VI-HPS Workshop, HLRS 2015

Tuesday, February 24th	
10:00 – 10:30	Performance Analysis with CrayPat (part 1)
10:30 – 11:00	CrayPat Walk Through
11:00 – 11:30	Performance Analysis with CrayPat (part 2)
11:30 – 11:45	Reveal Walk Through
Afternoon	Apply the tools to your own application

Performance Analysis with CrayPat

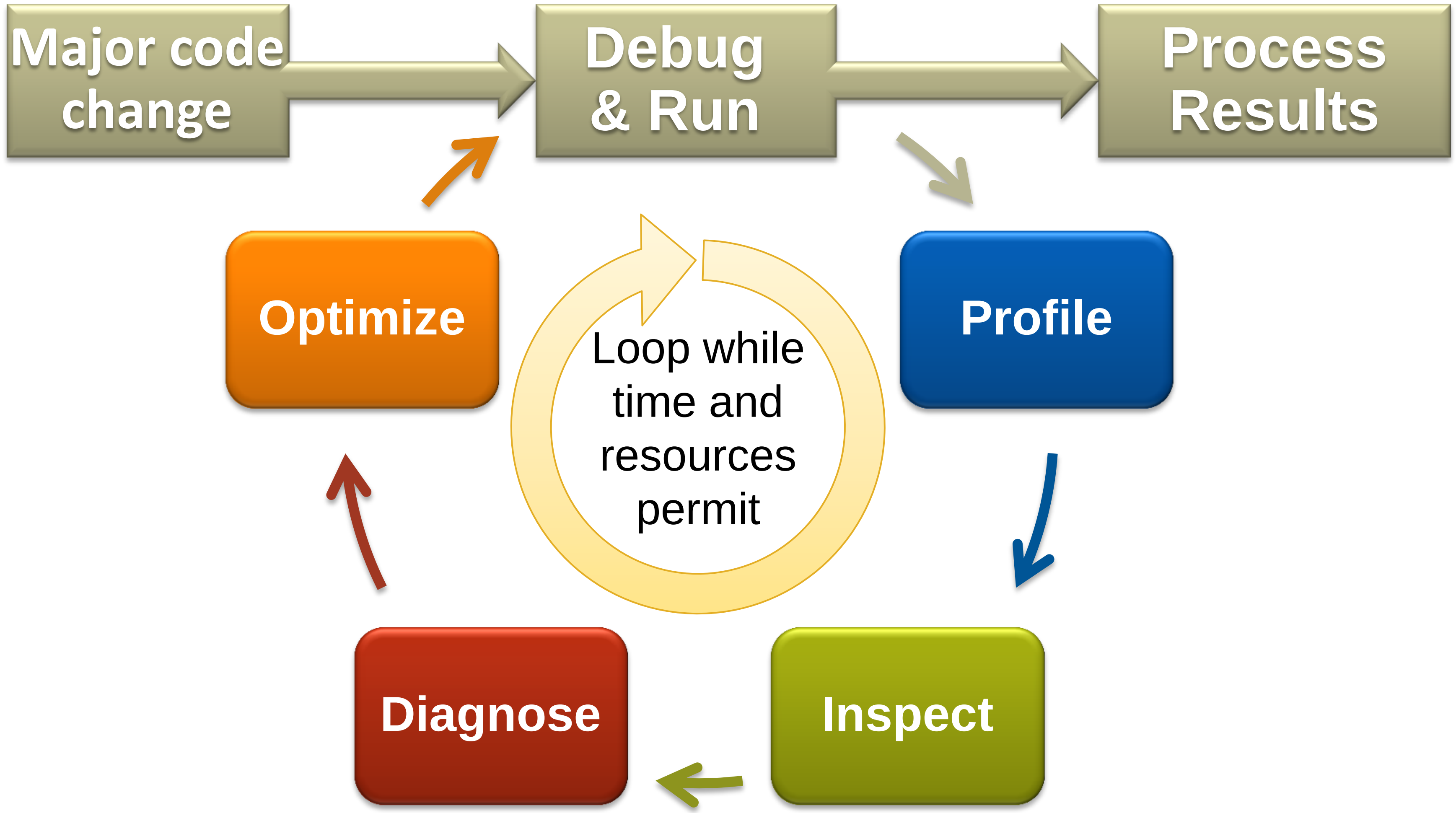
Part 1

Aniello Esposito
(esposito@cray.com)

