User-level Power Monitoring and Application Performance on Cray XC30 Supercomputers

Alistair Hart, Harvey Richardson
Cray Exascale Research Initiative Europe
King's Buildings
Edinburgh, UK
{*ahart,harveyr*}*@cray.com*

Jens Doleschal, Thomas Ilsche, Mario Bielert
Technische Universität Dresden,
ZIH
Dresden, Germany
{*jens.doleschal,thomas.ilsche,mario.bielert*}*@tu-dresden.de*

Matthew Kappel
Cray Inc.
St. Paul MN, USA
mkappel@cray.com

***Abstract*—In this paper we show how users can access and display new power measurement hardware counters on Cray XC30 systems (with and without accelerators), either directly or through extended prototypes of the Score-P performance measurement infrastructure and Vampir application performance monitoring visualiser.**

This work leverages new power measurement and control features introduced in the Cray XC supercomputer range and targeted at both system administrators and users.

We discuss how to use these counters to monitor energy consumption, both for complete jobs and also for application phases. We then use this information to investigate energy efficient application placement options on Cray XC30 architectures, including mixed use of both CPU and GPU on accelerated nodes and interleaving processes from multiple applications on the same node.

***Keywords*-energy efficiency; power measurement; application analysis**

I. INTRODUCTION

Energy and power consumption are increasingly important topics in High Performance Computing. Wholesale electricity prices have recently risen sharply in many regions of the world, including in the European states [1], prompting an interest in lowering energy consumption of HPC systems. Environmental (and political) concerns also motivate HPC data centres to reduce their "carbon footprints". This has driven an interest in energy-efficient supercomputing, as shown by the rise in popularity of the "Green 500" list of the most efficient HPC systems since its introduction in 2007 [2]. In many regions, the need to balance demand across the electricity supply grid has also led to a requirement to be able to limit the maximum power drawn by a supercomputing system.

Looking forward, energy efficiency is one of the paramount design constraints on a realistic exascale supercomputer. The United States Department of Energy has set a goal of 20MW [3] for such a machine. Compared to a typical petaflop system, this requires a thousand-fold leap in computational speed in only three times the power budget.

Energy efficiency goes beyond hardware design, however. To deliver sustained, but energy-efficient, performance of real applications will require software engineering decisions, both at the systemware level but also in the applications themselves. Such application decisions might be made when the software is designed or at runtime via an autotuning framework.

For these to be possible, fine-grained instrumentation is needed to measure energy and power usage not just of overall HPC systems, but of individual components within the architecture. This information also needs to be accessible not just to privileged system administrators but also to individual users of the system, and in a way that is easily correlated with the execution of their applications.

In this paper, we describe some ways that users can monitor the energy and power consumption of their applications when running on the Cray XC supercomputer range We exploit some of the new power measurement and control features that were introduced in the Cray Cascade-class architectures, aimed both at system-administrators and non-privileged users.

We demonstrate how users can access and display new power measurement hardware counters on Cray XC30 systems (with and without accelerators) to measure energy consumption by node-specific hardware components. Of particular interest to users is charting power consumption as an application runs, and correlating this with other information in full-featured performance analysis tools. We therefore are extending the Score-P scalable performance measurement infrastructure for parallel codes [4] to gather power measurement counter values on Cray XC30 systems (with and without accelerators). The framework is compatible with a number of performance analysis tools, and we show how one example, Vampir [5], displays this data in combination with application monitoring information. We describe how this was achieved and give some usage examples of these tools on Cray XC30 systems, based on benchmarks and large-scale applications.

We will use this information to illustrate some of the design decisions that users may then take when tuning their applications, using a set of benchmark codes. We will do this for two Cray XC30 node architecture designs, one based

solely on CPUs and another using GPU accelerators. For CPU systems, we will show that reducing the clockspeed of applications at runtime can reduce the overall energy consumption of applications. This shows that modern CPU hardware already deviates from the "received wisdom" that the most energy-efficient way to run an application is always the fastest although, as we discuss, the energy savings are perhaps not currently large enough to justify the increased runtime.

For hybrid systems, the GPU does indeed offer higher performance for lower energy consumption, but both of these are further improved by harnessing both CPU and GPU in the calculation, and we show a simple way of doing this.

The structure of this paper is as follows. In Sec. II we describe the Cray XC30 hardware used for the results in this paper, some relevant features of the system software and the procedure for reading the user-accessible power counters. We use these counters to measure the power draw of the nodes when idle. In Sec. III we present performance results (both runtime- and energy-based) when running some benchmark codes in various configurations on CPUs and/or GPUs. The energy results here were obtained using simple API calls; in Sec. IV we describe how the Score-P/Vampir framework has been modified to provide detailed tracing of application energy use, and present results both for synthetic benchmarks and a large-scale parallel application (Gromacs). Finally, we draw our conclusions in Sec. V.

This work was carried out in part as part of the EU FP7 project "Collaborative Research in Exascale Systemware, Tools and Applications (CRESTA)" [6]. Some early results from this work were presented in Ref. [7].

Related work reported at this conference can be found in Refs. [8], [9], [10].

II. The hardware considered

For this work, we consider two configurations of the "Cascade-class" Cray XC30 supercomputer. The first uses nodes containing two twelve-core Intel Xeon E5-2697 "Ivybridge" CPUs with a clockspeed of 2.7GHz (as used in, for instance, the UK "Archer" and ECMWF systems). The second uses hybrid nodes containing a single eight-core Intel Xeon E5-2670 "Sandybridge" CPU with a clockspeed of 2.6GHz and an Nvidia Tesla K20x "Kepler" GPU (as used in the Swiss "Piz Daint" system installed at CSCS). For convenience, we refer to these configurations as "Marble" and "Graphite" respectively.

A. Varying the CPU clockspeed at runtime

Linux OSes, including the Cray Linux Environment that runs on the Cray XC30 compute nodes, offer a CPUFreq governor that can be used to change the clock speed of the CPUs [11]. The default governor setting is "performance" on Cray XC30 systems, with CPUs consistently clocked to their highest settings. For this work, we concentrate on the "userspace" governor, which allows the user to set the CPU to a specific frequency ("p-state").

The p-state is labelled by an integer that corresponds to the CPU clock frequency in kHz. So a p-state label of 2600000 corresponds to a 2.6GHz clock frequency[1]. The only exception is that the Intel CPUs studied here have the option (depending on power and environmental constraints) of entering a temporary boost state above the nominal top frequency of the CPU; this option is available to the hardware if the p-state is set to 2601 (for a 2.6GHz processor). For the Sandybridge CPUs, the p-states ranged from 1200 to 2600, with a boost-able 2601 state above that (where the "turbo frequency" can be as high as 3.3GHz). For the Ivybridge CPUs, the range was from 1200 to 2700, with a boost-able 2701 state (where the frequency can reach 3.5GHz).

On Cray systems, the p-state of the node can be varied at application launch through the `--p-state` flag for the ALPS `aprun` job launch command (which also sets the governor to "userspace"). The range of available p-states on a given node is listed in file:

```
/sys/devices/system/cpu/cpu0/cpufreq/ \
      scaling_available_frequencies
```

B. Enhanced MPMD with Cray ALPS on the Cray XC30

The `aprun` command provides the facility to run more than one binary at a time as part of the same MPI application (sharing the `MPI_COMM_WORLD` communicator). Historically, this MPMD mode only supported running a single binary on all cores of a given node. In this work, we take advantage of a new capability to combine different binaries on the same node. This is done by using `aprun` to execute a script multiple times in parallel (once for each MPI rank in our job). Within this script, a new environment variable `ALPS_APP_PE` gives the MPI rank of that instance of the script. We can then use modular arithmetic (based on the desired number of ranks per node) to decide which MPI binary should be executed (and set appropriate control variables). Environment variable `PMI_NO_FORK` must be set in the main batch jobscript to ensure the binaries launch correctly. Fig. 1 gives an example of mixing (multiple) OpenACC and OpenMP executables on the same nodes, with appropriate `-cc` binding.

C. User-accessible power counters

The Cray Linux Environment OS provides access to a set of power counters via files in directory `/sys/cray/pm_counters`. Further details of these counters are implemented can be found in Ref. [8].

