D2.6.3 – Power measurement across algorithms

WP2: Underpinning and cross-cutting technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M30
Submission date:	28/02/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organization	CRAY UK
Version:	1.0
Status	Final
Author(s):	Alistair Hart (CRAY UK), Michele Wieland (UEDIN), Dmitry Khabi (HLRS), Jens Doleschal (TUD)
Reviewer(s)	Sebastian Schmieschek (UCL), Berk Hess (KTH)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.17	25/2/14	First version of the deliverable	Alistair Hart (CRAY UK), Michele Wieland (UEDIN), Dmitry Khabi (HLRS), Jens Doleschal (TUD)
0.24	27/2/14	Second version of the deliverable	Alistair Hart (CRAY UK), Michele Wieland (UEDIN), Dmitry Khabi (HLRS), Jens Doleschal (TUD)
0.96	4/3/14	Released to reviewers	Alistair Hart (CRAY UK), Michele Wieland (UEDIN), Dmitry Khabi (HLRS), Jens Doleschal (TUD)
1.00	24/3/14	Final version	Alistair Hart (CRAY UK), Michele Wieland (UEDIN), Dmitry Khabi (HLRS), Jens Doleschal (TUD)

Table of Contents

Index of Figures

Index of Tables

1 Executive Summary

This deliverable seeks to summarise the need for power and energy measurements on a variety of contemporary architectures. These architectures provide guidance to future exascale system design choices. We demonstrate that even on these architectures, there is now a clear tradeoff between runtime performance and energy or power consumption for different applications and algorithms. This shows that the time is now right for application developers to begin to consider these metrics when selecting algorithms for computation, even if the simple model that we introduce shows that the current savings are not yet significant enough to warrant different charging models for HPC systems.

Whilst overall energy and power consumption are appropriate measurements for single-purpose, standalone benchmark codes, real application developers require energy and power information to be integrated with other application-tracing performance data.

In Section 3, we describe how the Score-P and Vampir performance measurement and presentation tools have been adapted to provide a clear, easy-to-use view of evolving power and energy consumption during application execution.

In Section 4, we focus on two examples of current HPC architectures, the Cray XC30 with nodes containing either pure CPU or CPU plus GPU accelerator. We consider a number of different algorithm and runtime configuration choices and show how energy and power consumption are affected. We also dynamically varied the processor clockspeed. We present a number of different metrics by which algorithm choice can be tuned, relevant to different HPC scenarios.

In Section 5, we consider in more detail the power consumption of a multi core processor. Using the eight-core Sandy bridge processor we measured its power consumption when the algorithmic kernel operations ran on up to eight cores with all possible frequencies and with data in different hierarchy levels of memory. We introduce a new approximation formula for the power consumption, which is very suitable for the kernel operations considered. The results show that the question of energy consumption of both the algorithms and hardware is not trivial. We conclude that a more flexible strategy for choosing the optimal frequency and the number of active cores could lead to more efficient energy consumption of supercomputers.

In Section 6, we test the performance and energy usage of the NAS Parallel Benchmarks on low-power technologies that have their origins in the mobile processing world. Using two different quad-core ARM CPUs, one optimised for performance and the other for energy efficiency, we investigate the trade-off between performance (i.e. wall time to solution) and total energy consumed. Although the best absolute performance is always delivered by the more powerful A15 CPU, for certain workloads (and if pure performance is not a constraining factor) the low-power A7 CPU provides a real alternative in terms of overall energy efficiency. Although mobile processors are not a realistic resource for production HPC workloads, the results obtained on the ARM CPUs confirm that it is indeed possible to trade performance for energy efficiency (and vice versa).

We conclude that current CPUs (whether designed for HPC or low-power environments) are still far from satisfying the stringent performance and power budget requirements for an exascale architecture. Nonetheless, we do see that changes in algorithm and runtime configuration do have a marked effect on application energy efficiency of contemporary processors. We therefore feel that the field is now at a point where a meaningful co-design process can begin for the energy-efficient hardware and applications that are needed for an exascale era. For this to work, application developers need detailed feedback on how their changes affect power consumption, and the tools described in this deliverable are a vital step in this direction. Finally, a low-level understanding of power consumption (as we have begun here) will be needed

for systemware to provide energy-efficiency supporting interfaces between the hardware and the applications.

2 Introduction

Energy consumption is one the most important constraints on the design and construction of an exascale supercomputer. Indeed, if one were to attempt to build an exascale system using the components used in current petascale supercomputers, the final design would also need to include one or more neighbouring power stations, as the machine would draw at least 100MW of power.

The United States Department of Energy has instead set a goal of 20MW [1], requiring that a thousand-fold increase in computational speed over a petaflop supercomputer be achieved using only three times the power.

This is not just an engineering challenge. Exascale applications will need to be three-hundred times more energy efficient than their petascale ancestors. Whilst more efficient hardware will assist in this goal, the hardware advances will only benefit an application if it is sensitively attuned to the underlying architecture. Shared instruction streams will require applications that almost exclusively use long vector operations. Moving data will take more energy than floating point operations, so applications will need to be sensitive to data locality and to exploit increasingly-deep cache structures.