Outline

- **Introduction to performance analysis with CrayPat**
  - Different approaches to profiling: Sampling vs. Tracing
  - How to recompile and run your code for CrayPat.
  - Combining Sampling and Tracing: Automatic Performance Analysis
- **Collecting Hardware Performance counters.**
- **Compiler Feedback**
- **CrayPat API**
- **Short Introduction to Hands-on Exercises**

The Optimization Cycle

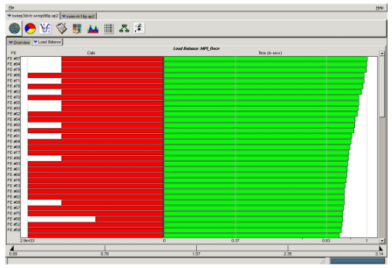


CrayPAT Overview

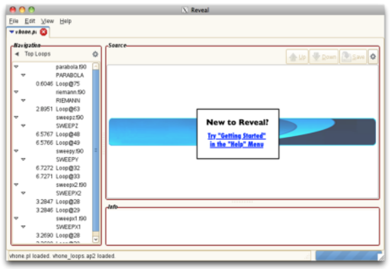
- **Assist the user with application performance analysis and optimization**
  - Provides concrete suggestions instead of just reporting data.
  - Work on user codes at realistic core counts with thousands of processes/threads
  - Integrate into large codes with millions of lines of code
- **Is an universal tool**
  - Basic functionality available to all compilers on the system
  - Additional functionality available for the Cray compiler (loop profiling)
- **Requires no source code or Makefile modification**
  - Automatic instrumentation at group (function) level such as mpi, io, …
  - Requires object files and archives for instrumentation and to be compiled with the wrapper scripts while the `perftools` module was loaded.
  - Able to generate instrumentation on optimized code.
  - Creates a new stand-alone instrumented program while preserving the original binary.

Components of CrayPat

- **Available through the `perftools` module:**
  - **pat_build** - Instruments the program to be analyzed (command line)
  - **pat_report** - Generates text reports from the performance data captured during program execution and exports data for use in other programs. (command line)
  - **Cray Apprentice2** - A graphical analysis tool that can be used to visualize and explore the performance data captured during program execution.



  - **Reveal** - A graphical source code analysis tool that can be used to correlate performance analysis data with annotated source code listings, to identify key opportunities for optimization.



Components of CrayPat (not discussed)

- **grid_order** - Generates MPI rank order information that can be used with the MPICH_RANK_REORDER environment variable to override the default MPI rank placement scheme and specify a custom rank placement. (For more information, see the **intro_mpi(3)** man page.)

- **pat_help** - Help system, which contains extensive usage information and examples. This help system can be accessed by entering pat_help at the command line.

- The individual components of CrayPat are documented in the following man pages (info on hardware counters will follow):
  - **intro_craypat(1)**
  - **pat_build(1)**
  - **pat_report(1)**
  - **pat_help(1)**
  - **grid_order(1)**
  - **app2(1)**
  - **reveal(1)**

- **craypat-lite –** Light weight profiling tool. (More details later on.)

Sampling and Event Tracing

- **CrayPAT provides two fundamental ways of profiling:**

1. **Sampling**
   - By taking regular snapshots of the applications call stack we can create a statistical profile of where the application spends most time.
   - Snapshots can be taken at regular intervals in time or when some other external event occurs, like a hardware counter overflowing
2. **Event Tracing**
   - Alternatively we can record performance information every time a specific program event occurs, e.g. entering or exiting a function.
   - We can get accurate information about specific areas of the code every time the event occurs
   - Event tracing code can be added automatically or included manually through API calls.

