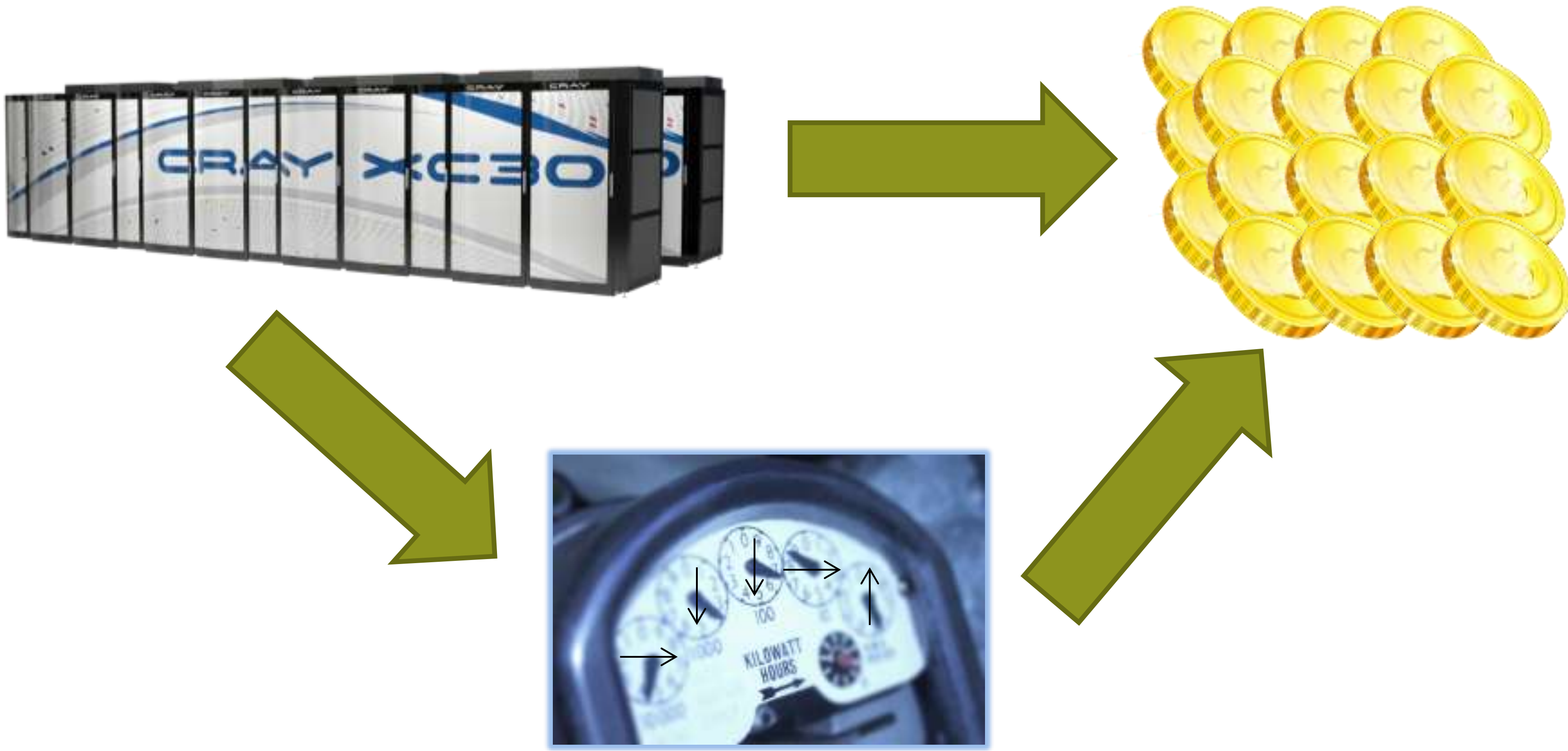
Introduction to performance analysis

Performance Analysis – Motivation (1)

