Hardware Performance Monitor (HPM) Toolkit

Users Guide

Christoph Pospiech
Advanced Computing Technology Center
IBM Research
pospiech@de.ibm.com
Phone: +49 – 351 – 86269826
Fax: +49 – 351 – 4758767

Version 3.2.2 - June 5, 2008

LICENSE TERMS

The IBM High Performance Computing Toolkit (IHPCT) is distributed under a nontransferable, nonexclusive, and revocable license. The HPM software is provided "AS IS". **IBM makes no warranties, expressed or implied, including the implied warranties of merchantability and fitness for a particular purpose**. IBM has no obligation to defend or indemnify against any claim of infringement, including, but not limited to, patents, copyright, trade secret, or intellectual property rights of any kind. IBM is under no obligation to maintain, correct, or otherwise support this software. IBM does not represent that the HPM Toolkit will be made generally available. IBM does not represent that any software made generally available will be similar to or compatible with the HPM Toolkit.

Contents

1 The HPM Toolkit

The HPM Toolkit was developed for performance measurement of applications running on IBM systems supporting the following processors and operating systems:

PPC970 AIX 5L or Linux

Power4 AIX 5L or Linux

Power5 AIX 5L or Linux

Power5+ AIX 5L or Linux

Power6 AIX 5L or Linux

BG/L Linux

BG/P Linux

The HPM Toolkit consists of:

- An utility **hpmcount**, which starts an application and provides at the end of execution wall clock time, hardware performance counters information, derived hardware metrics, and resource utilization statistics[1][2]
- An instrumentation library **libhpm**, which provides instrumented programs with a summary output containing the above information for each instrumented region in a program (resource utilization statistics is provided only once, for the whole section of the program that is instrumented). This library supports serial and parallel (MPI, threaded, and mixed mode) applications, written in Fortran, C, and C++.
- An utility **hpmstat** (depreciated)[2]
- A graphical user interface **PeekPerf**, for graphical visualization of the performance file generated by libhpm. This is described in a separate document.

Requirements:

On AIX 5L

- bos.pmapi, which is a product level file set provided with the AIX distribution, but not installed by default.

On Linux

- The kernel has to be recompiled with the perfctr patch. The patch can be obtained from the following URL
  http://user.it.uu.se/~mikpe/linux/perfctr/2.7/...
  .../perfctr-2.7.20.tar.gz

[1] For more information on the resource utilization statistics, please refer to the getrusage man pages.

[2] hpmcount and hpmstat are not available on BG/L and BG/P

- libperfctr.a compiled with 32-bit and 64-bit addressing (gcc -m32 and gcc -m64). The sources can be obtained from the following URL http://user.it.uu.se/~mikpe/linux/perfctr/2.7/... .../perfctr-2.7.20.tar.gz

On BG/L or BG/P none

The owner and developer for the previous HPM versions 1 and 2 was Luiz DeRose (ACTC).

2 hpmcount

2.1 Quick Start

hpmcount is essentially used the same way as the time command; the user just types

```
hpmcount <program>
```

As a result, hpmcount appends various performance information at the end of the screen (i.e. stdout) output. In particular it prints resource utilization statistics, hardware performance counter information and derived hardware metrics.

The resource usage statistics is directly taken from a call to *getrusage*(). For more information on the resource utilization statistics, please refer to the getrusage man pages. In particular, on Linux the man page for *getrusage*() states that not all fields are meaningful under Linux. The corresponding lines in hpmcount output have the value "n/a".

2.2 Usage

Sequential or shared memory programs

```
hpmcount [-o <name>] [-u] [-n] [-x] [-g <group>] \
         [-a <plugin>] <program>
```

MPI or hybrid parallel programs

```
poe hpmcount [-o <name>] [-u] [-n] [-x] [-g <group>] \
             [-a <plugin>] <program>
mpirun hpmcount [-o <name>] [-u] [-n] [-x] [-g <group>] \
                [-a <plugin>] <program>
```

or

```
hpmcount [-h] [-l] [-c]

where: <program>      program to be executed
       -h             displays this help message
       -o <name>      copies output to file <name>
       -u             make the file name <name> unique
       -n             no hpmcount output in stdout
                      when "-o" flag is used
       -x             adds formulas for derived metrics
       -g <group>     PM_API group number
       -a <plugin>    aggregate counters using the
                      plugin <plugin>
       -l             list groups
       -c             list counters and events
```

Detailed descriptions of selected options

-o <name> copies output to file <name>.

- The file name can be specified via option -o or via environment variable HPM_OUTPUT_NAME. The option takes precedence if there are conflicting specifications.
- The name <name>is actually expanded into three different file names.

  <name>.hpm is the file name for ASCII output — which is basically a one-to-one copy of the screen output.

  <name>.viz is the filename for XML output.

<name>.csv is the filename for output as comma separated value file. This is not yet implemented in the current release.

- Which of these output files are generated is governed by three additional environment variables. If none of those are set, only the ASCII output is generated. If at least one is set, the following rules apply.

  HPM_ASC_OUTPUT If set to 'Y[...]', 'y[...]' or '1', triggers the ASCII output.

  HPM_VIZ_OUTPUT If set to 'Y[...]', 'y[...]' or '1', triggers the XML output.

  HPM_CSV_OUTPUT If set to 'Y[...]', 'y[...]' or '1', triggers the csv output. This is not yet implemented in the current release.

- Unless the -a option is chosen (see section 2.7 Considerations for MPI Parallel Programs below), there is one output for each MPI task. To avoid directing all output to the same file, the user is advised to have a different name for each MPI task (using e.g. the -u flag below) or direct the file to a non-shared file system.

-u make the file name <name>(specified via -o) unique.

- A string

  `_<hostname>_<process_id>_<date>_<time>`

  is inserted before the last '.' in the file name.

- If the file name has no '.', the string is appended to the file name.
- If the only occurrence of '.' is the first character of the file name, the string is prepended, but the leading '_' is skipped.
- If the host name contains '.' ("long form"), only the portion preceding the first '.' is taken. In case a batch queuing system is used, the host name is taken from the execution host, not the submitting host.
- Similarly for MPI parallel programs, the host name is taken from the node where the MPI task is running. The addition of the process_id enforces different file names for MPI tasks running on the same node.
- If used for an MPI parallel program, hpmcount tries to extract the MPI task id (or MPI rank with respect to MPI_COMM_WORLD)

from the MPI environment. If successful, the process_id is replaced with the MPI task id.

- The date is given as dd.mm.yyyy, the time is given by hh.mm.ss in 24h format using the local time zone.
- This flag is only active when the -o flag is used.

-n no hpmcount output in stdout when "-o" flag is used.

- This flag is only active when the -o flag is used.
- Alternatively the environment variable HPM_STDOUT='no' can be used.

-x adds formulas for derived metrics

- For an explanation on derived metrics see section 2.5 below.
- For a formula reference see section 5.2 below.
- Alternatively the environment variable HPM_PRINT_FORMULA='yes' can be used.

-g <group[,<group>, ...]> specify group number(s)

- for an explanation on groups see sections 2.3 on Events and Groups and 2.4 on Multiplexing below.

-a <plug-in> aggregate counters using the plug-in <plug-in>.

-l list groups

- for an explanation on groups see section 2.3 on Events and Groups below.

-c list counters and events

- for an explanation on groups see section 2.3 on Events and Groups below.

2.3 Events and Groups

The hardware performance counters information is the value of special CPU registers that are incremented at certain events. The number of such registers is different for each architecture.

Processor Architecture	Number of performance counter registers
PPC970	8
Power4	8
Power5	6
Power5+	6
Power6	6
BG/L	52
BG/P	256

On both AIX and Linux, kernel extensions provide "counter virtualization", i.e. the user sees private counter values for his application. On a technical side, the counting of the special CPU registers is frozen and the values are saved whenever the application process is taken off the CPU and another process gets scheduled. The counting is resumed when the user application gets scheduled on the CPU.

The special CPU registers can count different events. On the Power CPUs the are restrictions which registers can count what events. A call to "hpmcount -c" will list all CPU counting registers and the events they can be monitoring.

Even more, there are lots of rules restricting the concurrent use of different events. Each valid combination of assignments of events to hardware counting registers is called a group. To make handling easier, a list of valid groups is provided. A call to "hpmcount -l" will list all available groups and the events they are monitoring. The -g option is provided to select a specific group to be counted by hpmcount. If -g is not specified, a default group will be taken according to the following table.

Processor Architecture	Number of groups	Default group
PPC970	41	23
Power4	64	60
Power5	148(140)	137
Power5+	152	145
Power6	195	127
BG/L	16	0
BG/P	4	0

The number of groups for Power5 is 140 for AIX 5.2, and 148 for Linux and AIX 5.3. The reason for this difference are different versions of bos.pmapi.The last group (139) was changed and 8 new groups were appended. If HPM is called with

```
hpmcount -g <a_new_group_number>
```

on AIX 5.2, it returns the following error message.

```
hpmcount ERROR - pm_set_program_mygroup:
pm_api : event group ID is invalid
```

2.4 Multiplexing

The idea behind multiplexing is to run several groups concurrently. This is accomplished by running the first group for a short time interval, then switching to the next group for the next short time interval, and keep doing this in a round robin fashion for the groups until the event counting is eventually stopped.

HPM supports multiplexing only on AIX5. The groups are specified as a comma separated list.

```
hpmcount -g 1,2 <program>
OR
export HPM_EVENT_SET='1,2'
hpmcount <program>
```

On Operating systems other than AIX5 this leads to the following error message.

```
HPM ERROR - Multiplexing not supported:
too many groups or events specified: 2
```

Of course multiplexing means that neither of the groups has been run on the whole code. Likewise it is unknown what fraction of the code was measured with which group. It is assumed that the workload is sufficiently uniform

that the measured event counts can be (more or less) safely calibrated as if the groups have been run separately on the whole code.

The default time interval for measuring one group on the application is 100ms. If the total execution time is shorter than this time interval, only the first group in the list will be measured. All other counter values are 0. This might result in NaNQ values for some derived metrics (see section 2.5 on derived metrics), if the formula for computing the derived metric requires division by the counter value 0.

The duration of this time interval can be controlled by setting the following environment variable.

```
export HPM_SLICE_DURATION=<integer value in ms>
```

This value is passed directly to AIX (more precisely bos.pmapi). AIX requires the value to lie between 10ms an 30s.

The form of the output depends on the chosen aggregator (see sub section 2.7.3 on Aggregators). Without specifying an aggregator (i.e. with the default aggregator), the data for each group is printed in a separate section with separate timing information. To have these data joined into one big group with more counters, one should use the local merge aggregator (loc_merge.so), which described below.

2.5 Derived Metrics

2.5.1 What are Derived Metrics

Some of the events are difficult to interpret. Sometimes a combination of events provide better information. In the sequel, such a recombination of basic events will be called derived metric. HPM also provides a list of derived metrics, which are defined in section Derived Metrics Description below.

Since each derived metric has its own set of ingredients, not all derived metrics are printed for each group. HPM automatically finds those derived metrics which are computable and prints them. As a convenience to the user, the option -x will print not only the value of the derived metric, but also its definition.

2.5.2 MFlop Issues

The two most popular derived metrics are the MFlop/s rate and the percentage of peak performance. The default group is chosen to make those two derived metrics available without special choice of the -g parameter.

