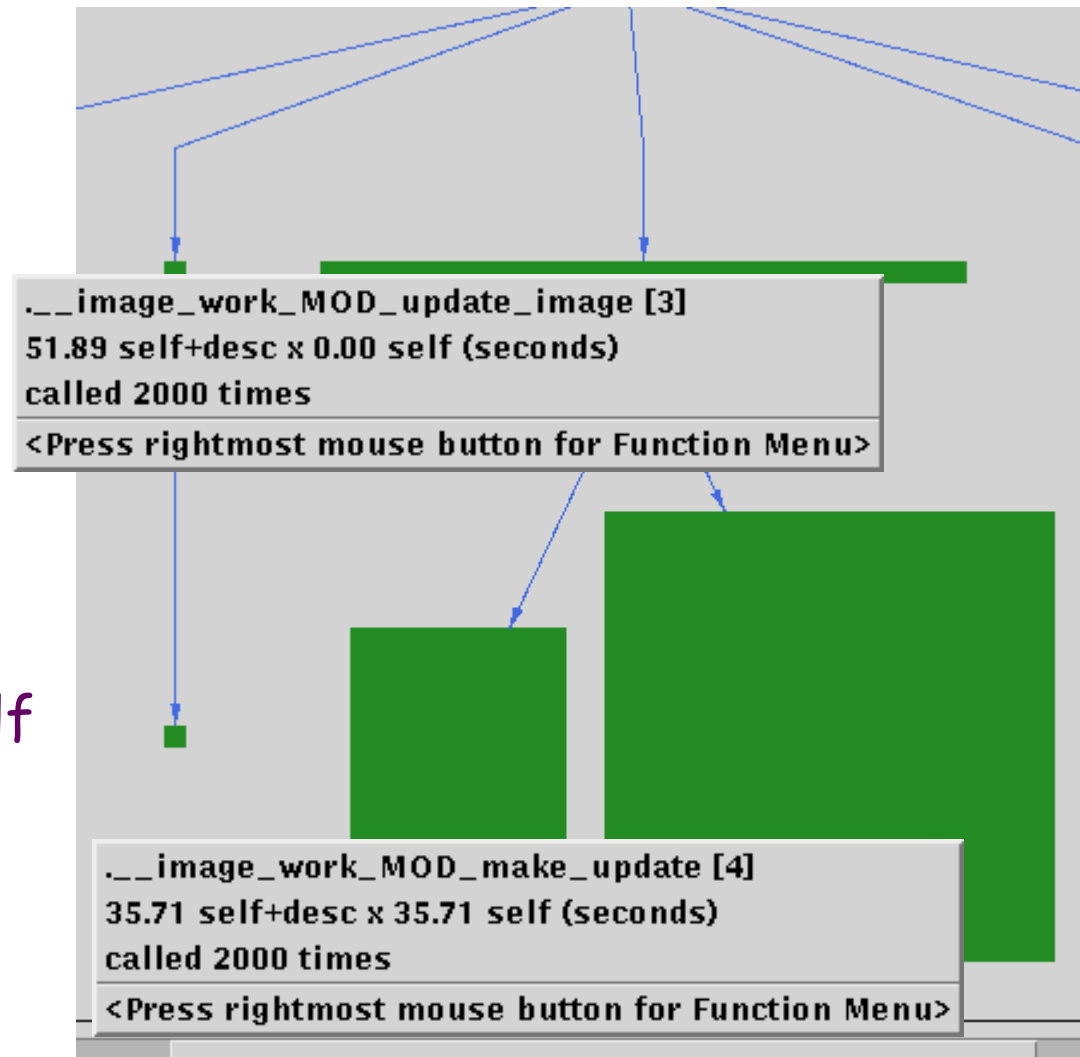


Call tree of your application

- Functions and subroutines are represented as green "function boxes"
 - Width: time spend in function and daughters
 - Height: time spend in function only
- Calls are represented by blue "call arcs"



- A left-click on a function box reveals the information box
- The information box details:
 - Name of function or subroutine
 - Its index
 - Time spent in itself and daughters
 - Time spend in itself
 - Number of invocations