Even if current applications with exascale ambitions (such as the six CRESTA codesign codes) survive to execute in that era, it is clear that there will need to be big algorithmic changes within the codes to exploit the extreme architectures. Without access to the technology of tomorrow, of course, it is difficult to predict exactly which algorithms will be most successful. Even when those algorithms with the greatest potential are identified, the absolute best will depend on the circumstances: the problem considered, the parallel decomposition, the details of the hardware, etc. An autotuning approach, such as that developed in the CRESTA project [2], will almost certainly be needed with the flexibility to allow tuning for a range of metrics (e.g. runtime performance or energy efficiency or some derived metric based on both).

While measuring power and energy consumption will be essential in the exascale era, there is increasing current interest in the topic. Wholesale electricity prices have recently risen sharply in many regions of the world, including in the European states represented in the CRESTA and other EU Exascale projects [3]. Environmental 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 interest in the "Green 500" list of the most efficient HPC systems since its introduction in 2007 [4].

It is important that we understand here what we mean by energy or power efficiency. In contrast to the performance-based "Top 500" list [5], which ranks systems on pure performance, the "Green 500" list uses a performance:power ratio. In areas with restricted power generation capacity, there is an additional interest in power-aware supercomputing, such that the performance of the system can be throttled back to reduce the immediate power consumption load on the electricity distribution grid to compensate for increased demand from other sectors. Finally, in mobile and embedded contexts, minimising energy consumption is important, e.g. to prolong battery life or to allow standalone use with wind or solar microgeneration systems. It is clear, then, that there are a number of different metrics that apply in different situations.

Nonetheless, both for exascale planning and for current computational systems, there are good reasons to study energy and power consumption of HPC applications. Such a study is also very timely from an architectural standpoint. In previous supercomputer generations, it was difficult to measure power consumption. The hardware to make the measurements was rarely included or, when available, tended to give a coarse-grained spatial and temporal view. Measurements were perhaps only available at the level of an entire rack or cabinet, and the sampling frequency was too low to be of use to application developers wishing to tune algorithms. In addition, the data was often not easily available to nonprivileged users, requiring administrator access to a system management workstation or similar external controller.

With many current HPC architectures, this has now changed. The new Cray Cascade-class XC30 supercomputers, for instance, now provide user-visible power and energy counters for each node that update ten times a second.

In addition, there is a general perception that tuning for energy efficiency is the same as tuning for performance. If hardware draws a constant amount of power regardless of computational load (as was broadly the case in the past), one minimises the energy consumption by minimising the time for which the application runs. Modern hardware, particularly CPUs, have a complicated hierarchy of performance states that draw different amounts of power, with the hardware dynamically switching between these in response to computational load. The hardware no longer draws a constant amount of power and, as we show in this deliverable, it is no longer always the case that the most energy-efficient way to carry out a calculation is with the smallest runtime.

Given these changes, it is now a good time to consider how algorithm choices affect power and energy consumption. This information can then be utilised by application developers and users to tune their applications (using one of the various metrics we outlined above and will present in more detail later).

At present, most HPC systems that implement user accounting do so purely on the basis of runtime. In such circumstances, there is no direct monetary incentive to the user to sacrifice computational performance to reduce the energy footprint. Indeed, as we discuss later, users and system administrators need to be careful that such a sacrifice really is actually beneficial over the lifetime of the system.

Looking forward, it is clear from the measurements in this deliverable that the time is now appropriate to look at more energy-based metrics for system accounting.

Looking further forward still, it is also clear that developers of applications with exascale ambitions now have the tools to begin to consider energy and power consumption when choosing and implementing algorithms. With a great deal of architectural innovation expected as we strive for exascale supercomputers, the choice of which algorithm is "best" will almost certainly also evolve. The important thing, however, is that the information is now available to begin this crucial step in rendering applications exascale-ready.

2.1 Purpose and overview of this deliverable

In this deliverable, we consider four inter-related themes. We present some results of measuring the power consumption on three different contemporary computing architecture classes. The first is an integrated supercomputer, using two examples from the Cray XC30 product line (one pure CPU, one with GPU accelerators), where we investigate how different algorithms, programming models and runtime configurations affect both energy consumption and application performance.

We then consider a more generic cluster architecture, which offers even greater visibility of the energy use with finer temporal resolution of measurements, which allows us to focus on measuring the energy cost of low-level operations, and developing models that could be used in systemware (runtime or OS software, or in lower-level hardware controllers) in a future energy-conserving HPC architecture.