- **Automatic Performance Analysis (APA) combines the two approaches.**
- **Loop profiling is a special flavor of event tracing.**

Sampling

Advantages

- Only need to instrument main routine
- Low Overhead – depends only on sampling frequency
- Smaller volumes of data produced

Disadvantages

- Only statistical averages available
- Limited information from performance counters

Event Tracing

Advantages

- More accurate and more detailed information
- Data collected from every traced function call not statistical averages

Disadvantages

- Increased overheads as number of function calls increases
- Huge volumes of data generated

The best approach is *guided tracing*.
e.g. Only tracing functions that are not small (i.e. very few lines of code) and contribute a lot to application's run time.
APA is an automated way to do this.

Exercise 1: Generate a Sampling Profile

```
> module load perftools
```

- Makes the default version of CrayPAT available
- Subsequent compiler invocations will automatically insert necessary hooks for profiling (not always up-to-date with latest third-party compilers)
- Binaries are *not* automatically instrumented

```
> make clean; make
> pat_build –S himeno.exe
```

- Builds code with profiling hooks, then instruments the binary
- Result named himeno.exe+pat

```
> aprun –n 24 ./himeno.exe+pat
> pat_report –o myrep.txt himeno+pat+*
```

- Running the “+pat” binary creates a data file *.xf or directory
- pat_report reads that data file and prints lots of human-readable performance data. Creates an *.ap2 file.

Table 2: Profile by Group, Function, and Line

```
 Samp% |    Samp | Imb. |   Imb. |Group
       |         | Samp | Samp% | Function
       |         |      |        |  Source
       |         |      |        |   Line
       |         |      |        |    PE=HIDE

 100.0% | 2063.0 |    -- |     -- |Total
|-----------------------------------------------------------------------------
|  82.3% | 1698.0 |    -- |     -- |USER
||----------------------------------------------------------------------------
||  77.2% | 1592.2 |    -- |     -- |jacobi
3|        |        |       |        | Himeno/test.samp/himeno.c
|||---------------------------------------------------------------------------
4|||  61.1% | 1260.6 | 32.4 |   2.9% |line.243
4|||   7.2% |   147.8 | 19.2 | 13.2% |line.257
4|||   4.3% |    89.5 | 17.5 | 18.7% |line.258
4|||   4.2% |    86.5 |   8.5 | 10.2% |line.260
|||===========================================================================
||   5.1% |   105.8 |    -- |     -- |initmt
3|        |         |       |        | Himeno/test.samp/himeno.c
||============================================================================
|  16.4% |   338.2 |    -- |     -- |ETC
||----------------------------------------------------------------------------
||  13.8% |   284.8 |   5.2 |   2.1% |__cray_scopy_HSW
||   2.6% |    53.5 |   4.5 |   8.9% |__cray_sset_HSW
||============================================================================
|   1.3% |    26.6 |    -- |     -- |MPI
|=============================================================================
```

Top function

Communication not relevant. Threshold of 0.5% can be cancelled with –T option.

Exercise 2: Generate a Tracing Profile

```
> module load perftools
```

- Makes the default version of CrayPAT available.

```
> pat_build -u -g mpi himeno.exe
```

- If your application is already built with `perftools` loaded you do not have to rebuild when switching the experiment.
- Traces MPI functions calls and functions defined in the program source files

```
> aprun -n 24 ./himeno.exe+pat
> pat_report -o myrep.txt himeno+pat+*
```

- Running the “+pat” binary creates a data file or directory
- pat_report reads that data file and prints lots of human-readable performance data. Creates an *.ap2 file.