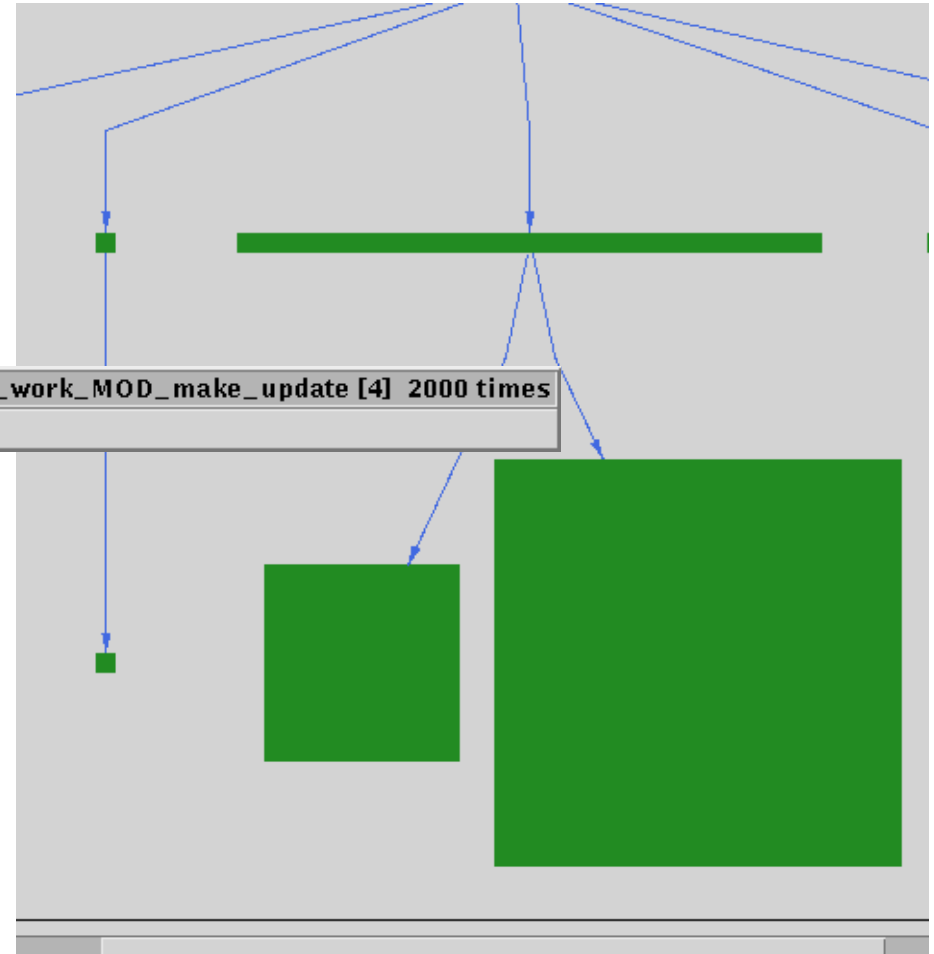


Getting details of a call arc

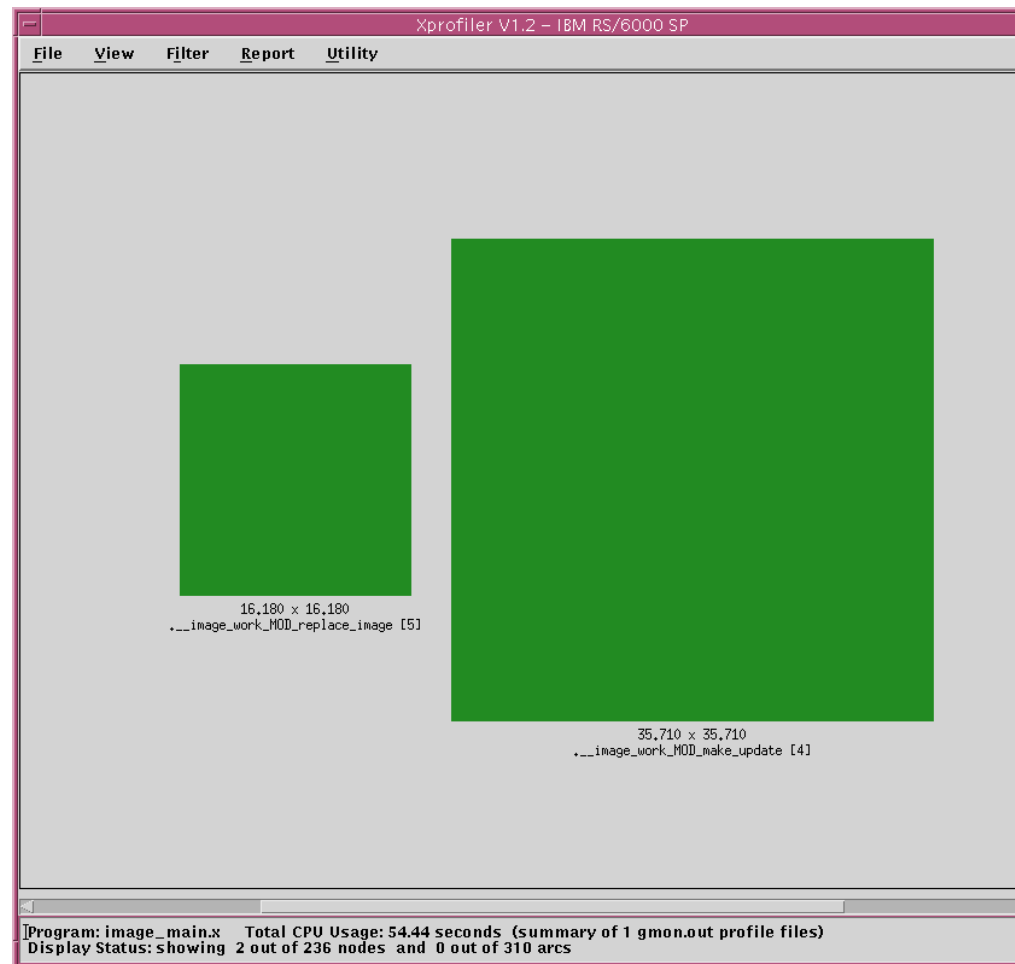
- A left-click on a call arc reveals its information box

```
._image_work_MOD_update_image [3] called ._image_work_MOD_make_update [4] 2000 times  
<Press rightmost mouse button for Arc Menu>
```

- The information box details:
 - Name of caller
 - Name of the callee
 - Their indices
 - Number of times the caller called the callee