Counter `power` records the instantaneous power consumption of the node in units of Watts. This node power includes CPU, memory and associated controllers and other hardware contained on the processor daughter card. It does

[1]For brevity, we drop the final three zeros when quoting p-state values.

```
# Excerpt from job script (written in bash)
export PMI_NO_FORK=1
export CRAY_CUDA_MPS=1
cclist='0:1:2,3:5,6'
aprun -n16 -N4 -cc $cclist ./wrapper.bash
```

```
#!/bin/bash
# wrapper.bash
node_rank=$((ALPS_APP_PE % 4))
if [ $node_rank -lt 2 ]; then
  export OMP_NUM_THREADS=1
  ./openacc.x
else
  export OMP_NUM_THREADS=2
  ./openmp.x
fi
```

Figure 1. Launching a 16-rank MPMD job with 4 ranks per node: two OpenACC and two OpenMP with 2 threads per rank.

not include the energy consumption of the (shared) Aries network controllers on that node, nor of any other hardware in the chassis or cabinet. Counter `energy` records the cumulative energy consumption of the node (in Joules) from some fixed time in the past.

The `power` and `energy` counters are available for all node architectures and always measure the consumption of the full node. On accelerated nodes, additional counters `accel_power` and `accel_energy` measure the part of the full node consumption that is due to the accelerator.

These counters are updated with a frequency of 10Hz, giving a time resolution of 100ms, which is fine-grained enough to resolve different phases of application execution, but not individual instructions (which would need to be measured using multiple repetitions in separate benchmarks). When the counters are updated, an integer-valued counter `freshness` is also incremented (although the value should not be interpreted as a timestamp). Using the Linux OS, changes to all counters are done atomically, so the values cannot be read until all counters have updated. The counters must, however, be read sequentially in a script or application. To ensure that a consistent set of values are obtained, the procedure for reading should be as follows.

The value of the `freshness` counter should be read twice: once immediately before reading the other counters, and then once immediately after they have all been read. If the two `freshness` values are the same, then we have a consistent set of counter readings. If not, we should repeat the process.

A simple bash shell script that implements this for just the energy counters is shown in Fig. 2. No inputs are expected by the script and the outputs are `ENERGY`, `ACCEL_ENERGY` and `FRESHNESS`.

Scripts like this are useful for measuring the energy used by the nodes when executing a complete application. If the application is executed using P nodes (usually with multiple processes per node), we can execute the script (once on each node, using command `aprun -nP -N1`) before and after the application execution. Some postprocessing of the job log file is then needed to collate the readings for each node and then subtract the "before" and "after" readings and sum the results across nodes[2].

The advantage of this approach is that it requires no modification of the application, but it cannot provide any information on how energy usage changed during the phases of the application. For instance, we may want to measure only the energy consumption for the main part of an application, excluding the initialisation process. For a multi-phase application, we may want to break down the energy consumption across the phases to target optimisation work.

For this reason, as part of the CRESTA project, we developed `pm_lib`, a small library with associated application programming interface (API) that allows Fortran, C or C++ programs to read multiple counters via a single library call [13].

Simple (and incomplete examples of use are given in Fig. 3. The user can insert measurement calls into an application at any point. This is at the cost of modifying the source code, both to insert the API calls and also to appropriately subtract readings within a given parallel process and then sum these differences over processes. Such manual instrumentation has a low measurement overhead, but is usually only practical for simple codes like the benchmarks that we discuss in Sec. III. For larger applications, where a greater resolution of measurements is required, a more systematic approach is required. An example of this is the Score-P/Vampir tracing framework that we discuss in Sec. IV. This framework, however, internally collects the counter information using a very similar API.

D. Idle power draw

The first stage in understanding the energy consumption of applications is to measure the baseline power consumption of an idle node in the various available p-states. We measured the energy consumption of each node (and, where relevant, of the accelerator on each node) while each CPU core executed a 10s sleep command. We then divide these figures by the time interval to give the mean power.

For Marble nodes, the base power increases fairly steadily from 91.3W to 119.5W as the p-state is raised from 1200 to 2700. In the boost-able 2701 state, the base power is essentially unchanged at 119.9W. This is not surprising; if the CPU is idle, we would not expect the hardware to use the boosting.

[2] This information can also be obtained using the Cray RUR reporting framework [12], if enabled.

```
PM=/sys/cray/pm_counters                                        # The location of the counters
good_freshness=0                                   # Value 1 when we have reliable measurements
while [ $good_freshness -eq 0 ]; do                          # Loop until measurements reliable
  start_freshness=$(cat $PM/freshness)
  ENERGY=$(awk '{print $1}' $PM/energy)
  ACCEL_ENERGY=$(awk '{print $1}' $PM/accel_energy)
  end_freshness=$(cat $PM/freshness)
  if [ $end_freshness -eq $start_freshness ]; then
    good_freshness=1                                            # Measurements are reliable...
    FRESHNESS=$end_freshness
  fi
done                                           # ... otherwise we should repeat the measurements
```

Figure 2. A simple bash shell script to reliably read the energy counters; output values are in upper case.

For Graphite nodes, the base node power increases smoothly from 59.6W to 67.1W as the p-state is raised from 1200 to 2600. If the boost-able 2601 state is used, the base power consumption jumps to 73.3W. Part of this node power is due to the GPU; it draws a relatively constant 14.6W, with the only deviation being for the top, boost-able p-state, when the accelerator draws 18.9W.

When an application runs, we can multiply these base power figures by the runtime to understand what portion of the energy consumption is the baseline, idle consumption of the nodes.

III. Benchmarks

The NAS Parallel Benchmarks (NPB) suite provides simple applications that mimic different phases of a typical Computational Fluid Dynamics (CFD) application [14]. We use version 3.3.1 of the suite, parallelised with MPI and compiled using the Cray Compilation Environment (CCE) with no further performance tuning.

The global problem size (known as "CLASS" within the NPB) and the number of MPI ranks are fixed at compile time; the latter is generally restricted to be a power of two. To ensure that we equally load the 24-core Marble nodes used in a calculation, we will run the benchmarks using 8 (of the possible 12) cores per CPU. For the exploratory studies in this paper, we use 4 nodes per calculation and the appropriate problem size is CLASS=C.

By default, each application reports the runtime and we modified the application using `pm_lib` to also measure the energy expenditure of the nodes. We can then calculate some derived metrics. The energy divided by the runtime gives the mean power expenditure. Given the number of floating point operations ("flop") made by the application, we can construct the floating point operations per second ("flop/s" or "flops") measure of runtime performance (as is used to rank supercomputers in the Top 500 list [15]). The flop count can either be done manually, counting the operations in the source code (as is done in the NPB), or using hardware counters in the CPU.

An alternative metric is the number of floating point operations carried out per Joule of expended energy (flop/J or, equivalently, flops/W), as used in the Green 500 list [2]; we will calculate this using the nodal energy expenditure.

A. *Marble results*

We begin by studying benchmark performance on the pure CPU Marble nodes, investigating how hyperthreads affect energy use. The benchmarks were run in two ways across 4 nodes. First, we compiled to use 64 ranks and ran these with one rank per physical core (`aprun -n64 -S8 -j1`). This was compared with compiling the same global problem size with 128 ranks and run utilising the two hyperthreads available on each Intel core (`aprun -n128 -S16 -j2`).

The results are shown in Fig. 4. Each row shows one of the five benchmarks considered. For each benchmark, the left graph shows application performance as reported by the benchmark. The right graph shows the combined energy consumption of the four nodes. In each case, we vary the p-state of the node between the lowest and highest settings. Each data point shown is the mean calculated from ten independent runs of the benchmark, occasionally excluding obvious outliers from the average.

As expected, we see a decrease in application performance as the p-state is reduced. As is also commonly seen, some applications (EP) give improved performance with hyperthreading, and some (CG, FT, MG) without. IS is an unusual case, as hyperthreading is advantageous at high p-state but not at lower clockspeeds. The energy consumption trends are inverse to those of the runtime; if no hyperthreading gives the best performance, it also gives the lowest energy consumption.