Table 1: Profile by Function Group and Function

```
 Time% |       Time |     Imb. |  Imb. |  Calls |Group
       |            |     Time | Time% |        | Function
       |            |          |       |        |  PE=HIDE

100.0% | 20.643909 |       -- |    -- | 1149.0 |Total
|-----------------------------------------------------------------
| 98.8% | 20.395989 |       -- |    -- |  219.0 |USER
||----------------------------------------------------------------
||  91.1% | 18.797060 | 0.115535 |  0.7% |    2.0 |jacobi
||   7.7% |  1.597866 | 0.006647 |  0.5% |    1.0 |initmt
||   0.0% |  0.000402 | 0.000167 | 33.5% |   53.0 |sendp3
||================================================================
|  1.2% |  0.239306 |       -- |    -- |  871.0 |MPI
||----------------------------------------------------------------
||   0.7% |  0.148981 | 0.094595 | 44.4% |  159.0 |MPI_Waitall
||   0.4% |  0.085824 | 0.023669 | 24.7% |  318.0 |MPI_Isend
||   0.0% |  0.004125 | 0.004316 | 58.4% |  318.0 |MPI_Irecv
||   0.0% |  0.000298 | 0.000013 |  4.8% |   55.0 |MPI_Allreduce
||   0.0% |  0.000033 | 0.000013 | 32.8% |    1.0 |MPI_Cart_create
||================================================================
|  0.0% |  0.008614 |       -- |    -- |   59.0 |MPI_SYNC
||----------------------------------------------------------------
||   0.0% |  0.006696 | 0.006627 | 99.0% |    2.0 |MPI_Barrier(sync)
||   0.0% |  0.001802 | 0.001399 | 77.6% |   55.0 |MPI_Allreduce(sync)
||   0.0% |  0.000061 | 0.000052 | 86.3% |    1.0 |MPI_Init(sync)
||   0.0% |  0.000056 | 0.000051 | 91.7% |    1.0 |MPI_Finalize(sync)
|=================================================================
```

User functions

Communication

Synchronisation

Options for Tracing

- **More information is given in the `pat_build` man page**
  - **-u** Create new trace intercept routines for those functions that are defined in the respective source file owned by the user.
  - **-w** Make tracing the default experiment and create new trace intercept routines for those functions for which no trace intercept routine already exists. If -t, -T, or the trace build directive are not specified, only those functions necessary to support the CrayPat runtime library are traced. If -t, -T, or the trace build directive are specified, and -w is not specified, only those function points that have pre-existing trace intercept routines are traced.
  - **-T tracefunc** Instrument program to trace the function references to tracefunc. This option applies to all user-defined entry points as well as to those that appear in the predefined function groups listed under the -g option. Use the nm or readelf command to determine function names to specify for tracing. The name of the function is the name used when the program is linked. For Fortran 90 and C++ programs, this is the mangled form of the name. If tracefunc begins with an exclamation point (!) character, references to tracefunc are not traced.
  - **-t tracefile** Instrument program to trace all function references listed in tracefile.
- **Only true function calls can be traced. Functions that are inlined by the compiler or that have local scope in a compilation unit cannot be traced.**

Options for Tracing

- More information is given in the `pat_build` man page
  - **-g tracegroup** Instrument the program to trace all function references belonging to the trace function group tracegroup. Only those functions actually executed by the program at runtime are traced. A selection of tracegroup values is:

blas	Basic Linear Algebra subprograms
netcdf	Network Common Data Form
HDF5	HDF5 I/O library
heap	dynamic heap
io	includes stdio and sysio groups
lapack	Linear Algebra Package
mpi	MPI
omp	OpenMP API
sysio	I/O system calls
system	system calls

- More information on the various tracegroup values is given in `$CRAYPAT_ROOT/share/traces` after loading the `perftools` module.

Files generated during regular Profiling

- **a.out+pat+PID-node[s|t].xf: raw data files**
  - Depending on the nature of the program and the environmental conditions in effect at the time of program execution, when executed, the instrumented executable generates one or more data files with the suffix .xf, where:
    - **a.out** is the name of the original program.
    - **PID** is the process ID assigned to the instrumented program at runtime.
    - **node** is the physical node ID upon which the rank zero process executed.
    - **s|t** is a one-letter code indicating the type of experiment performed, either **s** for sampling or **t** for tracing.
  - Use the `pat_report` command to view or dump the .xf file or export it to another file format for use with other applications, i.e. *.ap2 files.