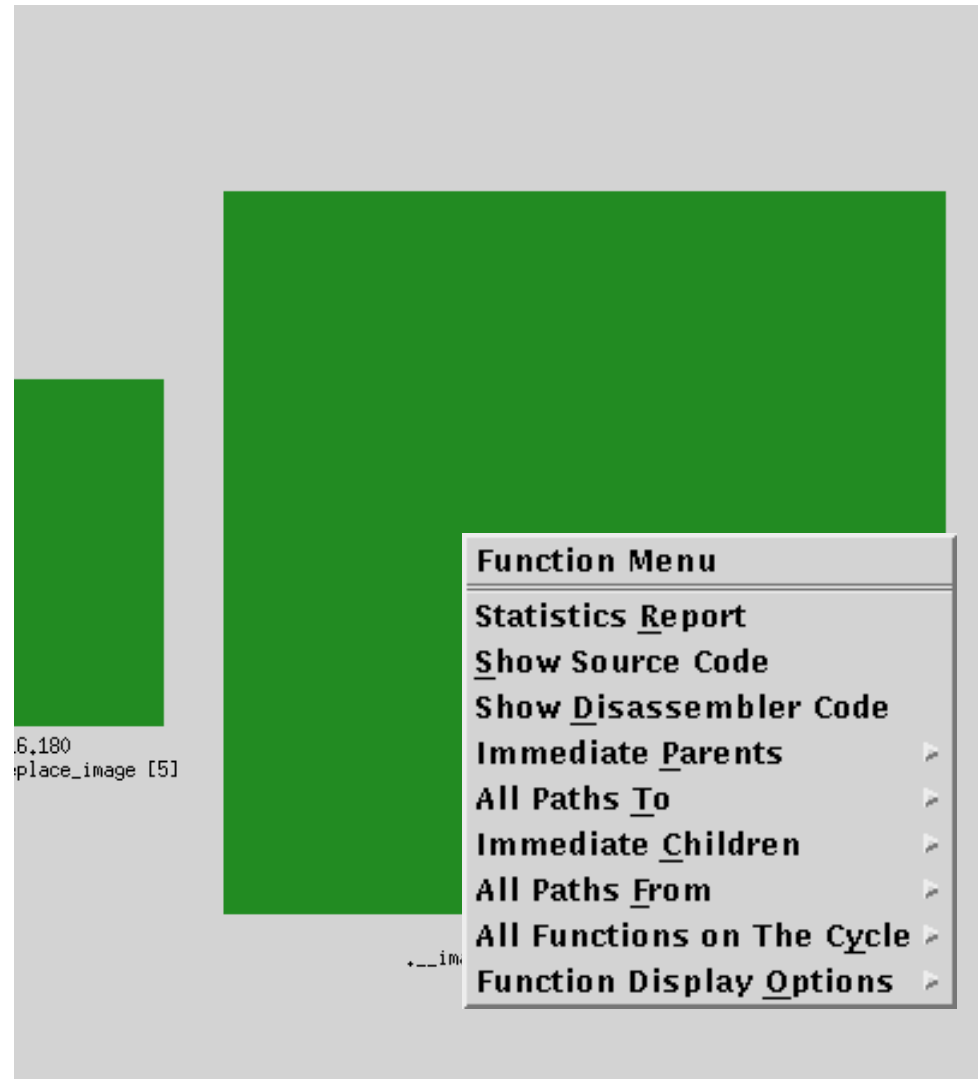


- To profile complex applications, it is advisable to start with the most time consuming functions
 - Filter
 - Filter by CPU Time ...
 - Select the number of functions you want to see
- The example picture is when selecting the two high scorers from the previous example

