



# Using the Hardware Performance Monitor Toolkit on HPCx

Joachim Hein

EPCC

The University of Edinburgh

Edinburgh EH9 3 JZ

Scotland, UK

November 3, 2003

## 1 The toolkit

The IBM Power4 processors used in the HPCx system provide a number of event counters. The *Hardware Performance Monitor Toolkit* reads these counters to obtain information on performance bottlenecks in your application code. The toolkit was developed by Luiz DeRose of the Advanced Computing Technology Center at IBM Research.

We provide a detailed description of the parts of the toolkit we believe are relevant to users of the HPCx system. The descriptions are tailored to the HPCx system, including the path names in order to make using the toolkit as easy as possible. We include various hints on the difficulties we faced when we started using this tool. The document has four main sections.

To read the counters for the entire application one can use the tool `HPMCOUNT` as described in section 2. `HPMCOUNT` is very easy to set up and does not require any modification of the source code. Its usage is even possible if you have access to an executable only. The disadvantages of using a global tool are also obvious. The output does not provide any hint on which part of the application contains the bottlenecks and all the setup and initialisation routines are included in the measurements.

To overcome these problems the code can be instrumented using `LIBHPM`, which is discussed in section 3. Obviously this requires access to the source to include the required calls. With `LIBHPM` it is possible to measure the performance of individual code segments.

In the following section 4, which is relevant to `HPMCOUNT` and `LIBHPM`, we give a comprehensive description of the different counters and metrics. This commented listing of the different numbers and their meaning was one of our key motivations for writing this document.

In the last section we discuss the visualisation tool `HPMVIZ`, which is useful to analyse the output of `LIBHPM`.

This document describes version **2.5.2** of the Hardware Performance Monitor Toolkit.

## 2 Global analysis with `HPMCOUNT`

`HPMCOUNT` investigates the performance of an entire application. However, several of the rates listed among the derived metrics make use of the wall-clock time. A typical run used for the

performance analysis of an application contains very sizable overheads. The initialisation of the application and the exchange of initial and final data with the mass storage are examples for the sources of these overheads. Such overheads can distort the results for these rates considerably when comparing short test jobs to proper production run. Despite these limitations, we feel HPMCOUNT is a very useful tool, particularly since it is so simple to use.

## 2.1 Running HPMCOUNT

The executable of HPMCOUNT is can be found

```
/usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount
```

### 2.1.1 Investigating serial code

To investigate the performance of a serial code named `serial_prog.x` on HPCx the LoadLeveler script should contain the line

```
/usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount serial_prog.x
```

This will analyse the performance of your program with HPMCOUNT's default settings and write the results of HPMCOUNT to the standard output.

**Pitfall:** If serial code has been compiled using the `mp-`prefix, e.g. `mpx1f` instead of `x1f` in the case of a Fortran-code or `mpx1c` instead of `x1c` in the case of a C-code, the above statement will automatically invoke `poe` in a way that HPMCOUNT will investigate the performance of `poe`. There are two solutions to this

- Recompile without the `mp-`prefix
- Run as a single-processor parallel job by calling

```
poe /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount serial_prog.x
```

### 2.1.2 Investigating parallel code

When investigating the performance of a parallel code named `parallel_prog.x`, the line

```
poe parallel_prog.x
```

in the original LoadLeveler script should be replaced by the line

```
poe /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount parallel_prog.x
```

**Important:** The ordering of the calls to `poe` and `hpmcount` is crucial. Having them in the wrong order or missing `poe` altogether will lead to HPMCOUNT examining the performance of `poe` instead of your application.

## 2.2 HPMCOUNT and its options

The behaviour of HPMCOUNT can be controlled by a number of command-line options. These have to be specified between the call to HPMCOUNT and the executable.

We describe ere the options which seem to us to be relevant to users of the HPCx system.

### 2.2.1 Getting help, option `-h`

Use

```
/usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -h
```

to get a short summary on the available command-line options.

### 2.2.2 Specifying an output file, option `-o`

With

```
poe /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -o outfile parallel_prog.x
```

HPMCOUNT will write the counter results to a set of files starting with the string specified as argument for the `-o` option, which is `outfile` in the above example. Without the option `-o` the results are written to the standard output. For parallel jobs, if `-o` is specified, there will be separate file for each processor.