For Power5, there is no group that supports enough counters to compute a MFlop/s rate. So an "Algebraic MFlop/s rate" was invented to bridge this gap. This derived metric counts floating point adds, subs and mults (including FMAs), but misses divides and square roots. If the latter two only occur in negligible numbers (which is desirable for a HPC code anyway), the "Algebraic MFlop/s rate" coincides with the usual definition of MFlop/s. Again, different counter groups can be used to check this hypothesis. Group 137 (which exploits the "Algebraic MFlop/s rate") is chosen the default group for Power5.

The derived metric "percent of peak performance" was added new in this version. For all, PPC970, Power4 and Power5, this is based on user time rather than wall clock time, as the result stays correct if multi-threaded (e.g. OpenMP) applications are run. For Power5, this is based on "Algebraic MFlop/s". This has not been tested on AIX 5.3, particularly when SMT is active.

Flip-flop blues: Compared to the previous version, the meanings of MFlip/s and MFlop/s are swapped. Not only many users were confused by the name "MFlip/s" being used for what they expected to be MFlop/s. Also Ptools (notably PAPI) is using a different nomenclature. HPM derived metric naming was changed to conform with this emerging standard.

On Power4, PPC970 and Power6 also weighted MFlop/s are available. These are like odinary MFlop/s, except that divisions enter the evaluation with a weight different from the other floating point operations. The weight factor is provided by the user through the evironment variable HPM_DIV_WEIGHT. If set to 1, the weighted MFlop/s coincide with the ordinary MFlop/s. HPM_DIV_WEIGHT can take any positive integer number.

If this environment variable is not set, no weighted MFlop/s are computed.

2.5.3 Latency Computation

In addition, users can provide estimations of memory, cache, and TLB miss latencies for the computation of derived metrics, with the following environment variables (please notice that not all flags are valid in all systems):

HPM_MEM_LATENCY latency for a memory load.

HPM_AVG_L3_LATENCY average latency for an L3 load.

HPM_L3_LATENCY latency for an L3 load within a MCM.

HPM_L35_LATENCY latency for an L3 load outside of the MCM.

HPM_AVG_L2_LATENCY average latency for an L2 load.

HPM_L2_LATENCY latency for an L2 load from the processor.

HPM_L25_LATENCY latency for an L2 load from the same MCM.

HPM_L275_LATENCY latency for an L2 load from another MCM.

HPM_TLB_LATENCY latency for a TLB miss.

When computing derived metrics that take into consideration estimated latencies for L2 or L3, the HPM Toolkit will use the provided "average latency" only if the other latencies for the same cache level are not provided. For example, it will only use the value set in HPM_AVG_L3_LATENCY, if at least one of the values of HPM_ L3_LATENCY and HPM_ L35_LATENCY is not set.

If environment variables are frequently used, they can be collected in a file like this.

```
$ cat docs/examples/HPM\_flags.env-example
export HPM\_MEM\_LATENCY=400
export HPM\_L3\_LATENCY=102
export HPM\_L35\_LATENCY=150
export HPM\_L2\_LATENCY=12
export HPM\_L25\_LATENCY=72
export HPM\_L275\_LATENCY=108
export HPM\_TLB\_LATENCY=700
export HPM\_DIV\_WEIGHT=5
```

The following command would enable these environment variables.

```
$ . cat docs/examples/HPM\_flags.env-example
```

2.6 Inheritance

On both, AIX and Linux, the "counter virtualization" and the group (i.e. set of events) that is actually monitored is inherited from the process to any of its children. Children in this context means threads or processes spawned by the parent process. AIX and Linux, however differ in the ability of the parent process to access the counter values of its children.

- On AIX all counter values of a process group can be collected.
- On Linux counter values are only available to the parent, if the child has finished.

hpmcount makes use of this inheritance. Therefore, if hpmcount is called for a program, the returned counter values are the sum of the counter values of the program and all of the threads and processes spawned by it - at the time the values are collected. For Linux this has to be restricted to the sum of counter values of all children that have finished at the time the values are collected. Even the latter is enough to catch the values of all threads of an OpenMP program.

Also, suppose some user is using a program to bind threads to CPUs. Further, suppose this program is called "taskset" and it would be used like this.

```
hpmcount taskset -c <num> <program_name>
```

hpmcount would first enable hardware event counting for the application /usr/bin/taskset. This command then spawns the program <program_name>, which inherits all event counter settings. At the end hpmcount would print counter values (and derived metrics) based on the sum of events for taskset and the called program. Since taskset is a very slim application, the hpmcount results would vastly represent the performance of the program <program_name>— which is what the user intended.

2.7 Considerations for MPI Parallel Programs

2.7.1 General Considerations

hpmcount is an inherently sequential program, looking only at the hardware performance counters of a single process (and its children, as explained in section 2.6 Inheritance). When started with "poe" or "mpirun", one instance of hpmcount is running for each MPI task. Unless additional action is taken as described in the following sub sections, all these instances of hpmcount are completely ignorant of each other. Consequently, each instance is writing its own output. If the option -o is used, each instance is using the same file name, which results in writing into the same file, if a parallel file system is used. Of course, this can be (and should be) prevented by making the file names unique through the '-u' option or the "HPM_UNIQUE_FILE_NAME" environment variable. Still it might be an unwanted side effect to have that many output files.

For this reason the -a option (or equivalently, the environment variable "HPM_AGGREGATE") triggers some aggregation before (possibly) restricting the output to a subset of MPI tasks. This formulation is deliberately vague, because there can be many ways to aggregate hardware performance counter information across MPI tasks. One way is to take averages, but maximum or minimum values could be also thought of. The situation is further complicated by allowing to run different groups on different MPI tasks. After all, the instances of hpmcount are independent, which should allow for this strategy. On architectures that allow for multiplexing (as described in section 2.4 Multiplexing), some tasks could use multiplexing, others may not. Of course averages, maxima and minima should only be taken on groups which are alike.

Therefore the -a option and the environment variable "HPM_AGGREGATE" take a value, which is the name of a plug-in that defines the aggregation strategy. Each plug-in is a shared object file containing two functions called distributor and aggregator.

2.7.2 Distributors

The motivating example for the distributor function is allowing a different hardware counter group on each MPI task. Of course this should be trivially possible if different environment variable settings (or different hpmcount op-

tions) can be passed to different MPI tasks. Although many MPI stacks allow for MPMD (multiple program multiple data) execution, it might be tedious and impractical to use the MPMD machinery just to vary the counting groups across the MPI tasks.

Therefore, the distributor is a subroutine that determines the MPI task id (or MPI rank with respect to MPI_COMM_WORLD) from the MPI environment for the current process, and (re)sets environment variables depending on this information. The environment variable may be any environment variable, not just HPM_EVENT_SET, which motivated this function.

Consequently, the distributor is called before any environment variable is evaluated by HPM. Command line options for hpmcount and hpmstat trigger (re)setting the corresponding environment variables, which are then passed to the distributor. Thus, the setting from the distributor takes precedence over both, options and global environment variable settings.

Of course, the aggregator has to adapt to the HPM group settings done by the distributor. This is why distributors and aggregators always come in pairs. Each plug-in is containing just one such pair.

2.7.3 Aggregators

The motivating example is the aggregation of the hardware counter data across the MPI tasks. In the simplest case this could be an average of the corresponding values. Hence this function is called

- after the hardware counter data have been gathered,
- before the derived metrics are computed.
- before these data are printed,

In a generalized view, the aggregator is taking the raw results and rearranges them for output.

Also, depending on the information of the MPI task id (or MPI rank with respect to MPI_COMM_WORLD) the aggregator sets (or doesn't set) a flag to mark the current MPI task for HPM printing.

2.7.4 Plug-ins shipped with the Tool Kit

The following plug-ins are shipped with the toolkit. They can be found in

```
$(IHPCT_BASE)/lib or  $(IHPCT_BASE)/lib64
```

mirror.so is the plug-in that is called when no plug-in is requested. The aggregator is mirroring the raw hardware counter data in a one-to-one fashion into the output function. Hence this name. It is also flagging each MPI task as printing task. The corresponding distributor is a void function. This plug-in doesn't use MPI and also works in a non-MPI context.

loc_merge.so does a local merge on each MPI task separately. It is identical to the mirror.so plug-in except for those MPI tasks that change the hardware counter groups in the course of the measurement (e.g. by multi-plexing). The different counter data, which are collected for only part of the measuring interval, are proportionally extended to the whole interval and joined into one big group that is entering derived metrics computation. This way, more derived metrics can be determined at the risk of computing garbage. The user is responsible for using this plug-in only when it makes sense to use it. It is also flagging each MPI task as printing task. The corresponding distributor is a void function. This plug-in doesn't use MPI and also works in a non-MPI context.

single.so does the same as mirror.so, but only on MPI task 0. The output on all other tasks is discarded. This plug-in uses MPI functions and can't be used in a sequential context.

average.so is a plug-in for taking averages across MPI tasks. The distributor is reading the environment variable HPM_EVENT_SET (which is supposed to be a comma separated list of group numbers) and distributes these group numbers in a round robin fashion to the MPI tasks. The aggregator is first building a MPI communicator of all tasks with equal hardware performance counting scenario. The communicator groups may be different from the original round robin distribution. This may happen if the conting group has been changed on some of the MPI tasks after the first setting by the distributor. Next the aggregator is taking the average across the subgroups formed by this communicator. Finally it is flagging the MPI rank 0 in each group as

printing host. This plug-in uses MPI functions and can't be used in a sequential context.

2.7.5 User defined Plug-ins

There can be no doubt that this set of plug-ins can only be a first starter kit and many more might be desirable. Rather than taking the average one could think of taking maximum or minimum. There is also the possibility of taking kind of a "history_merge.so" by blending in results from previous measurements. Chances are that however big the list of shipped plug-ins may be, the one just needed is missing from the set ("Murphy's law of HPM plug-ins"). The only viable solution comes with disclosing the interface between plug-in and tool and allowing for user defined plug-ins.

The easiest way to enable users to write their own plug-ins is by providing examples. Hence the plug-ins described above are provided in source code together with the Makefile that was used to generate the shared objects files. These files can be found in the following place.

```
$(IHPCT_BASE)/examples/plugins
```

2.7.6 Detailed Interface Description

Each distributor and aggregator is a function returning an integer which is 0 on success and $! = 0$ on error. In most cases the errors occur when calling a system call like malloc(), which sets the errno variable. If the distributor or aggregator returns the value of errno as return code, the calling HPM tool sees to an expansion of this errno code into a readable error message. If returning the errno is not viable, the function should return a negative value.

The function prototypes are defined in the following file.

```
$(IHPCT_BASE)/include/hpm_agg.h
```

This is a very short file with the following contents.

```
#include "hpm_data.h"
```

```
int distributor(void);

int aggregator(int num_in, hpm_event_vector in,
               int *num_out, hpm_event_vector *out,
               int *is_print_task);
```

The distributor has no parameters and is only required to (re)set environment variables (via setenv()).

The aggregator takes the current hpm values on each task as an input vector *in* and returns the aggregated values on the output vector *out* on selected or all MPI tasks. To have utmost flexibility, the aggregator is responsible to allocate the memory needed to hold the output vector *out*. The definition of the data types used for *in* and *out* are provided in the following file.

```
$(IHPCT_BASE)/include/hpm_data.h
```

Finally the aggregator is supposed to set (or unset) a flag to mark the current MPI task for HPM printing.