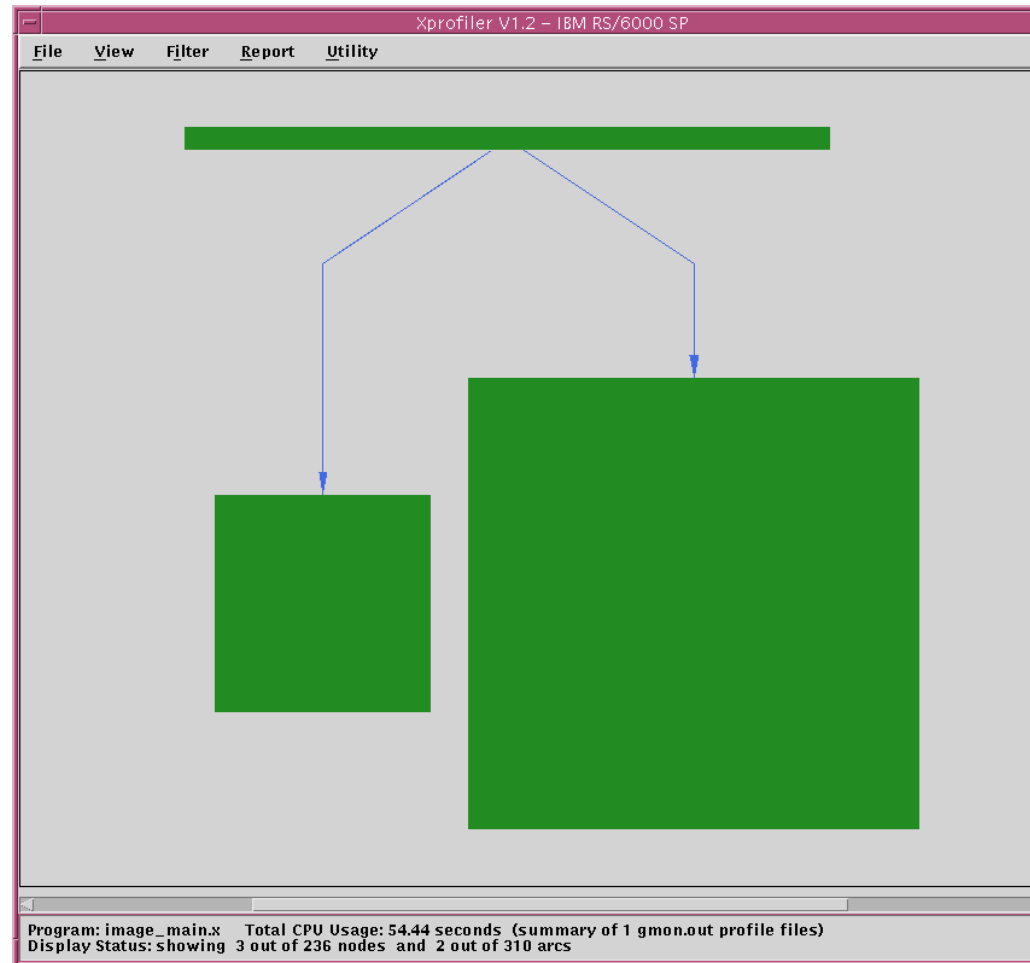


Getting the function menu

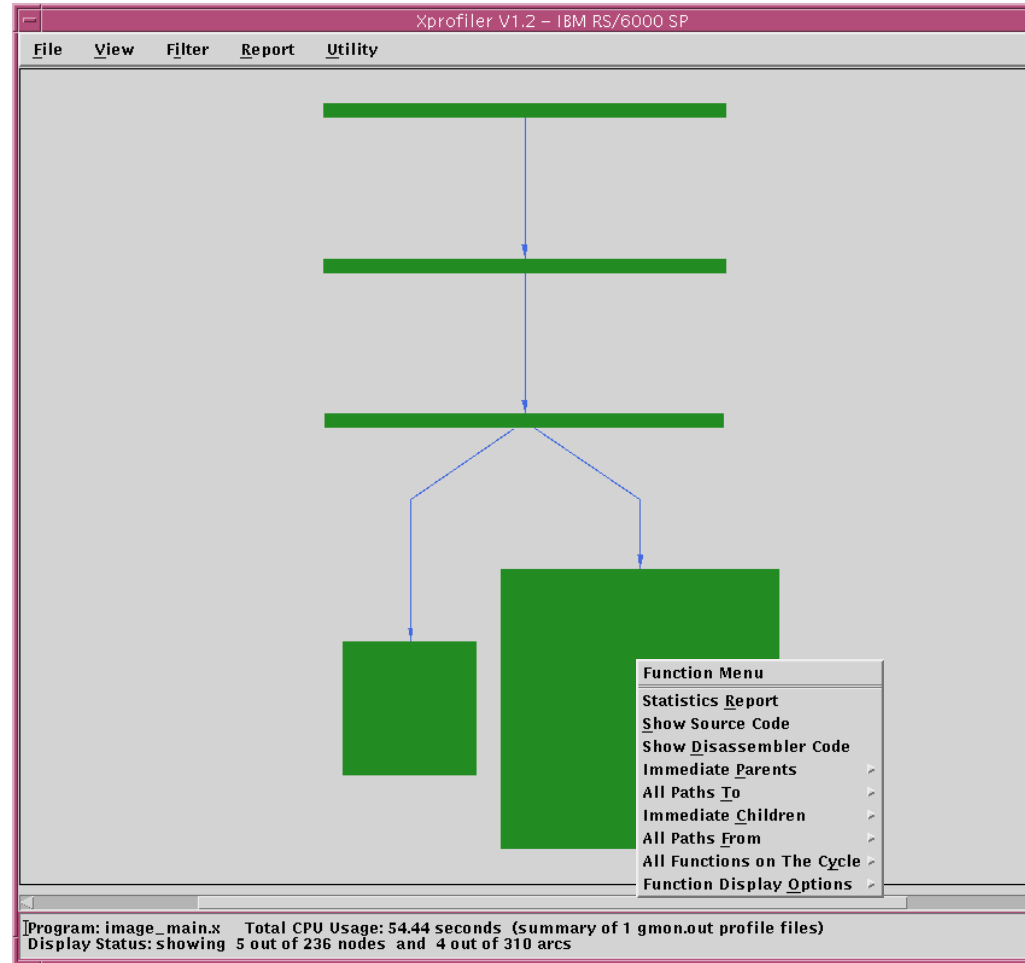
- Using the function menu we can build up a call tree
- A click with the right mouse button on the green function box brings up the "function menu"
- To add the calling routine to the tree choose:
 - Immediate Parents



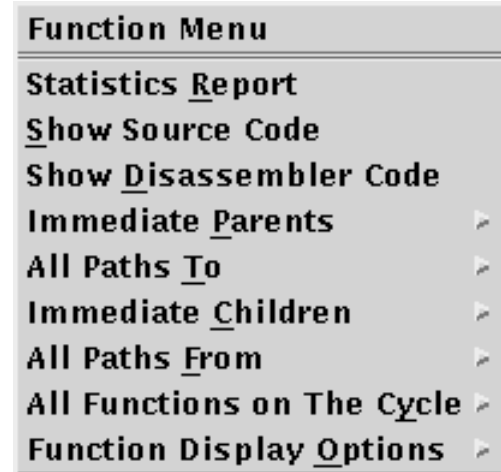
- This will add the calling routine
- In our example the two routines of the previous screen are daughters of the same routine.
xprofiler puts arcs from the caller to both of them



- This process might be iterated to build up to the main routine
- The option "*All Paths To*" from the function menu (right click) build this at once
- If this leads to a busy picture, use "*Undo*" from the Filter menu



- The Function Menu offers
 - Disassembler code profiling
 - Source code profiling
- For source code profiling, application needs to be compiled with the `-g` option
 - This disables `-qinline`, bad for performance of OO-codes (e.g. C++)
- The results of both options have to be read with care
 - It often picks the wrong instruction/line in the vicinity of the hot spot



Disassembler code profiling

Ticks/Instructions

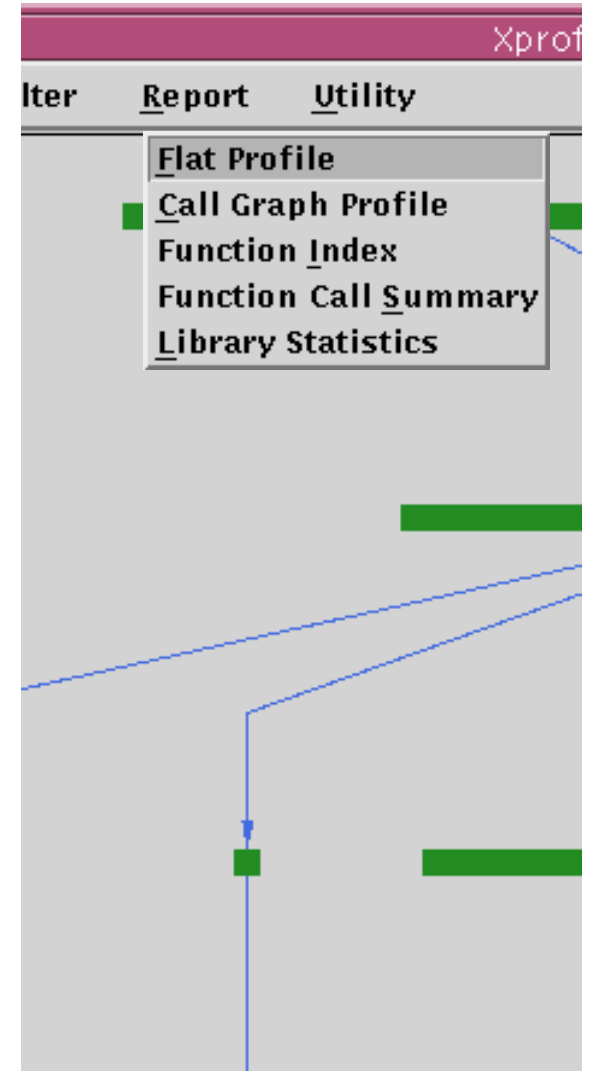
address	no. ticks per instr.	instruction	assembler code	NO LINE TABLE
10002584	1	C1A7FFFC lfs	13,0xfffc(7)	
10002588	3	D3E6FFF0 stfs	31,0xffff0(6)	
1000258C		EFFDE02A fadds	0x1f,0x1d,0x1c	
10002590	257	C3C7CB74 lfs	30,0xcb74(7)	
10002594		EF830032 fmul	0x1c,0x3,0	
10002598		C367E5C0 lfs	27,0xe5c0(7)	
1000259C		C347E5C8 lfs	26,0xe5c8(7)	
100025A0	211	EFA53028 fsubs	0x1d,0x5,0x6	
100025A4		C0680008 lfs	3,0x8(8)	
100025A8		C0A70000 lfs	5,0(7)	
100025AC		ECC1382A fadds	0x6,0x1,0x7	
100025B0	726	ECE2202A fadds	0x7,0x2,0x4	
100025B4		EC99C02A fadds	0x4,0x19,0x18	
100025B8		39080010 cal	8,0x10(8)	
100025BC		C047CB78 lfs	2,0xcb78(7)	
100025C0	225	FC20C090 fmr	1,24	
100025C4		ED080032 fmul	0x8,0x8,0	
100025C8		D126FFF4 stfs	9,0xffff4(6)	
100025CC		C127E5CC lfs	9,0xe5cc(7)	
100025D0	276	38E70010 cal	7,0x10(7)	
100025D4		ED6B6028 fsubs	0xb,0xb,0xc	