- ***.ap2 files: self contained compressed performance files.**
  - Normally about 5 times smaller than the corresponding set of *.xf files.
  - Only one *.ap2 per experiment compared to potentially multiple *.xf files.
  - Contains the information needed from the application binary and can be reused, even if the application binary is no longer available or if it was rebuilt.
  - Is independent on the version used to generate the ap2 file while the xf files are very version dependent.
  - It is the only input format accepted by Cray Apprentice2 and Reveal.
  - => Delete the xf files after you have the ap2 file.

Using pat_report

- **Always need to run `pat_report` at least once to perform data conversion**
  - Combines information from `xf` output (optimized for writing to disk) and binary with raw performance data to produce `ap2` file (optimized for visualization analysis)
  - Instrumented binary must still exist when data is converted!
  - Resulting `ap2` file is the input for subsequent `pat_report` calls and Reveal or Apprentice[2]
  - `xf` files and instrumented binary files can be removed once `ap2` file is generated.

- **Generates a text report of performance results**
  - Data laid out in tables
  - Many options for sorting, slicing or dicing data in the tables.
    ```
    > pat_report -O <table option> *.ap2
    > pat_report -O help (list of available profiles)
    ```
  - Volume and type of information depends upon sampling vs tracing.

Using pat_report

- **The performance numbers reported are in general an average over all tasks (also explains non-integer values)**
- **Not always meaningful**
  - Master-slave schemes
  - MPMD

```
 Time% |      Time |     Imb. |  Imb. |   Calls |Group
       |           |     Time | Time% |         | Function
       |           |          |       |         |  PE=HIDE

 100.0% | 20.643909 |       -- |    -- | 1149.0 |Total
|-----------------------------------------------------------
|  98.8% | 20.395989 |       -- |    -- |  219.0 |USER
||----------------------------------------------------------
||  91.1% | 18.797060 | 0.115535 |  0.7% |    2.0 |jacobi
||   7.7% |  1.597866 | 0.006647 |  0.5% |    1.0 |initmt
||   0.0% |  0.000402 | 0.000167 | 33.5% |   53.0 |sendp3
```

- **To solve this you can filter the *.ap2 file**

```
> pat_report -sfilter_input='condition' …
```

  - The 'condition' should be an expression involving 'pe' such as 'pe<1024' or 'pe%2==0'.
  - This option is also useful when the size of the full data file makes a report incorporating data from all PEs take too long or exceed the available memory

Combining Sampling and Tracing: APA

- **Motivation for Automatic Profiling Analysis:**
  - For programs that run for only a few seconds, there is no problem with using `pat_build` with the `-u` and `-g` mpi options to trace all user functions.
  - However with a large, long-running program such a trace will inject considerable overhead. It is better to limit tracing to those functions that consume the most time.
  - One can use a preliminary sampling experiment to determine and instrument those functions, referred to as automatic profiling analysis.

  - APA provides a simple procedure to instrument and collect performance data as a first step for novice and expert users.
  - Identifies top time consuming routines through sampling and provides instructions to trace only those routines.
  - Automatically creates instrumentation template customized to application for future in-depth measurement and analysis

Automatic Profiling Analysis (1/2)

```
> module load perftools
```

- Makes the default version of CrayPAT available.

```
> make clean; make
> pat_build himeno.exe
```

- The APA is the default experiment. No option needed.
- The pat_build generates a binary instrumented for sampling (different from the pure sampling shown before.)

```
> aprun -n 24 ./himeno.exe+pat
> pat_report -o myrep.txt himeno+pat+*
```

- Running the “+pat” binary creates a data file or directory.
- Applying pat_report to the *.xf generates an ***.apa** file in addition to the *.ap2 file.

Automatic Profiling Analysis (2/2)

```
> vi *.apa
```

- The *.apa file contains instructions for the next step, i.e. tracing. Modify it according to your needs.

```
> pat_build -O *.apa
```

- Generates an instrumented binary `himeno.exe+apa` for tracing according to the instructions in the *.apa file.

```
> aprun -n 24 ./himeno.exe+apa
> pat_report -o myrep.txt himeno+apa+*
```

- Running the "+apa" binary creates a new data file or directory.
- Applying pat_report to the *.xf generates a new*.ap2 file.