Form the above definitions it is apparent that the interface is defined in C-Language. While it is in principle possible to use another language for programming plug-ins, the user is responsible for using the same memory layout for the input and output variables. There is no explicit FORTRAN interface provided.

The hpm_event_vector *in* is a vector or list of *num_in* entries of type *hpm_data_item*. The latter is a struct containing members that describe the definition and the results of a single hardware performance counting task.

```
/*      NAME                INDEX */

#define HPM_NTIM 8
#define HPM_TIME_WALLCLOCK      0
#define HPM_TIME_CYCLE          1
#define HPM_TIME_USER           2
#define HPM_TIME_SYSTEM         3
#define HPM_TIME_START          4
#define HPM_TIME_STOP           5
#define HPM_TIME_OVERHEAD       6
#define HPM_TIME_INIT           7
```

```
typedef struct {
  int              num_data;
  hpm_event_info  *data;
  double           times[HPM_NTIM];
  int              is_mplex_cont;
  int              is_rusage;
  int              mpi_task_id;
  int              instr_id;
  int              count;
  int              is_exclusive;
  int              xml_element_id;
  char            *description;
  char            *xml_descr;
} hpm_data_item;

typedef hpm_data_item *hpm_event_vector;
```

- Counting the events from a certain HPM group on one MPI task is represented by a single element of type *hpm_data_item*.
- If multiplexing is used, the results span several consecutive elements, each dedicated to one HPM group that take part in the multiplex setting. On all but the first element the member *is_mplex_cont* is set to *TRUE_* to indicate that these elements are continuations of the first element belonging to the same multiplex setup.
- If HPM groups are changed during the measurement, the results for different groups are recorded in different vector elements, but no *is_mplex_cont* flag is set. This way results obtained via multiplexing can be distinguished from results obtained by ordinary group change.
- If libhpm and several instrumented sections are used, each instrumented code section will use separate elements of type *hpm_data_item* to record the results. Each of these will have the member *instr_id* set with the first argument of hpmStart and the logical member *is_exclusive* set to *TRUE_* or *FALSE_* depending on whether the element hold inclusive or exclusive counter results (see section 3.6 Inclusive and Exclusive Values for details). Then all these different elements are concatenated into a single vector.

- Finally, hpmcount, libhpm and hpmstat prepend the data from a call to getrusage() to this vector, so the rusage data form the vector element with index 0. This vector element is the only element with struct member *is_rusage* set to *TRUE_* to distinguish it from ordinary hardware performance counter data.

The output vector is of the same format. Each vector element enters the derived metrics computation separately (unless *is_rusage* == *TRUE_*). Then all vector elements (and the corresponding derived metrics) are printed in the order given by the vector *out*. The output of each vector element will be preceded by the string given in member *description* (which may include line feeds as appropriate). The XML output will be marked with the text given in *xml_descr*.

This way the input vector *in* is providing a complete picture of what has been measured on each MPI task. The output vector *out* is allowing complete control on what is printed on which MPI task in what order.

2.7.7 Getting the plug-ins to work

The plug-ins have been compiled with the following Makefile

```
$(IHPCT_BASE)/examples/plugins/Makefile
```

using this command.

```
<g>make ARCH=<appropriate_archtitecture>
```

The include files for the various architectures are provided in subdirectory make. The user is asked to note a couple of subtleties.

- The Makefile distinguishes "sequential" (specified in *PLUGIN_SRC*) and "parallel" plug-ins (specified in *PLUGIN_PAR_SRC*). The latter are compiled and linked with the MPI wrapper script for the compiler/ linker. Unlike a static library, generation of a shared object requires linking, not just compilation.
- On BG/L there are no shared objects, so ordinary object files are generated. On BG/L and BG/P just everything is parallel.

- If the MPI software stack requires the parallel applications to be linked with a special start-up code (like poe_remote_main() for IBM MPI on AIX), the shared object has to carry this start-up code. hpmcount is a sequential application. Therefore the start-up code has to be loaded and activated when the plug-in is loaded at run time. This turns the sequential application hpmcount into a parallel application "on the fly". This sounds complicated but works pretty seamlessly, at least for IBM MPI for user space protocol and MPICH and its variants. No further user action is required to make this work.

- There are, however, some restrictions to be observed when writing plug-in code. The MPI standard document disallows calling MPI_Init() twice on the same process. It appears that this is indeed not supported on the majority of MPI software stacks, not even if an MPI_Finalize() is called between the two invocations of MPI_Init().

- hpmcount is starting the user application via fork() and exec(). If hpmcount would be starting MPI_init() prior to the fork() (i.e. in the distributor), chances are that the parallel environment would be inherited across fork() and exec() and hence would collide with the MPI_Init() from the user application. Even an MPI_Finalize() at the end of the distributor does not help for the MPI_Init() calls in the user application and the aggregator. Hence the distributor must not call any MPI function.. The MPI task id should be extracted by inspecting environment variables that have been set by the MPI software stack.

- The aggregator, however, is executed in a different process than the user application, so it cannot participate from the parallel environment of the user application. It is usually fired up after the user application has finalized its MPI environment. Sitting on a different process the aggregator can and should initiate its own parallel environment. Fortunately, all tested MPI software stacks reacted cooperative when hpmcount out of a sudden turned into a parallel application and tried to reuse the parallel set up from the previously ended user application. The good news is that hpmcount turns parallel after the user application has ended. Hence there is no competition for interconnect resources between hpmcount and the user application.

- hpmcount uses a call to dlopen() to access the plug-in and makes use of its functions. There is no dlopen() on BG/L, but there is no fork() - and hence no hpmcount on BG/L and BG/P either.

- The "sequential" plug-ins are independent on the MPI software stack. Hence they can be continued to be used even if the MPI software is changed. The "parallel" plugins have to be re-compilated against the new MPI software stack.

2.8 Shared Object Trouble Shooting

Occasionally one of the following error messages occur.

Problem:

```
hpmcount: error while loading shared libraries: libperfctr.so.6:
cannot open shared object file: No such file or directory
```

Solution: The Linux versions are based on the perfctr kernel extension and libperfctr.a. Either libperfctr.a (more precisely libperfctr.so.6) is not there; please check the prerequisites 1. Or the library exists, but resides in an unusual place. In the latter case, please use the environment variable LD_LIBRARY_PATH like in the following example.

```
export LD_LIBRARY_PATH=~/perfctr-2.7.18/usr.lib
```

Problem:

```
hpmcount: error while loading shared libraries: liblicense.so:
cannot open shared object file: No such file or directory
```

Solution: Most probably, liblicense.so is searched in the wrong place. Please execute the following command (for bash).

```
export LD_LIBRARY_PATH=$IHPCT_BASE/lib:\
$IHPCT_BASE/lib64:$LD_LIBRARY_PATH
```

Problem:

```
$ hpmcount -a not_there.so hostname
HPM ERROR - Dynamic link error at dlopen():
not_there.so: cannot open shared object file:
No such file or directory
```

Solution: Either the plug-in is really not there (as the name indicates), or it is not found. In the latter case, (for bash) please execute the following command.

```
export LD_LIBRARY_PATH=$IHPCT_BASE/lib:\
$IHPCT_BASE/lib64:$LD_LIBRARY_PATH
```

3 LIBHPM

3.1 Quick Start

hpmcount (see section 2) provides hardware performance counter information and derived hardware metrics (see section 2.5) for the whole program. If this information is required for only part of the program, instrumentation with libhpm is required. Libhpm is a library that provides a programming interface to start and stop performance counting for an application program. The part of the application program between start and stop of performance counting will be called an instrumentation section. Any such instrumentation section will be assigned a unique integer number as section identifier. A simple case of an instrumented program may look as follows.

```
hpmInit( tasked, my program );
hpmStart( 1, outer call );
do_work();
hpmStart( 2, computing meaning of life );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( taskID );
```

Calls to hpmInit() and hpmTerminate() embrace the instrumented part, every instrumentation section starts with hpmStart() and ends with hpmStop(). The section identifier is the first parameter to the latter two functions. As shown in the example, libhpm supports multiple instrumentation sections, overlapping instrumentation sections, and each instrumented section can be called multiple times. When hpmTerminate() is encountered, the counted values are collected and printed.

The program above provides an example of two properly nested instrumentation sections. For section 1 we can consider the "exclusive" time and and "exclusive" counter values. By that we mean the difference of the values for section 1 and section 2. The original values for section 1 would be called "inclusive values" for matter of distinction. The terms "inclusive" and "exclusive" for the embracing instrumentation section are chosen to indicate whether counter values and times for the contained sections are included or excluded. For more details see section 3.6 on inclusive and exclusive values.

Libhpm supports OpenMP and threaded applications. There is only a thread safe version of libhpm. Either a thread-safe linker invocation (e.g. xlc_r, xlf_r) should be used or the libpthreads.a must be included in the list of libraries. For details on how to compile and link an application with Libhpm please go to section 3.13 on compiling and linking.

Notice that libhpm collects information and performs summarization during run-time. Thus, there could be a considerable overhead if instrumentation sections are inserted inside inner loops.

Libhpm uses the same set of hardware counters events used by hpmcount (see section 2).

If some error occurs the program is not automatically stopped. libhpm rather sets some error indicator and let the user handle the error. For details see section 3.7 on function reference.

3.2 Events and Groups

The hardware performance counters information is the value of special CPU registers that are incremented at certain events. The number of such registers is different for each architecture.

Processor Architecture	Number of performance counter registers
PPC970	8
Power4	8
Power5	6
Power5+	6
Power6	6
BG/L	52
BG/P	256

On both AIX and Linux, kernel extensions provide "counter virtualization", i.e. the user sees private counter values for his application. On a technical side, the counting of the special CPU registers is frozen and the values are saved whenever the application process is taken off the CPU and another process gets scheduled. The counting is resumed when the user application gets scheduled on the CPU.

The special CPU registers can count different events. On the Power CPUs the are restrictions which registers can count what events. A call to "hpmcount -c" will list all CPU counting registers and the events they can be monitoring.

Even more, there are lots of rules restricting the concurrent use of different events. Each valid combination of assignments of events to hardware counting registers is called a group. To make handling easier, a list of valid groups is provided. A call to "hpmcount -l" will list all available groups and the events they are monitoring. The group or event set to be used can be selected via the environment variable: HPM_EVENT_SET. If the environment variable HPM_EVENT_SET is not specified, a default group will be taken according to the following table.

Processor Architecture	Number of groups	Default group
PPC970	41	23
Power4	64	60
Power5	148(140)	137
Power5+	152	145
Power6	195	127
BG/L	16	0
BG/P	4	0

The number of groups for Power5 is 140 for AIX 5.2, and 148 for Linux and AIX 5.3. The reason for this difference are different versions of bos.pmapi.The last group (139) was changed and 8 new groups were appended. If HPM is called with

```
export HPM_EVENT_SET=<a_new_group_number>
```

on AIX 5.2, it returns the following error message.

```
hpmcount ERROR - pm_set_program_mygroup:
pm_api : event group ID is invalid
```

3.3 Multiplexing

The idea behind multiplexing is to run several groups concurrently. This is accomplished by running the first group for a short time interval, then switching to the next group for the next short time interval, and keep doing this in a round robin fashion for the groups until the event counting is eventually stopped.