Finally, we look at an architecture based on a low-power CPU design commonly used in mobile phones. The motivation for looking at such a low-power CPU is to provide a marked contrast to the approach taken in current supercomputers. Current generations of low-power processors are arguably not ready for wide scale HPC deployment, lacking, for instance, double precision arithmetic and ECC-corrected memory. As we see from the benchmarks we study, they are also still a long way from meeting the stringent power and performance requirements of an exascale processor. They do, however, show what can be done *in extremis* with current technology, which can provide indications of features that could be incorporated in upcoming generations of the server-class CPUs typically used in HPC systems. More speculatively, most current HPC systems are focused on numerically-intensive calculations where floating point performance is of paramount importance. There is, however, a growing use of "big

data" HPC systems that are more aimed at relational problems such as graph analytics, such as the Cray Urika and IBM Watson appliances. The human brain is another example that performs well in pattern recognition and association benchmarks within a power budget of a few tens of Watts. Whilst the biology of the brain is very different to the silicon of a mobile phone processor, it does indicate that there is no reason to rule out extreme low power processors from all HPC fields. Whilst the precise processor choices differ from that used in the Mont Blanc EU Exascale project [6] (we consider quad-core ARM Cortex-A15 and Cortex-A7 processors, compared to the dual-core ARM Cortex-A9 in BSC's Tibidabo system [7][8][9]), they are sufficiently similar that the results should be relevant to that project. In fact, the System-on-Chip (SoC) selected as the building block for the first Mont-Blanc prototype [10] is the Samsung Exynos 5 Dual, which features a dual-core A15 processor plus an integrated ARM GPU. The SoC used for the work presented here, the Exynos 5 Octa, is a later generation that was released in the middle of 2013.

The energy and power measurements on these architectures were done explicitly, in some cases with specialist knowledge and hardware, and with some effort to derive meaningful information from the raw data. Whilst this is fine for research projects like CRESTA, this approach neither scales well to large systems, nor does it apply well to application developers who do not have such a direct interest in learning about the hardware. In these cases, it is clear that the power and energy consumption information should be provided as part of a more integrated view of application performance, as provided by performance analysis tools like Vampir or CrayPAT. Not only are such tools easier to use and more scalable, they also provide an easy tracing mechanism to look at the how the power consumption varies as the application runs, synchronised with the program tracing. This can be done without such a tool, but it is much more difficult to associate power fluctuations with phases of an application.

In this deliverable, we describe the work that has been done to incorporate power measurement profiling into the Score-P application tracing framework, and to then be able to display this information in the graphical Vampir tool, with examples of its use on the Cray XC30 systems.

The structure of this Deliverable is as follows. In Section 3, we describe the developments in the Score-P and Vampir tools. In Section 4, we investigate how power and energy consumption is affected by algorithm and runtime configuration choices on contemporary supercomputer architectures. In Section 5, we study in more detail how power usage can be understood and controlled within a modern multicore CPU. In Section 6, we study energy and power consumption for benchmarks running on a processor designed specifically for low-power environments. Having drawn interim conclusions at the end of each Section, we make some final, overall concluding remarks in Section 7.

We first begin in Section 2.4.1 by briefly reviewing some features of the Cray XC30 supercomputer architecture, as this will be relevant to Sections 3 and 4.

2.2 Glossary of Acronyms

In addition to the established Système International d'Unités of metric units and prefixes, we use the following acronyms:

ARM	RISC-based computer design approach, widely used in embedded systems
CCE	Cray Compilation Environment
CPU	central processing unit
CrayPAT	Performance analysis tool provided by Cray
CUDA	Compute Unified Device Architecture, GPU platform and programming model provided by NVIDIA

flop or op	floating point operation, a common unit of computation
flops or flop/s or op/s	floating point operations per second, a measure of computational performance
FT	Fourier transform, transformation of signals between time and frequency domains
GPU	Graphics Procssing Unit
MPI	Message Passing Interface, distributed memory programming interface
OpenMP	Shared memory multiprocessing programming interface
OS	Operating system
RAM	Random-access memory, main memory of computational node

2.3 Choice of benchmarks

Energy and power consumption should clearly be measured when the systems under test are running a computational load. Ideally, of course, this should be the CRESTA co-design applications running on peta-scale system with a suitably-sized problem. The CRESTA benchmark suite aims to capture this behaviour. Even running these same codes at smaller scale is likely to significantly change the characteristics of the code.

As the systems considered in Sections 5 and 6 contain only a few CPUs, we opt instead to use simpler benchmarks. For Section 5, which concentrates on low-level computational operations, these benchmarks are especially simple and specially constructed. For Section 6, with the especially simple ARM processor board, we use version 3.3.1 of the NAS Parallel Benchmarks [13] running on CPU nodes and parallelised with MPI. This benchmark suite was constructed to be representative of the different execution phases of a typical Computational Fluid Dynamics (CFD) application, and is thus particularly relevant to CRESTA co-design applications like NEK5000 and OpenFOAM. Alternative benchmark suites, like HPCC or SPEC, are less relevant in this regard.

For Sections 3 and 4, using a larger supercomputer, we would ideally have liked to use the CRESTA benchmark suite. This was not possible in this study, mainly because the power monitoring facilities only became available near the end of the project, and initially on smaller test systems (and initially only available within Cray). By using the same NAS Parallel Benchmarks on both a supercomputer and a low-power CPU, we could make a direct comparison of energy consumption.