*.apa File after Sampling Experiment

```
# ----------------------------------------------------------------------
#       Collect the default PERFCTR group.

  -Drtenv=PAT_RT_PERFCTR=default
...
# ----------------------------------------------------------------------

#       Libraries to trace.

  -g mpi
# ----------------------------------------------------------------------

#       User-defined functions to trace, sorted by % of samples.


  -w  # Enable tracing of user-defined functions.
      # Note: -u should NOT be specified as an additional option.

# 77.44% 3751 bytes
         -T jacobi

# 5.04% 2467 bytes
         -T initmt
# ----------------------------------------------------------------------

-o himeno.exe+apa # New instrumented program.
```

Suggestion to collect Performance counters

Augment this list if needed, i.e. `–g mpi,io`

Add or remove functions as needed.

Create the binary for tracing

General Remarks

- **Always check that the instrumenting binary has not affected the run time notably compared to the original**
- **Collecting event traces on large numbers of frequently called functions, or setting the sampling interval very low can introduce a lot of overhead (check `trace-text-size` option to `pat_build`)**
- **MUST run on Lustre**
  - Avoid running on the home directory. Use a workspace.
- **The runtime analysis can be modified through the use of environment variables of the form `PAT_RT_*`**
  - Number of files used to store raw data:
    - 1 file created for program with 1 – 256 processes
    - √n files created for program with 257 – n processes
    - Ability to customize with PAT_RT_EXPFILE_MAX
  - Check the `PAT_LD_OBJECT_TMPDIR` variable if you cannot preserve the original build tree.

Hardware Performance Counters

- **CrayPat supports the use of hardware counters to collect hardware events**
  - Most counters accessed through the PAPI interface.
  - Predefined sets of hardware counters are specified that can be instrumented for performance analysis experiment.
  - Number of simultaneous counters limited by hardware.
- **CrayPat provides information at the function call level on hardware features like caches, vectorization and memory bandwidth. Very useful feature for understanding application performance bottlenecks.**
- **HWPC collection can slow down the execution notably.**
  - Should be used within a tracing experiment only for a small set of functions or ideally through an automatic performance analysis.

Hardware Counters Selection

- **HW counter collection enabled with `PAT_RT_PERFCTR` environment variable (not set by default)**

```
export PAT_RT_PERFCTR=<event list> | <group>
```

  - Counter events are specified in a comma-separated list. Event names and groups from any and all components may be mixed as needed. To list the names of the individual events on your system, use the `papi_avail` and `papi_native_avail` commands which are explained in the `papi_counters` man page.
  - Alternatively, counter group numbers can be used in addition to or in place of individual event names, to specify one or more predefined performance counter groups. A set number can be used to select a group of predefined hardware counters events (recommended). The groups are given in the hwpc man page (contents in $CRAYPAT_ROOT/share/counters/)
  - An overview of events is given in `pat_help->counters->haswell`
  - Aries network performance counters is found in the `nwpc(5)` man page.
  - Intel Running Average Power Limit and Cray Power Management in `rapl(5)`, and info on Performance API (PAPI) in `intro_papi(5)`.

Haswell HW counter groups (hwpc man page)

```
Table 5. Intel Haswell Event Sets
    ------------------------------------------------------------
    Group        Description
    ------------------------------------------------------------
    0            D1 with instruction counts
    1            Summary with cache and TLB metrics (default)
    2            D1, D2, and L3 metrics
    3-5          Not used
    6            Micro-op queue stalls
    7            Back-end stalls
    8            Instructions and branches
    9            Instruction cache
    10           Cache hierarchy
    19           Prefetches
    23           Summary with cache and TLB metrics (same as 1)
    ------------------------------------------------------------
    Cray XC40 and Cray XC40-AC systems only: Hardware performance counters
    do not support floating-point operations.
```

Most useful for measuring cache efficiency. List of events is given in $CRAYPAT_ROOT/share/counters