Even with the `-o` option HPMCOUNT will write some information to `stdout`. This can be changed with the `-n` option.

### 2.2.3 No output to `stdout`, option `-n`

With `-n`

```
poe /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -o outfile -n parallel_prog.x
```

no output will be written to the standard output, if an output file is specified with `-o`.

### 2.2.4 Changing the event set, option `-g`

For a power 4 system such as HPCx the event counters are grouped into 61 different event sets. The `-g` option allows you to select the event set. The example below selects event set number 5

```
poe /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -g 5 parallel_prog.x
```

The `-g` takes a single integer between 0 and 60 as an argument. The default event set is 60, which will be used if `-g` is not specified. Luiz DeRose considers the event sets 5, 53, 58, 59 and 60 as particularly useful. We describe these five sets in section 4 in great detail. Use HPMCOUNT's `-l` option to enquire about the remaining event sets.

### 2.2.5 Listing the available event sets, option `-l`

```
/usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -l
```

will list the raw counters available for each of the 61 event sets.

### 2.2.6 Include system activity, option `-k`

To include the system activity on behalf of your process, which is not normally included, use the `-k` option

```
poe /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -k parallel_prog.x
```

This option is a recent addition, which was first introduced in version 2.5.1 of HPM toolkit.

## 2.3 Useful hints on using HPMCOUNT

### 2.3.1 Output

By default HPMCOUNT writes to `stdout`, which can be changed with the `-o` option of HPM-COUNT. For a parallel job writing to `stdout`, we recommend setting `MP_STDOUTMODE=ordered` to prevent the processors from writing interleaved. This has to be set in the LoadLeveler script. The following example is for `ksh`:

```
export MP_STDOUTMODE=ordered
```

When using the `-o` option, each processor writes its own output and setting `MP_STDOUTMODE=ordered` is not required.

## 3 Investigating code segments with LIBHPM

Instrumenting your code with LIBHPM allows you to target individual code segments. Instrumentation also allows you to exclude overheads from input, output and initialisation routines from the analysis easily. This way it is possible to obtain more reliable results from quite short test jobs.

### 3.1 Instrumentation routines

The LIBHPM consists of a set of routines which are used to instrument your code. Note the instrumentation routines for Fortran have an “f\_” prefix on their names. Since using LIBHPM for Fortran requires preprocessing, some of the instrumentation names appear to be case sensitive.

Care must be taken with respect to placing instrumentation in the innermost loops. Obviously instrumentation in the innermost loops will lead to sizable overheads.

We will explain the commands we believe are most relevant to users of HPCx. For further information check the original documentation by Luiz DeRose [1].

#### 3.1.1 Header files

- **Header files for Fortran:** Every Fortran file containing any of the instrumentation routines below has to include the line:

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

This is not Fortran and needs to be interpreted by the preprocessor. See subsection 3.3 for details on how to invoke the preprocessor.

- **Header files for C:** Every C file containing any of the instrumentation routines below has to use the line

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

which is standard C. No preprocessor is needed unless your code does need it anyway.

#### 3.1.2 Initialising the tool: hpmInit

Initialise the toolkit with the subroutine `hpmInit`

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

- `taskid` has to be different for each MPI task.
  - For a **serial code**, a zero can be entered for `taskid`.
  - In a **pure MPI code** it is recommended to set `taskid` to the processor rank in the communicator `MPI_comm_world`.
  - For a **pure OpenMP code** the command `hpmInit` or `f_hpminit` has to be placed outside a parallel region. As in the serial case we recommend entering a zero for `taskid`. If the counters are started and stopped inside a parallel region, the commands `hpmTstart` and `hpmTstop` have to be used, to which the thread identification can be passed.
  - For a **mixed mode code using MPI and OpenMP**, again, the command `hpmInit` or `f_hpminit` has to be placed outside a parallel region. As in the MPI case we recommend setting `taskid` to the processor rank in the communicator `MPI_comm_world`. If the counters are started and stopped inside a parallel region, the commands `hpmTstart` and `hpmTstop` have to be used, to whom the thread identification can be passed.
- `name` will become part of the name of the `*.viz` file for the HPMVIZ visualisation tool.