Unless stated otherwise, the NAS Parallel Benchmark codes were compiled as-is and without code modifications and with no particular effort to optimised performance on the chosen systems. We chose to focus the five kernel benchmarks (as opposed to the "mini-app" codes) from the NAS Parallel Benchmark suite:

1. CG – Conjugate Gradient
2. FT – 3D Fast Fourier Transform
3. MG – Multi-grid
4. IS – Integer Sort
5. EP – Embarrassingly Parallel

IS, MG and CG are memory intensive; CG, MG and FT stress communication. A full characterisation of these codes can be found in [14].

The benchmarks provide a number of pre-programmed problem sizes, which are selected using the CLASS parameter in the provided Makefiles. We select the appropriate value for this to allow the benchmarks to execute in a reasonable time on the chosen hardware. This was usually CLASS=C for server-class CPUs in HPC systems (this Section) and CLASS=A for mobile-class processors (Section 6).

2.4 The hardware we consider

Sections 3 and 4 of this study consider power and energy measurement on current supercomputer architectures. For this work, we consider two configurations of the "Cascade-class" Cray XC30 supercomputer, which we describe below.

Section 5 considers a more generic workstation system, and Section 6 looks at a small test system. The hardware for these will be described at the start of the relevant sections.

2.4.1 The Cray XC30 architecture

The first configuration uses nodes containing two twelve-core Intel Xeon E5-2697 "Ivybridge" CPUs with a clockspeed of 2.7GHz. This configuration of the Cray XC30 matches that in, for instance, the UK *ARCHER* system installed at EPCC. The second XC30 configuration uses blades based on a hybrid node architecture. Each node contains a single eight-core Intel Xeon E5-2670 "Sandybridge" CPU with a clockspeed of 2.6GHz and an Nvidia Tesla K20x "Kepler" GPU. This configuration is the same as is used in the Swiss *Piz Daint* system installed at CSCS. For convenience, we refer to the pure-CPU nodes as "Marble" in this report and the GPU-accelerated nodes as "Graphite", replicating the project names used when the systems were developed.

For reference, we give a brief review of the architecture of the Cray XC30. Two identical nodes are built on a single "processor daughter card" (PDC). Two of these cards (again, with identical nodes) then fit into a blade chassis, which houses a single Aries network controller that is shared between the four nodes on the blade. Sixteen of these blades (64 nodes) slot into a chassis (blades containing different node architectures can be mixed at this point). Within a chassis, an all-to-all network connection is made between Aries controllers via copper backplane connections (the level-1 network). Three chassis are housed in a single cabinet (192 nodes), with pairs of cabinets sharing a common cooling infrastructure (a "group" of 384 nodes). The all-to-all network between nodes in a chassis is now supplemented by a further all-to-all network between the six chassis in a group using copper cables (the level-2 network). Finally, a level-3 all-to-all network is configured between groups of cabinets using optical cable bundles, with the size of the cable bundle configurable to meet bandwidth and cost requirements. With these three all-to-all networks, the minimal routing between two nodes involves at most 5 hops (two electrical, one optical and then two further electrical), although the network will adaptively use non-minimal routing to avoid congestion.

2.4.1.1 Cray XC30 power and energy counters

The Cray Linux Environment OS provides access to a set of power counters via files in directory /sys/cray/pm_counters (which we abbreviate here to $PM) on the compute nodes. Counter $PM/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 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 $PM/energy records the cumulative energy consumption of the node (in Joules) from some fixed time in the past.

The $PM/power and $PM/energy counters are available for all node architectures and always measure the consumption of the full node. On accelerated nodes, additional counters $PM/accel_power and $PM/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 $PM/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 so, to ensure that a consistent set of values are obtained, the procedure for reading should be as follows.

First the freshness should be read and stored. The power and/or energy counters should then be read sequentially. The freshness should then be read once more. If the initial and final values of the freshness agree, the values of the other counters are accepted, otherwise we should repeat the procedure. This process is illustrated in the example script in (which is a simplified example of a user-constructed script, rather than a Cray-supported product).

For the work in this Section, we concentrate on the information in the $PM/energy and $PM/accel_energy counters. By measuring the values of these counters at the start and end of program execution, we can calculate the node and accelerator energy consumption.

2.4.1.2 P-states on the Cray XC30

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 on the fly, e.g. to save battery power on mobile devices, because the lower the clock speed, the less power the CPU consumes. Here we investigate this functionality as a mechanism for reducing the energy consumption of applications on HPC systems.

A description of the governors, and their available settings can be found Ref [11]. The default governor setting is "performance" on Cray XC30 systems, with CPUs consistently clocked to their highest settings, which also permit the CPU to enter internal "boost" states as and when the hardware decides.

For this work, we concentrate on the "userspace" governor, which allows the user to set the CPU to a specific frequency (known as a p-state). 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")

On a Cray XC30, the range of available p-states on a given node is listed in file:

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

The p-state is an integer that corresponds to the CPU clock frequency measured in kHz. So a CPU frequency of 2.6GHz translates to a p-state label of 2600000. 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 upper frequency of the CPU; this option is available to the hardware if the p-state is set to 2601000 (for a 2.6GHz processor).

For convenience in this report, when quoting p-states, we will drop the final three zeros, so "2600" should be interpreted as being in units of MHz, and so translates to a clock frequency of 2.6GHz.

