Agenda

- Overview of performance tools @ RES
- Perfminer
- Profile
- Papiex
- HPMcount
- CPI Breakdown Analisys
- Paraver

Some performance analysis tools

DynInst: is an api to allow dynamic injection of code into a running program

*** **GPROF:** the GNU Profiler. Have some visual interfaces like ***kprof***

HPCToolkit: (http://hpctoolkit.org/)An integrated suite of tools for measurement and analysis of program performance. It works by sampling an execution of a multi-threaded and/or multiprocess program using hardware performance counters

*** **PAPI: Performance** API. A portable interface to hardware performance counters on modern microprocessors

*** **Valgrind:** Is a GPL'd system for debugging and profiling x86-Linux programs. It supports tools to either detect memory management and threading bugs, or profile performance. It works for any language and the assembler. Has some visual front ends

VTUNE: Performance Analyzer tool from Intel Corporation. graph or analyzing a set of tuning events. It works with C/C++/Fortran/.NET/Java and other applications on Linux or Windows, but only when running on selected Intel hardware.

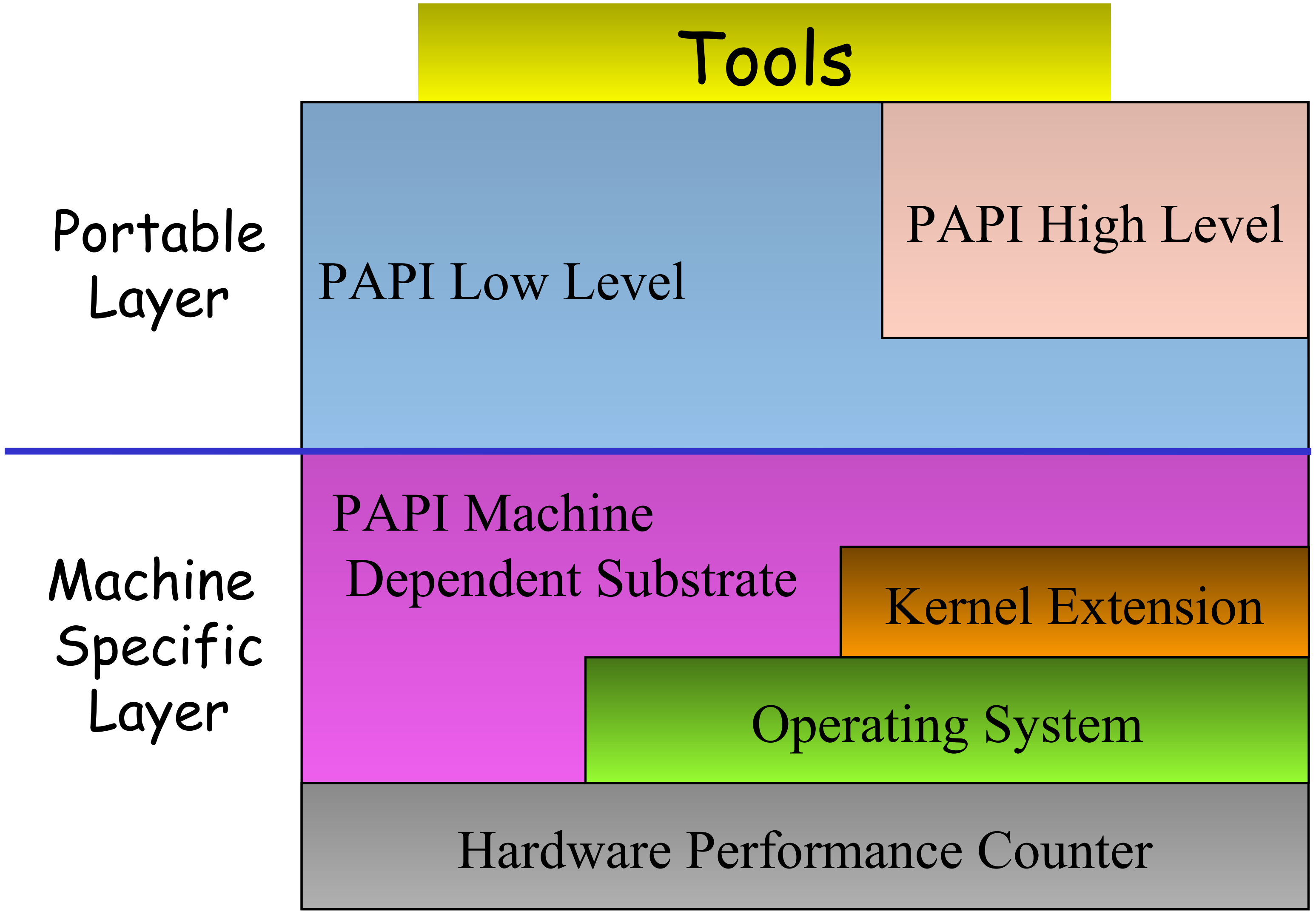
WARPP: Parallel Application Simulator and Performance Toolkit, developed by the University of Warwick High Performance Systems Group for analyzing the performance of high performance parallel/distributed applications

Xperf: Part of the Windows Performance Tools suite that comes with the Windows SDK, XPerf relies on the Event Tracing for Windows (ETW) infrastructure to provide rich support for symbol decoding, sample profiling and capture of call stacks on kernel events. Works on Windows Vista, Server 2008 and above version.

*** Available in MN

Papi: Qué es

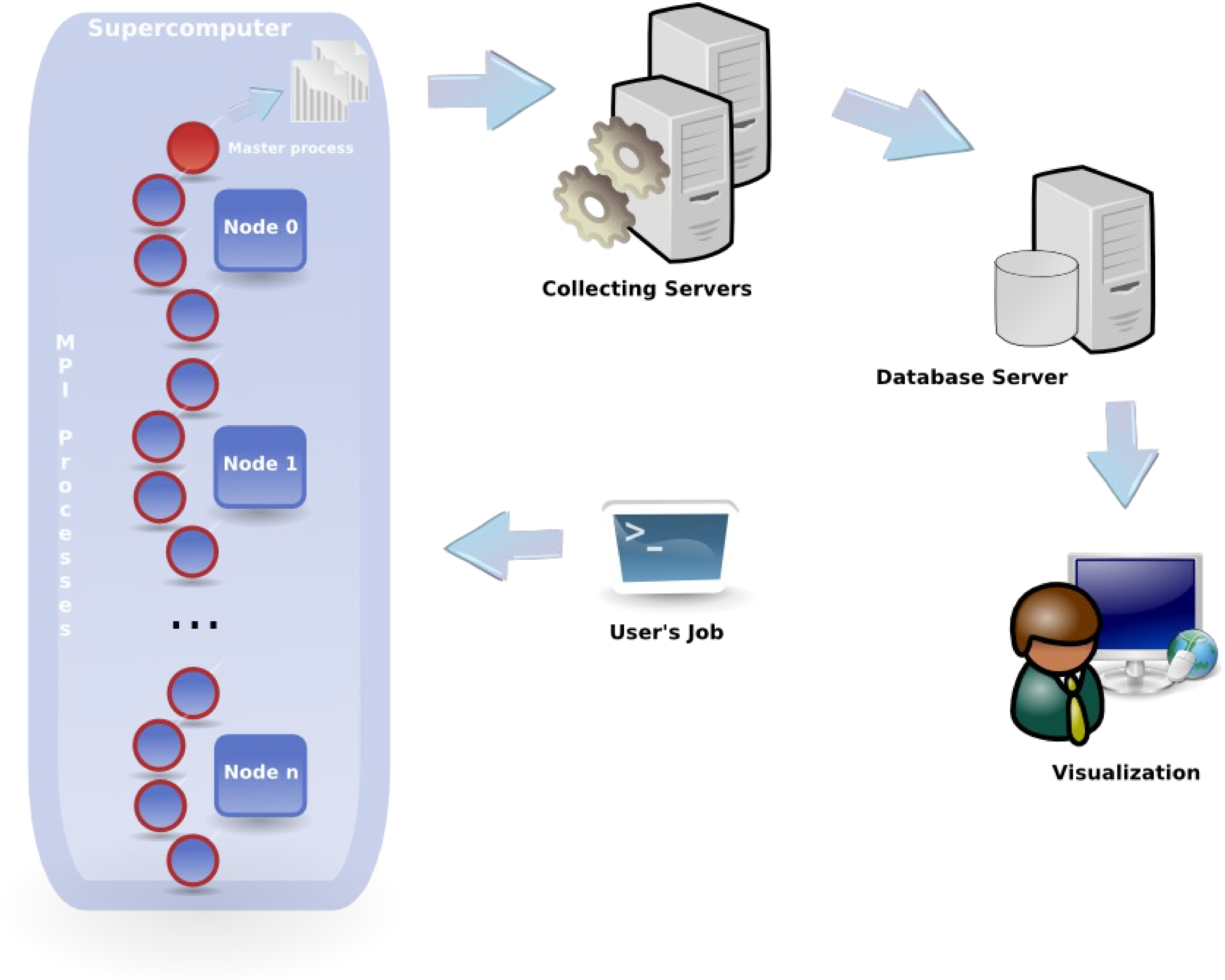
- **P**erformance **A**pplication **P**rogramming **I**nterface
- Implementación de una API portable y eficiente para acceder a los contadores hardware de rendimiento del procesador
  - Permite acceder a los contadores de eventos del procesador fácilmente
    - Eventos predefinidos
    - Eventos nativos
  - Sólo puede sacar de forma simultanea determinados contadores (limitación del procesador)
    - Solución: Multiplexación