HPM supports multiplexing only on AIX5. The groups are specified as a comma separated list.

```
export HPM_EVENT_SET='1,2'
```

On Operating systems other than AIX5 this leads to the following error message.

```
HPM ERROR - Multiplexing not supported:
too many groups or events specified: 2
```

Of course multiplexing means that neither of the groups has been run on the whole code. Likewise it is unknown what fraction of the code was measured with which group. It is assumed that the workload is sufficiently uniform that the measured event counts can be (more or less) safely calibrated as if the groups have been run separately on the whole code.

The default time interval for measuring one group on the application is 100ms. If the total execution time is shorter than this time interval, only the first group in the list will be measured. All other counter values are 0. This might result in NaNQ values for some derived metrics (see section 3.4 on derived metrics), if the formula for computing the derived metric requires division by the counter value 0.

The duration of this time interval can be controlled by setting the following environment variable.

```
export HPM_SLICE_DURATION=<integer value in ms>
```

This value is passed directly to AIX (more precisely bos.pmapi). AIX requires the value to lie between 10ms an 30s.

The form of the output depends on the chosen aggregator (see sub section 3.12.3 on Aggregators). Without specifying an aggregator (i.e. with the default aggregator), the data for each group is printed in a separate section with separate timing information. To have these data joined into one big group with more counters, one should use the local merge aggregator (loc_merge.so), which described below.

3.4 Derived Metrics

3.4.1 What are Derived Metrics

Some of the events are difficult to interpret. Sometimes a combination of events provide better information. In the sequel, such a recombination of basic events will be called derived metric. HPM also provides a list of derived metrics, which are defined in section Derived Metrics Description below.

Since each derived metric has its own set of ingredients, not all derived metrics are printed for each group. HPM automatically finds those derived metrics which are computable and prints them. As a convenience to the user, not only the value of the derived metric, but also its definition will be printed out, if the environment variable HPM_PRINT_FORMULA is set.

3.4.2 MFlop Issues

The two most popular derived metrics are the MFlop/s rate and the percentage of peak performance. The default group is chosen to make those two derived metrics available without special choice of the environment variable HPM_EVENT_SET.

For Power5, there is no group that supports enough counters to compute a MFlop/s rate. So an "Algebraic MFlop/s rate" was invented to bridge this gap. This derived metric counts floating point adds, subs and mults (including FMAs), but misses divides and square roots. If the latter two only occur in negligible numbers (which is desirable for a HPC code anyway), the "Algebraic MFlop/s rate" coincides with the usual definition of MFlop/s. Again, different counter groups can be used to check this hypothesis. Group 137 (which exploits the "Algebraic MFlop/s rate") is chosen the default group for Power5.

The derived metric "percent of peak performance" was added new in this version. For all, PPC970, Power4 and Power5, this is based on user time rather than wall clock time, as the result stays correct if multithreaded (e.g. OpenMP) applications are run. For Power5, this is based on "Algebraic MFlop/s". This has not been tested on AIX 5.3, particularly when SMT is active.

Flip-flop blues: Compared to the previous version, the meanings of MFlip/s and MFlop/s are swapped. Not only many users were confused by the name "MFlip/s" being used for what they expected to be MFlop/s. Also Ptools (notably PAPI) is using a different nomenclature. HPM derived metric naming was changed to conform with this emerging standard.

On Power4, PPC970 and Power6 also weighted MFlop/s are available. These are like odinary MFlop/s, except that divisions enter the evaluation with a weight different from the other floating point operations. The weight factor is provided by the user through the evironment variable HPM_DIV_WEIGHT. If set to 1, the weighted MFlop/s coincide with the ordinary MFlop/s. HPM_DIV_WEIGHT can take any positive integer number.

If this environment variable is not set, no weighted MFlop/s are computed.

3.4.3 Latency Computation

In addition, users can provide estimations of memory, cache, and TLB miss latencies for the computation of derived metrics, with the following environment variables (please notice that not all flags are valid in all systems):

HPM_MEM_LATENCY latency for a memory load.

HPM_AVG_L3_LATENCY average latency for an L3 load.

HPM_L3_LATENCY latency for an L3 load within a MCM.

HPM_L35_LATENCY latency for an L3 load outside of the MCM.

HPM_AVG_L2_LATENCY average latency for an L2 load.

HPM_L2_LATENCY latency for an L2 load from the processor.

HPM_L25_LATENCY latency for an L2 load from the same MCM.

HPM_L275_LATENCY latency for an L2 load from another MCM.

HPM_TLB_LATENCY latency for a TLB miss.

When computing derived metrics that take into consideration estimated latencies for L2 or L3, the HPM Toolkit will use the provided "average latency" only if the other latencies for the same cache level are not provided. For example, it will only use the value set in HPM_AVG_L3_LATENCY, if at least one of the values of HPM_ L3_LATENCY and HPM_ L35_LATENCY is not set.

If environment variables are frequently used, they can be collected in a file like this.

```
$ cat docs/examples/HPM\_flags.env-example
export HPM\_MEM\_LATENCY=400
export HPM\_L3\_LATENCY=102
export HPM\_L35\_LATENCY=150
export HPM\_L2\_LATENCY=12
export HPM\_L25\_LATENCY=72
export HPM\_L275\_LATENCY=108
export HPM\_TLB\_LATENCY=700
export HPM\_DIV\_WEIGHT=5
```

The following command would enable these environment variables.

```
$ . cat docs/examples/HPM\_flags.env-example
```

3.5 Inheritance

The "counter virtualization" and the group (i.e. set of events) that is actually monitored is inherited from the process to any of its children, inparticular threads that are spawned vi OpenMP. But there are differences among the various operating systems.

- On AIX all counter values of a process group can be collected.
- On Linux and Blue Gene counter values are only available to the parent, when the child has finished.

To make use of that concept, libhpm provides two flavors of start and stop functions.

- hpmStart and hpmStop start and stop counting on all processes and threads of a process group.
- hpmTstart and hpmTstop start and stop counting only for the thread from which they are called.

On Linux and Blue Gene the first flavor of start and stop routines can not be properly implemented, because the parent has no access to the counting environment of the child before this child has ended. Therefore the functionality of hpmStart/hpmStop is disabled on Linux and Blue Gene. The calls to hpmStart/hpmStop are folded into calls to hpmTstart/hpmTstop. As a result, they are identical and can be freely mixed on Linux and Blue Gene. This is, however, not advisable to do, as an instrumentation like this would not port to AIX.

3.6 Inclusive and Exclusive Values

3.6.1 What are Exclusive Values ?

For a motivation of the term "exclusive values" please look at sub section 3.1 Quick Start above. This program snippet provides an example of two properly nested instrumentation sections. For section 1 we can consider the exclusive time and and exclusive counter values. By that we mean the difference of the values for section 1 and section 2. The original values for section 1 would be called inclusive values for matter of distinction. The terms inclusive and exclusive for the embracing instrumentation section are chosen to indicate whether counter values and times for the contained sections are included or excluded.

Of course the extra computation of exclusive values generates overhead which is not always wanted. Therefore the computation of exclusive values is only carried out if the environment variable HPM_EXCLUSIVE_VALUES is set to Y[...], y[...] or 1 or if the HPM_ONLY_EXCLUSIVE parameter is used as outlined in the following section.

The exact definition of "exclusive" is based on parent-child relations among the instrumented sections. Roughly spoken, the exclusive value for the parent is derived from the inclusive value of the parent reduced by the inclusive value of all children. In previous versions of HPM, instrumented sections had to be properly nested. This generated a natural parent-child relation. Since the children didn't overlap (if there was more than one child anyway), the exclusive duration was naturally defined to be the inclusive duration of the parent minus the sum of the inclusive durations of all children.

In this version of HPM, instrumented sections need not be to be properly nested, but can overlap in arbitrary fashion. Unfortunately, this destroys (or at least obscures) the natural parent-child relations among instrumented sections and complicates the definition of exclusive values.

3.6.2 Parent-Child Relations

HPM provides an automatic search for parents, which is supposed to closely mimic the behavior for strictly nested instrumented regions. It roughly follows the way that the previous version of HPM was going. For strictly nested instrumented sections, the call to hpmStart or hpmTstart for the parent has

to occur prior to the corresponding call for the child. In a multi-threaded environment, however, this causes problems, if the children are executed on different threads. In a kind of race condition, a child may mistake his brother for his father. This generates weird parent child relations, which change with every execution of the program. Actually, the previous version of HPM could exhibit this behavior in certain circumstances. To avoid the race condition safely, the search is restricted to calls from the own thread only, as only these exhibit a race condition free call history. The parent found in this history is the last call of the same kind (i.e. both were started with hpmStart or both are started with hpmTstart or their corresponding FORTRAN equivalences) that has not posted a matching hpmStop or hpmTstop meanwhile. If no parent is found that matches these rules, the child is declared an orphan. As a consequence, automatic parent child relations are never established across different threads.

There may be situations where the automatic parent child relations prove unsatisfactory. To help this matter, new calls in the HPM API have been introduced to enable the user to establish the relations of his choice. These functions are hpmStartx and hpmTstartx and their FORTRAN equivalents. The additional "x" in the function name could be interpreted as "extended" or "explicit". The first two parameters of this function are the instrumented section ID and the ID of the parent instrumented section.

The user has the following choices for the parent ID.

HPM_AUTO_PARENT This triggers the automatic search and is equivalent the classical start routines hpmStart and hpmTstart. Indeed the classical routines are implemented through calls to hpmStartx and hpmTstartx with parent ID specified as HPM_AUTO_PARENT.

HPM_ONLY_EXCLUSIVE This is essentially the same as the previous HPM_AUTO_PARENT, but sets the exclusive flag to TRUE on this instance only. The environment variable HPM_EXCLUSIVE_VALUES sets this flag globally for all instrumented sections.

HPM_NO_PARENT This suppresses any parent child relations.

an integer This has to point to an instrumented section with the following restrictions.

- It has to exist when the call to hpmStartx or hpmTstartx is made.

- It has to be of the same kind (i.e. both were started with hpmStart or both are started with hpmTstart or their corresponding FORTRAN equivalences)

Otherwise HPM will exit with an error message like the following.

```
hpmcount ERROR - Illegal instance id specified
```

In the very last consequence, the user is responsible for establishing meaningful parent child relations. The good news is that these parent child relations only affect the computation of exclusive values. In the worst case, these are just bogus, but this wouldn't effect any other result.

3.6.3 Handling of Overlap Issues

As the user can establish almost arbitrary parent child relations, the definition of the explicit duration or explicit counter values is far from being obvious.

Each instrumented section can be represented by the corresponding subset of the time line. Actually this subset is a finite union of intervals with the left or lower boundaries marked by calls to hpmStart[x]/hpmTstart[x], the right or upper boundaries are marked by calls to hpmStop/hpmTstop. The duration is just the accumulated length of this union of intervals. The counter values are the number of those events that occur within this subset of time.

Basically, the exclusive times and values are the times and values, when no child has a concurrent instrumented section. Hence the main step in defining the meaning of exclusive values is defining the subset of the time line to which they are associated. This is done in several steps.

- Represent the parent and every child by the corresponding subset of the time line (henceforth called the parent set and the child sets).
- Take the union of the child sets.
- Reduce the parent set by the portion that is overlapping with this union.
- Using set theoretic terms, we take the difference of the parent set with the union of the child sets.

The exclusive duration is just the accumulated length of the resulting union of intervals. The exclusive counter values are the number of those events that occur within this subset of time.