Search Engine: (regular expressions supported)

xprofiler ticked fast floating point instructions (e.g. fadds) instead to the slow loads (lfs)

The report menu

- The report menu offers
 - Flat Profile similar to prof
 - Call Graph Profile similar to gprof
 - Function Index
 - Function Call Summary, which lists the routines by their call counts
 - Library Statistics, a flat profile listing the time spend in the different libraries



- Hardware event counters were originally invented to assist the development of the processor hardware
- They proved valuable tools for the performance optimisation of applications
- On HPCx the *Hardware Performance Monitor Toolkit* is available to read the counters
 - Use HPMCOUNT for a global analysis
 - Use LIBHPM to analyse code segments (instrumentation)
 - HPMVIZ is a simple GUI to read the output from LIBHPM

- The individual events of the Toolkit are grouped into 61 sets
- Within a single run only one event set can be investigated
- To enquire about the raw counters available within the event sets use
`/usr/local/packages/actc/hpmtk/pwr4/bin/hpcount -l`
- Event sets may contain derived metrics

- Use event set 60 (default) to count:

Float divide (hardware)	Fused multiply-add	Operation count on FPU	Number of cycles
Float stores	Number of instructions	Float loads	

- The derived metrics include:

Floating point instructions + FMAs (flips)
Flip rates (Wall clock and User time) ***VERY USEFUL***
FMA percentage
Flips/load-store

- Use event set **56** to count:

Data TLB misses	Instruction TLB misses	L1 cache load misses	L1 cache store misses
Number of cycles	Number of instructions	L1 data store references	L1 data load references

- The derived metrics include:

TLB miss rate (cycles and loads)

Instruction rates (loads, cycles, wall clock)

- **Rem:** Due to prefetching and multiple load units on the Power4 chip, the L2 traffic can not be calculated from L1 misses. The derived metrics referring to the L2 will be renamed in future releases

- Nomenclature of cache levels and locations:

Level 2	Level 2 on same chip
Level 2.5	Level 2 on different chip, same MCM
Level 2.75	Level 2 on different MCM, inaccessible (phase 1)
Level 3	Level 3 on same MCM
Level 3.5	Level 3 on different MCM, inaccessible (phase 1)

- Rem: For parallel codes, if you use shared memory (e.g. OpenMP) or shared memory MPI inside the logical partition, you will note level 2.5 traffic.

- Use event set 5 to count:

Loads from memory	Loads from L3	Loads from L3.5	Loads from L2
Loads fr. L2.5 'read only'	Loads fr. L2.5 'exclusive'	Loads L2.75 'read only'	Loads L2.75 'exclusive'

- The derived metrics include:

Total loads from L2 and L3

Traffic, bandwidth and miss rates for L2, L3 and Memory

- Rem: Interpretation needs care, see streams example below

- **HPMCOUNT** investigates the performance of the entire application
 - Pros:
 - Very easy to use
 - No need to modify the source
 - Cons:
 - No information on which part of the code performs poorly
 - Several rates are calculated with respect to Wallclock time. Due to overheads these rates can be severely distorted when running short test jobs
- **Executable:**
`/usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount`

- For code compiled for serial execution (xlf, xlc)
`<path>/hpmcount -g <set> my_prog.x`
- For code compiled for MPI (e.g. mpxlf, mpxlc)
`poe <path>/hpmcount -g <set> my_prog.x`
- Pitfall: `poe` and the order of `poe` and `<path>/hpmcount` is crucial. Otherwise HPMCOUNT examines the performance of `poe`.

- Getting help
 - h
- Specifying an output file (default standard out)
 - o *<outfile>*
- Specifying the event set (default set 60)
 - g *<set>*
- Listing the available event sets
 - l

- Use LIBHPM to investigate the performance of individual code segments
- Can exclude the effect from input, output and initialisation routines, allowing for reliable results from short test runs
- This requires instrumentation of the codes with a few simple routines

- Every file containing any of the LIBHPM instrumentation calls needs to contain a header file:

- FORTRAN

```
#include "f_hpm.h"
```

- All FORTRAN instrumentation carries an "f_" prefix
- LIBHPM is not standard FORTRAN and needs a preprocessor
- Some of the commands are case sensitive!

- C

```
#include "libhpm.h"
```

- Initialise the tool with

```
f_hpminit(taskid, name)
```

```
hpmInit(taskid, name)
```

- **taskid**: identifies the MPI-task (put 0 for serial)
- **name**: will become part of the *.viz file

- Close the tool with

```
f_hpmterminate(taskid)
```

```
hpmTerminate(taskid)
```

- **taskid**: put same as for the hpmInit call

- Forgetting hpmTerminate: no output file

- Start the counters with
 - `f_hpmstart(instid, label)`
 - `hpmStart(instid, label)`
 - `instid`: unique number identifying the section,
 $0 \leq \text{instid} \leq 100$ (default)
 - `label`: string
- Stop the counters with
 - `f_hpmstop(instid)`
 - `hpmStop(instid)`
 - `instid`: unique number identifying the section,
has to match the argument of `hpmStart`
- Use `hpmTstart` and `hpmTstop` in threaded regions