For the Sandybridge CPUs, the p-states ranged from 1200 to 2600, with a boost-able 2601 state above that (where the so-called "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). The highest p-state common to both CPUs was 2500.

2.4.1.3 Enhanced MPMD with Cray ALPS on the Cray XC30

The Cray ALPS 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). This MPMD mode, however, has historically only worked if each node runs a single binary (although different nodes in the same aprun instance can execute different binaries). For the work done here, particularly when we mix CPU and GPU executables on the same node, we take advantage of a new way to run 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. It is important, however, that environment variable PMI_NO_FORK=1 is set in the main batch jobscript to ensure the binaries launch correctly.

```
#!/bin/bash
PM=/sys/cray/pm_counters # The location of the counters
good_freshness=0 # Value 1 when we have reliable measurements
# Loop until we have reliable measurements
while [ $good_freshness -eq 0 ]; do
# Measure the freshness at the start of the measurements
       start_freshness=$(cat ${PM}/freshness)
       ENERGY=$(awk '{print $1}' ${PM}/energy)
       ACCEL_ENERGY=$(awk '{print $1}' ${PM}/accel_energy)
# Measure the freshness at the end of the measurements
       end_freshness=$(cat ${PM}/freshness)
# If the freshness didn't change, then we can exit the loop
              if [ $end_freshness -eq $start_freshness ]; then
              good_freshness=1 # Record we have reliable values
              FRESHNESS=$end_freshness
       fi
# ... otherwise we should repeat the measurements
done
```

Figure 1. A simple example bash script to read energy and accel_energy counters.

3 Tools for power measurement and visualisation

With whole job and system energy monitoring, only the amount of energy for a given period is of interest. By contrast, monitoring of energy and power consumption during performance analysis of applications is more challenging. Application performance analysis relies on detailed power and energy information to correlate this information with the fine-grained events of the application and, at the same time, we have to take care about 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 derived from energy measurements) can also provide misleading information.

In this section, we present the infrastructure needed to monitor power and energy counters with the application monitoring system Score-P and the challenges for a detailed performance analysis. We classify metrics by their synchronicity and their scope to select the best monitoring strategy and conclusion for a given metric. In addition, we visualized the monitoring data of two benchmarks (a synthetic load-idle pattern benchmark, and the CUDA HPL implementation) with the performance visualizer Vampir and analyzed the characteristics and behaviour of the energy and power metrics.

3.1 Score-P metric plugin interface

Since version 1.2 the application measurement system Score-P is able to record generic and user-defined hierarchical performance counters. This is done with a "metric plugins" interface to address the level of complexity both of today's machine architectures and of those 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 are loaded at application runtime by the measurement system.

External counters are by default special counters with own properties, as well as requirements for the monitoring of these counters. Therefore, we classify the external counters by their synchronicity and their scope to select the best monitoring strategy.

Metrics by Synchronicity

In terms of synchronicity, we basically divide the metrics into synchronous and asynchronous metrics. Whereas synchronous metrics are correlated with the events of the instrumented application, the asynchronous metrics are independent from events of the application.

In the following we define metric classes, which are used within Score-P, and differ from each other by the time a counter is called to get an update of its value and which and how many information will be written:

Strictly Synchronous Metrics

A strictly synchronous metric is called whenever an event during the application monitoring occurs and the actual value will then be attached to this event. One important property of strictly synchronous metrics is that they can be recorded only at the occurrence of an event and must be recorded at each event. Every time the metric is called to deliver its current value, it must return exactly one value. An example for this type of metric is a counter for the number of past events.

Synchronous Metrics

A synchronous metric is called whenever an event occurs to get an update of its value. It can be recorded only at the occurrence of an event, but in contrast to the strictly synchronous metrics there is no need to write a value at each event. When a synchronous metric is called, it may provide an update of its value (exactly one) or not (no value). An example for this type of metric is the state of the machine or the node at the occurrence of an event.

Asynchronous Event Metrics

An asynchronous metric event is called whenever an event occurs to get an update of its value. In contrast to the synchronous metrics, it allows to provide an arbitrary number of timestamp-value-pairs at one call. These values are not attached to a single event. An example for this metric is the information of the machine stored in a database by a daemon and values can be requested for a specific interval.

Asynchronous Metrics

In contrast to asynchronous event metrics, these metrics can be recorded at arbitrary points in time. An example for this metric is the power information of the machine from a power meter.

Metrics by Scope

Furthermore, we divide the metrics by the scope for which the metric is valid. Examples for the different levels of scope can be the different levels of the machine hierarchy, e.g., rack, blade, or the software level, e.g., process, thread. Within each level of scope only one instance is allowed to record the values of a metric.

3.2 Cray XC30 energy and power monitoring

As described in Section 2.4.1.1, Cray's energy and power counters are periodically updated with a 10Hz frequency. The scope of these metrics is per node and the 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 and can be therefore classified as asynchronous. To reduce the overhead during a specific event we decided to collect all energy and power values with asynchronous metrics. We therefore have to start an additional thread that polls the files and checks if updates are available, collect the new values, and link it with an actual timestamp. At the end of the monitoring the vector of timestamp-value-pairs are 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 to choose 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.