3.6.4 Computation of Exclusive Values for Derived Metrics

The task of computing exclusive values for derived metrics may sound complicated at first. But it is actually very simple, given the work already done in the previous subsections. The basic observation is that we are given a subset of the time line that is associated to the notion of "exclusive values". There is no need to care about the way how this set was constructed; just assume the interval boundaries are marked by calls to hpmStart and hpmStop for a new "virtual" instrumented section. In this case it is obvious how to compute the derived metrics - just apply the usual definitions !

3.7 Function Reference

The subroutines provided by libhpm have no return value. In case of some error the global integer variable hpm_error_count is updated. The user is encouraged to check this variable to have his application gracefully exit from HPM errors. As it is difficult for FORTRAN programs to accesss C global variables, there is a logical function

```
f_hpm_error()
```

which returns TRUE if an error occurred inside libhpm and FALSE otherwise. For parallel FORTRAN programs the user is encouraged to implement a subroutine like this.

```
C---------------------- Error Check --------------------
      SUBROUTINE F_HPM_ERRCHK

      INTEGER ierr
      LOGICAL f_hpm_error
      EXTERNAL f_hpm_error

      IF ( f_hpm_error() ) THEN
```

```
         CALL MPI_Abort(MPI_COMM_WORLD, 4, ierr)
      END IF

      RETURN
      END
```

A parallel C version would look like this.

```
#ifdef HPM
#include "libhpm.h"
#define hpm_errchk if (hpm_error_count) MPI_Abort(MPI_COMM_WORLD,4)
#endif /* HPM */
```

The API provided by libhpm is now descibed in detail.

```
    hpmInit( taskID, progName )
  f_hpminit( taskID, progName )
```

- taskID is an integer value. It is now depreciated. In earlier version this was indicating the node ID. It is no longer used and can be set to any value.
- progName is a string with the program name. If the environment Variable HPM_OUTPUT_NAME is not set, this string will be used as a default value for the output name.

```
    hpmStart( instID, label )
  f_hpmstart( instID, label )
```

- instID is the instrumented section ID. It should be > 0 and ≤ 1000.
- To run a program with more than 1000 instrumented sections, the user should set the environment variable HPM_NUM_INST_PTS. In this case, instID should be less than the value set for HPM_NUM_INST_PTS.
- Label is a string containing a label, which is displayed by PeekPerf.

```
    hpmStartx( instID, par\_ID, label )
  f_hpmstartx( instID, par\_ID, label )
```

- instID is the instrumented section ID. It should be > 0 and ≤ 1000.
- To run a program with more than 1000 instrumented sections, the user should set the environment variable HPM_NUM_INST_PTS. In this case, instID should be less than the value set for HPM_NUM_INST_PTS.
- par_ID is the instrumentation ID of the parent section (see section 3.6 Inclusive and Exclusive Values)
- Label is a string containing a label, which is displayed by PeekPerf.

```
  hpmStop( instID )
f_hpmstop( instID )
```

- For each call to hpmStart, there should be a corresponding call to hpmStop with matching instID.
- If not provided explicitely, in implicit call to hpmStop will be made at hpmTerminate.

```
  hpmTstart( instID, label )
f_hpmtstart( instID, label )
  hpmTstartx( instID, par\_ID, label )
f_hpmtstartx( instID, par\_ID, label )

  hpmTstop( instID )
f_hpmtstop( instID )
```

- In order to instrument threaded applications, one should use the pair hpmTstart and hpmTstop to start and stop the counters independently on each thread. Notice that two distinct threads using the same instID will generate an error. See the Section 3.11 on multi-threaded issues for examples.

```
  hpmGetTimeAndCounters( numCounters, time, values )
f_GetTimeAndCounters ( numCounters, time, values )
  hpmGetCounters( values )
f_hpmGetCounters ( values )
```

- These functions have been temporarily disabled in this release. They will be reintroduced in the next release.

```
  hpmTerminate( taskID )
f_hpmterminate( taskID )
```

- All active instrumented code sections will receive an hpmStop
- This function will generate the output.
- If the program exits without calling hpmTerminate, no performance information will be generated.

3.8 Measurement Overhead

As in previous versions of HPM, the instrumentation overhead is caught by calls to the wall clock timer at entry and exit of calls to hpmStart[x], hpmStop, hpmTstart[x], hpmTstop. The previous version tried to eliminate (or hide) the overhead from the measured results. The current version just prints the timing of the accumulated overhead (separate for every instrumented section) in the ASCII output (*.hpm file).

The idea is to let the user decide what to do with this information.

- If the overhead is several orders of magnitude smaller than the total duration of the instrumented section, he can safely ignore the overhead timing.
- If the overhead is in the same order as the total duration of the instrumented section, he may wonder whether to trust the results anyway.
- If the overhead is within 20% of the measured wall clock time, a warning is printed to the ASCII output file.

To make the use of libhpm thread safe, mutexes are set around each call to hpmStart[x], hpmStop, hpmTstart[x], hpmTstop, which adds to the measurement overhead. If the user is running on one thread only, the setting of the mutexes can be suppressed by setting the environment variable HPM_USE_PTHREAD_MUTEX to '0', 'no' or 'NO'.

3.9 Output

If no environment variable is specified, libhpm will write two files. These contain (at least roughly) the same information, but use different formats.

- The file name can be specified via environment variable HPM_OUTPUT_NAME=<name>.
- If HPM_OUTPUT_NAME is not set, the string "progName" as specified in the second parameter to hpmInit is taken as default (see sub section 3.7 Function Reference above).
- The name <name>is actually expanded into three different file names.

  <name>.hpm is the file name for ASCII output — which is basically a one-to-one copy of the screen output.

  <name>.viz is the filename for XML output.

  <name>.csv is the filename for output as comma separated value file. This is not yet implemented in the current release.

- Which of these output files are generated is governed by three additional environment variables. If none of those are set, the ASCII and the XML output is generated. If at least one is set, the following rules apply.

  HPM_ASC_OUTPUT If set to 'Y[...]', 'y[...]' or '1', triggers the ASCII output.

  HPM_VIZ_OUTPUT If set to 'Y[...]', 'y[...]' or '1', triggers the XML output.

  HPM_CSV_OUTPUT If set to 'Y[...]', 'y[...]' or '1', triggers the csv output. This is not yet implemented in the current release.

- The filename can be made unique by setting the environment variable HPM_UNIQUE_FILE_NAME=1. This triggers the following changes.
  - A string

    ```
    _<hostname>_<process_id>_<date>_<time>
    ```

    is inserted before the last '.' in the file name.
  - If the file name has no '.', the string is appended to the file name.
  - If the only occurrence of '.' is the first character of the file name, the string is prepended, but the leading '_' is skipped.

  - If the host name contains '.' ("long form"), only the portion preceding the first '.' is taken. In case a batch queuing system is used, the host name is taken from the execution host, not the submitting host.
  - Similarly for MPI parallel programs, the host name is taken from the node where the MPI task is running. The addition of the process_id enforces different file names for MPI tasks running on the same node.
  - If used for an MPI parallel program, hpmcount tries to extract the MPI task id (or MPI rank with respect to MPI_COMM_WORLD) from the MPI environment. If successful, the process_id is replaced with the MPI task id.
  - The date is given as dd.mm.yyyy, the time is given by hh.mm.ss in 24h format using the local time zone.

- By default there is no output to stdout. This can be changed via setting the environment variable HPM_STDOUT to either '1', 'yes' or 'YES'. In this case, the contents of the ASCII output file are copied to stdout.
- If the environment variable HPM_PRINT_FORMULA is set to either '1', 'yes' or 'YES', the ASCII output of each derived metric is followed by the formula used to compute this derived metric. For details on derived metrics see section 3.4.

3.10 Examples of Use

The following examples show how to instrument a code with calls to libhpm. For details on how to compile and link an application with Libhpm please go to section 3.13 on compiling and linking.

3.10.1 C and C++

```
declaration:
     #include "libhpm.h"
use:
     hpmInit( 0, "my program" );
     if (hpm_error_count) exit(4);
     hpmStart( 1, "outer call" );
```

```
if (hpm_error_count) exit(4);
do\_work();
hpmStart( 2, "computing meaning of life" );
if (hpm_error_count) exit(4);
do\_more\_work();
hpmStop( 2 );
if (hpm_error_count) exit(4);
hpmStop( 1 );
if (hpm_error_count) exit(4);
hpmTerminate( taskID );
if (hpm_error_count) exit(4);
```

The syntax for C and C++ is the same. libhpm routines are declared as having extern "C" linkage in C++.

3.10.2 Fortran

Fortran programs should call the functions with prefix ?f_?. Also, notice that the following declaration is required on all source files that have instrumentation calls.

```
declaration:
     #include "f_hpm.h"
use:
     call f_hpminit( 0, "my_program" )
     if (hpm_error_count > 0) call exit(4)
     call f_hpmstart( 1, "Do Loop"  )
     if (hpm_error_count > 0) call exit(4)
     do <some_loop_header>
       call do_work()
       call f_hpmstart( 5, "computing meaning of life" );
       if (hpm_error_count > 0) call exit(4)
       call do_more\_work();
       call f_hpmstop( 5 );
       if (hpm_error_count > 0) call exit(4)
     end do
     call f_hpmstop( 1 )
     if (hpm_error_count > 0) call exit(4)
     call f_hpmterminate( taskID )
```

```
        if (hpm_error_count > 0) call exit(4)
```

3.11 Multi-Threaded Program Instrumentation Issues

When placing instrumentation inside of parallel regions, one should use different ID numbers for each thread, as shown in the following Fortran example:

```
!$OMP PARALLEL
!$OMP&PRIVATE (instID)
      instID = 30+omp\_get\_thread\_num()
      call f\_hpmtstart( instID, "computing meaning of life" )
!$OMP DO
      do ...
        do\_work()
      end do
      call f\_hpmtstop( instID )
!$OMP END PARALLEL
```

If two threads are using the same ID numbers for call to hpmTstart or hpmTstop, libhpm will exit with the following error message.

```
   hpmcount ERROR - Instance ID on wrong thread
```

There is a subtlety for currently only AIX, which supports true hpmStart/ hpmStop routines. See the subsection 3.5 Inheritance for an explanation of the difference between hpmStart/hpmStop and hpmTstart/hpmTstop. It is legal to use hpmStart/hpmStop inside of a parallel region, but this use is strongly discouraged. To understand this, one has to recall that the different threads of a workgroup run totally unsynchronized. A call to hpmStart would not only record the values of the calling thread, but also of all other threads in the same process group. Only the calling thread is at a defined point in the program; the other threads might be at any other statement in the parallel region. So basically, numbers of apples are added to numbers of pears, and the result is almost ever some bogus number. Thus, hpmTstart/hpmTstop should be used inside parallel regions.

Outside of the parallel region (in particular embracing a parallel section), it can well make sense to use hpmStart/hpmStop. In a parallel section, several

threads are spawned to take over part of the workload. hpmStart/hpmStop would catch the activity of all spawned threads, hpmTstart/hpmTstop only considers the thread that is active before and after the parallel section.