The most interesting result here is the pattern of energy consumption as the p-state is varied. If the node power consumption were independent of the p-state, we would expect the energy consumption to be proportional to the runtime and thus to increase as the p-state of the nodes was reduced. The most energy efficient way to run would therefore be the fastest. We see this pattern for the EP and

```
! Fortran
use pm_lib
type(pm_counter) :: counters(3)=[ &
    PM_COUNTER_FRESHNESS, &
    PM_COUNTER_ENERGY, &
    PM_COUNTER_ACCEL_ENERGY ]
integer(kind=i64) :: values(3)

call pm_init
nc = pm_get_counters(3,counters,values,1)
```

```
// C/C++
#include 'pm_lib.h'
pm_counter_e counters[3]=\{ \
   PM_COUNTER_FRESHNESS, \
   PM_COUNTER_ENERGY, \
   PM_COUNTER_ACCEL_ENERGY \};
pm_counter_value values[3];

pm_init();
nc = pm_get_counters(3,counters,values,1);
```

Figure 3. Simple (and incomplete) examples reading three counters from Fortran or C/C++. Setting the final argument of the measurement call to 1 ensures the counters are read consistently (i.e. atomically).

(to a lesser extent) IS benchmarks. For the CG, FT and MG benchmarks, however, we see a pattern where the most energy efficient way to run is with an intermediate value of the p-state. By sacrificing overall runtime, we can increase the energy efficiency.

This effect has not been commonly seen; on most previous HPC architectures, tuning an application to minimise energy consumption would be the same as tuning to maximise performance. On more modern architectures like the Cray XC30, it is now clear that these are two separate (but probably still not completely independent) operations. Of course, we are limited here to measuring the energy consumption of the nodes, omitting that of the network and other cabinet hardware (notably power conversion losses and cooling), as well as that of all the ancillary infrastructure (rotating storage, login nodes...). Using a linear Power Usage Effectiveness (PUE) model, we may approximate these as being proportional to the node energy use, but on the Cray XC30 this information is not easily accessible to a non-privileged user, so we do not focus on it here[3]. Also, as we discuss later, the energy/runtime trade-offs that we see (even based on node power alone) are currently not significant enough to be economically viable. For these reasons, we postpone discussion of the additional system power draws to possible later studies.

The results so far have used the default -O2 optimisation

[3]Ref. [9] discusses this in more detail.

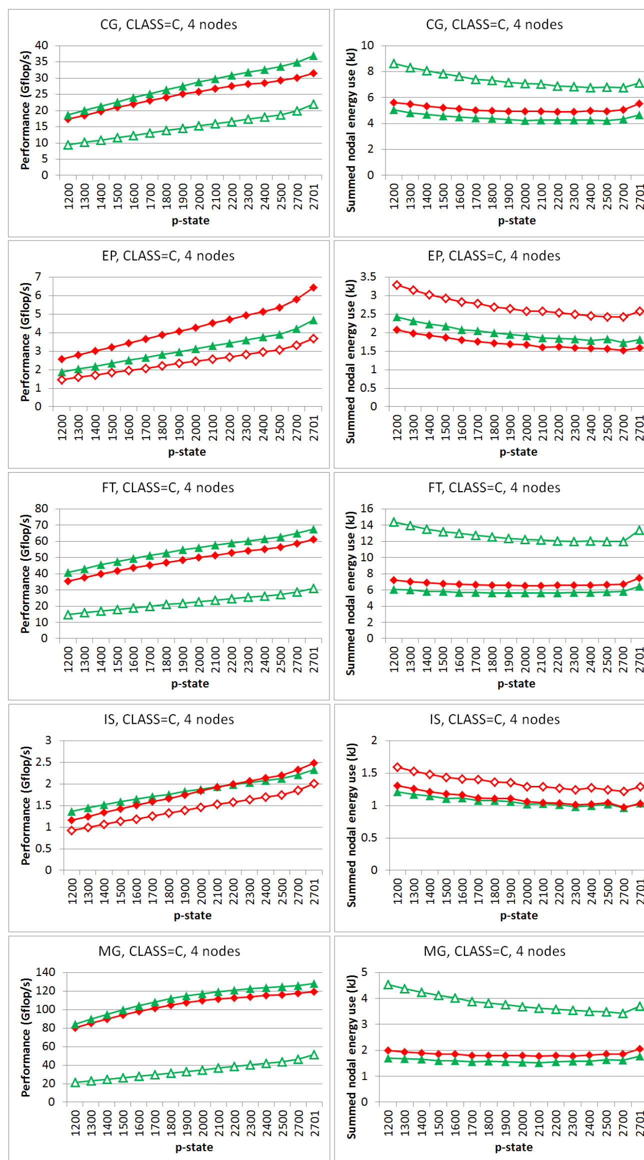

Figure 4. Performance of the NAS Parallel benchmarks on Marble nodes. Diamonds (red) and triangles (green) denote runs with and without hyperthreading, respectively. Solid symbols are compiled with -O2 optimisation; open symbols denote unoptimised -O0 compilation.

level for CCE. An interesting question is whether the runtime gains from optimisation justify potentially higher power consumption. The short answer is "yes", as is shown by the open symbols in Fig. 4 (which uses the hyperthreading setting that worked best with -O2). In all cases optimisation gives faster codes which overall consume less energy. The mean power consumption of the node is indeed higher with optimisation, but this is more than compensated by the decreased runtime.

Further, more thorough investigations of intermediate optimisation levels are clearly warranted. A reduced optimisation level could be used if there were a need to cap

the maximal power consumption (as opposed to the total energy consumption) of the system, but the same effect could be achieved using a lower p-state without recompiling the applications.

So far, we have studied energy consumption as we change system parameters, but not looked at the effects of using different algorithms. As an example of this, we can compare the pure MPI codes used so far with a hybrid MPI/OpenMP programming model. Using only a small number of nodes where MPI scaling should be good, we are unlikely to see much benefit on Marble nodes, but we shall see later that a hybrid programming model is very important on accelerated Graphite nodes.

The NPB suite does not include hybrid versions for most of the benchmarks, so we developed a hybrid MPI/OpenMP version of the MG benchmark. Time did not permit hybridisation of all the codes, and MG has features that are representative of a good range of HPC codes. Scoping features of the Cray Reveal tool helped in this port, with some code changes. No particular effort was made to optimise the OpenMP directives, but scaling appeared good for the threadcounts considered here.

For the hybrid investigation, we reduced the number of MPI ranks per node whilst increasing `OMP_NUM_THREADS` to keep the total threadcount per NUMA node constant (8 for each CPU on Marble nodes). Again, we executed the CLASS=C on four nodes, so the `aprun` options were: `-n64 -S8 -d1`; `-n32 -S4 -d2`; `-n16 -S2 -d4`; `-n8 -S1 -d8`. Options `-j1 -ss` were used throughout (the latter to ensure that cores on a CPU access only their local memory controller).

As expected on Marble nodes, threading did not significantly change runtime performance or energy consumption on four nodes, and the same pattern was seen of minimum energy consumption occuring at an intermediate p-state value.

B. Graphite results

We now turn our attention to energy measurements on accelerated nodes containing a CPU and associated GPU.

Using the pure MPI version of the NPB codes, we carried out a similar study of hyperthreading and optimisation using the CPUs of four Graphite nodes. Eight MPI ranks (16 with hyperthreading) were used per node. We obtained the same pattern of results seen for the Marble nodes. Once more, an intermediate p-state value led to the lowest energy consumption for computationally intensive CG, FT and MG benchmarks.