### 3.1.3 Closing call: `hpmTerminate`

This subroutine will finalise the instrumentation and write the output files. If you omit this call, no output will be generated.

```
f_hpmterminate(taskid)
hpmTerminate(taskid)
```

The argument `taskid` has to match the `taskid` of `hpmInit` in 3.1.2. For a threaded code, `hpmTerminate` has to be placed outside a parallel region.

### 3.1.4 Starting the counters in a non-threaded region: `hpmStart`

To start the counters outside a threaded region, use the command:

```
f_hpmstart(instid, label)
hpmStart(instid, label)
```

- `instid` has to be set to a unique number, identifying each instrumented section. `instid` has to be positive and can be as large as 100. However the maximum possible value of `instid` can be changed with the environment variable `HPM_NUM_INST_PTS`.
- `label` has to be set to a string. The label will be included into the output files to assist in matching the results to the correct code section.

Instrumented sections can be nested. An instrumented section can be entered several times during code execution. This will restart the counting. Placing instrumentation inside a loop is a typical example for restarting. If a set of counters is restarted the counters will be added to. See section 3.2.2 for an example code with nested and restarted instrumented sections.

### 3.1.5 Stopping the counters in a non-threaded region: `hpmStop`

To stop the counter inside a non-threaded region use:

```
f_hpmstop(instid)
hpmStop(instid)
```

- `instid` has to match the first argument of `hpmStart`.

### 3.1.6 Starting the counters in a threaded region: `hpmTstart`

To start the counters inside a parallel region, use the command

```
f_hpmtstart(instid, label)
hpmTstart(instid, label)
```

- `instid` identifies the counter and has to be larger than 0. The default for its maximum value is 100, which can be changed with the environment variable `HPM_NUM_INST_PTS`. If `instid` is set to the same value for all threads, the results will contain the sum of the counters for all threads. If they are set differently for each thread, you will get separate results for each thread. Different values of `instid` for each thread can be achieved with the OpenMP enquiry function `OMP_GET_THREAD_NUM()`. See section 3.2.3 for an example code.
- `label` is a string which is passed to the output files. It helps to match the results to the instrumented sections they originate from.

### 3.1.7 Stopping the counters in a threaded region: `hpmTstop`

To stop the counters inside a parallel region, use:

```
f_hpmtstop(instid)
hpmTstop(instid)
```

- `instid` has to match the first argument of the corresponding `hpmTstart`.

## 3.2 Examples for instrumented code

### 3.2.1 A simple serial code in C

The following is a C version of the famous “*Hello World*” code. It has been instrumented to investigate the performance of the `printf` statement.

```
#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);

}
```

### 3.2.2 Example using MPI (Fortran)

The following is an example for a code using MPI. This code runs successfully on HPCx. The variable `me_world` has been set to the rank of the task in the MPI communicator `MPI_comm_world`. This is done inside the subroutine `make_mpi_world()`.

We instrumented three regions inside the code. The first region, with the label “`kern`”, measures the performance of the entire kernel of the code. The second region, labeled “`communication`”, measures the performance of the routine “`Halo_swap`”. Finally the third region, labeled “`working`”, measures the performance of the routine “`Update_image()`”. The second and third region are placed inside an iteration loop. For each iteration, their counters will be increased appropriately once they are restarted. Please note that the regions “`communication`” and “`working`” are nested inside the region “`kern`”.

```
program image_main

    use image_param
    use image_mpi_comm
    use image_work
    use image_time
    use time_tool
```

```

implicit none

#include "f_hpm.h"

real(kind=8) :: a_time, b_time, c_time
real(kind=8) :: s_time

integer iter

Call make_mpi_world()
Call make_cart_world()
Call edge_init('r')

! make sure we start together
Call sync_this(cart_comm)

! set times to 0
call zero_timers()

call f_hpminit( me_world, "case_image_normal_nohott" )

call f_hpmstart(1, "kern" )

s_time = now_time()
iterloop: do iter = 1, Nb_iter

    call f_hpmstart( 2, "communication" )
    a_time = now_time()

    Call Halo_swap(image)

    call f_hpmstop( 2 )

    call f_hpmstart( 3, "working" )
    b_time = now_time()

    Call Update_image()

    c_time = now_time()

    call f_hpmstop( 3 )
    halo_time = halo_time + b_time - a_time
    update_time = update_time + c_time - b_time

enddo iterloop

call f_hpmstop(1)
elapsed_time = c_time - s_time

Call Print_times()

call f_hpmterminate( me_world )

Call shut_mpi_world()