- Measured sections can be nested
- Both calls hpmstart & hpmstop take about 10 μ s
 - Instrumented regions have to be longer than several 100 μ s for the rates quoted among the derived metrics to become reliable

```
#include <stdio.h>
#include <stdlib.h>

#include "libhpm.h"

main(int argc, char *argv[]){
    hpmInit(0, "hpm_hello");
    hpmStart(1, "print_count");

    printf("Hello world!\n");

    hpmStop(1);
    hpmTerminate(0);
}
```

- Example measures the performance of printf
- The measured region gets an identifier 1 and the label "print_count"
- The name of *.viz file will include the string "hpm_hello"

- To invoke the preprocessor on HPCx use the `-qsuffix` option of `xlf`

```
-qsuffix=cpp=f
```

```
-qsuffix=cpp=f90
```

- Example: Compiling FORTRAN90

```
xlf90_r -qsuffix=cpp=f90 \  
-o my_prog.x my_prog.f90 \  
-I/usr/local/packages/actc/hpmtk/include \  
-L/usr/local/packages/actc/hpmtk/pwr4/lib \  
-lhpm -lpmpi
```

- For OpenMP replace `-lhpm` with `-lhpm_r`, add `-qsmp=omp`

- **Compiling C**

```
xlc_r -o my_prog.x my_prog.c \  
-I/usr/local/packages/actc/hpmtk/include \  
-L/usr/local/packages/actc/hpmtk/pwr4/lib \  
-lhpm -lpmpi -lm
```

- For OpenMP replace `-lhpm` with `-lhpm_r` and add `-qsmp=omp`

- Use `HPM_EVENT_SET` to select the event set:

```
export HPM_EVENT_SET=5
```

- `HPM_OUTPUT_NAME` changes the name of the output files. It overwrites the argument of `hpmInit`. To get output files which include the event set specify the following in you ll-script:

```
export HPM_EVENT_SET=5
```

```
export HPM_OUTPUT_NAME=hpm_set${HPM_EVENT_SET}
```

- LIBHPM generates two output files per MPI-task
 1. *.hpm
 2. *.viz
- Both files contain the same data
- The *.hpm file is a plain ASCII file. Investigate with UNIX tools such as `more`, `grep`, `awk`
- Use `HPMVIZ` to read the *.viz file

Example for an *.hpm file

```
joachim@bronzite:~  
Window Edit Options Help  
Instrumented section: 1 - Label: copy - process: 0  
file: stream_d.f, lines: 182 <--> 192  
Count: 10  
Wall Clock Time: 6.246742 seconds  
Average duration: 0.624674  
Standard deviation: 0.0097997  
  
PM_DATA_FROM_L3 (Data loaded from L3) : 1243741  
PM_DATA_FROM_MEM (Data loaded from memory) : 5478966  
PM_DATA_FROM_L35 (Data loaded from L3.5) : 0  
PM_DATA_FROM_L2 (Data loaded from L2) : 37136612  
PM_DATA_FROM_L25_SHR (Data loaded from L2.5 shared) : 8  
PM_DATA_FROM_L275_SHR (Data loaded from L2.75 shared) : 0  
PM_DATA_FROM_L275_MOD (Data loaded from L2.75 modified) : 0  
PM_DATA_FROM_L25_MOD (Data loaded from L2.5 modified) : 2  
  
Total loads from L2 : 37.137 M  
L2 load traffic : 4753.488 MBytes  
L2 load bandwidth : 760.955 MBytes/sec  
L2 Load miss rate : 15.328 %  
Total loads from L3 : 1.244 M  
L3 load traffic : 159.199 MBytes  
L3 load bandwidth : 25.485 MBytes/sec  
L3 Load miss rate : 81.499 %  
Memory load traffic : 2805.231 MBytes  
Memory load bandwidth : 449.071 MBytes/sec  
hpm_stream_0_set5_0000.hpm (36%)
```

- HPMVIZ is a simple visualisation tools to investigate the output from LIBHPM
- Starting HPMVIZ
`/usr/local/packages/actc/hpmtk/pwr4/hpmviz &`
- You may list the `*.viz` file(s) before the `&`

The screenshot shows the hpmviz application window. On the left, a table titled 'stream_bench' lists instrumented sections. The 'copy' section is highlighted. On the right, a source code display shows the Fortran code for the 'copy' section, including timing and parallelization directives.

Label	Count	ExcSec	IncSec
add	10	7.44	7.44
copy	10	4.609	4.609
scale	10	4.749	4.749
triad	10	7.439	7.439