Perfminer overview

- Main Objectives:
  - To obtain performance information of **ALL** applications:
    - Generic
    - Transparent to the user
    - Minimum impact in:
      - Applications
      - System and its configuration
  - Gather the information in a scalable way
  - Storage in a Database
  - Methods to visualize and analyze
- **Perfminer:** Porting to MareNostrum from the original solution at PDC (Sweden)

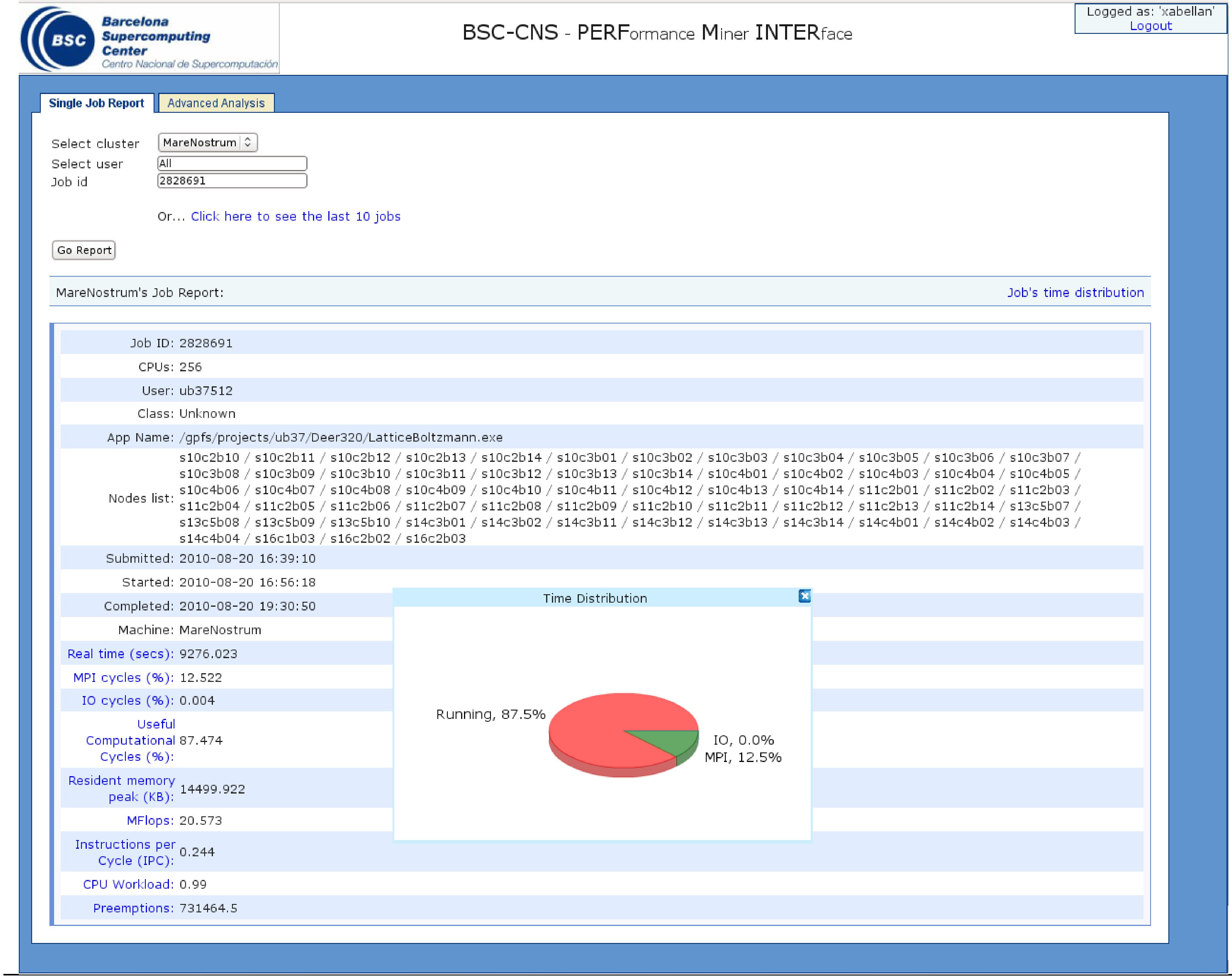
Perfminer architecture



Perfminer: key points

- Obtaining information:
    - Usage of papiex
        - PAPI and native counters
        - Memory usage
        - MPI and I/O general statistics
- Integration with SLURM
    - Plugin loads required instrumentation environment
    - Sends the raw data obtained to a collecting server
- Collecting server(s)
    - Reception and process of data
    - Flush to DB
- Perfminter: User interface

Web-based report



Logged as: 'xabellan'
Logout

BSC-CNS - PERFormance Miner INTERface

Select cluster: MareNostrum
Select user: All
Job id: 2828691

Or... Click here to see the last 10 jobs

MareNostrum's Job Report:

Job's time distribution

Job ID:	2828691
CPUs:	256
User:	ub37512
Class:	Unknown
App Name:	/gpfs/projects/ub37/Deer320/LatticeBoltzmann.exe
Nodes list:	s10c2b10 / s10c2b11 / s10c2b12 / s10c2b13 / s10c2b14 / s10c3b01 / s10c3b02 / s10c3b03 / s10c3b04 / s10c3b05 / s10c3b06 / s10c3b07 / s10c3b08 / s10c3b09 / s10c3b10 / s10c3b11 / s10c3b12 / s10c3b13 / s10c3b14 / s10c4b01 / s10c4b02 / s10c4b03 / s10c4b04 / s10c4b05 / s10c4b06 / s10c4b07 / s10c4b08 / s10c4b09 / s10c4b10 / s10c4b11 / s10c4b12 / s10c4b13 / s10c4b14 / s11c2b01 / s11c2b02 / s11c2b03 / s11c2b04 / s11c2b05 / s11c2b06 / s11c2b07 / s11c2b08 / s11c2b09 / s11c2b10 / s11c2b11 / s11c2b12 / s11c2b13 / s11c2b14 / s13c5b07 / s13c5b08 / s13c5b09 / s13c5b10 / s14c3b01 / s14c3b02 / s14c3b11 / s14c3b12 / s14c3b13 / s14c3b14 / s14c4b01 / s14c4b02 / s14c4b03 / s14c4b04 / s16c1b03 / s16c2b02 / s16c2b03
Submitted:	2010-08-20 16:39:10
Started:	2010-08-20 16:56:18
Completed:	2010-08-20 19:30:50
Machine:	MareNostrum
Real time (secs):	9276.023
MPI cycles (%):	12.522
IO cycles (%):	0.004
Useful Computational Cycles (%):	87.474
Resident memory peak (KB):	14499.922
MFlops:	20.573
Instructions per Cycle (IPC):	0.244
CPU Workload:	0.99
Preemptions:	731464.5

Time Distribution

Profile

- GPROF: GNU Profiler

- "gprof" produces an execution profile of C, Pascal, or Fortran77 programs calculating the amount of time spent in each routine .