We also investigated the effect of OpenMP threading on the node for the MG benchmark, using up to 8 threads per MPI rank. As with Marble nodes, threading had little effect on any of the performance metrics, except when we reached 8 threads per rank, when the runtime performance was around 10% lower (and energy consumption consequently

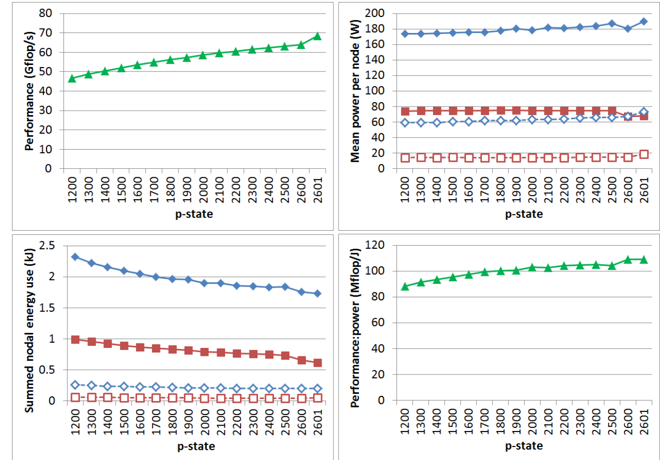

Figure 5. Performance of the MPI/OpenACC version of MG running on Graphite nodes. Open symbols show the idle power draw/energy use for the entire node (upper line) and the GPU (lower line).

higher). The best performance:power ratio obtained was approximately 100 Mflop/J, compared to around 120 Mflop/J for the Marble nodes, showing that the idle accelerator is not unduly penalising the energy efficiency of the system.

We now consider just using the GPU of the Graphite blades. Given the host-based execution model, at least one core of the CPU must be used to control the GPU, so by this we mean a code where all computational tasks have been ported to the GPU and no significant data transfers occur between CPU and GPU, beyond initialisation data (outside the timed region) and MPI send/receive buffers. We ported the MG benchmark to the GPU in this way using the directive-based OpenACC programming model and ran this on 4 Graphite nodes using one MPI rank per node (and without "G2G" MPI optimisation). Again, we did not investigate kernel tuning with the OpenACC.

The results are shown in Fig. 5. Once more, we see that lower p-states result in lessened performance. Profiling the OpenACC code with CrayPAT showed that reduced p-states affected not only the rate of (unpinned) CPU/GPU data transfers but also kernel performance. The difference to using the CPU is that the mean power per node is relatively constant, so the total energy used to complete the calculation just increases as we reduce the p-state, rather than showing a minimum at some intermediate CPU clockspeed.

The best performance from the GPU was just under 70 Gflop/s, compared to just over 50 Gflop/s for the full CPU.

The OpenACC code is faster, but (as we expect) not significantly. To make best use of accelerated nodes, we would really like to use both CPU and GPU for the computation.

If an application is sufficiently task-based, we can efficiently make full use of an accelerated node by overlapping the computation of independent tasks on the CPU and

the GPU. An example of this approach is the GROMACS code. This approach is, however, difficult to implement in many applications. Alternatively, the application can be re-engineered so that within the executable binary, some MPI ranks will execute on the GPU and some on the CPU. Again, this requires a lot of work in the application to allow the code to compile in this form.

A third approach is to compile the application in two ways, one targeting the CPU and one the GPU, and then run them as a single MPI job on the Cray XC30 using the extended MPMD mode in Fig. 1. The complication is that we would like the MPI ranks to be load balanced, but (for simplicity) they will probably each calculate equally-sized portions of the global problem and a GPU is much faster than a single CPU core. We should therefore make sure that the CPU version of the code has OpenMP threading in addition to the MPI parallelism.

We did this using the versions of MG developed for this paper. We use one OpenACC rank per node, and reserve one CPU core per node for this. We then choose a number of CPU-only MPI ranks between 1 and 7 per node and compile both versions of the code to consequently use between 8 and 32 ranks in total. We then execute both versions together on four nodes with increasing OpenMP threadcounts per CPU-specific rank, such that the total number of threads per node does not exceed 7.

For each number of ranks we select the value of OMP_NUM_THREADS that gives best overall performance. For 32 ranks, it must be 1; for 16 ranks, it is 2; and for 8 ranks it is 6 (rather than 7). The results for these are compared with the pure CPU and pure GPU results in Fig. 6. Looking at the Performance plot, we see that the best hybrid CPU/GPU version of the code delivers nearly 90 Gflop/s compared to 70 Gflop/s for just the GPU or 50 Gflop/s for just the CPU. This is obtained using the highest value of OMP_NUM_THREADS, which is expected to be the most balanced configuration. Increasing the number of MPI ranks rapidly degrades performance. The fastest hybrid CPU/GPU combination is also (within small fluctuations) the most efficient in terms of energy use for the full range of p-states (matching the pure GPU at high values and the pure CPU at lower values). Expressing these results in terms of a performance:power ratio, the best hybrid CPU/GPU combination is a clear winner, giving around 110 Mflop/J. As before, this is obtained at an intermediate value of the p-state.

IV. Application performance monitoring

In the previous Section, we considered a set of simple benchmarks. For more complicated applications, we would like to monitor the power usage as the application runs and, crucially, present this data as part of an integrated view of the application performance. Collection of the power counter information should therefore be integrated into a

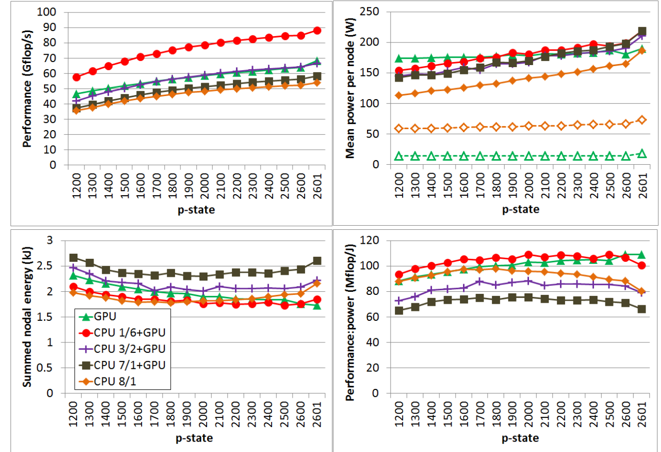

Figure 6. Hybrid CPU/GPU performance of the MG benchmark. "CPU N/d" denotes N ranks of the MPI/OpenMP application per node, each with d threads. "GPU" denotes one rank of the MPI/OpenACC application per node. Open symbols denote the idle power draw of the entire node (upper line) and of the GPU (lower line).

wider performance measurement suite. For the Cray XC30 platforms, this has been done in the Cray Performance Analysis Toolkit (CrayPAT) [10]. We describe a separate implementation using the event-based tracing module of the monitoring system Score-P [4], with the detailed trace files then analysed with the performance visualiser Vampir [5].

In contrast to job energy monitoring, detailed monitoring of energy and power during performance analysis of applications is more challenging. This new information must be correlated with the fine-grained events of the application, whilst remaining mindful of the accuracy and intrusion of the monitoring process. In addition, missing a sample of the measured power and energy counters can result in misleading results and conclusions. Derived metrics (such as the average power of the last interval calculated from energy measurements) must thus be treated with care.

In this Section, we present the infrastructure needed to read the power and energy counters in the application monitoring system Score-P for Cray XC30 platforms, and the challenges for a detailed performance analysis. In addition, we visualised the monitoring data of two benchmarks (a synthetic load-idle pattern benchmark, and the HPL CUDA code) and the real-life Gromacs application [16] (one of the exascale applications defined within the CRESTA project) with the performance visualiser Vampir and analysed the characteristics and behaviour of the energy and power metrics.