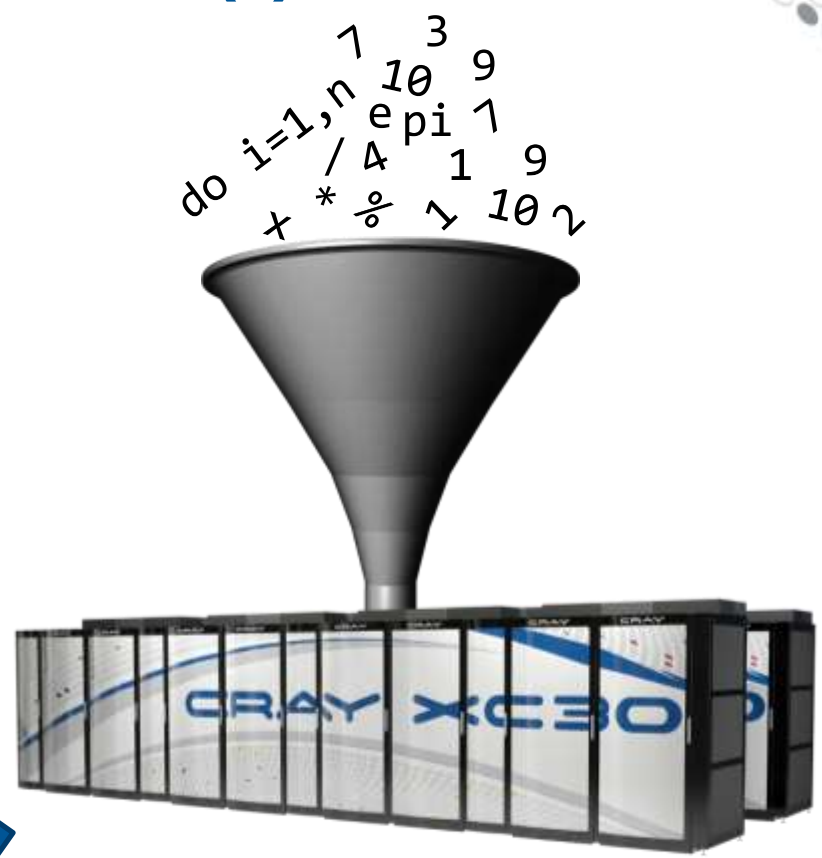


Even the most reasonably priced supercomputer costs money to buy and needs power to run (money)

Performance Analysis – Motivation (2)