3.3 Performance and metric visualisation with Vampir

The energy and power metrics are value-pairs (the timestamp and the associated value) and by their nature they only reflect a value at a specific point in time. Therefore they 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, cannot be done without any knowledge about correlation factors.

Performance visualisers like Vampir decouple power and energy metrics from the event stream and display the behaviour over time in special timelines, e.g., 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. In combination with timelines, which visualise the event stream over time, e.g., master timeline, a visual correlation of these different types of information will become possible.

Figure 2 shows the visual representation of a MPI parallel application using hyper-threading, running on four nodes with 32 processes each (left upper part), and corresponding node power and freshness information with Vampir (left lower part).

3.3.1 Synthetic load idle pattern

We developed a synthetic load-idle benchmark with decreasing intervals to check if the energy and energy and power counter can capture and reflect this alternating switch of extreme states. Figure states. Figure 3 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 benchmark it is obvious that there is a correlation between the load-idle patterns and the node energy and power metrics. At the end of the benchmark when the length of the load-idle pattern for each state is 50ms the energy and power metrics (especially the instantaneous power) are not able to reflect the state changes, which is clear since the sampled sampled instantaneous values are only valid for one point in time. This behaviour can be seen in

be seen in

Figure 4.

3.3.2 HPL CUDA

To demonstrate how information about accelerators is collected and displayed, we executed the executed the HPL CUDA application [12], which uses MPI processes, OpenMP threads, and CUDA kernels on a Cray XC30. In addition to the traditional application behaviour monitoring we recorded the energy and power counter for the nodes and the installed graphic cards.

Figure 5 shows the colour-coded visualisation of CUDA HPL with Vampir. The topmost timeline shows the behaviour of the processes, threads, and CUDA streams over time for an interval of 2 seconds. **Error! Reference source not found.**

We can convincingly show that the node power mainly depends on the execution of the CUDA kernels (blue boxes) for this application. Another interesting fact is that instantaneous power values are hard to interpret and can lead to misleading decisions. This can be seen in **Error! Reference source not found.** Figure 6within the third and the fifth timeline, which display the instantaneous node power and respectively the instantaneous accelerator power. Since we have seen that the accelerator power has a strong impact on the node power, we can see in this monitoring interval that the decrease of the node power is two samples earlier than for the accelerator power. The rates of change of the node power and respectively of the accelerator power 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, visualized within the second timeline of **Error! Reference source not found.**. Curiously, the energy metric curve appears smoother than the curves for the instantaneous power metric and for the application event information. This effect is often due to digital filtering, although this is not carried out for the metrics, and we are investigating this effect further.

3.4 Conclusions of this Section

We conclude that even on today's supercomputing architectures, the power and energy consumption of the node changes markedly as an application moves between different phases of execution. Tracing and visualising this is important, both to allow developers to identify and then focus on the most expensive parts of their application, but also to provide feedback on how algorithm choices and other changes affect the overall power and energy consumption of the application. By allowing this information to be collected and displayed within the wider application tracing and visualisation tools Score-P and Vampir, an integrated view of the entire application performance data is possible, including when accelerators are used.