```

end

### 3.2.3 Example using OpenMP (FORTRAN)

The following is an example of a simple code using OpenMP. The LIBHPM has been initialised and terminated outside the parallel region, using the `f_hpminit` and `f_hpmterminate` calls. The counters are started and stopped inside the parallel region using `f_hpmtstart` and `f_hpmtstop`. The first argument of both calls has been set differently for each thread, using the OpenMP enquiry function `OMP_GET_THREAD_NUM()`.

```
program OMP_PRINT

  implicit none

  #include "f_hpm.h"

  integer :: myid, nthreads
  integer :: OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM

  call f_hpminit(0,"hpmfile")

  !$OMP PARALLEL default(none) private(myid) &
  !$OMP shared(nthreads)

  ! Determine the number of threads and their id
  myid = OMP_GET_THREAD_NUM()
  nthreads = OMP_GET_NUM_THREADS();

  !$OMP BARRIER

  call f_hpmtstart(1+myid,"printcounter")

  print*, 'myid = ', myid, ' nthreads ', nthreads

  call f_hpmtstop(1+myid)

  !$OMP END PARALLEL

  call f_hpmterminate(0)

end program OMP_PRINT
```

## 3.3 Compiling instrumented Fortran

As explained in subsection 3.1.1 all files containing instrumentation need to contain the statement

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

which is not standard Fortran and needs to be interpreted by the preprocessor. On HPCx the preprocessor can be invoked easily with the `-qsuffix` option of the `xlf` Fortran compiler

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

depending on whether the file is called `filename.f` or `filename.f90`.

To compile the instrumented code `my_prog.f90` on HPCx and link it against the HPM toolkit use

```
mpxlf90_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 -lpmapi
```

For a code `old_prog.f` this becomes:

```
mpxlf_r -qsuffix=cpp=f -o old_prog.x old_prog.f \  
-I/usr/local/packages/actc/hpmtk/include \  
-L/usr/local/packages/actc/hpmtk/pwr4/lib -lhpm -lpmapi
```

For an code using OpenMP, the `-lhpm` has to be replaced by `-lhpm_r` and the option `-qsmp=omp` has to be added.

### 3.4 Compiling instrumented C

To compile instrumented C code, use the statement:

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

For an code using OpenMP, the `-lhpm` has to be replaced by `-lhpm_r` and the option `-qsmp=omp` has to be added.

### 3.5 Controlling the runtime environment for instrumented runs

Various aspects of the behaviour of LIBHPM can be controlled with the help of the following environment variables. On HPCx the environment variables should be specified inside the LoadLeveler script.

#### 3.5.1 HPM\_EVENT\_SET

The individual counters are combined into sets. With `HPM_EVENT_SET` one can select between the different event sets. If `HPM_EVENT_SET` is not specified the default set 60 will be used. The following example is for `ksh`

```
export HPM_EVENT_SET=5
```

The counters and derived metrics of the different sets are discussed in section 4.

#### 3.5.2 HPM\_OUTPUT\_NAME

The environment variable `HPM_OUTPUT_NAME` changes the names of the output file `*.hpm` and the visualisation file `*.viz`. It overwrites what you specified in the second argument of the `hpmInit` call. This can be used to specify the selected event set in the name of the output files.

```
export HPM_EVENT_SET=56  
export HPM_OUTPUT_NAME=hpm_output_set${HPM_EVENT_SET}
```

The above syntax is for `ksh`. Please modify for your favorite shell. For a three processor run this will generate the following output files

```
hpm_output_set56_0000.hpm  
hpm_output_set56_0000.viz  
hpm_output_set56_0001.hpm  
hpm_output_set56_0001.viz  
hpm_output_set56_0002.hpm  
hpm_output_set56_0002.viz
```

The 4-digit number is derived from the `taskid` of the instrumentation routine `hpmTerminate`.

### 3.5.3 HPM\_NUM\_INST\_PT

The environment variable `HPM_NUM_INST_PTS` controls the number of instrumented regions, which are maximally possible. The default is 100 if `HPM_NUM_INST_PTS` is not specified.

## 4 HPM event sets and explanation of the output

On a IBM Power4 architecture (such as HPCx) different event sets have to be selected. When using `HPMCOUNT` use the `-g` option and when running code instrumented with `LIBHPM` use the environment variable `HPM_EVENT_SET` to specify the event set.