3.12 Considerations for MPI Parallel Programs

3.12.1 General Considerations

libhpm is inherently sequential, looking only at the hardware performance counters of a single process (and its children, as explained in section 3.5 Inheritance). When started with "poe" or "mpirun", each MPI task is doing its own hardware performance counting and these instances are completely ignorant of each other — unless additional action is taken as described in the following sub sections. Consequently, each instance is writing its own output. If the environment variable HPM_OUTPUT_NAME is used, each instance is using the same file name, which results in writing into the same file, if a parallel file system is used. Of course, this can be (and should be) prevented by making the file names unique through the HPM_UNIQUE_FILE_NAME environment variable. Still it might be an unwanted side effect to have that many output files.

For this reason the the environment variable HPM_AGGREGATE triggers some aggregation before (possibly) restricting the output to a subset of MPI tasks. This formulation is deliberately vague, because there can be many ways to aggregate hardware performance counter information across MPI tasks. One way is to take averages, but maximum or minimum values could be also thought of. The situation is further complicated by allowing to run different groups on different MPI tasks. On architectures that allow for multiplexing (as described in section 3.3 Multiplexing), some tasks could use multiplexing, others may not. Of course averages, maximum and minimum values should only be taken on groups which are alike.

Therefore the environment variable HPM_AGGREGATE take a value, which is the name of a plug-in that defines the aggregation strategy. Each plug-in is a shared object file containing two functions called distributor and aggregator.

On Blue Gene/L there are no shared objects. Therefore the plug-ins are simple object files. The HPM_AGGREGATE environment variable is not used on Blue Gene/L, but the plug-ins are statically linked with the library.

3.12.2 Distributors

The motivating example for the distributor function is allowing a different hardware counter group on each MPI task. Therefore, the distributor is a subroutine that determines the MPI task id (or MPI rank with respect to MPI_COMM_WORLD) from the MPI environment for the current process, and (re)sets environment variables depending on this information. The environment variable may be any environment variable, not just HPM_EVENT_SET, which motivated this function.

Consequently, the distributor is called before any environment variable is evaluated by HPM. The settings of the evironment variables done in the distributor take precedence over global settings.

Of course, the aggregator has to adapt to the HPM group settings done by the distributor. This is why distributors and aggregators always come in pairs. Each plug-in is containing just one such pair.

3.12.3 Aggregators

The motivating example is the aggregation of the hardware counter data across the MPI tasks. In the simplest case this could be an average of the corresponding values. Hence this function is called

- after the hardware counter data have been gathered,
- before the derived metrics are computed.
- before these data are printed,

In a generalized view, the aggregator is taking the raw results and rearranges them for output.

Also, depending on the information of the MPI task id (or MPI rank with respect to MPI_COMM_WORLD) the aggregator sets (or doesn't set) a flag to mark the current MPI task for HPM printing.

3.12.4 Plug-ins shipped with the Tool Kit

The following plug-ins are shipped with the toolkit. They can be found in

`$(IHPCT_BASE)/lib or $(IHPCT_BASE)/lib64`

mirror.so is the plug-in that is called when no plug-in is requested. The aggregator is mirroring the raw hardware counter data in a one-to-one fashion into the output function. Hence this name. It is also flagging each MPI task as printing task. The corresponding distributor is a void function. This plug-in doesn't use MPI and also works in a non-MPI context.

loc_merge.so does a local merge on each MPI task separately. It is identical to the mirror.so plug-in except for those MPI tasks that change the hardware counter groups in the course of the measurement (e.g. by multi-plexing). The different counter data, which are collected for only part of the measuring interval, are proportionally extended to the whole interval and joined into one big group that is entering derived metrics computation. This way, more derived metrics can be determined at the risk of computing garbage. The user is responsible for using this plug-in only when it makes sense to use it. It is also flagging each MPI task as printing task. The corresponding distributor is a void function. This plug-in doesn't use MPI and also works in a non-MPI context.

single.so does the same as mirror.so, but only on MPI task 0. The output on all other tasks is discarded. This plug-in uses MPI functions and can't be used in a sequential context.

average.so is a plug-in for taking averages across MPI tasks. The distributor is reading the environment variable HPM_EVENT_SET (which is supposed to be a comma separated list of group numbers) and distributes these group numbers in a round robin fashion to the MPI tasks. The aggregator is first building a MPI communicator of all tasks with equal hardware performance counting scenario. The communicator groups may be different from the original round robin distribution. This may happen if the conting group has been changed on some of the MPI tasks after the first setting by the distributor. Next the aggregator is taking the average across the subgroups formed by this communicator. Finally it is flagging the MPI rank 0 in each group as printing host. This plug-in uses MPI functions and can't be used in a sequential context.

3.12.5 User defined Plug-ins

There can be no doubt that this set of plug-ins can only be a first starter kit and many more might be desirable. Rather than taking the average one could think of taking maximum or minimum. There is also the possibility of taking kind of a "history_merge.so" by blending in results from previous measurements. Chances are that however big the list of shipped plug-ins may be, the one just needed is missing from the set ("Murphy's law of HPM plug-ins"). The only viable solution comes with disclosing the interface between plug-in and tool and allowing for user defined plug-ins.

The easiest way to enable users to write their own plug-ins is by providing examples. Hence the plug-ins described above are provided in source code together with the Makefile that was used to generate the shared objects files. These files can be found in the following place.

```
$(IHPCT_BASE)/examples/plugins
```

3.12.6 Detailed Interface Description

Each distributor and aggregator is a function returning an integer which is 0 on success and $! = 0$ on error. In most cases the errors occur when calling a system call like malloc(), which sets the errno variable. If the distributor or aggregator returns the value of errno as return code, the calling HPM tool sees to an expansion of this errno code into a readable error message. If returning the errno is not viable, the function should return a negative value.

The function prototypes are defined in the following file.

```
$(IHPCT_BASE)/include/hpm_agg.h
```

This is a very short file with the following contents.

```
#include "hpm_data.h"

int distributor(void);

int aggregator(int num_in, hpm_event_vector in,
               int *num_out, hpm_event_vector *out,
               int *is_print_task);
```

The distributor has no parameters and is only required to (re)set environment variables (via setenv()).

The aggregator takes the current hpm values on each task as an input vector *in* and returns the aggregated values on the output vector *out* on selected or all MPI tasks. To have utmost flexibility, the aggregator is responsible to allocate the memory needed to hold the output vector *out*. The definition of the data types used for *in* and *out* are provided in the following file.

```
$(IHPCT_BASE)/include/hpm_data.h
```

Finally the aggregator is supposed to set (or unset) a flag to mark the current MPI task for HPM printing.

Form the above definitions it is apparent that the interface is defined in C-Language. While it is in principle possible to use another language for programming plug-ins, the user is responsible for using the same memory layout for the input and output variables. There is no explicit FORTRAN interface provided.

The hpm_event_vector *in* is a vector or list of *num_in* entries of type *hpm_data_item*. The latter is a struct containing members that describe the definition and the results of a single hardware performance counting task.

```
/*      NAME                INDEX */

#define HPM_NTIM 8
#define HPM_TIME_WALLCLOCK      0
#define HPM_TIME_CYCLE          1
#define HPM_TIME_USER           2
#define HPM_TIME_SYSTEM         3
#define HPM_TIME_START          4
#define HPM_TIME_STOP           5
#define HPM_TIME_OVERHEAD       6
#define HPM_TIME_INIT           7


typedef struct {
  int              num_data;
  hpm_event_info  *data;
  double           times[HPM_NTIM];
```

```
  int              is_mplex_cont;
  int              is_rusage;
  int              mpi_task_id;
  int              instr_id;
  int              count;
  int              is_exclusive;
  int              xml_element_id;
  char            *description;
  char            *xml_descr;
} hpm_data_item;

typedef hpm_data_item *hpm_event_vector;
```

- Counting the events from a certain HPM group on one MPI task is represented by a single element of type *hpm_data_item*.
- If multiplexing is used, the results span several consecutive elements, each dedicated to one HPM group that take part in the multiplex setting. On all but the first element the member *is_mplex_cont* is set to *TRUE_* to indicate that these elements are continuations of the first element belonging to the same multiplex setup.
- If HPM groups are changed during the measurement, the results for different groups are recorded in different vector elements, but no *is_mplex_cont* flag is set. This way results obtained via multiplexing can be distinguished from results obtained by ordinary group change.
- If several instrumented sections are used, each instrumented code section will use separate elements of type *hpm_data_item* to record the results. Each of these will have the member *instr_id* set with the first argument of hpmStart and the logical member *is_exclusive* set to *TRUE_* or *FALSE_* depending on whether the element hold inclusive or exclusive counter results (see section 3.6 Inclusive and Exclusive Values for details). Then all these different elements are concatenated into a single vector.
- Finally, the data from a call to getrusage() is prepended to this vector. So the rusage data form the vector element with index 0. This vector element is the only element with struct member *is_rusage* set to *TRUE_* to distinguish it from ordinary hardware performance counter data.

The output vector is of the same format. Each vector element enters the derived metrics computation separately (unless *is_rusage* == *TRUE_*). Then all vector elements (and the corresponding derived metrics) are printed in the order given by the vector *out*. The output of each vector element will be preceded by the string given in member *description* (which may include line feeds as appropriate). The XML output will be marked with the text given in *xml_descr*.

This way the input vector *in* is providing a complete picture of what has been measured on each MPI task. The output vector *out* is allowing complete control on what is printed on which MPI task in what order.

3.12.7 Getting the plug-ins to work

The plug-ins have been compiled with the following Makefile

```
$(IHPCT_BASE)/examples/plugins/Makefile
```

using this command.

```
<g>make ARCH=<appropriate_archtitecture>
```

The include files for the various architectures are provided in subdirectory make. The user is asked to note a couple of subtleties.

- The Makefile distinguishes "sequential" (specified in *PLUGIN_SRC*) and "parallel" plug-ins (specified in *PLUGIN_PAR_SRC*). The latter are compiled and linked with the MPI wrapper script for the compiler/ linker. Unlike a static library, generation of a shared object requires linking, not just compilation.
- On BG/L there are no shared objects, so ordinary object files are generated. On BG/L and BG/P just everything is parallel.
- If the MPI software stack requires the parallel applications to be linked with a special start-up code (like poe_remote_main() for IBM MPI on AIX), the shared object has to carry this start-up code. hpmcount is a sequential application. Therefore the start-up code has to be loaded and activated when the plug-in is loaded at run time. This turns the

sequential application hpmcount into a parallel application "on the fly". This sounds complicated but works pretty seamlessly, at least for IBM MPI in user space protocol and MPICH and its variants. No further user action is required to make this work.

- There are, however, some restrictions to be observed when writing plug-in code. The MPI standard document disallows calling MPI_Init() twice on the same process. It appears that this is indeed not supported on the majority of MPI software stacks, not even if an MPI_Finalize() is called between the two invocations of MPI_Init().

- The distributor is called by hpmInit(). If it would contain MPI calls, this would enforce to have MPI_Init() prior to hpmInit(). To lift this restriction, the distributor must not call any MPI function. If the distributor is supposed to work with hpmcount as well, this restriction is required as well (see 2.7.7 for details). The MPI task id should be extracted by inspecting environment variables that have been set by the MPI software stack.

- The aggregator, however, usually can't avoid calling MPI functions. Before calling MPI_Init(), it has to check whether the instrumented application has already done so. If the instrumented application is an MPI application, it cannot be called after MPI_Finalize(). The aggregator is called by hpmTerminate(). Hence hpmTerminate() has to be called between the calls to MPI_Init() and MPI_Finalize().