Example: HW counter data and derived metrics

```
==============================================================================
  USER / jacobi
------------------------------------------------------------------------------
  Time%                                                    91.0%
  Time                                                 18.783816 secs
  Imb. Time                                             0.131366 secs
  Imb. Time%                                                0.8%
  Calls                                0.106 /sec            2.0 calls
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                    1874027894
  CPU_CLK_THREAD_UNHALTED:THREAD_P                   52330735798
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                   15309079
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                   9590363
  L1D:REPLACEMENT                                     2490612461
  L2_RQSTS:ALL_DEMAND_DATA_RD                         1255673984
  L2_RQSTS:DEMAND_DATA_RD_HIT                          495319777
  MEM_UOPS_RETIRED:ALL_LOADS                          7905309689
  User time (approx)                  18.783 secs    46977527366 cycles  100.0% Time
  CPU_CLK                              2.792GHz
  TLB utilization                     317.49 refs/miss     0.620 avg uses
  D1 cache hit,miss ratios             68.5% hits          31.5% misses
  D1 cache utilization (misses)         3.17 refs/miss     0.397 avg hits
  D2 cache hit,miss ratio              69.5% hits          30.5% misses
  D1+D2 cache hit,miss ratio           90.4% hits           9.6% misses
  D1+D2 cache utilization              10.40 refs/miss     1.300 avg hits
  D2 to D1 bandwidth                4080.191MiB/sec  80363134952 bytes
  Average Time per Call                                 9.391908 secs
  CrayPat Overhead : Time               0.0%
```

Raw counters

derived

Example: Observations and suggestions

```
D1 + D2 cache utilization:

    7.7% of total execution time was spent in 1 functions with combined
    D1 and D2 cache hit ratios below the desirable minimum of 80.0%.
    Cache utilization might be improved by modifying the alignment or
    stride of references to data arrays in these functions.

    D1+D2 cache hit ratio   Time%   Function

     58.9%                   7.7%   initmt


TLB utilization:

    7.7% of total execution time was spent in 1 functions with fewer
    than the desirable minimum of 200 data references per TLB miss. TLB
    utilization might be improved by modifying the alignment or stride
    of references to data arrays in these functions.

       LS per TLB DM   Time%   Function

       5.21             7.7%   initmt
```

Compiler Feedback (CCE)

- With CCE use `-rm` for Fortran or `-hlist=a` for C/C++
- For each source file a corresponding `*.lst` file is created.

```
%%%    L o o p m a r k   L e g e n d    %%%

    Primary Loop Type        Modifiers
    ------- ---- ----        ---------
    A - Pattern matched      a - atomic memory operation
                             b - blocked
    C - Collapsed            c - conditional and/or computed
    D - Deleted
    E - Cloned
    F - Flat - No calls      f - fused
    G - Accelerated          g - partitioned
    I - Inlined              i - interchanged
    M - Multithreaded        m - partitioned
                             n - non-blocking remote transf.
                             p - partial
                             r - unrolled
                             s - shortloop
    V - Vectorized           w - unwound
```

```
  191.    C-------------<   for(i=0 ; i<MIMAX ; ++i)
  192.    C C-----------<     for(j=0 ; j<MJMAX ; ++j)
  193.    C C VCr2------<       for(k=0 ; k<MKMAX ; ++k){
  194.    C C VCr2                a[0][i][j][k]=0.0;
  195.    C C VCr2                a[1][i][j][k]=0.0;
  196.    C C VCr2                a[2][i][j][k]=0.0;
  197.    C C VCr2                a[3][i][j][k]=0.0;
202.    C C VCr2                c[1][i][j][k]=0.0;
  203.    C C VCr2                c[2][i][j][k]=0.0;
  204.    C C VCr2 A---<>         p[i][j][k]=0.0;

CC-6005 CC: SCALAR File = himeno.c, Line = 193
  A loop was unrolled 2 times.

CC-6204 CC: VECTOR File = himeno.c, Line = 193
  A loop was vectorized.

CC-6231 CC: VECTOR File = himeno.c, Line = 204
  A statement was replaced by a library call.
```

API for adding User Instrumentation

- **The CrayPat API calls enable you to insert functions into your source code that write special tracing records into the experiment data file at runtime**

  - API calls are supported in both Fortran and C. After the perftools module is loaded, the include files that define the CrayPat API can be found in the `$CRAYPAT_ROOT/include` directory and consist of the C header file, `pat_api.h`, and the Fortran and Fortran 77 header files, `pat_apif.h` and `pat_apif77.h`, respectively.