Luiz DeRose recommends five different sets as particularly useful. These are discussed in the following subsections. Each set consists of a number of raw counters and a set of derived metrics, which are often easier to judge than the raw counters. The `-1` option in section 2.2.5 of `HPMCOUNT` provides a complete listing of the raw counters for all sets.

Several of these derived metrics refer to rates which use either wall-clock time or user time. When investigating the entire program using `HPMCOUNT`, the wall-clock time will include various overheads, such as the time needed for starting the application, initialisation routines etc. In particular for the short test jobs normally used in performance analysis, these overheads will take up a large portion of the total execution time. Hence derived metrics using wall-clock time will be severely distorted in comparison to long production runs, which might lead to misleading conclusions. When instrumenting the application code using `LIBHPM`, it is easy to exclude the overheads from the measured code segments and derived metrics using the wall clock time will produce meaningful results even when used in short runs.

The user time is only available in those sets which feature the counter `PM_CYC`. `PM_CYC` counts the processor cycles consumed by the application. The user time is calculated by dividing the number of cycles by the processor frequency.

### 4.1 Cycles, instructions, floats including divides, multiply-adds, loads and stores

**Event set 60** is the default if no event set is specified. Use it for counts of cycles, total instruction and various floating point operations. The floating point operations include: *divides, multiply-adds<sup>1</sup>, load, stores and completed floating point instructions*. Among the derived metrics you can find an estimate of the floating point performance (excluding store operations) of the code section under investigation.

Raw counters	
<code>PM_FPU_FDIV</code>	Number of floating point divisions (hardware)
<code>PM_FPU_FMA</code>	Number of floating point multiply-additions
<code>PM_FPU0_FIN</code>	Operations on floating point unit 0 producing a result
<code>PM_FPU1_FIN</code>	Operations on floating point unit 1 producing a result
<code>PM_CYC</code>	Number of processor cycles
<code>PM_FPU_STF</code>	Number of floating point stores (by floating point unit)
<code>PM_INST_CMPL</code>	Number of completed instructions
<code>PM_LSU_LDF</code>	Number of floating point loads (by load store unit)

<sup>1</sup>The Power4 processor can process `a+b*c` in a single operation called a floating point multiply-add (FMA). For top performance it is crucial to get a high FMA percentage.

<b>Derived Metrics</b>	
Utilization rate	User time divided by wall-clock time in percent
Load and store operations	Total number of floats loaded and stored in 1000000 operations
Instructions per load/store	Completed instructions divided by result of the previous line
MIPS	Completed instructions divided by wall-clock time in 1000000/s
Instructions per cycle	Completed instructions divided by number of cycles
HW Float point instructions per Cycle	Sum of result-producing operations on both FPUs divided by the number of cycles
Floating point instructions + FMAs (flips)	Sum of result-producing operations on both FPUs plus the number of executed floating point multiply-additions minus the stores by the FPUs <i>Each FMA contains 2 calculations in a single instruction. Store instructions contain no calculation.</i>
Flip rate (flips/WCT)	Result of the previous line divided by the wall-clock time in 1000000/sec <i>Includes many overheads for HPMCOUNT, more useful for LIBHPM</i>
Flips/user time	As above but with user time instead of wall-clock time
FMA percentage	Twice the number of floating point multiply adds divided by the flips in percent
Computation intensity	Number of flips divided by the total number of floats loaded and stored

## 4.2 Cycles, instructions, TLB and level 1 data cache

**Event set 56** gives access to the hardware counters for cycles, instructions, the translation lookaside buffer (TLB) and the level 1 data cache. Note the 64 kB per processor level 1 instruction cache is not covered by this event set.

**Remark on level 1 data cache store misses:** the level 1 data cache of HPCx has a store-through policy. Any data written to level 1 cache will immediately be written to level 2 cache as well. A level 1 data cache store miss will not establish the relevant cache line on the level 1 data cache. We expect level 1 data cache store misses to have at most a very minor impact on the performance. This remark also concerns some of the derived metrics in the table further down.