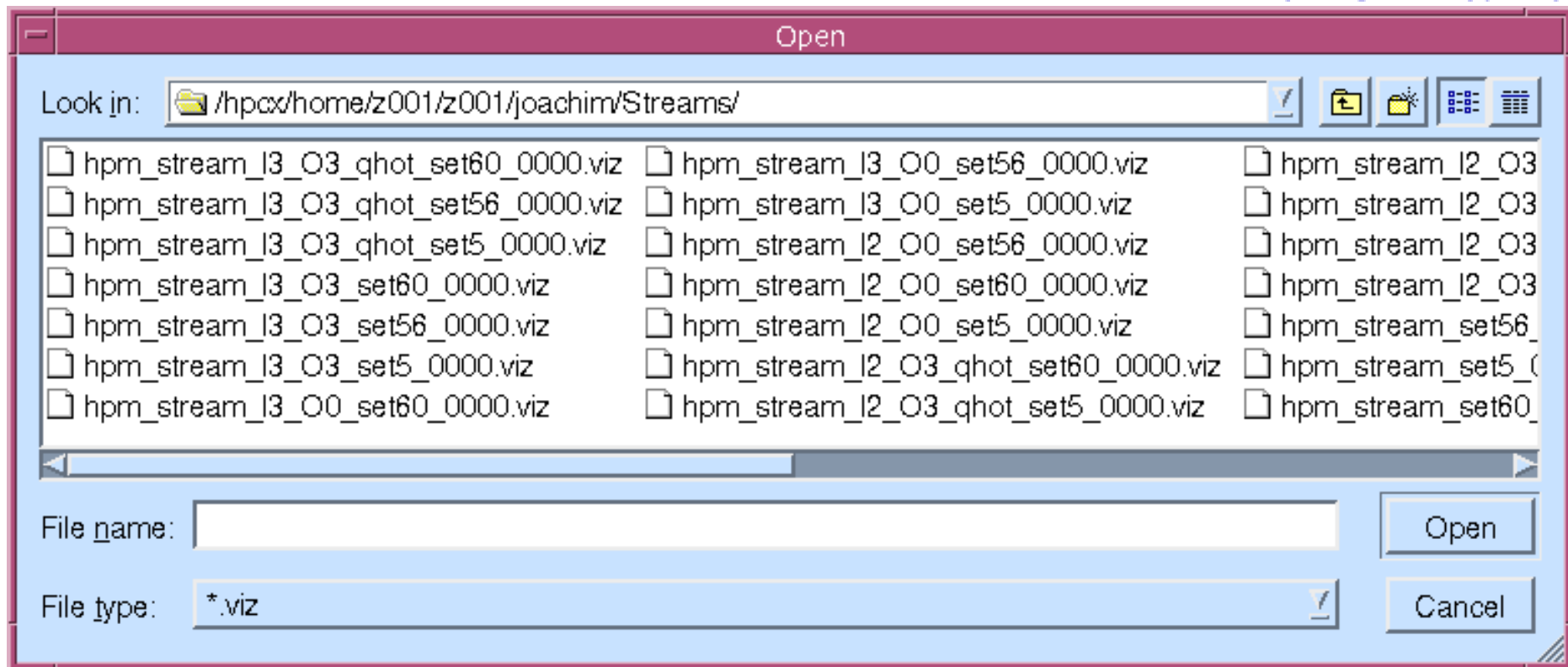
```
Call f_hpmstart(1,"copy")
t = mysecond()
a(1) = a(1) + t
!$OMP PARALLEL DO
DO 30 j = 1,n
  c(j) = a(j)
30 CONTINUE
t = mysecond() - t
c(n) = c(n) + t
times(1,k) = t
Call f_hpmstop(1)

Call f_hpmstart(2,"scale")
t = mysecond()
c(1) = c(1) + t
!$OMP PARALLEL DO
DO 40 j = 1,n
  b(j) = scalar*c(j)
40 CONTINUE
t = mysecond() - t
b(n) = b(n) + t
```

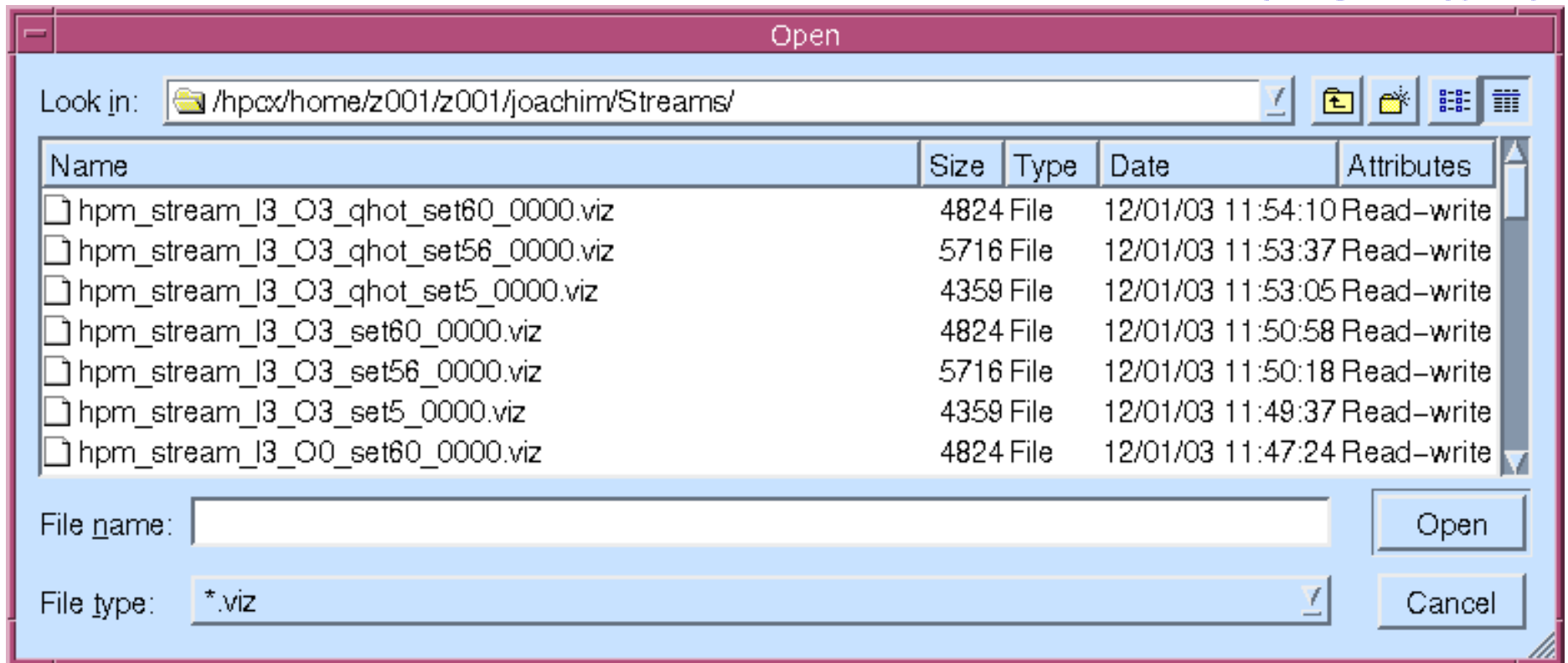
List of instrumented sections

- Left click for source code
- Right click for metrics window

Source code display



- Use the file window for loading *.viz files



- Click top right hand corner to get detailed view
- Allows sorting with respect to date (very useful with parallel jobs)

Metric Browser: copy

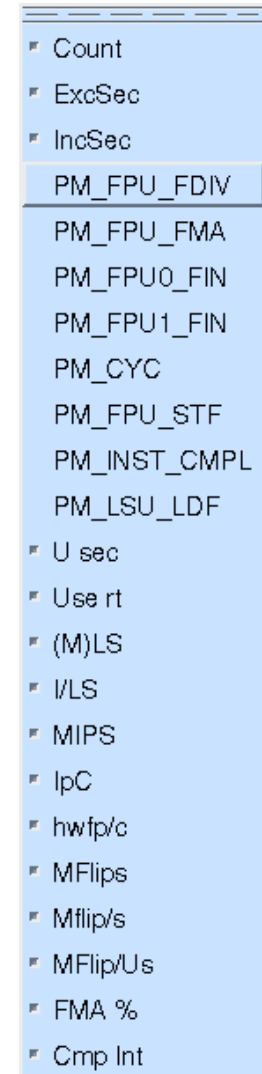
Node	Thread	Count	ExcSec	IncSec	U sec	Use rt	(M)LS	I/LS	MIPS	IpC	hwfp/c	MFlips	Mflip/s	MFlip/Us	FMA %	Cmp Int
0	0	10	4.609	4.609	4.597	99.755	1452.41	1.084	341.752	0.264	0.123	0	0	0	0	0