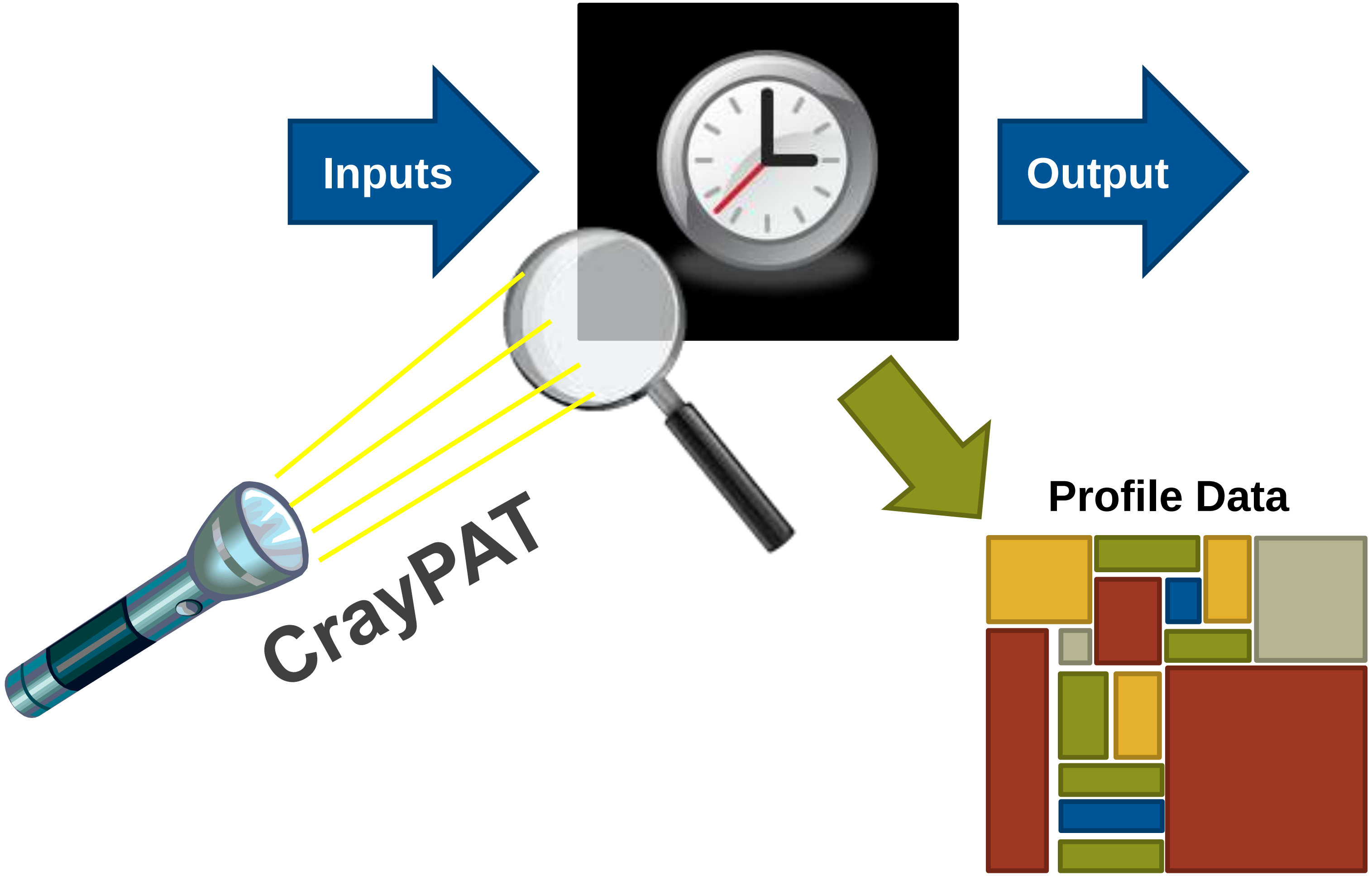
We want to get the most science and engineering through the system as possible.

The more efficient codes are the more productive scientists and engineers can be.



Performance Analysis – Motivation (3)

To optimise code we must know what is taking the time



Sampling and Event Tracing

- **When we instrument a binary, we have to choose when we will collect performance information:**

1. **Sampling**
   - By taking regular snapshots of the applications call stack we can create a statistical profile of where the spends most time.
   - Snapshots can be taken at regular intervals in time or when some other external even 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.

- **`pat_build` options define how binaries are instrumented, for sampling or 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.

CrayPAT's Design Goals

- **Assist the user with application performance analysis and optimization**
  - Help user identify important and meaningful information from potentially massive data sets
  - Help user identify problem areas instead of just reporting data
  - Bring optimization knowledge to a wider set of users
- **Focus on ease of use and intuitive user interfaces**
  - Lightweight and automatic program instrumentation
  - Automatic Profiling Analysis mode to bootstrap the process
- **Target scalability issues in all areas of tool development**
  - Work on user codes at realistic core counts with thousands of processes/threads
  - Integrate into large codes with millions of lines of code
- **Be a universal tool**
  - Basic functionality available to all compilers on the system
  - Additional functionality available from the Cray compiler

The Three Stages of CrayPAT

- **There are three fundamental stages with accompanying tools**
  1. Instrumentation
     - Use **`pat_build`** to apply instrumentation to program binaries
  2. Data Collection
     - Transparent collection via CrayPAT's run-time library
  3. Analysis
     - Interpreting and visualizing collected data using a series of post-mortem tools:
       1. **`pat_report`**: a command line tool for generating text reports
       2. **Cray Apprentice²**: a graphical performance analysis tool
       3. **Reveal**: Graphical performance analysis and code restructuring tool