Raw counters	
PM_DTLB_MISS	Number of data TLB misses
PM_ITLB_MISS	Number of instruction TLB misses
PM_LD_MISS_L1	level 1 data cache load misses
PM_ST_MISS_L1	level 1 data cache store misses
PM_CYC	Number of processor cycles
PM_INST_CMPL	Number of completed instructions
PM_ST_REF_L1	level 1 <b>data</b> cache store references
PM_LD_REF_L1	level 1 <b>data</b> cache load references

**Remark on derived level2 metrics:** Several of the names of the following derived metrics refer to the level 2 cache. However, these derived metrics are calculated from the level 1 load and store misses. On the HPCx system these numbers should be interpreted as level 1 misses and not as level 2 access as their name suggests. Comparing the level 1 data load misses (counter PM\_LD\_MISS\_L1) to the actual load operations from cache level 2, level 3 and main memory (use event set 5 in subsection 4.3), we observed between 0.5 and 10 times as many level 1 misses as there are load operations. Our investigations indicate data prefetching as a possible trigger for data loads, which have no corresponding level 1 miss. On the other hand the Power4 processor has two load/store units. Both units encountering a level 1 miss on the same level 2 or level 3 cache line, would explain those cases when we measured up to twice as many level 1 misses as data loads.

Derived Metrics	
Utilization rate	User time divided by wall-clock time in percent
% TLB misses per cycle	Data TLB misses divided by the number of cycles
Avg number of loads per TLB miss	level 1 data cache load references divided by the number of data TLB misses
Total L2 data cache accesses	Sum of <b>level 1</b> load and store misses <i>Please read the above remark!</i>
% accesses from L2 per cycle	<b>Level 1</b> load and store misses divided by cycle number <i>Please read the above remark!</i>
L2 traffic	<b>Level 1</b> load and store misses multiplied by the 128 byte cache line size <i>Please read the above remark!</i>
L2 bandwidth	The previous line divided by the wall-clock time <i>For HPMCOUNT the wall-clock time may include considerable overheads.</i> <i>Please read the above remark!</i>
Load and store operations	Sum of the L1 data cache load and store references
Instructions per load/store	Number of completed instructions divided by the above.
Avg number of loads per load miss	Counter PM_LD_REF_L1 divided by counter PM_LD_MISS_L1
Avg number of store per store miss	Counter PM_ST_REF_L1 divided by counter PM_ST_MISS_L1
Avg number of load/stores per D1 miss	$(PM\_LD\_REF\_L1 + PM\_ST\_REF\_L1)/(PM\_LD\_MISS\_L1 + PM\_ST\_REF\_L1)$
L1 cache hit rate	Number of level 1 load and store references not resulting in a level 1 miss. Measured in % relative to the total level 1 load and store references.
MIPS	Completed instructions divided by wall-clock time in 1000000/s
Instructions per cycle	Completed instructions divided by number of cycles

### 4.3 Loading from memory, level 2 and level 3 cache

Use **Event set 5** for investigations of the loads from main memory, the level 2 and level 3 caches.

Nomenclature of cache levels and locations	
level 2	level 2 cache on same chip as processor
level 2.5	level 2 cache on different chip but same MCM as processor
level 2.75	level 2 cache on different MCM, <i>inaccessible on HPCx, since outside LPAR</i>
level 3	level 3 cache on same MCM
level 3.5	level 3 cache on different MCM, <i>inaccessible on HPCx, since outside LPAR</i>

Raw counters	
PM_DATA_FROM_MEM	Load operations from main memory
PM_DATA_FROM_L3	Load operations from level 3 cache
PM_DATA_FROM_L35	Load operations from level 3.5 cache
PM_DATA_FROM_L2	Load operations form level 2 cache
PM_DATA_FROM_L25_SHR	Load operations from level 2.5 cache in ‘read only’ state
PM_DATA_FROM_L25_MOD	Load operations from level 2.5 cache in ‘exclusive’ state
PM_DATA_FROM_L275_SHR	Load operations from level 2.75 cache, ‘read only’ state
PM_DATA_FROM_L275_MOD	Load operations from level 2.75 cache, ‘exclusive’ state