Good result

Poor results

- Right click on section in main window
- Set precision (double or single)
- Highlights results it thinks are good or poor

- By default, the metrics window of HPMVIZ shows the derived metrics of the group only
- Use the Metrics Options pull down to activate raw counters and deactivate derived metrics as needed



Case study: Streams benchmark copy

Array (doubles):	17E3 (L2)	70E6 (Mem)
Stores (set 60):	17E3	70E6
Loads(set 60):	17E3	70E6
L1-Store (set 56):	17E3	70E6
L1-Load (set 56):	17E3	70E6
L1-S-miss (set 56):	17E3	70E6
L1-L-miss (set 56):	0.15E3	0.66E6
Loads from L2 (5):	1.1E3	3.5E6
Loads from L3 (5):	0	130
Loads from Mem (5):	0	0.9E6

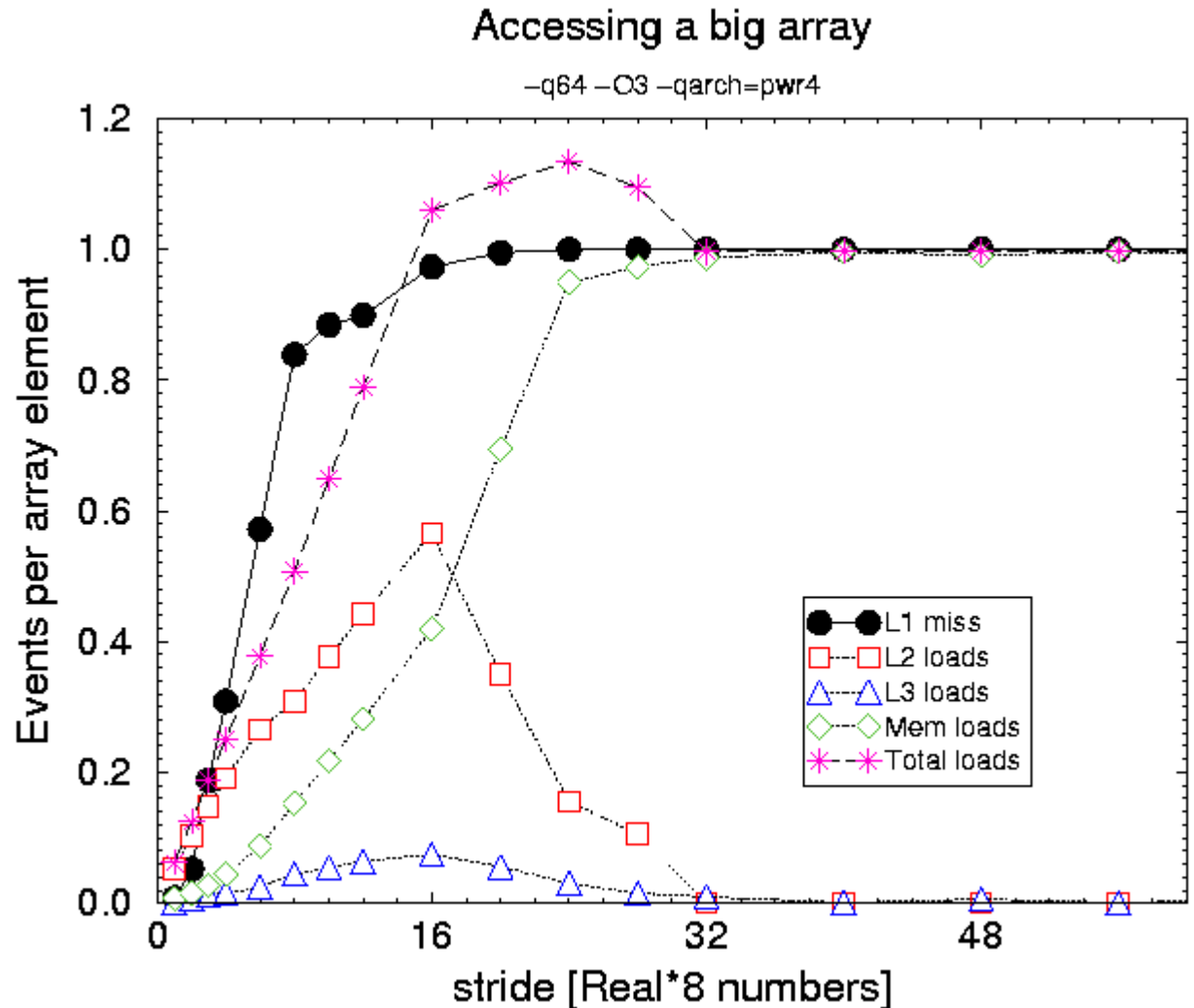
Loads from L1:
16 × Loads (L2 + L3 + Mem)

Fitting into L2: no loads from L3 and Memory

Code fitting on Memory: most loads still from L2 (prefetching)

2nd Case Study: Sum of an array

- No simple relation between L1 misses and loads
 - Prefetching
 - Two LSU
- Stride larger than two cache lines: all loads triggered by L1 misses and data loaded from memory



- **General**

- User Guide to the HPCx Service

www.hpcx.ac.uk/support/introduction

- **xprofiler**

- IBM Parallel Environment for AIX
Operation and use, Volume 2

www.hpcx.ac.uk/support/documentation/IBMdocuments/a2274261.pdf

- **HPM Toolkit**

- Hardware Performance Monitor (HPM) Toolkit

www.hpcx.ac.uk/support/documentation/IBMdocuments/HPM.html

- Using the Hardware Performance Monitor Toolkit on HPCx

www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0307_choose.html