- libhpm uses a call to dlopen() to access the plug-in and makes use of its functions. There is no dlopen() on Blue Gene/L, so plug-ins are statically linked to the application. On Blue Gene/P, both ways to access the plug-ins can be used.

3.13 Compiling and Linking

3.13.1 Dynamic Linking

libhpm inspects the value of `HPM_AGGREGATE` at run time and links to the appropriate distributor and aggregator (see section 3.12 for details) via a call to dlopen at run time. This in general requires dynamic linking of the application. In many cases this is transparent to the user.

The easiest way to find the right compile and link options is to inspect the examples.

```
. /usr/local/ihpct_2.2/env_sh
# replace the above PATH with the actual location
# where HPCT is installed.
cp -r $IHPCT_BASE/examples/hpm .
cd hpm
ls make
gmake ARCH=power_aix swim_mpi redBlackSOR
# replace power_aix with the actual architecture
# you are compiling on.
# inspect the sub directory make for getting
# the right value for ARCH.
```

On AIX the following compile and link statement can be observed from the above gmake command. For better readability, the echoes from gmake are splitted over several lines. Also PATH names have been replaced with the environment variable `$IHPCT_BASE` wherever appropriate.

```
mpxlf_r -c  -O3 -q64 -g -I$IHPCT_BASE/include \
      -qsuffix=cpp=f -qrealsize=8 -WF,-DHPM swim_mpi.f
mpxlf_r -O3 swim_mpi.o -o swim_mpi -q64 \
      -lxlf90 -L$IHPCT_BASE/lib64 -lhpm -llicense \
      -lpmapi -lm

xlc_r -O3 -DHPM -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c redBlackSOR.c
xlc_r -O3 -DHPM -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c relax.c
xlc_r -O3 -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT redBlackSOR.o relax.o \
      -o redBlackSOR -q64 -lxlf90 -L$IHPCT_BASE/lib64 \
      -lhpm -llicense -lpmapi -lm
```

On Linux, the same gmake command leads to the following statements. Please note that compared to the statements above, there is no `-lpmapi` (or rather `-lperfctr`). This is due to the different way how the GNU loader handles unresolved external references in subsequent libraries.

```
mpfort -c  -O3 -q64 -g -I$IHPCT_BASE/include \
     -qsuffix=cpp=f -qrealsize=8 -WF,-DHPM swim_mpi.f
mpfort -O3 swim_mpi.o -o swim_mpi -q64 \
     -L$IHPCT_BASE/lib64 -lhpm -llicense

xlc_r  -O3 -DHPM -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c redBlackSOR.c
xlc_r  -O3 -DHPM -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c relax.c
xlc_r -O3 -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT redBlackSOR.o relax.o \
      -o redBlackSOR -q64 -L$IHPCT_BASE/lib64 -lhpm -llicense
```

Finally, on BlueGene/P, the following statements can be observed. Please note the `-dynamic` in the linker statement. This is necessary, because the default linkage is static on BlueGene.

```
/bgsys/drivers/ppcfloor/comm/bin/mpif90 -c \
      -O3 -g -I$IHPCT_BASE/include \
      -x f77-cpp-input -std=gnu -Wp,-DHPM swim_mpi.f
/bgsys/drivers/ppcfloor/comm/bin/mpif90 -O3 \
       swim_mpi.o -o swim_mpi -dynamic \
       -L$IHPCT_BASE/lib -llicense -lhpm

/bgsys/drivers/ppcfloor/comm/bin/mpicc -O3 -DHPM -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       -c redBlackSOR.c
/bgsys/drivers/ppcfloor/comm/bin/mpicc -O3 -DHPM -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       -c relax.c
/bgsys/drivers/ppcfloor/comm/bin/mpicc -O3 -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       redBlackSOR.o relax.o -o redBlackSOR -dynamic \
       -L$IHPCT_BASE/lib -llicense -lhpm
```

Please note that on BlueGene/P, the targets involving OpenMP do not compile with the options given in the example Makefile.

If `mpicc` is being replaced by `mpixlc_r`, the `-dynamic` has to be replaced by `-qnostaticlink`. There is no corresponding option for `mpixlf90_r`.

```
/bgsys/drivers/ppcfloor/comm/bin/mpixlc_r -O3 -DHPM -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       -c redBlackSOR.c
/bgsys/drivers/ppcfloor/comm/bin/mpixlc_r -O3 -DHPM -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       -c relax.c
/bgsys/drivers/ppcfloor/comm/bin/mpixlc_r -O3 -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       redBlackSOR.o relax.o -o redBlackSOR -qnostaticlink \
       -L$IHPCT_BASE/lib -llicense -lhpm
```

Also, please note that on BlueGene/P, environment variables have to be passed to the MPI tasks. This is demonstrated in the following example.

```
export LD_LIBRARY_PATH=\
     $IHPCT_BASE/lib:/bgsys/drivers/ppcfloor/runtime/SPI
mpirun -np 8 -mode VN -partition r000n120-c16i1 \
     -env BG_MAXALIGNEXP=1 \
     -env LD_LIBRARY_PATH=$LD_LIBRARY_PATH \
     -env HPM_UNIQUE_FILE_NAME=yes \
     -env HPM_AGGREGATE=average.so ./swim_mpi
```

3.13.2 Static Linking

If, for some reason, dynamic linking is not an option, there is also a "plan b". For instance, the current version 8.4.1 of TotalView on BlueGene/P does not support dynamic linking. Also the targets involving OpenMP in the above example appear to require a different linkage.

To do this, the call to dlopen (as well as the other dlxxx calls) are linked to a fake library. This fake library turns the aggregator and distributor into unresolved external references. Hence the plugin needs to be statically linked to the application as well. As a result, the environment variable `HPM_AGGREGATE` will be disregarded.

This is easiest and most needed on BlueGene/P. Here the following link statements will do. The second example is a serial program, so taking the average plugin does not make sense - although is does not hurt, as on BG/P, everything is a parallel program.

```
/bgsys/drivers/ppcfloor/comm/bin/mpif90 -c \
      -O3 -g -I$IHPCT_BASE/include \
      -x f77-cpp-input -std=gnu -Wp,-DHPM swim_mpi.f
/bgsys/drivers/ppcfloor/comm/bin/mpif90 -O3 \
       swim_mpi.o -o swim_mpi \
       -L$IHPCT_BASE/lib -lhpm -llicense \
       $IHPCT_BASE/lib/fake_dlfcn.o \
       $IHPCT_BASE/lib/average.o

/bgsys/drivers/ppcfloor/comm/bin/mpicc -O3 -DHPM -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       -c redBlackSOR.c
/bgsys/drivers/ppcfloor/comm/bin/mpicc -O3 -DHPM -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       -c relax.c
/bgsys/drivers/ppcfloor/comm/bin/mpicc -O3 -g \
       -I$IHPCT_BASE/include -DHPM -DDYNAMIC -DPRINT \
       redBlackSOR.o relax.o -o redBlackSOR \
       -L$IHPCT_BASE/lib -lhpm -llicense -lm \
       $IHPCT_BASE/lib/fake_dlfcn.o \
       $IHPCT_BASE/lib/mirror.o
```

On AIX, the commands look just accordingly. The only changes are that libhpm and liblicense are specified explicitely in static form to force the linker to resolve them by static linking. MPI is still linked dynamically.

```
mpxlf_r -c  -O3 -q64 -g -I$IHPCT_BASE/include \
      -qsuffix=cpp=f -qrealsize=8 -WF,-DHPM swim_mpi.f
mpxlf_r -O3 swim_mpi.o -o swim_mpi -q64 \
      $IHPCT_BASE/lib64/libhpm.a \
      $IHPCT_BASE/lib64/fake_dlfcn.o \
      $IHPCT_BASE/lib64/average.o \
      $IHPCT_BASE/lib64/liblicense.a \
      -lpmapi -lm

xlc_r -O3 -DHPM -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c redBlackSOR.c
xlc_r -O3 -DHPM -q64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c relax.c
xlc_r -O3 -q64 -g -I$IHPCT_BASE/include \
```

```
      -DHPM -DDYNAMIC -DPRINT redBlackSOR.o relax.o \
      -o redBlackSOR -q64 -lxlf90 \
      $IHPCT_BASE/lib64/libhpm.a \
      $IHPCT_BASE/lib64/fake_dlfcn.o \
      $IHPCT_BASE/lib64/mirror.o \
      $IHPCT_BASE/lib64/liblicense.a \
      -lpmapi -lm
```

On linux, disabling the dlopen and dlsym calls generates conflicts with other libraries that make use of these functions, like libxlfmath.so. Unfortunately, this rules out the xl-compilers for this exercise. The GNU compilers, however, generate executables that work correctly.

```
export MP_COMPILER=gfortran
mpfort -c  -O3 -m64 -g -I$IHPCT_BASE/include \
      -x f77-cpp-input -std=gnu -Wp,-DHPM swim_mpi.f
mpfort -O3 swim_mpi.o -o swim_mpi -m64 \
      $IHPCT_BASE/lib64/libhpm.a \
      $IHPCT_BASE/lib64/fake_dlfcn.o \
      $IHPCT_BASE/lib64/average.o \
      $IHPCT_BASE/lib64/liblicense.a \
      -L/opt/perfctr-2.7.20/64/lib -lperfctr

gcc  -O3 -DHPM -m64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c redBlackSOR.c
gcc  -O3 -DHPM -m64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT -c relax.c
gcc  -O3 -m64 -g -I$IHPCT_BASE/include \
      -DHPM -DDYNAMIC -DPRINT redBlackSOR.o relax.o \
      -o redBlackSOR \
      $IHPCT_BASE/lib64/libhpm.a \
      $IHPCT_BASE/lib64/fake_dlfcn.o \
      $IHPCT_BASE/lib64/mirror.o \
      $IHPCT_BASE/lib64/liblicense.a \
      -L/opt/perfctr-2.7.20/64/lib -lperfctr -lm
```

4 hpmstat

4.1 Quick Start

hpmstat is a simple system wide monitor based on hardware performance counters. For most of the functionality of hpmstat, root priviliges are required. The usage is very similar to that of the vmstat command. For a start the (root) user simply types the following.

```
hpmstat
```

As a result, hpmstat lists various performance information on the screen (i.e.stdout output). In particular it prints resource utilization statistics, hardware performance counter information and derived hardware metrics.

The resource usage statistics is directly taken from a call to *getrusage*(). For more information on the resource utilization statistics, please refer to the getrusage man pages. In particular, on Linux the man page for *getrusage*() states that not all fields are meaningful under Linux. The corresponding lines in hpmstat output have the value "n/a".

4.2 Usage

Detailed descriptions of selected options

-o <name> copies output to file <name>.

- For parallel programs the user is advised to have a different name for each MPI task (using e.g. the -u flag below) or direct the file to a non-shared file system.

-u make the file name <name>(specified via -o) unique.

- A string

  `_<hostname>_<process_id>_<date>_<time>`

is inserted before the last '.' in the file name.

- If the file name has no '.', the string is appended to the file name.
- If the only occurance of '.' is the first character of the file name, the string is prepended, but the leading '_' is skipped.
- If the hostname contains '.' ("long form"), only the portion preceding the first '.' is taken. In case a batch queuing system is used, the hostname is taken from the execution host, not the submitting host.
- Similarly for MPI parallel programs, the hostname is taken from the node where the MPI task is running. The addition of the process_id enforces different file names for MPI tasks running on the same node.
- The date is given as dd.mm.yyyy, the time is given by hh:mm.ss in 24h format using the local time zone.
- The file name created that way contains a ':' to make the time easily recognizable. Unfortunately, some Windows versions don't like ':' in file names. Please remember to rename the file before transferring it to a Windows system.
- This flag is only active when the -o flag is used.