- **Documentation is provided via**
  - The `pat_help` system
  - And the traditional `man craypat`

Instrumentation

- All instrumentation is done by `pat_build`, a stand-alone utility that automatically instruments an existing application for performance collection

- **Requires no source code or makefile modification by default**
  - Automatic instrumentation at group (function) level
    - Example groups: mpi, io, heap, math SW, …

- **Performs link-time instrumentation**
  - **Requires object files to still exist, have been compiled with the wrapper scripts while the perftools module was loaded**
  - Able to generates instrumentation on optimized code
  - Creates a new stand-alone instrumented program
  - Preserves original binary

- **To use the tools perftools must be loaded during the compile , at linking and at instrumentation (but not runtime)**
  - `module load perftools`

Creating and running a sampling binary

- `pat_build` **creates sampling binaries by default**
- **To build a binary with sampling instrumentation, run:**
  - `pat_build <exe>`
- **This will create a new executable in the form.**
  - `<exe>+pat`
- **Run this executable as normal in place of the original.**
- **Profiling data will be created in the form of**
  - `*s*.xf` files (s for sampling)
  - Or a directory containing multiple `*s*.xf` files

Creating event tracing binaries

- **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
- **Enabled with `pat_build` `–g`, `-u`, `-T`, `-t` or `–w` options**
  - `-w` instructs `pat_build` to create trace points in the binary for user functions (required if user functions need to be traced)
  - `-g` enables tracing of system functions and system libraries, e.g. `mpi`, `blas`, `caf`, `upc`, `fftw`
  - `-u` creates instrumentation for ALL the user defined functions
  - `-T` creates instrumentation for specific user function (may be defined multiple times for different functions, or limited regular expressions)
  - `-t` specifies a file containing a list of functions to create instrumentation for.
- **A new binary will be created which can be run in place of the original.**
- **Data is output in `*.t.xf` file or files (`t` for tracing) in the run directory**

-g tracegroup (subset)

blas	Basic Linear Algebra subprograms
CAF	Co-Array Fortran (Cray CCE compiler only)
HDF5	HDF5 I/O library
heap	dynamic heap
io	includes stdio and sysio groups
lapack	Linear Algebra Package
math	ANSI math
mpi	MPI
omp	OpenMP API
omp-rtl	OpenMP runtime library
pthreads	POSIX threads
shmem	SHMEM
sysio	I/O system calls
system	system calls
upc	Unified Parallel C (Cray CCE compiler only)

For a full list, please see man pat_build

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 Apprentice^2
  - `xf` 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.

Why Should I generate an ".ap2" file?

- The ".ap2" file is a self contained compressed performance file
- Normally it is about 5 times smaller than the ".xf" file
- Contains the information needed from the application binary
  - 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
  - The xf files are very version depending
- It is the only input format accepted by Cray Apprentice^2
- => Delete the xf files after you have the ap2 file

Some important options to pat_report -O

callers	Profile by Function and Callers
callers+hwpc	Profile by Function and Callers
callers+src	Profile by Function and Callers, with Line Numbers
callers+src+hwpc	Profile by Function and Callers, with Line Numbers
calltree	Function Calltree View
heap_hiwater	Heap Stats during Main Program
hwpc	Program HW Performance Counter Data
load_balance_program+hwpc	Load Balance across PEs
load_balance_sm	Load Balance with MPI Sent Message Stats
loop_times	Loop Stats by Function (from -hprofile_generate)
loops	Loop Stats by Inclusive Time (from -hprofile_generate)
mpi_callers	MPI Message Stats by Caller
profile	Profile by Function Group and Function
profile+src+hwpc	Profile by Group, Function, and Line
samp_profile	Profile by Function
samp_profile+hwpc	Profile by Function
samp_profile+src	Profile by Group, Function, and Line

For a full list see pat_report -O help

Break

Automatic Profile Analysis

A two step process to create an guided event trace binary.