Derived Metrics	
Total loads from L2	Sum of loads from L2, L2.5 & L2.75 in 1000000 operations <i>This includes data and instructions</i>
L2 load traffic	The above multiplied by a 128 byte cache line <i>Cache line size of L1 and L2 is 128 bytes</i>
L2 load bandwidth	The above divided by the wall-clock time <i>Includes many overheads for HPMCOUNT, more useful for LIBHPM</i>
L2 load miss rate	Sum of loads from L3, L3.5 and Memory divided by the sum of loads from L2, L2.5, L2.75, L3, L3.5 and Memory
Total loads from L3	Sum of loads from L3 & L3.5 in 1000000 operations
L3 load traffic	The above multiplied by a 128 byte cache line <i>Cache line size of L2 relevant</i>
L3 load bandwidth	The above divided by the wall-clock time <i>Includes many overheads for HPMCOUNT, more useful for LIBHPM</i>
L3 load miss rate	Sum of loads from Memory divided by the sum of loads from L3, L3.5 and Memory
Memory load traffic	Loads from memory times a 512 byte cache line <i>Cache line size of L3 is relevant</i>
Memory load bandwidth	The above divided by the wall-clock time <i>Includes many overheads for HPMCOUNT, more useful for LIBHPM</i>

#### 4.4 Cycles, instructions, loads from level 3 and memory

Use **Event set 58** to investigate counts of cycles, total instructions and loads from level 3 cache or memory. This event set lacks the counter for the level 2.75 cache, which is irrelevant on HPCx because of the use of LPARs, for a full analysis of the level 2 caches. Use event set 5 in subsection 4.3 for an analysis of all level 2 caches. All raw counters of this event set have already been discussed in event set 60 in subsection 4.1 or event set 5 in subsection 4.3. Only the derived metrics “% loads from L3 per cycle” and “% loads from memory per cycle” are new here.

Nomenclature of cache levels and locations	
level 2	level 2 cache on same chip as processor
level 2.5	level 2 cache on different chip but same MCM as processor
level 3	level 3 cache on same MCM
level 3.5	level 3 cache on different MCM, <i>inaccessible on HPCx, outside LPAR</i>

Raw counters	
PM_DATA_FROM_MEM	Load operations from main memory
PM_DATA_FROM_L3	Load operations from level 3 cache
PM_DATA_FROM_L35	Load operations from level 3.5 cache
PM_DATA_FROM_L2	Load operations from level 2 cache
PM_DATA_FROM_L25_SHR	Load operations from level 2.5 cache in 'read only' state
PM_DATA_FROM_L25_MOD	Load operations from level 2.5 cache in 'exclusive' state
PM_CYC	Number of processor cycles
PM_INST_CMPL	Number of completed instructions

Derived Metrics	
Utilization rate	User time divided by wall-clock time in percent
Total loads from L3	Sum of loads from L3 & L3.5 in 1000000 operations
% loads from L3 per cycle	The previous result divided by the number of cycles in percent
L3 load traffic	The above multiplied by a 128 byte cache line <i>cache line size of L2 relevant</i>
L3 load bandwidth	The above divided by the wall-clock time <i>Includes many overheads for HPMCOUNT, more useful for LIBHPM</i>
L3 load miss rate	Sum of loads from Memory divided by the sum of loads from L3, L3.5 and Memory
% loads from memory per cycle	Load operations from main memory divided by the number of cycles in percent
Memory load traffic	Loads from memory times a 512 byte cache line <i>Cache line size of L3 is relevant</i>
Memory load bandwidth	The above divided by the wall-clock time <i>Includes many overheads for HPMCOUNT, more useful for LIBHPM</i>
MIPS	Completed instructions divided by wall-clock time in 1000000/s
Instructions per cycle	Completed instructions divided by number of cycles

## 4.5 Cycles, instructions, fixed point and floating point counters

Using **Event set 53** gives access to counters of cycles, total instructions, floating point and fixed point operations. The counter for the usage of hardware square roots (**FSQRT**) is unique to this event set. Further floating point counters can be found in event set 60 in subsection 4.1.

Raw counters	
PM_FPU_FDIV	Number of floating point divisions (hardware)
PM_FPU_FMA	Number of floating point multiply-additions
PM_FXU_FIN	Number of fixed point operations producing a result
PM_FPU_FIN	Number of floating point operations producing a result
PM_CYC	Number of processor cycles
PM_FPU_FSQRT	Number of floating point square roots (hardware)
PM_INST_CMPL	Number of completed instructions
PM_FPU_FMOV_FEST	Number of floating point instructions FMOV or FEST

**Remark** The derived metrics of this event set feature HW floating point instructions. Please note that these include floating point store operations, which are not normally included in performance measures. Please check event set 60 in subsection 4.1 for derived metrics with the floating point store operations excluded.