  - `int PAT_region_begin (int id, char *label)`
    - id is a unique identifier for the region,
    - Label is the description that will appear in profiling output.
  - `int PAT_region_end (int id)`
    - id must match begin call.

- **Fortran equivalents, like MPI, are subroutines with extra final integer argument for return value**
- **More information is given in the `pat_build` man page. For further examples of using CrayPat API calls in source code, see the topic "API" in the `pat_help` system.**

PAT Regions example

```
include "pat_apif.h“
...
call PAT_region_begin( 1, "step 1", istat )
! the execution of this code segment will appear in
! CrayPAT output as “step 1”
...
call PAT_region_end( 1, istat )
...
call PAT_region_begin( 2, "step 2", istat )
! the execution of this code segment will appear in
! CrayPAT output as “step 2”
...
call PAT_region_end( 2, istat )
...
```

`-DCRAYPAT` defined by CCE compilers

PAT_region example

```
  100.0% | 58225.2 |     -- |    -- |Total
 |-----------------------------------------------
 |  91.2% | 53072.9 |     -- |    -- |USER
 ||----------------------------------------------
 ||  43.9% | 25571.3 |  388.7 |  1.5% |calc_force_
 ||  29.7% | 17292.9 |  289.1 |  1.6% |calc_p_
 ||  14.3% |  8305.5 |   75.5 |  0.9% |pair_table_
 ||   1.4% |   844.2 |   74.8 |  8.2% |predict_
 ||==============================================
 |   7.5% |  4363.8 |     -- |    -- |MPI
 ||----------------------------------------------
 ||   3.8% |  2229.9 |  905.1 | 28.9% |MPI_SENDRECV
 ||   2.1% |  1208.5 | 1050.5 | 46.6% |MPI_BARRIER
 ||   1.4% |   829.7 |  487.3 | 37.1% |MPI_ALLREDUCE
 ||==============================================
 |   1.4% |   788.1 |     -- |    -- |ETC
 |===============================================
```

- But calc_force is 494 lines and Calc_P 334 lines long.
- Introduce 4 PAT regions to the code, two for Calc_Force and two for Calc_P, according to the steps annotated in the code.

PAT_region example

```
100.0% | 58359.5 |      -- |     -- |Total
|-----------------------------------------------
|  90.9% | 53023.8 |      -- |     -- |USER
||----------------------------------------------
||  43.1% | 25131.3 |   510.7 |   2.0% |#3.force_step1
||  28.9% | 16879.8 |   345.2 |   2.0% |#1.p_step1
||  14.3% |  8317.4 |    66.6 |   0.8% |pair_table_
||   1.4% |   834.7 |    79.3 |   8.7% |predict_
||==============================================
|   7.8% |  4551.9 |      -- |     -- |MPI
||----------------------------------------------
||   3.9% |  2249.3 |   941.7 |  29.6% |MPI_SENDRECV
||   2.3% |  1330.7 |  1269.3 |  48.9% |MPI_BARRIER
||   1.5% |   878.0 |   496.0 |  36.2% |MPI_ALLREDUCE
||==============================================
|   1.3% |   783.5 |      -- |     -- |ETC
|===============================================
```

Narrowed down to some ½ of the lines of code – could refine further

Hands-On Sessions

- **Simple codes are provided for the workshop but users are highly encouraged to experiment with their own applications.**
  - **Himeno Benchmark:** Iterative solution of the Poisson equation by means of an iterative Jacobi method. MPI versions for C and Fortran. Make copies of the `startfiles/` directories.
  - **VH1**: The code used for the Reveal walk through.
- **Copy the codes from:**

  `/zhome/academic/HLRS/hlrs/hpcaespo/VI-HPS_Tuning_Workshop`
- **The CrayPAT and Reveal step-by-step guides are described in a PDF document.**
- **Remember: Do not run any computational job on the home file system.**