Program Instrumentation - Automatic Profiling Analysis

- **Automatic profiling analysis** (APA)
- Provides simple procedure to instrument and collect performance data as a first step for novice and expert users
- Identifies top time consuming routines
- Automatically creates instrumentation template customized to application for future in-depth measurement and analysis

Steps to Collecting Performance Data

- **Access performance tools software**

```
% module load perftools
```

- **Build application keeping .o files (CCE: `-h keepfiles`)**

```
% make clean
% make
```

- **Instrument application for automatic profiling analysis**
  - You should get an instrumented program `a.out+pat`

```
% pat_build -O apa a.out
```

We are telling `pat_build` that the output of this sample run will be used in an APA run

- **Run application to get top time consuming routines**
  - You should get a performance file (“`<sdatafile>.xf`”) or multiple files in a directory `<sdatadir>`

```
% aprun … a.out+pat  (or qsub <pat script>)
```

Steps to Collecting Performance Data (2)

- Generate text report and an `.apa` instrumentation file

```
% pat_report -o my_sampling_report [<sdatafile>.xf |
   <sdatadir>]
```

- Inspect `.apa` file and sampling report

- Verify if additional instrumentation is needed

Generating Event Traced Profile from APA

- **Instrument application for further analysis (a.out+apa)**

```
% pat_build -O <apafile>.apa
```

- **Run application**

```
% aprun … a.out+apa  (or  qsub <apa script>)
```

- **Generate text report and visualization file (.ap2)**

```
% pat_report -o my_text_report.txt [<datafile>.xf | <datadir>]
```

- **View report in text and/or with Cray Apprentice²**

```
% app2 <datafile>.ap2
```

Modifying CrayPAT's collection behaviour

Changing how and which data are collected at runtime

Launching instrument variables

- **Once a binary has been instrumented for either sampling or tracing it should be run in place of the original binary.**
  - Always check that instrumenting the binary has not affected the run time compared to the original binary
  - Collecting event traces on large numbers of frequently called functions, or setting the sampling interval very low can introduce a lot of overhead.

- **MUST run on Lustre**
  - Avoid running on the home directory, use a `/wrk`

- **The runtime analysis can be modified through the use of environment variables**
  - All runtime CrayPAT environment variables are of the form `PAT_RT_*`

Example Runtime Environment Variables

- **Optional timeline view of program available**
  - `export PAT_RT_SUMMARY=0`
  - View trace file with Cray Apprentice[2]

- **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`

- **Request hardware performance counter information:**
  - `export PAT_RT_HWPC=<HWPC Group>`
  - Can specify events or predefined groups

API for controlling tracing

- `#include <pat_api.h>`
- `int PAT_state (int state)`
  - State can have one of the following:
    - `PAT_STATE_ON`
    - `PAT_STATE_OFF`
    - `PAT_STATE_QUERY`
- `int PAT_record (int state)`
  - Controls the state for all threads on the executing PE. As a rule, use `PAT_record()` unless there is a need for different behaviors for sampling and tracing
    - `int PAT_sampling_state (int state)`
    - `int PAT_tracing_state (int state)`
- `int PAT_trace_function (const void *addr, int state)`
  - Activates or deactivates the tracing of the instrumented function
- `int PAT_flush_buffer (void)`

Fortran equivalents, like MPI, are subroutines with extra final integer argument for return value

API for adding user instrumentation

- **Users are able to define their own trace points via the region API.**

- `#include <pat_api.h>`
- `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` is a unique identifier for the region, must match begin call.

Fortran equivalents, like MPI, are subroutines with extra final integer argument for return value

Trace On / Trace Off Example

```
include "pat_apif.h“
!  Turn data recording off at the beginning of execution.
call PAT_record( PAT_STATE_OFF, istat )
...
!  Turn data recording on for two regions of interest.
call PAT_record( PAT_STATE_ON, istat )
…
call PAT_region_begin( 1, "step 1", istat )
...
call PAT_region_end( 1, istat )
…
call PAT_region_begin( 2, "step 2", istat )
...
call PAT_region_end( 2, istat )
…
!  Turn data recording off again.
call PAT_record( PAT_STATE_OFF, istat )
…
```

-DCRAYPAT defined by CCE compilers