A. *Cray XC30 energy and power monitoring with Score-P*

The application measurement system Score-P has been able to record external generic and user-defined hierarchical performance counters since version 1.2. This is done with a flexible "metric plugins" interface to address the complexity

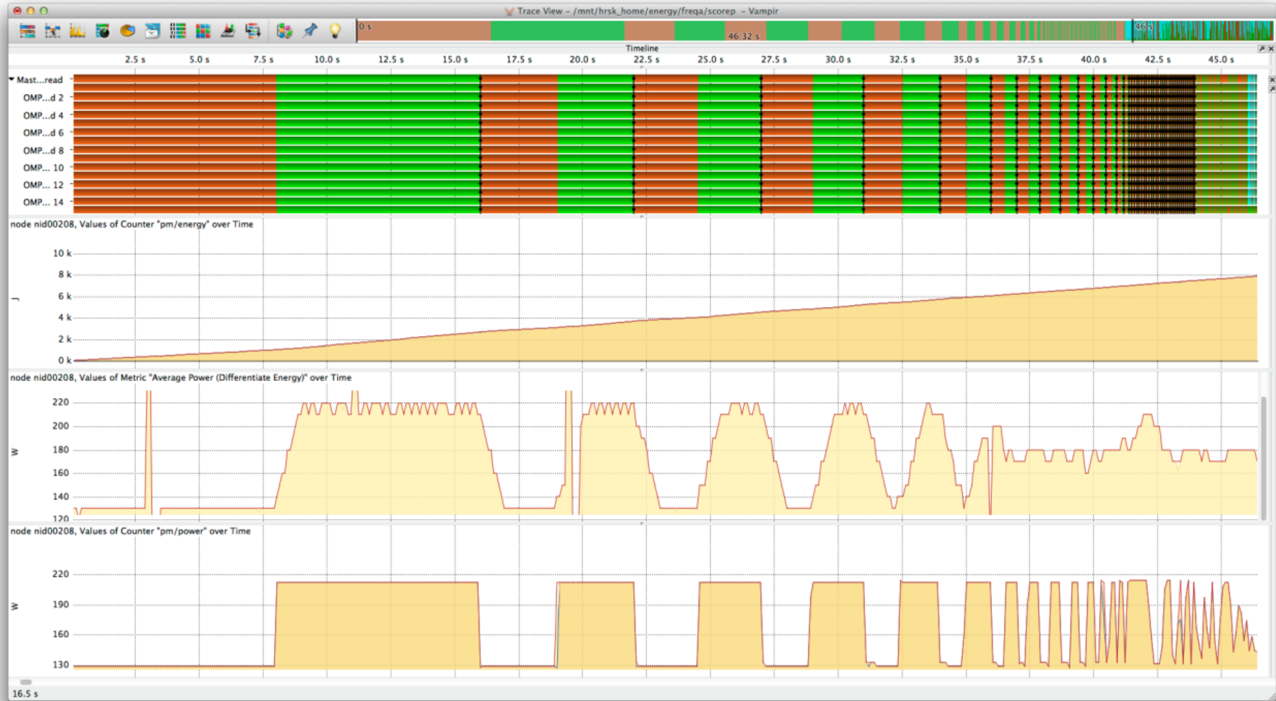

Figure 7. Load-idle benchmark with colour-coded visualisation of the load-idle regions (topmost timeline, load-idle regions are coloured in green respectively in brown) and corresponding energy (second timeline), average power derived from energy (third timeline), and instantaneous power information (lowest timeline) with Vampir for an interval of 46.3s.

of machine architectures both today and in the future. The metric plugin interface provides an easy way to extend the core functionality of Score-P to record additional counters, which can be defined in external libraries and loaded at application runtime by the measurement system. We built a Score-P metric plugin to monitor the application external energy and power information on Cray platforms during the application measurement with the following properties on synchronicity and scope.

As described above, the Cray XC30 energy and power counters are updated with a frequency of 10Hz. The scope of these metrics is per node and therefore the Score-P measurement system instructs and steers the first process of each node to start the metric plugin.

Energy and power information are generally independent of the occurrence of events during the program monitoring. To reduce the overhead during a specific event we decided to collect all energy and power values asynchronously. We start an additional thread that polls the files and checks if updates are available, collects the new values, and links it with an actual timestamp. At the end of the monitoring the vector of timestamp-value pairs is written to disk together with the application monitoring information. The frequency of the update check can be set by the developer of the metric plugin and should be chosen under the constraints of accuracy and overhead. For a 10Hz counter we suggest an update check frequency higher than the original frequency of the counter, e.g. 100Hz, to ensure that no update of the energy and power information will be missed. The age of the sample therefore depends on the latency of the internal measurement infrastructure and the update check frequency.

B. *Performance and metric visualisation with Vampir*

The energy and power metrics are timestamp-value pairs and, by their nature, only reflect a specific point in time and cannot be easily correlated to the events of the processes and threads. In addition, these metrics are valid within a scope with a set of processes and threads. Therefore, the aggregation of energy and power metrics within the function statistic of each process/thread (which is usually done by profiling approaches) requires knowledge of correlation factors.

In the performance visualiser Vampir [5], we decouple power and energy metrics from the event stream and display the behaviour over time in special timelines. For instance, the "counter timeline" can be used to visualise the behaviour of one metric for a specific node over time, or the "performance radar" can be used to visualise the behaviour of one metric for a set of nodes over time. With these combinations of timelines (such as the "master timeline"), which visualise the event stream over time, visual correlation of the different

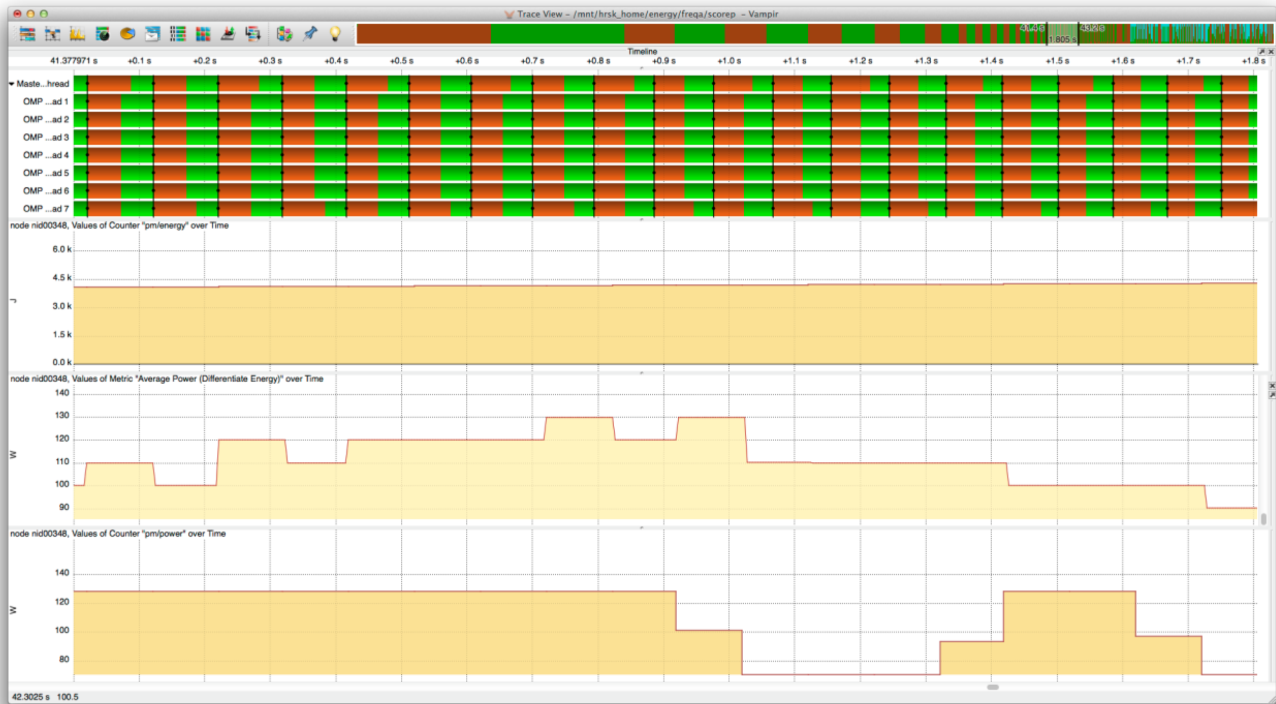

Figure 8. Length of load-idle pattern of 50ms. The instantaneous power (lowest timeline) is not able to reflect this alternating pattern. Even the average power derived from the energy (third timeline) is not able to reflect this alternating pattern.

information is possible.