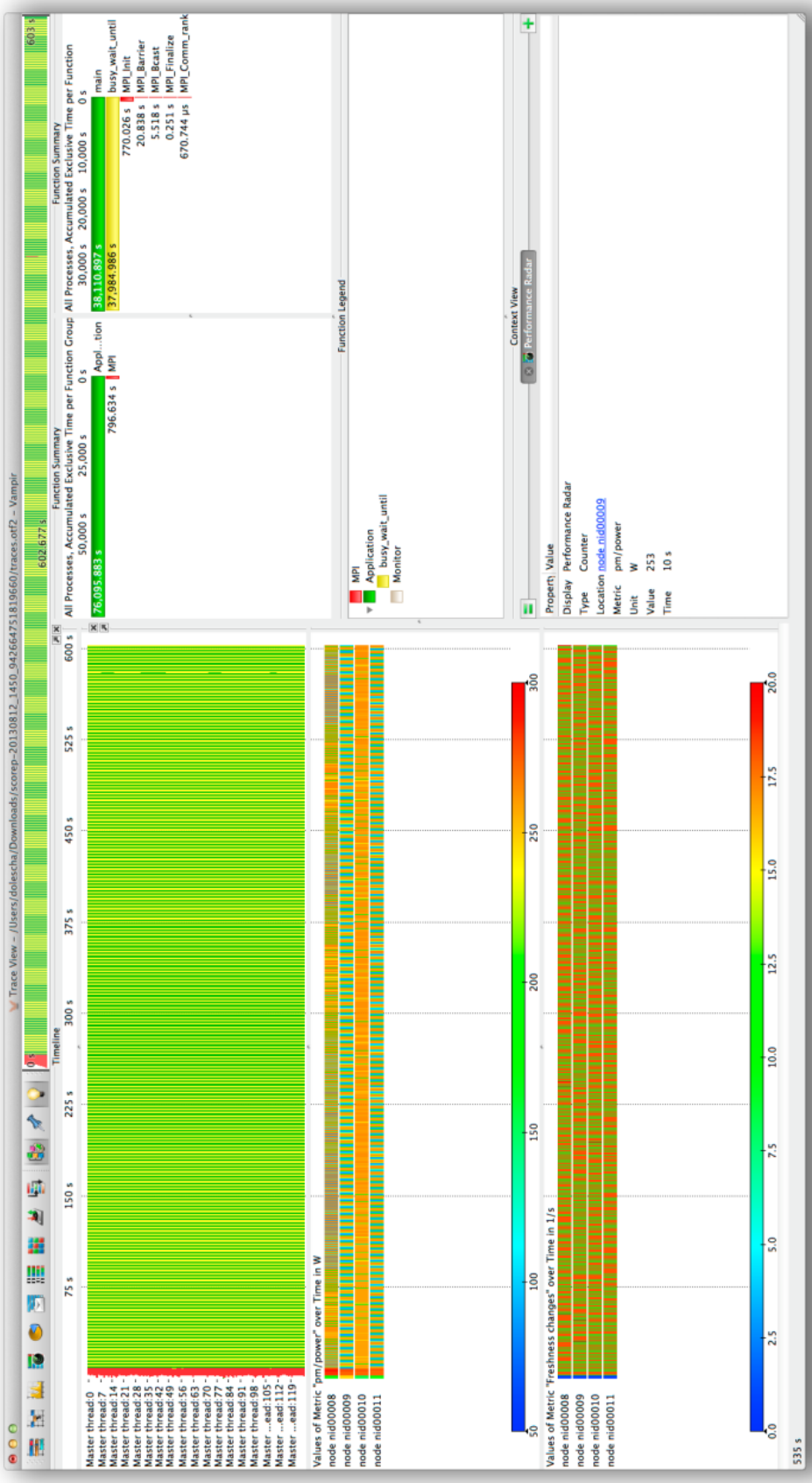

Figure 2 Performance visualisation with Vampir of the application behaviour and node power and freshness information of a MPI parallel application using hyper-threading, running on four nodes with 32 processes each monitored on a Cray XC30.