- Use:
  - ➔ Compile with -g -pg options
  - ➔ The binary execution will generate a file named ***gmon.out***
  - ➔ Execute ***“gprof <binary>”*** to see the application's profile.

  Note:

  ➔The environment variable **GMON_OUT_PREFIX** controls the name of the gmon.out file.

  ➔If **GMON_OUT_PREFIX** is not set in MPI applications, all the outputs will be overwritten into the same file. So **ALWAYS** set **GMON_OUT_PREFIX** when profiling MPI applications.

Profile

- GPROF: GNU Profiler

- EXAMPLE

```
Flat profile:

Each sample counts as 0.01 seconds.
  %      cumulative   self               self      total
 time   seconds   seconds    calls  ms/call  ms/call    name
 94.69     37.14    37.14     1981    18.75    18.75   .bmod
  2.37    38.07     0.93       105     8.87     8.87   .fwd
  2.30    38.97     0.90        98     9.20     9.20   .bdiv
  0.64    39.22     0.25         1   250.32   250.32   .genmat
  0.08    39.25     0.03         7     4.29     4.29   .lu0
  0.05    39.27     0.02                                .main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```

Profile

- GPROF: GNU Profiler

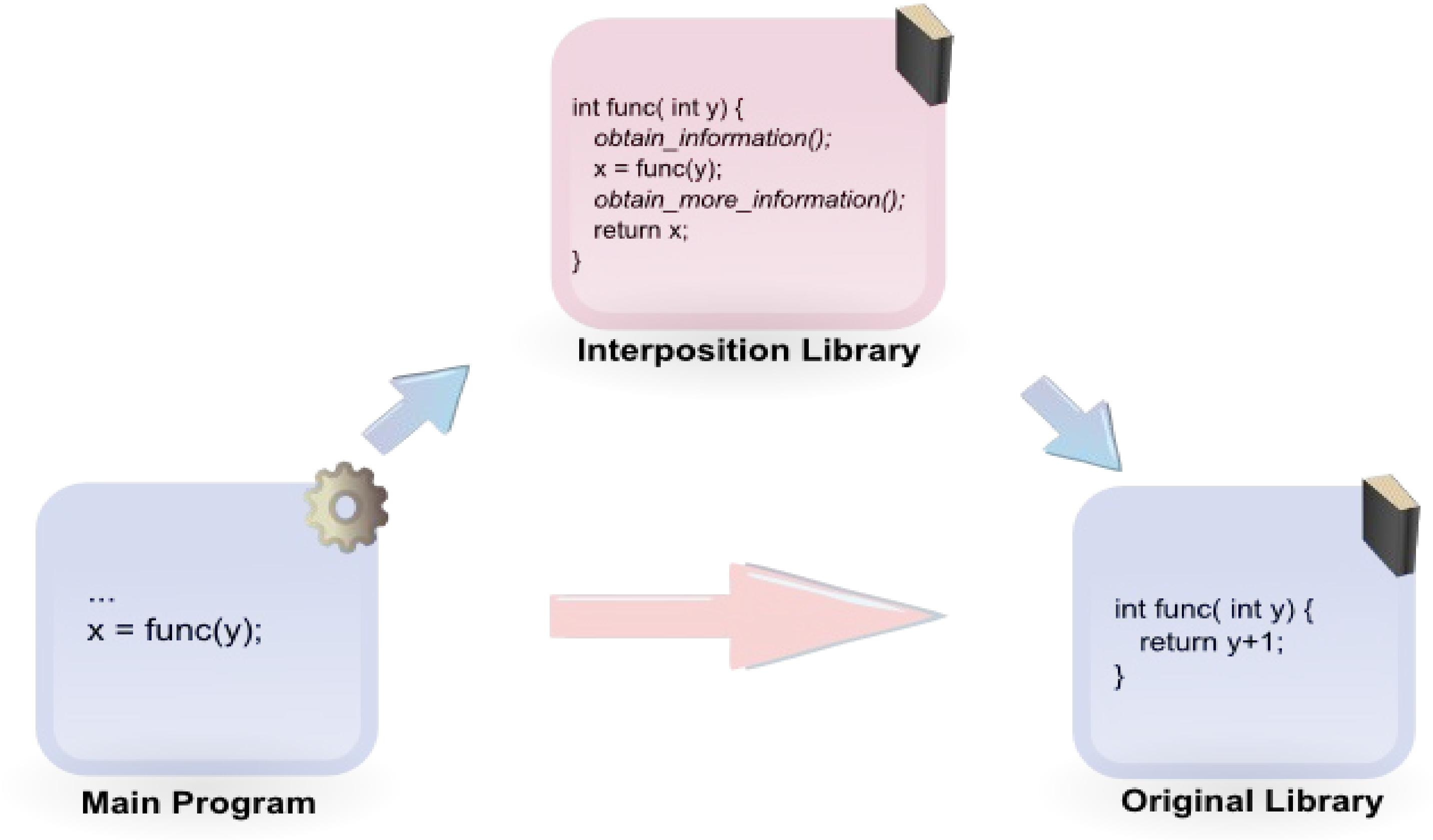
- EXAMPLE

```
Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.03% of 39.27 seconds

Index   % time    self  children    called     name
                                                   <spontaneous>
[1]    100.0    0.02   39.25                    .main [1]
                 37.14    0.00    1981/1981        .bmod [2]
                  0.93    0.00     105/105         .fwd [3]
                  0.90    0.00      98/98          .bdiv [4]
                  0.25    0.00       1/1           .genmat [5]
                  0.03    0.00       7/7           .lu0 [6]
-----------------------------------------------
              37.14    0.00    1981/1981        .main [1]
[2]     94.6   37.14    0.00    1981         .bmod [2]
-----------------------------------------------
               0.93    0.00     105/105         .main [1]
[3]      2.4    0.93    0.00     105         .fwd [3]
-----------------------------------------------
               0.90    0.00      98/98          .main [1]
[4]      2.3    0.90    0.00      98         .bdiv [4]
-----------------------------------------------
               0.25    0.00       1/1           .main [1]
[5]      0.6    0.25    0.00       1         .genmat [5]
-----------------------------------------------
               0.03    0.00       7/7           .main [1]
[6]      0.1    0.03    0.00       7         .lu0 [6]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.
```

```
Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index      A unique number given to each element of the table.
                Index numbers are sorted numerically.
                The index number is printed next to every function name so
                it is easier to look up where the function in the table.