1) Synthetic load idle pattern: To test this framework, we developed a synthetic load-idle benchmark that repetitively executes regions with high and low power requirement. The transitions between the different phases are synchronized across processes and threads. Thus, the resulting power consumption over time can be considered to be a square wave, if side effects are not taken into account. However, we reduce the wave period while executing the benchmark to check whether the energy and power counter can capture and reflect this alternating switch of extreme states. Fig. 7 shows the colour-coded visualisation of this benchmark. In the upper part the load-idle regions are coloured in green and respectively in brown. In the lower part the behaviour of the three metrics (node energy, average node power [derived from the node energy using a backward difference operator] and the native instantaneous node power) over time for the first 45s is displayed. In the beginning of the execution, there is an obvious correlation between the load-idle patterns and the node energy and power metrics. At the end of the benchmark, when the duration of each state of the load-idle pattern is 50ms, the energy and power metrics (especially the instantaneous power) can no longer resolve the state changes. These aliasing effects can be seen in Fig. 8. We therefore can only provide meaningful results if the duration of a state (or a set of states) is longer than the time between two energy and power samples, ensuring that there will be at least one sample that can be monitored and correlated with this region.

2) HPL CUDA: To demonstrate how information about accelerators is collected and displayed, we executed the HPL CUDA benchmark [17] on a Cray XC30, using MPI processes, OpenMP threads and CUDA kernels. In addition to the traditional application behaviour monitoring we recorded the energy and power counter for the nodes and the installed graphic cards. Fig. 9 shows the colour-coded visualisation with Vampir. The topmost timeline shows the behaviour of the processes, threads, and CUDA streams over time for an interval of 2 seconds. The second timeline displays the instantaneous node power over time. The third timeline displays the instantaneous graphic card power and the lowest timeline displays the node “exclusive” power (i.e. without the graphic card) derived from the energy for the first of the four nodes (nid00348).

It is clear from Vampir that the node power usage is driven by the execution of the CUDA kernels (blue boxes) for this application. We note also that the instantaneous power values are hard to interpret and can lead to misleading decisions. This can be seen in in the third and fifth timelines in Fig. 10, which display the instantaneous node and accelerator power

Figure 9. Colour-coded visualisation of HPL CUDA with Vampir. The topmost timeline shows the behaviour of the processes, threads, and CUDA streams over time for an interval of 2s. The second timeline displays the instantaneous node power over time. The third timeline displays the instantaneous graphic card power and the lowest timeline displays the board exclusive power without the graphic card derived from the energy for the first node nid00348 out of the four nodes.

draws (respectively). Since we have seen that the accelerator power has a strong impact on the node power, it is interesting to note that the decrease of the node power is two samples earlier than for the accelerator power. The rates of change of the node and accelerator power draws are displayed in the fourth and the lowest timeline. The main conclusion is that it is hard to trust only in the instantaneous values, and further work is needed to understand why this occurs. It is better to compare it with an average power for the last interval derived by a difference operator on the energy metric, visualised within the second timeline of Fig. 10. Curiously, the average node power curve (second timeline) appears smoother than the curve for the instantaneous node power (third timeline) and the application event timeline (topmost timeline). This effect is often due to digital filtering, although this is not carried out for these metrics. We are investigating this effect further.

3) Gromacs: Finally, we monitored a tri-hybrid MPI/OpenMP/CUDA version of Gromacs 4.6.5 running on a Cray XC30 with four nodes, with each node hosting one MPI process with six OpenMP CPU threads and two GPU CUDA streams. Fig. 11 shows the colour-coded performance visualisation with Vampir of 4000 iterations with corresponding timelines for the energy, instantaneous board power, average board power derived from the board energy, instantaneous accelerator power, and average accelerator power derived from the accelerator energy. The colour-coded visualisation of the board energy (second timeline) gives us a first rough impression of the dynamic load balancing behaviour of Gromacs. The second node consumes the most energy (8507J) for this application run of 49.393s, followed by the third (8252J), first (8031J) and fourth (7859J) nodes. This can be also observed within the various power timelines. In addition, we can identify two intervals within this run (from 7 to 10 seconds and from 17 to 20 seconds), when the accelerators have increased power consumption. We are investigating the cause of this. We also encounter an artefact where the average accelerator power derived from the energy is zero on the first and fourth nodes (around 28.5 seconds into the run). This failure to update the counters was due to a bug in the system software that has since been fixed.

Fig. 12 shows a zoomed-in, colour-coded visualisation of 80 iterations over an interval of 0.993s. The timelines show: events on all four nodes (white background); the events of the first (nid00348, blue background) and second (nid00349, green background) nodes; and instantaneous power information for the four boards and accelerators. Statistics for the

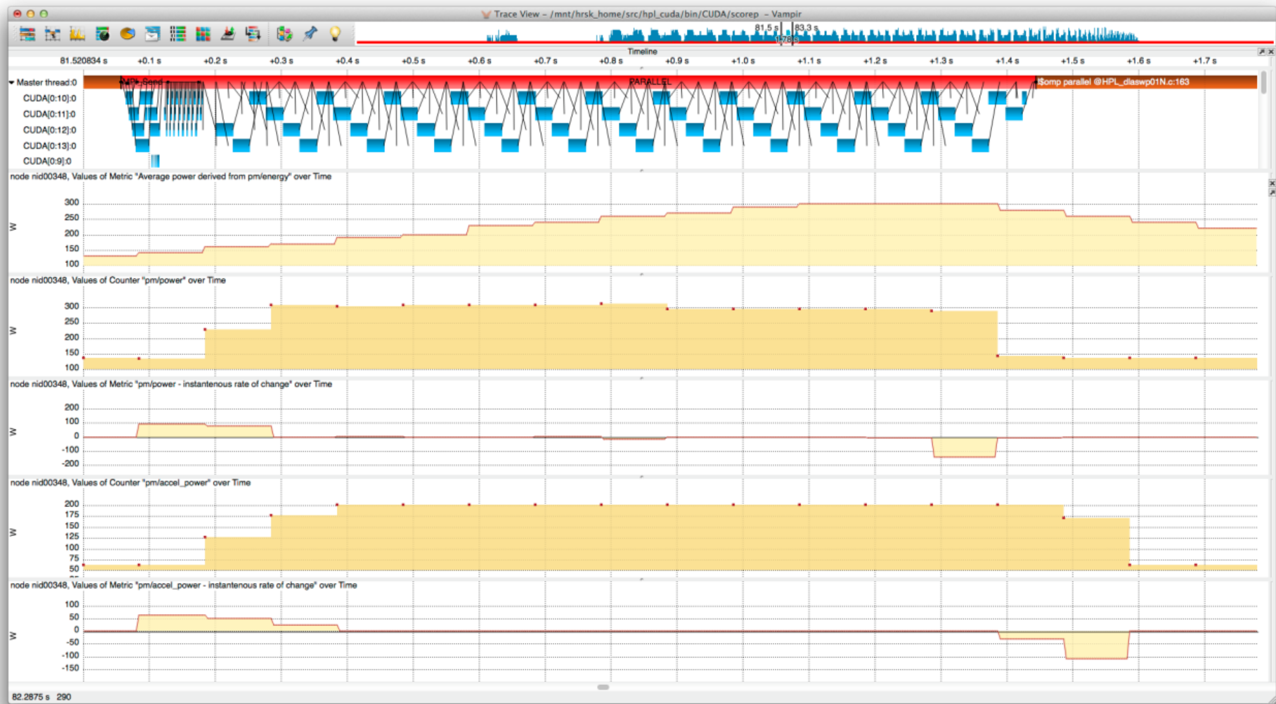

Figure 10. Colour-coded performance visualisation with Vampir of the first node of the HPL CUDA application for an interval of 1.8s with according timelines for the events (topmost), average node power derived from the node energy by a backwards difference operator (second timeline), instantaneous node power(third timeline), rate of change of the node power (fourth timeline), instantaneous accelerator power (fifth timeline) and rate of change of the accelerator power (lowest timeline).

exclusive time on the right part of the figure. It is interesting to see that the second node (nid00349) has a higher power consumption than the first node (nid00348) mainly caused by the accelerator power consumption peaking at 99W on the second node, compared to 87W for the first node. The dynamic load-balancing of the CUDA kernels (dark blue boxes in the timelines and statistic displays) places a higher computational load on the second node (0.59s) than on the first node (0.35s). However, since the CUDA kernels of Gromacs are highly optimized and have only a very short runtime, the frequency of the energy and power counter is too low to properly investigate the energy and power consumption of individual kernels. Instead, we suggest investigating the energy and the average power derived by a difference operator on the energy information for a set of kernels, as in the 80 iterations in Fig. 12.