Derived Metrics	
Utilization rate	User time divided by wall-clock time in percent
MIPS	Completed instructions divided by wall-clock time in 1000000/s
Instructions per cycle	Completed instructions divided by number of cycles
HW Float point instructions per Cycle	Sum of result producing operations on both FPUs divided by the number of cycles
HW floating point / user time	Sum of result producing operations on both FPUs divided by the user time, in 1000000 instructions per second
HW floating point rate	Sum of result producing operations on both FPUs divided by the wall-clock time, in 1000000 instructions per second

## 5 Visualisation with HPMVIZ

Each task of a LIBHPM instrumented code will generate two output files. One has a file extension `.hpm` and the other `.viz`. The `.hpm` is a plain ASCII file which contains all the results. Use standard UNIX tools like `more`, `awk` and `grep` to investigate their contents. Depending on the task at hand, this might be preferable to using HPMVIZ.

The `.viz` files are the input file for the HPMVIZ visualisation tool, which comes as a part of the HPM toolkit.

### 5.1 Starting HPMVIZ

To start HPMVIZ type:

```
/usr/local/packages/actc/hpmtk/pwr4/bin/hpmviz &
```

at the command-prompt. You can list the `.viz` files to be examined between the command and the “&” if you want.

## 5.2 Loading files

To select the file from inside HPMVIZ, click on “File” in the top right corner of the window, and on “Open Data” in the pull-down menu which appears. This will get you a browser to select the `.viz` file(s) to analyse.

The browser window is quite self explanatory. Here we want to point out the two buttons in the top right-hand corner. They switch between the “Detail View” and the more compact “List View”. In “Detail View” by clicking on “Date” the browser will sort the files with respect to their creation date. This can be helpful when locating all output files belonging to a single parallel job.

### 5.2.1 Visualising parallel jobs

When working on a parallel job with LIBHPM, by default each process generates its own `*.viz` file. We worked out two convenient ways to load these multiple files in the visualisation tool:

- On the **UNIX command-line**: List all the `*.viz` files on the command-line after the `hpmviz` command. This is particular useful when using wildcards for the file names originating from jobs with large processor counts.
- In the **browser** of the file menu of the visualisation tool: Select multiple files by holding the **Control**-key while clicking on them. If the files are consecutive, you can select the first file by clicking on it and the last file by holding down the **Shift**-key while clicking on it. In both cases it is helpful that you can sort the files by name, size, date etc. when using the “Detail view” format.

When comparing different runs it is useful to have different root names for the `*.viz` files. Change the second argument of the `hpmInit` command and recompile or use the environment variable `HPM_OUTPUT_NAME`.

## 5.3 The HPMVIZ main window

The main window is split in the middle. On the left-hand side it displays a list of the instrumented sections. They are labled with the second argument of the command `hpmStart` described in section 3.1.4. The column “Count” shows how often the instrumented section was entered in the run under investigation. This is followed by two timings. Both are derived from the wall-clock time. “IncSec” gives the time spent in this section. In the column `ExcSec`, the time spent in instrumented regions, which are nested inside the region under investigation, has been subtracted. The right-hand side of the main window displays the source code.

By clicking with the left mouse button on one of the sections in the left-hand side, the relevant code section will be highlighted in the right-hand window. Clicking on an instrumented section with the right mouse button will open a “metrics” window for this section.

## 5.4 The metrics window

The metrics window lists the performance result for each task and thread. The columns “Count”, “ExecSec” and “IncSec” have been explained for the main window in section 5.3 already. The following columns list the performance data for the various metrics.

Individual metrics can be switched on and off in the pull-down menu “Metric Options”. By default all raw counter are off and all derived metrics are on.

The visualisation tool will highlight all results in red which it regards as poor and show those in light grey which are good. To us this is the key feature of HPMVIZ.

**Important:** Whether a result is good or bad depends on the precision of the code, which is directly obvious for a metric such as loads per load miss. Use the pull-down menu “Precision” to set “single” or “double” precision depending on your code.

## References

- [1] Luiz DeRose, “*Hardware Performance Monitor (HPM) Toolkit*”,  
<http://www.hpcx.ac.uk/support/documentation/IBMdocuments/HPM.html>