     % time     This is the percentage of the `total' time that was spent
                in this function and its children.  Note that due to
                different viewpoints, functions excluded by options, etc,
                these numbers will NOT add up to 100%.

     self       This is the total amount of time spent in this function.

     children   This is the total amount of time propagated into this
                function by its children.

     called     This is the number of times the function was called.
                If the function called itself recursively, the number
                only includes non-recursive calls, and is followed by
                a `+' and the number of recursive calls.

     name       The name of the current function.  The index number is
                printed after it.  If the function is a member of a
                cycle, the cycle number is printed between the
                function's name and the index number.
```

Papiex: Cómo funciona

- **Monitor:** Instrumentación dinámica en tiempo de ejecución
  - Inicio y final del proceso o thread
  - Funciones MPI
  - Funciones I/O
- *Hooks*: implementados en la librería de papiex



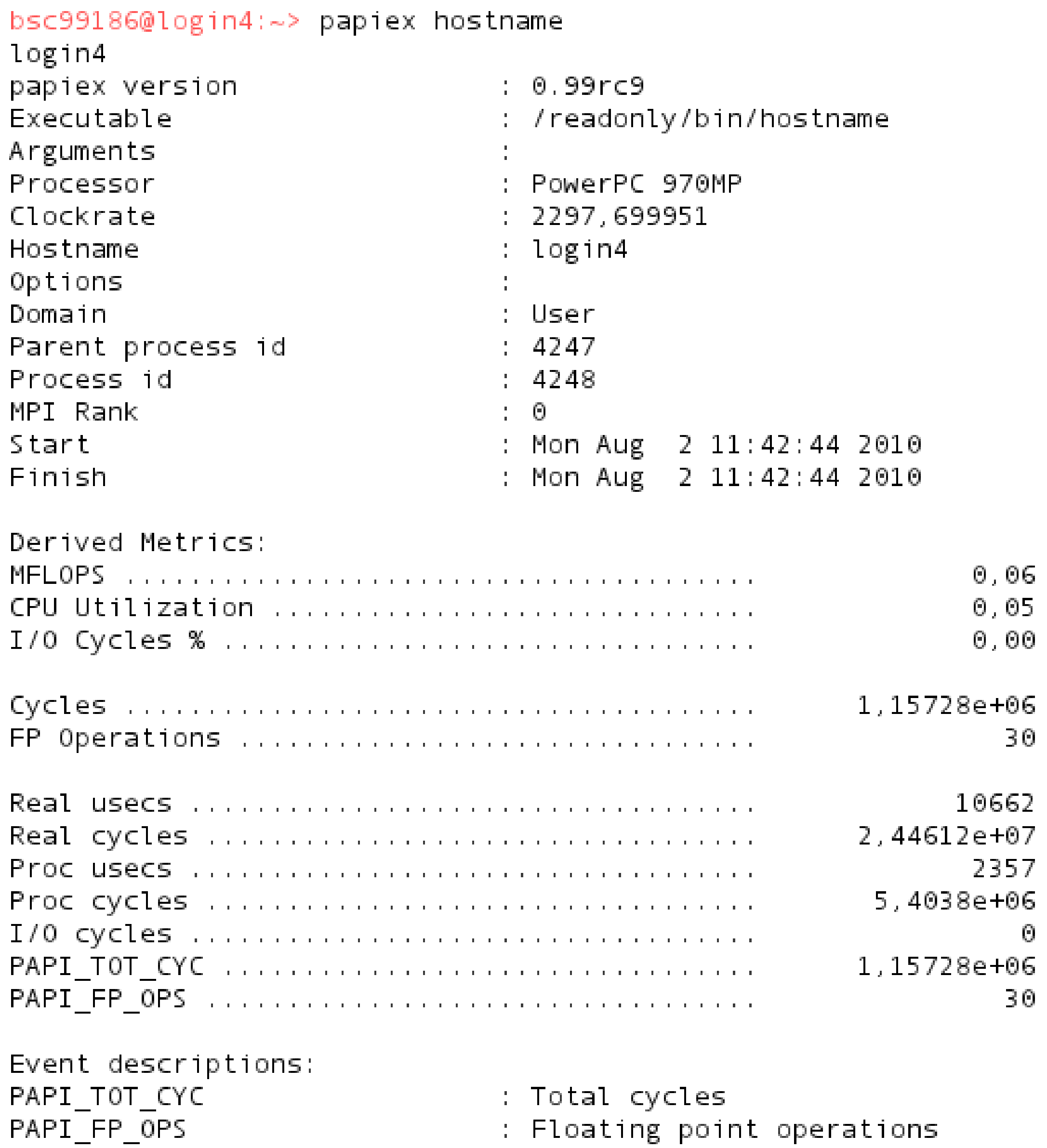
```
int func( int y) {
   obtain_information();
   x = func(y);
   obtain_more_information();
   return x;
}
```

```
...
x = func(y);
```

```
int func( int y) {
   return y+1;
}
```

Papiex: Cómo funciona

- **Papiex:**
  - Ejecutable
    - Wrapper para cargar el entorno (opciones, LD_PRELOAD)
  - Librerías de instrumentación
    - Opciones a través de variables de entorno:
      - PAPIEX_OPTIONS
      - PAPIEX_OUTPUT
  - Ejemplos de métricas:
    - Tiempo (total, MPI, I/O)
    - Ciclos, instrucciones, IPC…
    - MFLOPS
    - Fallos de cache, uso de memoria
    - Etc.



```
bsc99186@login4:~> papiex hostname
login4
papiex version              : 0.99rc9
Executable                  : /readonly/bin/hostname
Arguments                   :
Processor                   : PowerPC 970MP
Clockrate                   : 2297,699951
Hostname                    : login4
Options                     :
Domain                      : User
Parent process id           : 4247
Process id                  : 4248
MPI Rank                    : 0
Start                       : Mon Aug  2 11:42:44 2010
Finish                      : Mon Aug  2 11:42:44 2010

Derived Metrics:
MFLOPS ......................................            0,06
CPU Utilization .............................            0,05
I/O Cycles % ................................            0,00

Cycles ......................................     1,15728e+06
FP Operations ...............................              30

Real usecs ..................................           10662
Real cycles .................................     2,44612e+07
Proc usecs ..................................            2357
Proc cycles .................................      5,4038e+06
I/O cycles ..................................               0
PAPI_TOT_CYC ................................     1,15728e+06
PAPI_FP_OPS .................................              30

Event descriptions:
PAPI_TOT_CYC                : Total cycles
PAPI_FP_OPS                 : Floating point operations
```

Papiex: Cómo funciona

- Resultados:
  - Salida de error
  - Ficheros
- Estructura de los ficheros:
  - Ejecución secuencial:
    - Un fichero: <binario>.papiex.<hostname>.<pid>
  - Ejecución MPI
    - Un directorio: <binario>.papiex.<hostname>.<pid0>
      - all_tasks.summary
      - Ficheros o directorios: task_0, … task_n
  - Ejecución con threads
    - Un directorio: <binario>.papiex.<hostname>.<pid0>
      - process.summary
      - Ficheros thread_0, … thread_n

HPMcount

- Installed at:

  /gpfs/apps/IHPCT/2.2-4/bin/hpmcount

  - "hpmcount -l" : lists the native HWC groups available.
  - IBM tool
  - Access to native events as well as derived metrics

```
Group 38:
PM_INST_CMPL ----------- Instructions completed
PM_INST_CMPL ----------- Instructions completed
PM_FXU_FIN ------------- FXU produced a result
PM_FXU1_BUSY_FXU0_IDLE - FXU1 busy FXU0 idle
PM_FXU_IDLE ------------ FXU idle
PM_FXU_BUSY ------------ FXU busy
PM_FXU0_BUSY_FXU1_IDLE - FXU0 busy FXU1 idle
PM_CYC ----------------- Processor cycles