-n no hpmstat output in stdout when "-o" flag is used.

- This flag is only active when the -o flag is used.

-x adds formulas for derived metrics

-g <group[,<group>, ...]> specify group number(s)

- for an explanation on groups see sections 2.3 on Events and Groups and 2.4 on Multiplexing below.

-a <plug-in> aggregate counters using the plugin <plug-in>.

-l list groups

- for an explanation on groups see section 2.3 on Events and Groups below.

-c list counters and events

- for an explanation on groups see section 2.3 on Events and Groups below.

4.3 Events and Groups

The hardware performance counters information is the value of special CPU registers that are incremented at certain events. The number of such registers is different for each architecture.

Processor Architecture	Number of performance counter registers
PPC970	8
Power4	8
Power5	6
Power5+	6
Power6	6
BG/L	52
BG/P	256

On both AIX and Linux, kernel extensions provide "counter virtualization", i.e. the user sees private counter values for his application. On a technical side, the counting of the special CPU registers is frozen and the values are saved whenever the application process is taken off the CPU and another process gets scheduled. The counting is resumed when the user application gets scheduled on the CPU.

The special CPU registers can count different events. On the Power CPUs the are restrictions which registers can count what events. A call to "hpmstat -c" will list all CPU counting registers and the events they can be monitoring.

Even more, there are lots of rules restricting the concurrent use of different events. Each valid combination of assignments of events to hardware counting registers is called a group. To make handling easier, a list of valid groups is provided. A call to "hpmstat -l" will list all available groups and the events they are monitoring. The -g option is provided to select a secific group to be counted by hpmstat. If -g is not specified, a default group will be taken according to the following table.

Processor Architecture	Number of groups	Default group
PPC970	41	23
Power4	64	60
Power5	148(140)	137
Power5+	152	145
Power6	195	127
BG/L	16	0
BG/P	4	0

The number of groups for Power5 is 140 for AIX 5.2, and 148 for Linux and AIX 5.3. The reason for this difference are different versions of bos.pmapi.The last group (139) was changed and 8 new groups were appended. If HPM is called with

```
hpmstat -g <a_new_group_number>
```

on AIX 5.2, it returns the following error message.

```
hpmstat ERROR - pm_set_program_mygroup:
pm_api : event group ID is invalid
```

5 Reference

5.1 List of Environment Variables

Environment variable	Text reference
HPM_AGGREGATE	2.7.1, 3.12.1
HPM_ASC_OUTPUT	2.2, 3.9
HPM_DIV_WEIGHT	2.5.2, 3.4.2
HPM_EVENT_SET	3.2
HPM_EXCLUSIVE_VALUES	3.6
HPM_L25_LATENCY	2.5.3, 3.4.3
HPM_L275_LATENCY	2.5.3, 3.4.3
HPM_L2_LATENCY	2.5.3, 3.4.3
HPM_L35_LATENCY	2.5.3, 3.4.3
HPM_L3_LATENCY	2.5.3, 3.4.3
HPM_MEM_LATENCY	2.5.3, 3.4.3
HPM_NUM_INST_PTS	3.7
HPM_OUTPUT_NAME	2.2, 3.9
HPM_PRINT_FORMULA	2.2, 3.9, 3.4
HPM_SLICE_DURATION	2.4, 3.3
HPM_STDOUT	2.2, 3.9
HPM_TLB_LATENCY	2.5.3, 3.4.3
HPM_UNIQUE_FILE_NAME	2.2, 3.9
HPM_USE_PTHREAD_MUTEX	3.8
HPM_VIZ_OUTPUT	2.2, 3.9

5.2 Derived Metrics Description

```
Utilization rate
    100.0 * user_time / wall_clock_time

Total FP load and store operations
    fp_tot_ls = (PM_LSU_LDF + PM_FPU_STF) * 0.000001

MIPS
    PM_INST_CMPL * 0.000001 / wall_clock_time
```

```
Instructions per cycle
    (double)PM_INST_CMPL / PM_CYC

Instructions per run cycle
    (double)PM_INST_CMPL / PM_RUN_CYC

Instructions per load/store
    (double)PM_INST_CMPL/(PM_LD_REF_L1 + PM_ST_REF_L1)

% Instructions dispatched that completed
    100.0 * PM_INST_CMPL / PM_INST_DISP

Fixed point operations per Cycle
    (double)PM_FXU_FIN / PM_CYC

Fixed point operations per load/stores
    (double)PM_INST_CMPL/(PM_LD_REF_L1 + PM_ST_REF_L1)

Branches mispredicted percentage
    100.0* (PM_BR_MPRED_CR + PM_BR_MPRED_TA) / PM_BR_ISSUED

number of loads per load miss
    (double)PM_LD_REF_L1 / PM_LD_MISS_L1

number of stores per store miss
    (double)PM_ST_REF_L1 / PM_ST_MISS_L1

number of load/stores per L1 miss
    (  (double)PM_LD_REF_L1 + (double)PM_ST_REF_L1)
    / ((double)PM_ST_MISS_L1 + (double)PM_LD_MISS_L1)

L1 cache hit rate
    100.0 * (1.0 - ((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1)
             / ((double)PM_ST_MISS_L1 + (double)PM_LD_MISS_L1))

number of loads per TLB miss
    (double)PM_LD_REF_L1 / PM_DTLB_MISS

number of loads/stores per TLB miss
    ((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1) / PM_DTLB_MISS
```

```
Total Loads from L2
    tot_ld_L2 = sum((double)PM_DATA_FROM_L2*) / (1024*1024)

L2 load traffic
    L1_cache_line_size * tot_ld_L2

L2 load bandwidth per processor
    L1_cache_line_size * tot_ld_L2 / wall_clock_time

Estimated latency from loads from L2
    ( HPM_L2_LATENCY*(double)PM_DATA_FROM_L2
    + HPM_L25_LATENCY*(double)sum(PM_DATA_FROM_L25*)
    + HPM_L275_LATENCY*(double)sum(PM_DATA_FROM_L275*))
                                              * cycle_time

% loads from L2 per cycle
    100.0 * tot_ld_L2 / PM_CYC

Total Loads from local L2
    tot_ld_l_L2 = (double)PM_DATA_FROM_L2 / (1024*1024)

local L2 load traffic
    L1_cache_line_size * tot_ld_l_L2

local L2 load bandwidth per processor
    L1_cache_line_size * tot_ld_l_L2 / wall_clock_time

Estimated latency from loads from local L2
    HPM_L2_LATENCY * (double)PM_DATA_FROM_L2 * cycle_time

% loads from local L2 per cycle
    100.0 * (double)PM_DATA_FROM_L2 / PM_CYC

Total Loads from L3
    tot_ld_L3 = sum((double)PM_DATA_FROM_L3*) / (1024*1024)

L3 load traffic
    L2_cache_line_size * tot_ld_L3

L3 load bandwidth per processor
```

```
    L2_cache_line_size * tot_ld_L2 / wall_clock_time

Estimated latency from loads from L3
    ( HPM_L3_LATENCY*(double)PM_DATA_FROM_L3
    + HPM_L35_LATENCY*(double)PM_DATA_FROM_L35) * cycle_time

% loads from L3 per cycle
    100.0 * (double)sum(PM_DATA_FROM_L3*) / PM_CYC

Total Loads from local L3
    tot_ld_l_L3 = (double)PM_DATA_FROM_L3 / (1024*1024)

local L3 load traffic
    L2_cache_line_size * tot_ld_l_L3

local L3 load bandwidth per processor
    L2_cache_line_size * tot_ld_l_L3 / wall_clock_time

Estimated latency from loads from local L3
    HPM_L3_LATENCY * (double)PM_DATA_FROM_L3 * cycle_time

% loads from local L3 per cycle
    100.0 * (double)PM_DATA_FROM_L3 / PM_CYC

Total Loads from memory
    tot_ld_mem = (double)PM_DATA_FROM_MEM / (1024*1024)

memory load traffic
    L3_cache_line_size * tot_ld_mem

memory load bandwidth per processor
    L3_cache_line_size * tot_ld_mem / wall_clock_time

Estimated latency from loads from memory
    HPM_MEM_LATENCY * (double)PM_DATA_FROM_MEM * cycle_time

% loads from memory per cycle
    100.0 * (double)PM_DATA_FROM_MEM / PM_CYC

Total Loads from local memory
    tot_ld_lmem = (double)PM_DATA_FROM_LMEM / (1024*1024)
```

```
local memory load traffic
    L3_cache_line_size * tot_ld_lmem

local memory load bandwidth per processor
    L3_cache_line_size * tot_ld_lmem / wall_clock_time

Estimated latency from loads from local memory
    HPM_MEM_LATENCY * (double)PM_DATA_FROM_LMEM * cycle_time

% loads from local memory per cycle
    100.0 (double)PM_DATA_FROM_LMEM / PM_CYC

% TLB misses per cycle
    100.0 * (double)PM_DTLB_MISS / PM_CYC

% TLB misses per run cycle
    100.0 * (double)PM_DTLB_MISS / PM_RUN_CYC

Estimated latency from TLB misses
    HPM_TLB_LATENCY * (double)PM_DTLB_MISS * cycle_time

HW float point instructions (flips)
    (flips = (double) PM_FPU_FIN) * 0.000001      -- or --
    (flips = (double)(PM_FPU0_FIN + PM_FPU1_FIN))* 0.000001

HW float point instructions per cycle
    flips / PM_CYC

HW float point instructions per run cycle
    flips / PM_RUN_CYC

HW floating point instr. rate (HW flips / WCT)
    flips * 0.000001 / wall_clock_time

HW floating point instructions/ user time
    flips * 0.000001 / user_time

Total floating point operations
    (flops = (double)(PM_FPU0_FIN + PM_FPU1_FIN
                      +PM_FPU_FMA  - PM_FPU_STF )) *0.000001
```

```
Flop rate (flops / WCT)
    flops  * 0.000001/ wall_clock_time

Flops / user time
    flops  * 0.000001/ user_time

Algebraic floating point operations
    (aflops = (double)(PM_FPU_1FLOP + 2*PM_FPU_FMA)) * 0.000001

Algebraic flop rate (flops / WCT)
    aflops * 0.000001 / wall_clock_time

Algebraic flops / user time
    aflops * 0.000001 / user_time

Weighted Floating Point operations
    (wflops = flops + (HPM_DIV_WEIGHT-1)*PM_FPU_FDIV) * 0.000001

Weighted flop rate (flops / WCT)
    wflops * 0.000001 / wall_clock_time

Weighted flops / user time
    wflops * 0.000001 / user_time

FMA percentage
    100.0 * 2*PM_FPU_FMA /  flops (on Power4)
    100.0 * 2*PM_FPU_FMA / aflops (on Power5)
    100.0 * 2*PM_FPU_FMA /  flops (on Power6)

Computation intensity
    flops / fp_tot_ls

% of peak performance
    100.0 *  flops * cycle_time/ (4*user_time) (on Power4)
    100.0 * aflops * cycle_time/ (4*user_time) (on Power5)
    100.0 *  flops * cycle_time/ (4*user_time) (on Power6)
```