V. Conclusions

With energy and power efficiency increasingly important in HPC, developers require a fine-grained view of energy consumption in their applications. This information will allow algorithmic and runtime decisions to be made in the software to tune performance against a wider range of energy-based metrics, rather than just runtime.

Understanding how energy is used when executing real-life applications can then also feed back into design of processors and other HPC architecture components in a cyclic process of co-design. This should allow the large further energy savings required to move towards the predicted exascale era of supercomputing.

In this paper, we have shown how application energy and power consumption of the node-exclusive (rather than shared blade- and cabinet-level) components can be measured by users on Cray XC systems. We have concentrated on two methods for achieving this. The first is relatively coarse-grained, either on a per-application level (from the jobscript) or via the user inserting API calls into the application code. While this is both sufficient and tractable for the exploratory work using simple, single-purpose benchmarks presented in this paper, it is often difficult to extract such exemplars from real applications. A whole-application tracing framework removes the need for this, and we described the framework for doing this and its implementation in the widely-used Score-P and Vampir packages.

As examples of the tuning decisions that a user can make based on this information, we investigated some aspects of running the NAS Parallel Benchmark codes on two versions of the Cray XC30 architectures: one CPU-based and one

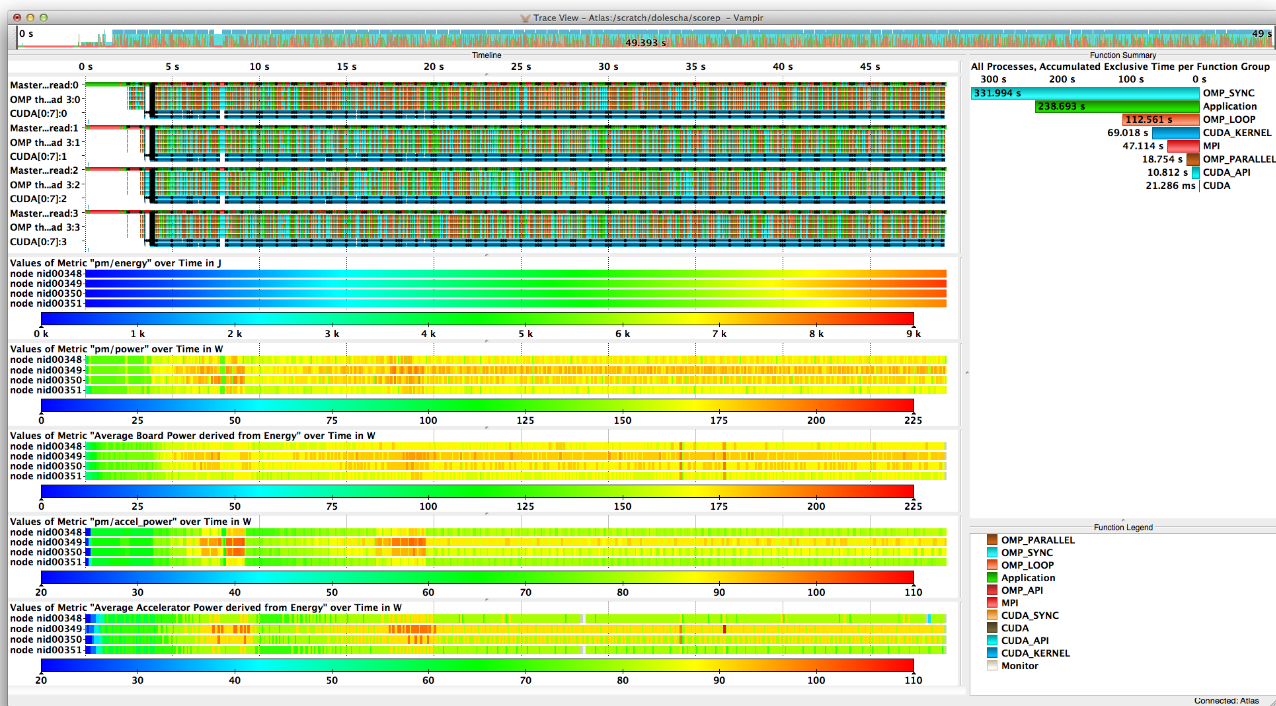

Figure 11. Colour-coded visualisation of 4000 iterations of a hybrid version of Gromacs running on four nodes (with each node hosting one MPI process with six CPU threads and two GPU CUDA streams running on the accelerator) for an interval of 49.393s with according timelines for the events on all four nodes (topmost) and corresponding energy (second timeline), instantaneous power (third timeline), average board power derived from energy (fourth timeline), instantaneous accelerator power (fifth timeline), average accelerator power derived from accelerator power (lowest timeline) for the four nodes, and according statistics for the exclusive time on the right part of the figure.

also using GPU accelerators. In particular, we studies how both runtime and energy consumption of an application is changed as the CPU clockspeed is varied via the p-state. For both node architectures, if the application is run on the CPU, we found that the overall nodal energy consumption could be reduced by running at a clockspeed intermediate between the highest (default) and lowest settings. This demonstrates that modern HPC hardware has already moved beyond the previously true "fastest is most energy efficient" position.

We also investigated how introducing OpenMP threading in the MG benchmark changed energy consumption; for the small testcases we studied here it neither saved nor cost significant amounts of energy. In contrast to the CPU results, when running on GPUs using an OpenACC port of the MG benchmark, reducing the CPU p-state uniformly increased both the application runtime and energy consumption.

We also demonstrated that the best performance for the MG benchmark (both in terms of runtime and energy consumption) came from using both CPU and GPU in the parallel calculation, and we demonstrated how to do this using separate MPI/OpenMP and MPI/OpenACC binaries. OpenMP threading is very important on the CPU; without this the poor load balancing leads to reduced runtime performance and energy efficiency.

We have used the power-instrumented versions of Score-P and Vampir to trace and display the execution of a range of applications, from synthetic benchmarks to the highly-optimised, GPU-accelerated version of the Gromacs Molecular Dynamics package. The ability to correlated the instantaneous power draw of the nodes' components with other measures of application progression is very important. In particular, we showed how the Vampir trace can be used to diagnose the asymmetric power consumption of the nodes, showing this is due to the code's automatic scheduling of CUDA kernels.

A. A trade-off model for runtime and energy consumption

We have seen that varying the p-state of the CPU can reduce the overall energy consumption of an application, at the cost of it taking longer to run. If energy consumption is the primary concern of the user or system provider, this is a good thing. The downside, however, is that fewer jobs can be run during the lifetime of the system and the depreciation of the capital cost per job is therefore higher.

We can develop a (very) simple model of this trade-off[4]. If

[4]We thank Mark Bull for first drawing our attention to this issue.