Group 39:
PM_INST_CMPL ------ Instructions completed
PM_CYC ------------ Processor cycles
PM_FXLS1_FULL_CYC - Cycles FXU1/LS1 queue full
PM_FXLS0_FULL_CYC - Cycles FXU0/LS0 queue full
PM_FXU_IDLE ------- FXU idle
PM_FXU_BUSY ------- FXU busy
PM_FXU0_FIN ------- FXU0 produced a result
PM_FXU1_FIN ------- FXU1 produced a result
```

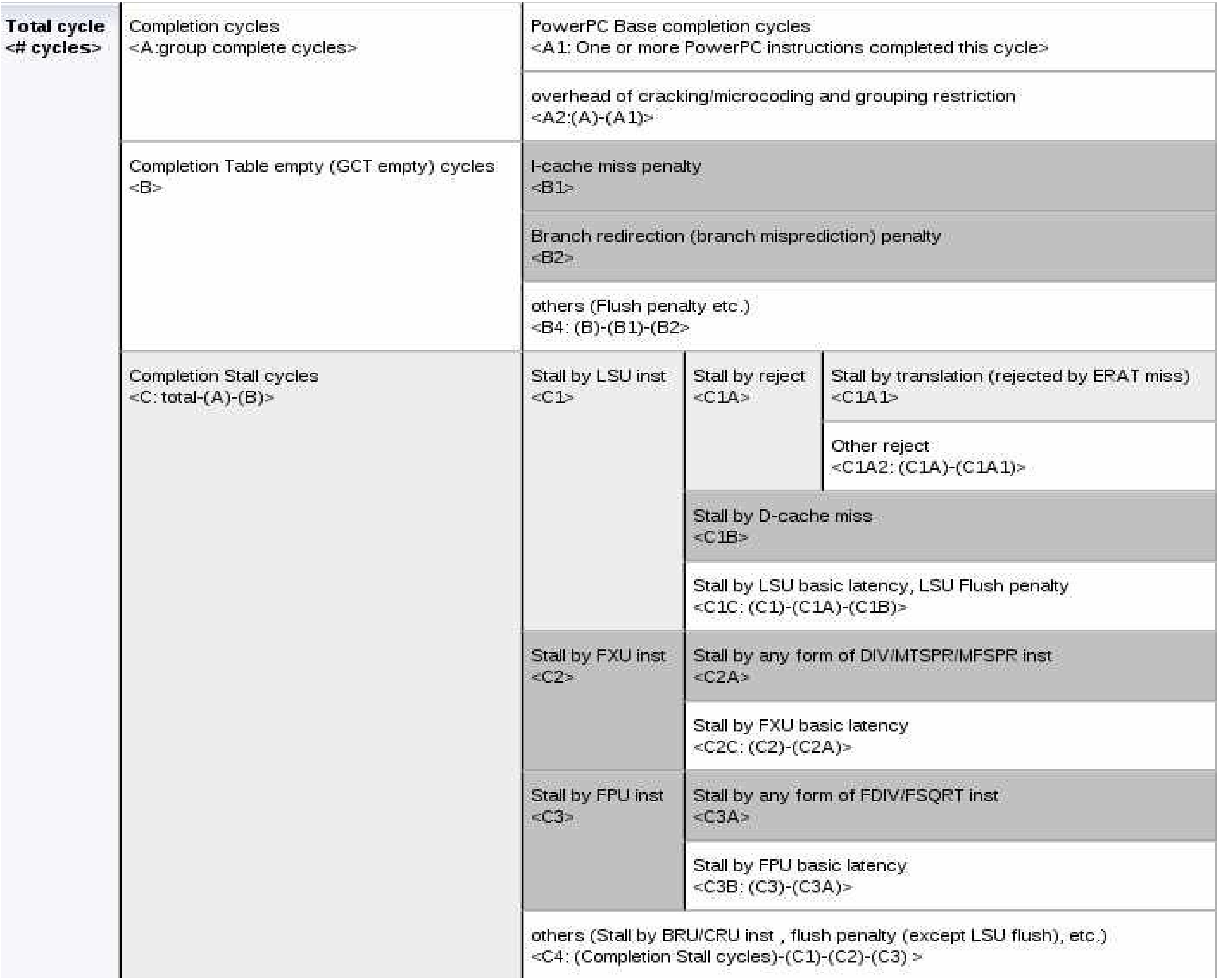
Global View (CPI-Stack Breakdown analysis)

- Introduction:

- *Cycles per instruction* (CPI) is a measurement for analyzing the performance of a workload. It is defined as the number of cpu cycles needed to complete an instruction: **CPI = Total Cycles / Number of Instructions Completed**
  - Indicates underutilization of resources
  - ↑ Lower bond in the ppc970 is 0.25 (theoretical limit: 4 ↓instructions per cycle)
  - Depends on the architecture.
  - Need access to hardware counter.
  - Not all needed counters can be read at the same time due to hardware limitations.

Global View (CPI-Stack Breakdown analysis)

- Application Cycle Distribution Snapshot:



Total cycle **<# cycles>**	Completion cycles <A:group complete cycles>	PowerPC Base completion cycles <A1: One or more PowerPC instructions completed this cycle>		
		overhead of cracking/microcoding and grouping restriction <A2:(A)-(A1)>		
	Completion Table empty (GCT empty) cycles <B>	I-cache miss penalty <B1>		
		Branch redirection (branch misprediction) penalty <B2>		
		others (Flush penalty etc.) <B4: (B)-(B1)-(B2)>		
	Completion Stall cycles <C: total-(A)-(B)>	Stall by LSU inst <C1>	Stall by reject <C1A>	Stall by translation (rejected by ERAT miss) <C1A1>
				Other reject <C1A2: (C1A)-(C1A1)>
			Stall by D-cache miss <C1B>	
			Stall by LSU basic latency, LSU Flush penalty <C1C: (C1)-(C1A)-(C1B)>	
		Stall by FXU inst <C2>	Stall by any form of DIV/MTSPR/MFSPR inst <C2A>	
			Stall by FXU basic latency <C2C: (C2)-(C2A)>	
		Stall by FPU inst <C3>	Stall by any form of FDIV/FSQRT inst <C3A>	
			Stall by FPU basic latency <C3B: (C3)-(C3A)>	
		others (Stall by BRU/CRU inst , flush penalty (except LSU flush), etc.) <C4: (Completion Stall cycles)-(C1)-(C2)-(C3) >		

Global View (CPI-Stack Breakdown analysis)

- Implementation in MN:

- **Path to Deployment:** *gpfs/projects/bsc99/bsc99704/CPI_Stack_Breakdow*n

  - Template File*: “CPIStack_Breakdown.ods” : to be filled with the gathered output (HWC)*
  - “paraver” directory: contains all the things needed to do a detailed CPI stack breakdown analysis using Paraver.
  - Scripts:

    “cpisba_mpi.sh” : for MPI application

    *"cpisba.sh" : for s*ecuential application

- **Overview:**

  - Gather needed counters (we use ***hpmcount***): Four runs will be needed because ***not all counters can be read at the same*** time. So careful select the wall_clock_time of the submission script.

    Example of a submission script:

```
#!/bin/bash
# @ job_name = test
# @ initialdir = .
# @ output = mpi_%j.out
# @ error = mpi_%j.err
# @ total_tasks = 8
# @ wall_clock_limit = 1:00:00

/gpfs/projects/bsc99/bsc99704/CPI_Stack_Breakdown/cpisba_mpi.sh <binario> <argumentos>
```

Global View (CPI-Stack Breakdown analysis)

- Implementation in MN: (cont.)

- **Overview:**

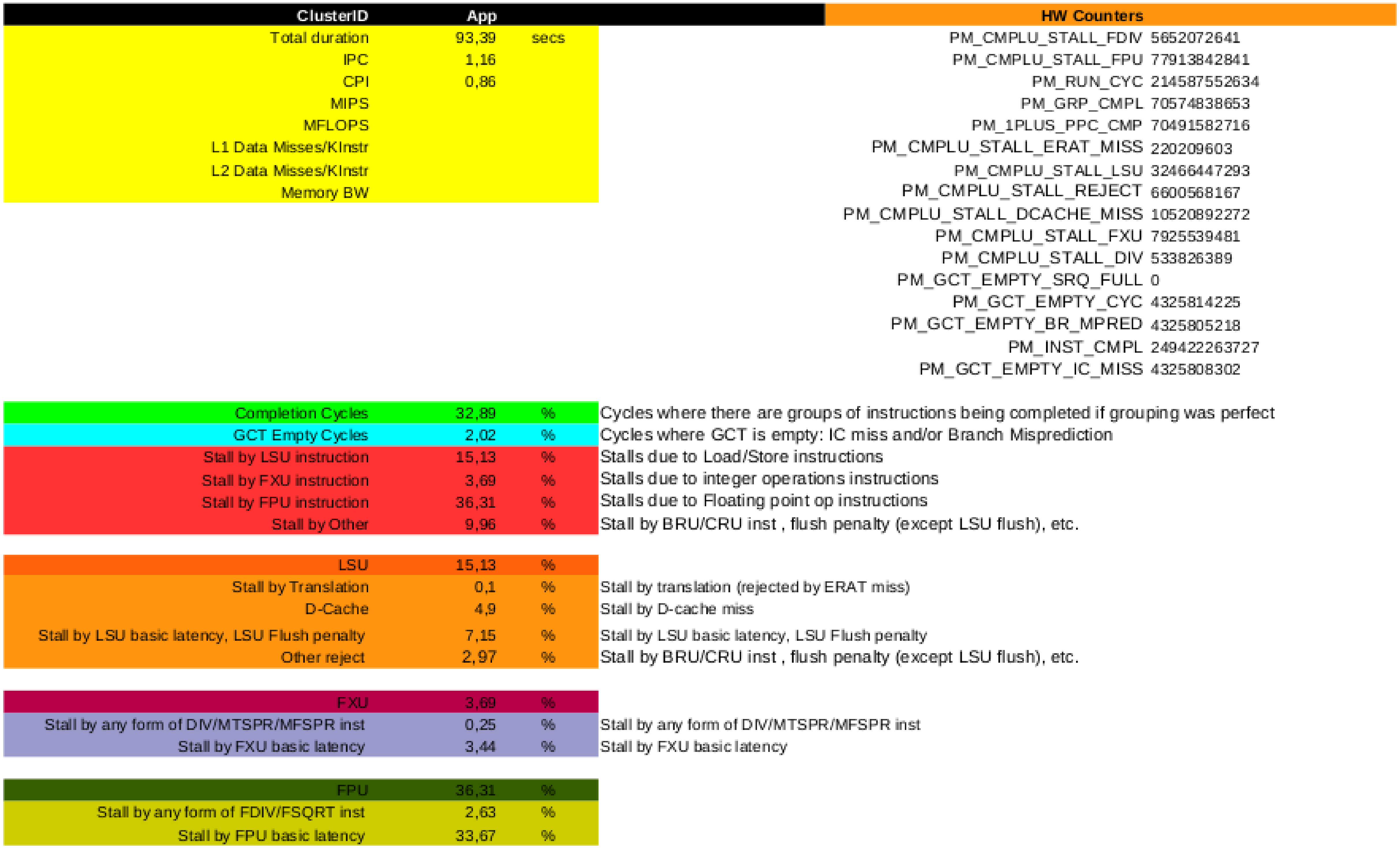
1) At the end. In your ***.out** file you should end up with something like:

```
PM_CMPLU_STALL_FDIV: 62879865499
PM_CMPLU_STALL_FPU: 253027962454
PM_RUN_CYC: 1434930927367
PM_GRP_CMPL: 191558015451
PM_1PLUS_PPC_CMP: 187199925619
PM_CMPLU_STALL_ERAT_MISS: 137567636255
PM_CMPLU_STALL_LSU: 762239025471
PM_CMPLU_STALL_REJECT: 322779066900
PM_CMPLU_STALL_DCACHE_MISS: 309334934749
PM_CMPLU_STALL_FXU: 29359134884
PM_CMPLU_STALL_DIV: 3239196680
PM_GCT_EMPTY_SRQ_FULL: 0
PM_GCT_EMPTY_CYC: 185840390144
PM_GCT_EMPTY_BR_MPRED: 185840077006
PM_INST_CMPL: 656689888192
PM_GCT_EMPTY_IC_MISS: 185840207081
```

- Insert this data into the Template File (“*CPIStack_Breakdown.ods”*) ...

Global View (CPI-Stack Breakdown analysis)

- Implementation in MN: (cont.)

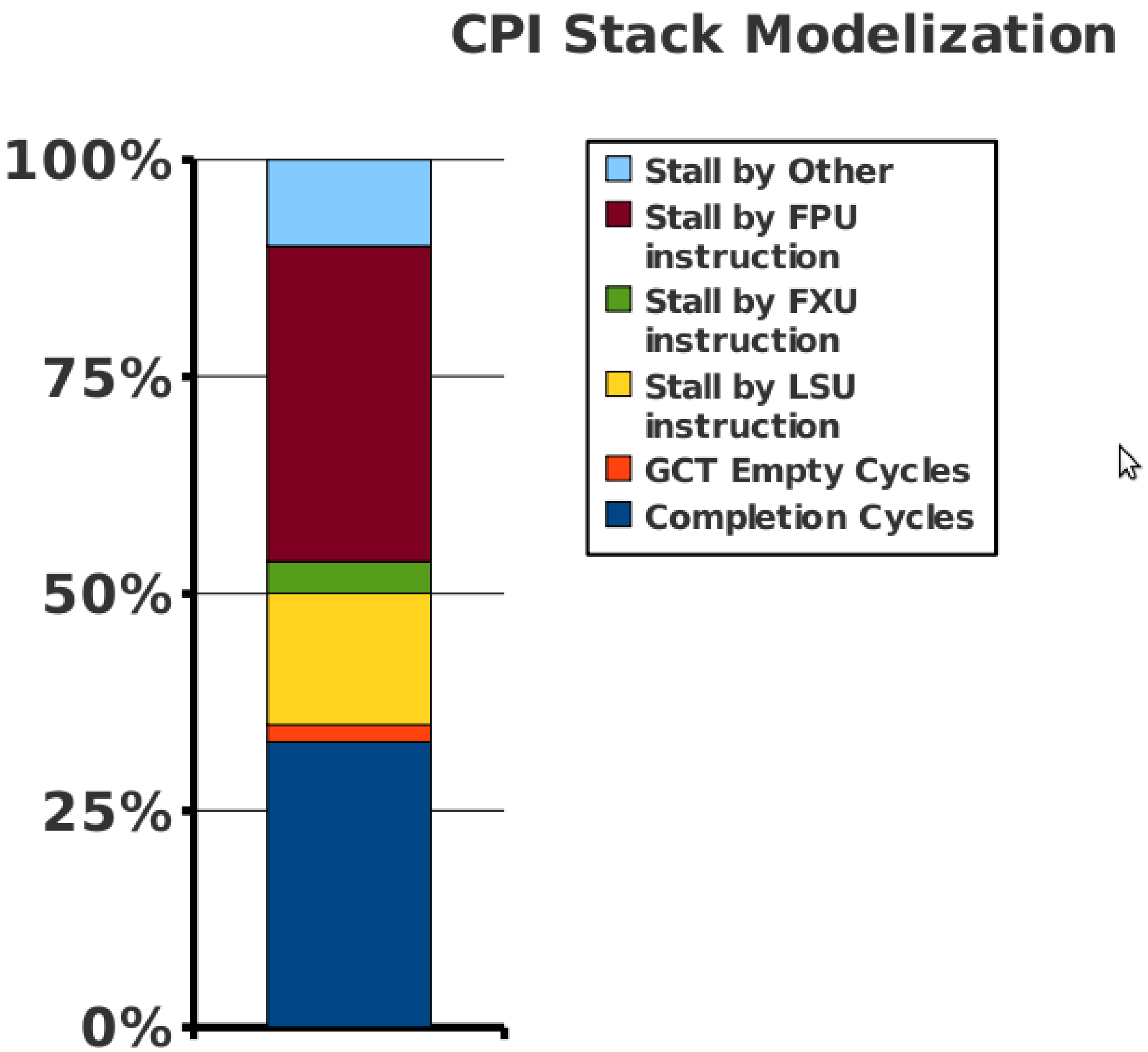


ClusterID	App	
Total duration	93,39	secs
IPC	1,16	
CPI	0,86	
MIPS		
MFLOPS		
L1 Data Misses/KInstr		
L2 Data Misses/KInstr		
Memory BW		

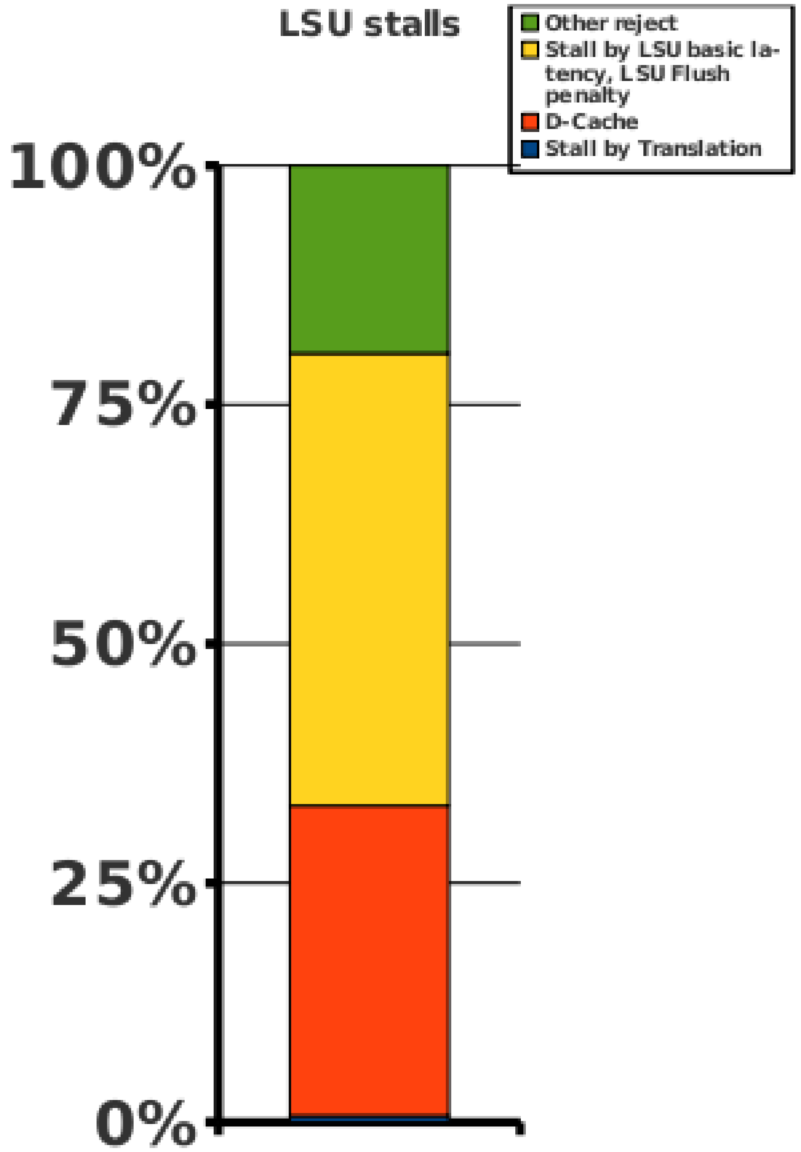
HW Counters	
PM_CMPLU_STALL_FDIV	5652072641
PM_CMPLU_STALL_FPU	77913842841
PM_RUN_CYC	214587552634
PM_GRP_CMPL	70574838653
PM_1PLUS_PPC_CMP	70491582716
PM_CMPLU_STALL_ERAT_MISS	220209603
PM_CMPLU_STALL_LSU	32466447293
PM_CMPLU_STALL_REJECT	6600568167
PM_CMPLU_STALL_DCACHE_MISS	10520892272
PM_CMPLU_STALL_FXU	7925539481
PM_CMPLU_STALL_DIV	533826389
PM_GCT_EMPTY_SRQ_FULL	0
PM_GCT_EMPTY_CYC	4325814225
PM_GCT_EMPTY_BR_MPRED	4325805218
PM_INST_CMPL	249422263727
PM_GCT_EMPTY_IC_MISS	4325808302

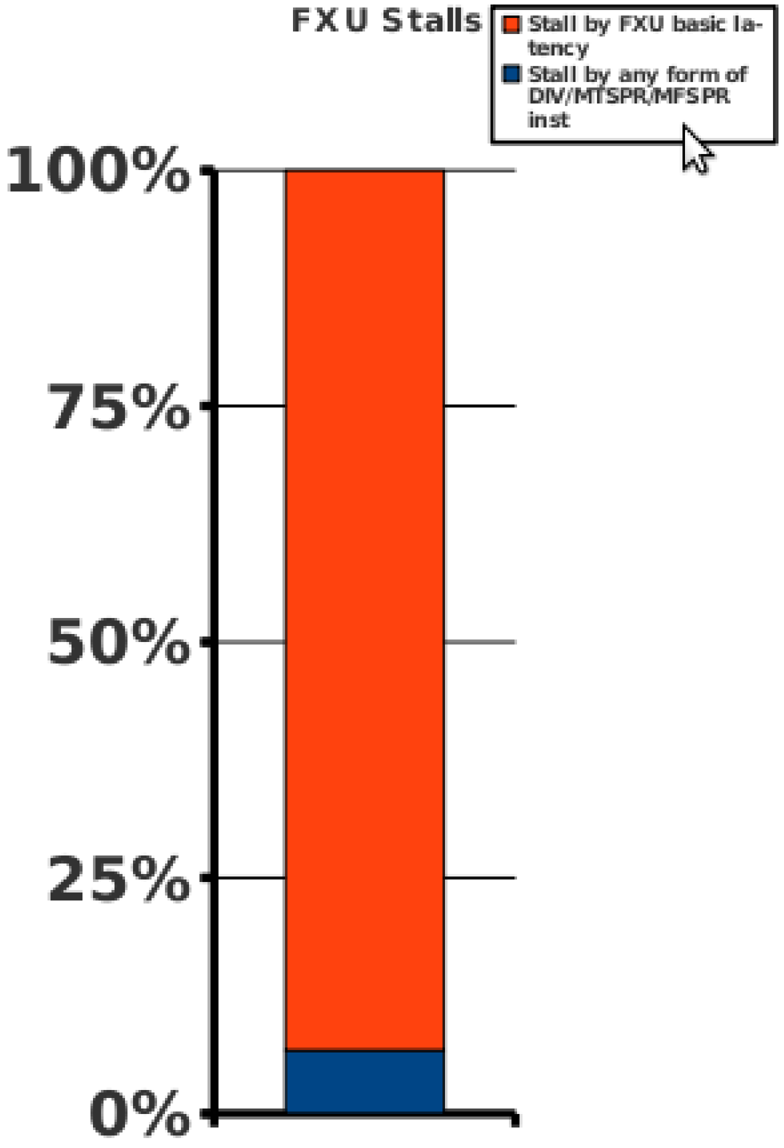
Completion Cycles	32,89	%	Cycles where there are groups of instructions being completed if grouping was perfect
GCT Empty Cycles	2,02	%	Cycles where GCT is empty: IC miss and/or Branch Misprediction
Stall by LSU instruction	15,13	%	Stalls due to Load/Store instructions
Stall by FXU instruction	3,69	%	Stalls due to integer operations instructions
Stall by FPU instruction	36,31	%	Stalls due to Floating point op instructions
Stall by Other	9,96	%	Stall by BRU/CRU inst , flush penalty (except LSU flush), etc.
LSU	15,13	%	
Stall by Translation	0,1	%	Stall by translation (rejected by ERAT miss)
D-Cache	4,9	%	Stall by D-cache miss
Stall by LSU basic latency, LSU Flush penalty	7,15	%	Stall by LSU basic latency, LSU Flush penalty
Other reject	2,97	%	Stall by BRU/CRU inst , flush penalty (except LSU flush), etc.
FXU	3,69	%	
Stall by any form of DIV/MTSPR/MFSPR inst	0,25	%	Stall by any form of DIV/MTSPR/MFSPR inst
Stall by FXU basic latency	3,44	%	Stall by FXU basic latency
FPU	36,31	%	
Stall by any form of FDIV/FSQRT inst	2,63	%	
Stall by FPU basic latency	33,67	%	

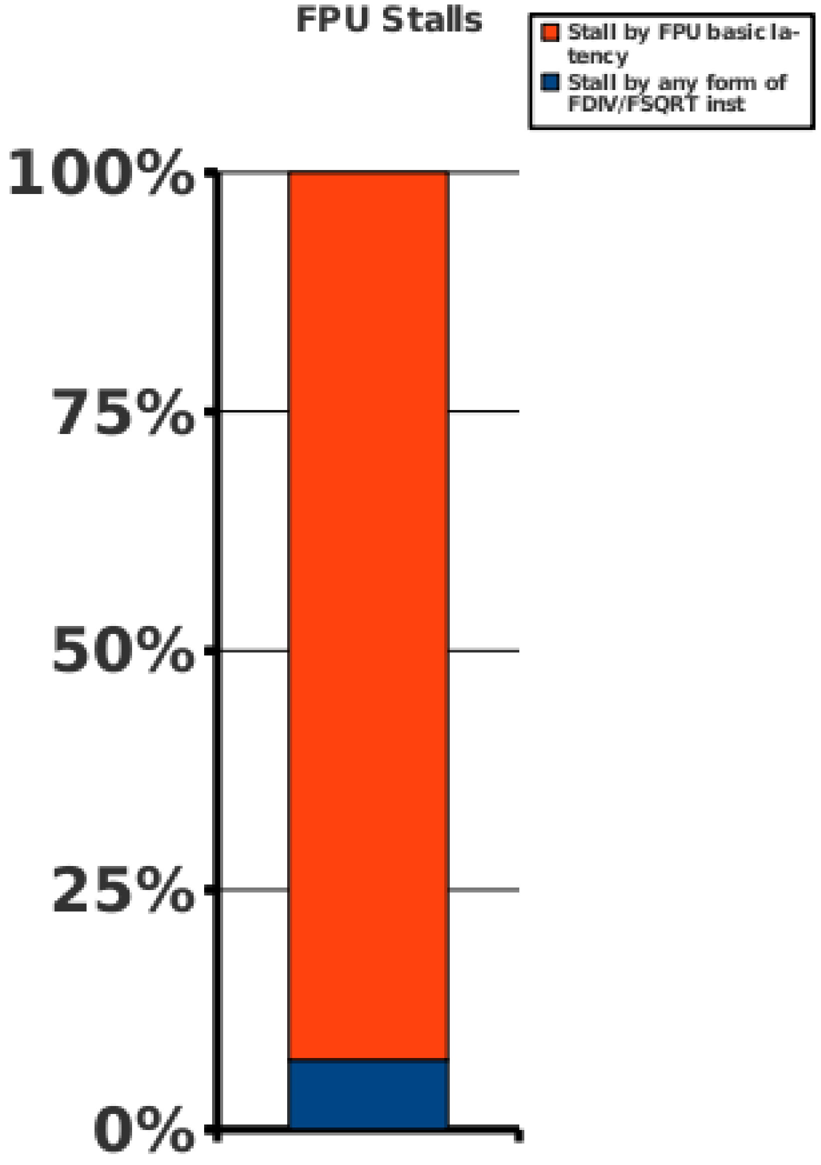
Global View (CPI-Stack Breakdown analysis)



Global View (CPI-Stack Breakdown analysis)







CPI Breakdown Analysis (Detailed View with Paraver)

- For a detailed CPI Breakdown analysis using Paraver it is necessary to trace the application using **MPITRACE** (out of the scope of this presentation)

  - **MPITRACE**: xml driven tool based on PAPI to obtain performance data between events (MPI or user-defined) or by sampling. It can also read HWC.

  - It can be use to read the necessary HWC needed to do the CPI breakdown analysis.

  - Using Paraver and its ability to create and manage custom configurations we can obtain different views to see the same info as in the *Global View* but zooming into a section instead of the whole runtime.

CPI Breakdown Analysis (Detailed View with Paraver)

- **Path to Deployment:** */gpfs/projects/bsc99/bsc99704/CPI_Stack_Breakdow*n/paraver
  - “mpitrace.xml” : XML file to obtain drive MPITRACE