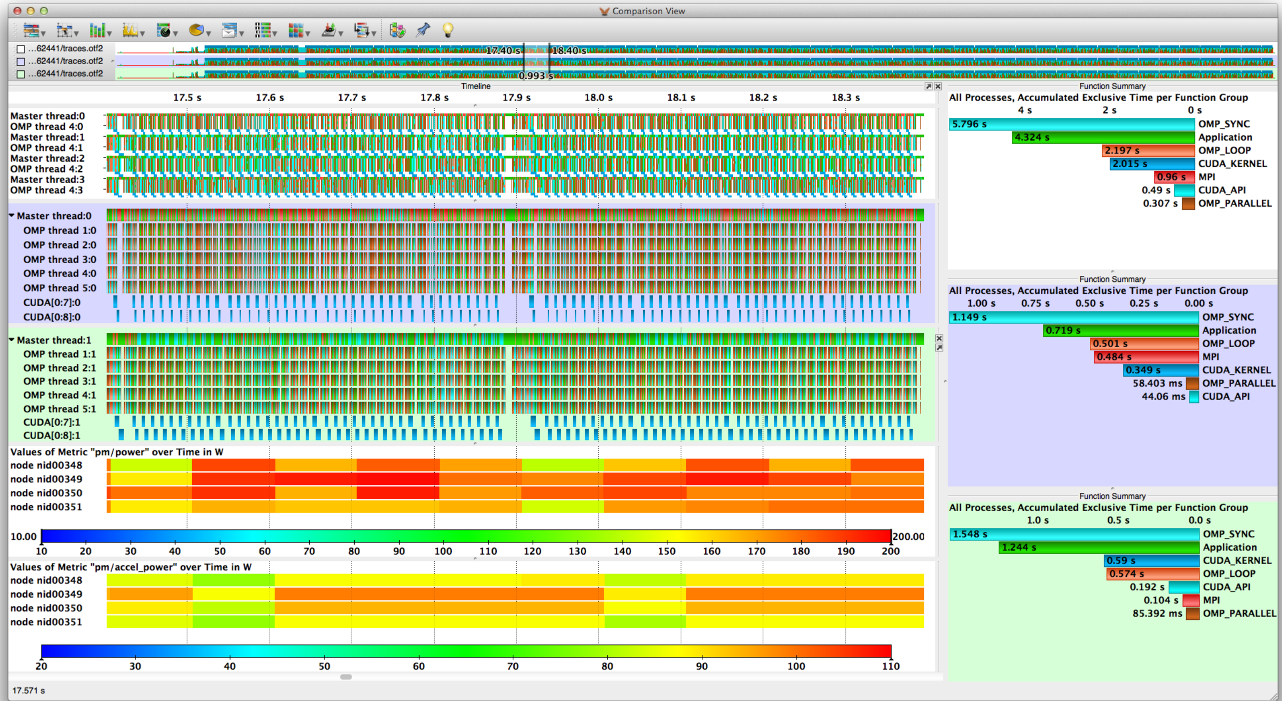

Figure 12. Colour-coded visualisation of 80 iterations of a hybrid version of Gromacs running on four nodes (with each node hosting one MPI process with six CPU threads and two GPU CUDA streams running on the accelerator) for an interval of 0.993s with according timelines for the events on all four nodes (white background), the events of the first node nid00348 (blue background), the events of the second node nid00349 (green background), instantaneous power information for the four boards and accelerators. Statistics for the exclusive time on the right part of the figure.

the initial cost of the system is S and it will run for T years, the capital cost depreciates at rate of S/T per year. If each year we run N (assumed similar) jobs, with an electricity cost of C, then the overall financial cost per job is:

$$K_0 = \frac{S}{NT} + \frac{C}{N} \, . \tag{1}$$

If we run the jobs in a more energy-efficient, throttled manner, the runtime changes by a factor $R > 1$, whilst the energy consumed consequently changes by a factor $E \leq 1$. We will therefore run R fewer jobs per year and the cost becomes:

$$K_1 = \frac{SR}{NT} + \frac{CE}{N} \, . \tag{2}$$

Overall, this change is financially viable if $K_1 < K_0$. If running costs are dominant, this favours reducing E as much as possible, regardless of the runtime penalty. At the other extreme, if the capital depreciation is the main concern, we should keep R at its minimum value of 1. Between these limits, the payback time T_{pb} measures how long we would need to run a system with throttled applications before $K_1 \leq K_0$:

$$T_{pb} = \frac{R-1}{1-E} \times \frac{S}{C} \, . \tag{3}$$

As a very rough figure, a typical contemporary HPC system has a ratio of annual electricity bill to initial system cost around $C/S = 5\%$.

In the case of MG benchmark running on 4 nodes, if we compare the performance at the top p-state to that which gives the best energy efficiency, we find that a 5% increase in runtime ($R = 1.05$) is balanced by a 15% decrease in energy use ($E = 0.85$). Given this, the payback time T_{pb} is 6.7 years, far longer than the three to five year lifetime of most HPC systems. This estimate was for the p-state giving the lowest energy consumption. Even if we look across the entire range of p-states for all benchmarks, there is only one instance where the payback time is less than five years. So, even though downclocking the CPUs reduces node energy use, we are not yet in an era where this is a cost-effective way to run an HPC system.

At present, most HPC systems implement user accounting based purely on runtime. There is, however, increasing interest in "billing by the Joule"; charging for the energy used in executing an application. The analysis above suggests that a pure energy expenditure metric could be overly simplistic and not reflect the capital depreciation costs in running a system. A more in-depth analysis, covering all HPC system and HPC data centre components, is clearly needed. The model

above does not include the energy consumption of shared architectural infrastructure (e.g. network- and cabinet-level), which needs to be divided somehow between the executing jobs. If we assume that the fractional change in nodal energy consumption is reflected in the overall energy expenditure of the system (i.e. a linear PUE model to include network and cabinet infrastructure), the above conclusions still hold. If, however, the additional components contribute a more constant overhead (as seemed to be seen in Ref. [9]), the model would need to be refined.

Individual, non-privileged users cannot, however, access information about the shared components, so this work goes beyond the scope of this paper.

Even based on the above analysis, the conclusions would already change if energy prices were to rise by, say, 30%, reducing T_{pb}. Alternatively, if the decision were made to procure a long-lifetime system, it would also make reducing performance more attractive from the overall cost perspective. These benefits would, at least currently, probably be outweighed by efficiency gains in new systems available midway through this long-lifetime procurement.

In conclusion, then, we appear to have reached a point where not only is energy consumption important in HPC, but also where it is measureable and can be visibly influenced by choices made by application developers. An exascale supercomputer is unlikely to be built from current hardware, but we can now (from a measurement and visualisation perspective) begin a meaningful co-design process for energy-efficient exascale supercomputers and applications.

ACKNOWLEDGMENT

We thank S. Martin (Cray Inc.), M. Weiland (EPCC) and D. Khabi (HLRS) for valuable discussions and assistance. This work has been supported in part by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

REFERENCES

[1] "EU energy, transport and GHG emissions trends to 2050 reference scenario 2013," (Accessed 26.Feb.14).

[2] "The Green 500 list." [Online]. Available: http://www.green500.org

[3] P. Beckman *et al.*, "A decadal DOE plan for providing exascale applications and technologies for DOE mission needs," (Accessed 26.Feb.14). [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/mar10/Awhite.pdf

[4] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, 2012, pp. 79–91.

[5] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool Set," in *Tools for High Performance Computing*, 2008, pp. 139–155.

[6] A. Hart, M. Wieland, D. Khabi, and J. Doleschal, "Power measurement across algorithms," Mar. 2014, deliverable D2.6.3, EU CRESTA project (in review).

[7] T. Ilsche and J. Doleschal, "Application energy monitoring on hpc systems - measurement and models," in *Adept Workshop on Efficient Modelling of Parallel Computer Systems*, Vienna, Austria, Jan. 2014. [Online]. Available: http://www.adept-project.eu/images/slides/HiPEAC2014_ThomasIlsche1.pdf

[8] S. Martin and M. Kappel, "Cray XC30 power monitoring and management," in *Proc. Cray User Group (CUG) conference*, Lugano, Switzerland, May 2014.

[9] G. Fourestey, B. Cumming, L. Gilly, and T. Schulthess, "First experiences with validating and using the Cray power management database tool," in *Proc. Cray User Group (CUG) conference*, Lugano, Switzerland, May 2014.

[10] H. Poxon, "New functionality in the Cray Performance Analysis and Porting Tools," in *Proc. Cray User Group (CUG) conference*, Lugano, Switzerland, May 2014.

[11] D. Brodowski and N. Golde, "Linux CPUFreq Governors," (Accessed 24.Feb.14). [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[12] A. Barry, "Resource utilization reporting," in *Proc. Cray User Group (CUG) conference*, Napa CA., U.S.A., May 2013. [Online]. Available: https://cug.org/proceedings/cug2013_proceedings/includes/files/pap103.pdf

[13] H. Richardson, "`pm_lib`: A power monitoring library for Cray XC30 systems," Jan. 2014, EU CRESTA project.

[14] "The NAS Parallel Benchmarks," (Accessed 24.Feb.14). [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

[15] "The Top 500 list." [Online]. Available: http://www.top500.org

[16] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[17] "The CUDA HPL package," (Accessed 28.Feb.14). [Online]. Available: https://github.com/avidday/hpl-cuda