```
<?xml version='1.0'?>

<trace enabled="yes" home="/gpfs/apps/CEPBATOOLS/mpitrace-mx-20100204/64"
 initial-mode="detail"
 type="paraver"
 xml-parser-id="Id: xml-parse.c 150 2010-02-03 13:48:00Z harald $">

  <mpi enabled="yes">
    <counters enabled="yes" />
  </mpi>

  <callers enabled="yes">
    <mpi enabled="yes">1-3</mpi>
    <sampling enabled="yes">1-5</sampling>
  </callers>

  <counters enabled="yes">

    <cpu enabled="yes" starting-set-distribution="cyclic">

      <!-- para CPI breakdown analisys -->
      <set enabled="yes" domain="all" >
        PM_INST_CMPL,PM_CYC,PM_GRP_CMPL,PM_1PLUS_PPC_CMPL
      </set>

      <set enabled="yes" domain="all" >
        PM_INST_CMPL,PM_CYC,PM_CMPLU_STALL_ERAT_MISS,PM_CMPLU_STALL_LSU
      </set>

      <set enabled="yes" domain="all" >
        PM_INST_CMPL,PM_CYC,PM_CMPLU_STALL_REJECT,PM_CMPLU_STALL_DCACHE_MISS
```

CPI Breakdown Analysis (Detailed View with Paraver)

```
      </set>

      <set enabled="yes" domain="all" >
        PM_INST_CMPL,PM_CYC,PM_CMPLU_STALL_FXU,PM_CMPLU_STALL_DIV,PM_GCT_EMPTY_SRQ_FULL
      </set>

      <set enabled="yes" domain="all" >
        PM_INST_CMPL,PM_CYC,PM_CMPLU_STALL_FPU,PM_CMPLU_STALL_FDIV
      </set>

      <set enabled="yes" domain="all" >
        PM_INST_CMPL,PM_CYC,PM_GCT_EMPTY_CYC,PM_GCT_EMPTY_BR_MPRED,PM_GCT_EMPTY_IC_MISS
      </set>

      <!-- Extended statistics sets -->
      <set enabled="yes" domain="all" >
        PM_DTLB_MISS, PM_ITLB_MISS, PM_LD_MISS_L1, PM_ST_MISS_L1, PM_CYC, PM_INST_CMPL, PM_ST_REF_L1, PM_LD_REF_L1
      </set>

      <set enabled="yes" domain="all" >
        PM_DATA_FROM_L2, PM_INST_CMPL, PM_DATA_FROM_MEM, PM_LD_MISS_L1_LSU0, PM_1PLUS_PPC_CMPL, PM_CYC, PM_LD_MISS_L1_LSU1, PM_LD_REF_L1
      </set>

    </cpu>

  </counters>

  <storage enabled="yes">
    <trace-prefix enabled="yes">TRACE</trace-prefix>
    <size enabled="no">80</size>
    <temporal-directory enabled="yes" make-dir="no">/scratch</temporal-directory>
    <final-directory enabled="yes" make-dir="no">/home/bsc99/bsc99704/EXAMPLES/vasp-slowdown/TRACES</final-directory>
    <gather-mpits enabled="no" />
  </storage>

  <!-- Buffer configuration -->
  <buffer enabled="yes">
    <size enabled="yes">1500000</size>
    <circular enabled="no" />
  </buffer>

</trace>
```

CPI Breakdown Analysis (Detailed View with Paraver)

- **Path to Deployment:** */gpfs/projects/bsc99/bsc99704/CPI_Stack_Breakdow*n/paraver
  - “trace.sh” : script to use as a wrapper in the submission script

```
#!/bin/bash

export MPTRACE_CONFIG_FILE=$HOME/HANDY/CPI_Stack_Breakdown/paraver/mpitrace.xml
export MPITRACE_ON=1
export LD_LIBRARY_PATH=/gpfs/apps/PAPI/3.7.1/64/lib:/gpfs/apps/CEPBATOOLS/mpitrace-mx-20100204/64/lib:/gpfs/apps/CEPBATOOLS/mpitrace-mx-20100204/32/lib
export LD_PRELOAD=libmpitrace.so

$@
```

  - “Filters” : this directory contains useful xml files to pre-process the trace. Sometimes (most of the time, actually) the traces need to be pre-processed: traces tend to be large unmanageable files...
  - “CPI_cfgs” : Core configuration files. CFGs to create the derived metrics and load each of the previously described views

- Use:
  1. Generate the trace for the application
  2. Pre-process the trace to make it “manageable”
  3. Open Paraver
  4. Load trace
  5. Select section of interest
  6. Load desired cfg file
  7. Back to step 5 until finished.

Paraver-based analysis

Need a Paraver Tutorial :)