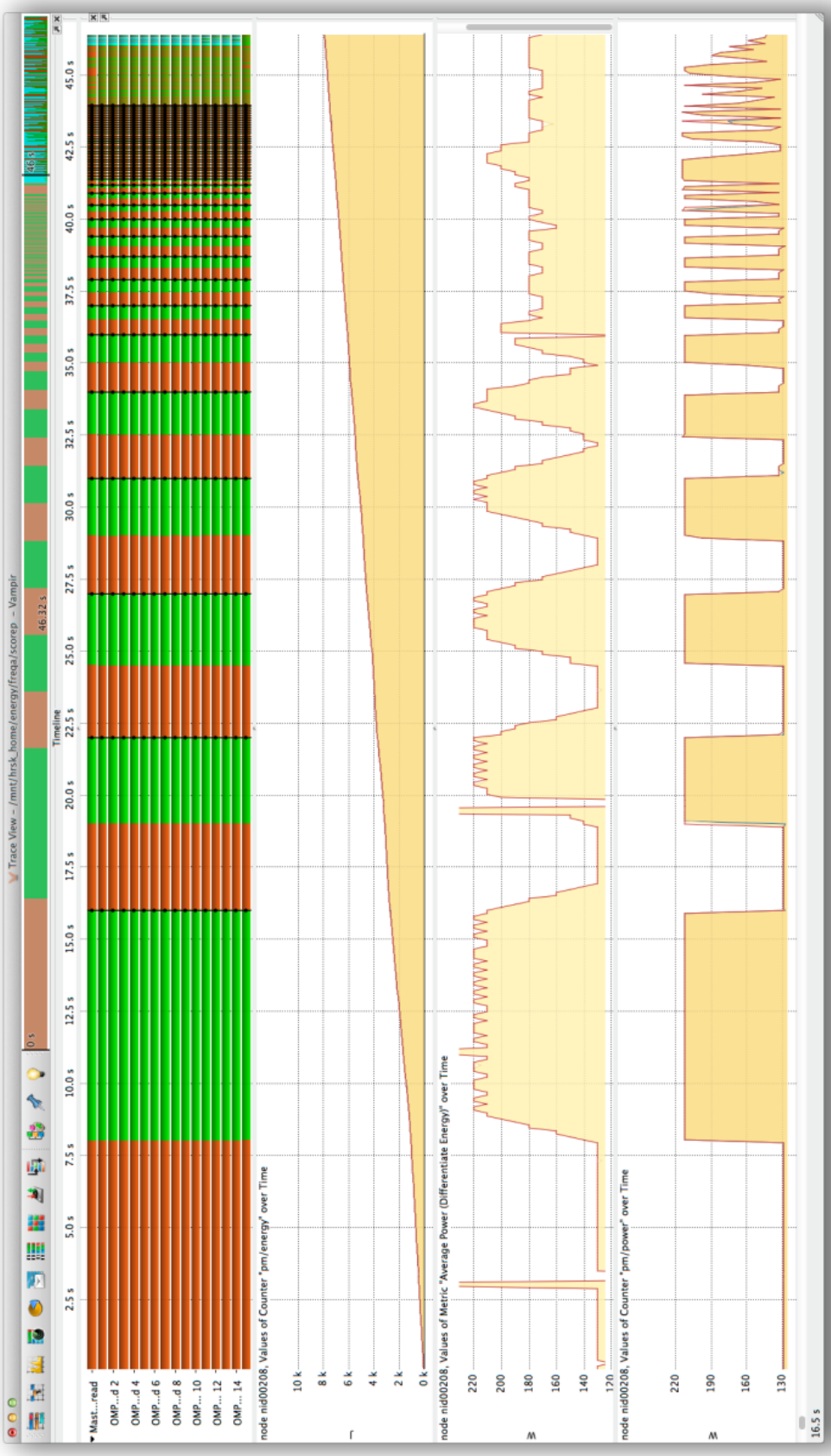

Figure 3 Load-idle benchmark with colour-coded visualisation of the load-idle regions and corresponding energy, average power derived from energy, and power information with Vampir.

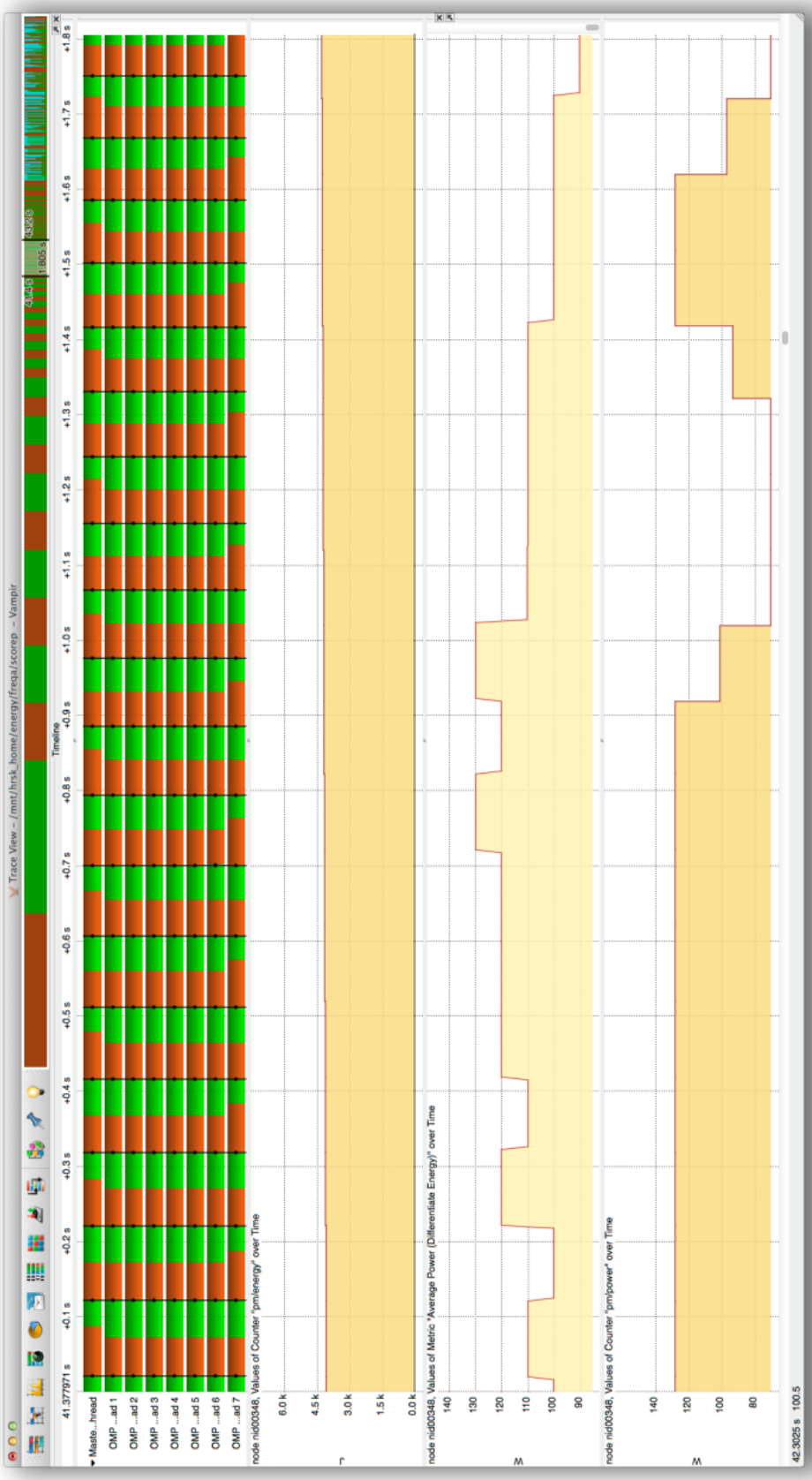

Figure 4: Length of load-idle pattern of 50ms. The instantaneous power metric (lowest timeline) is not able to reflect this alternating pattern. Even the average power derived from the energy is not able to reflect this alternating pattern.

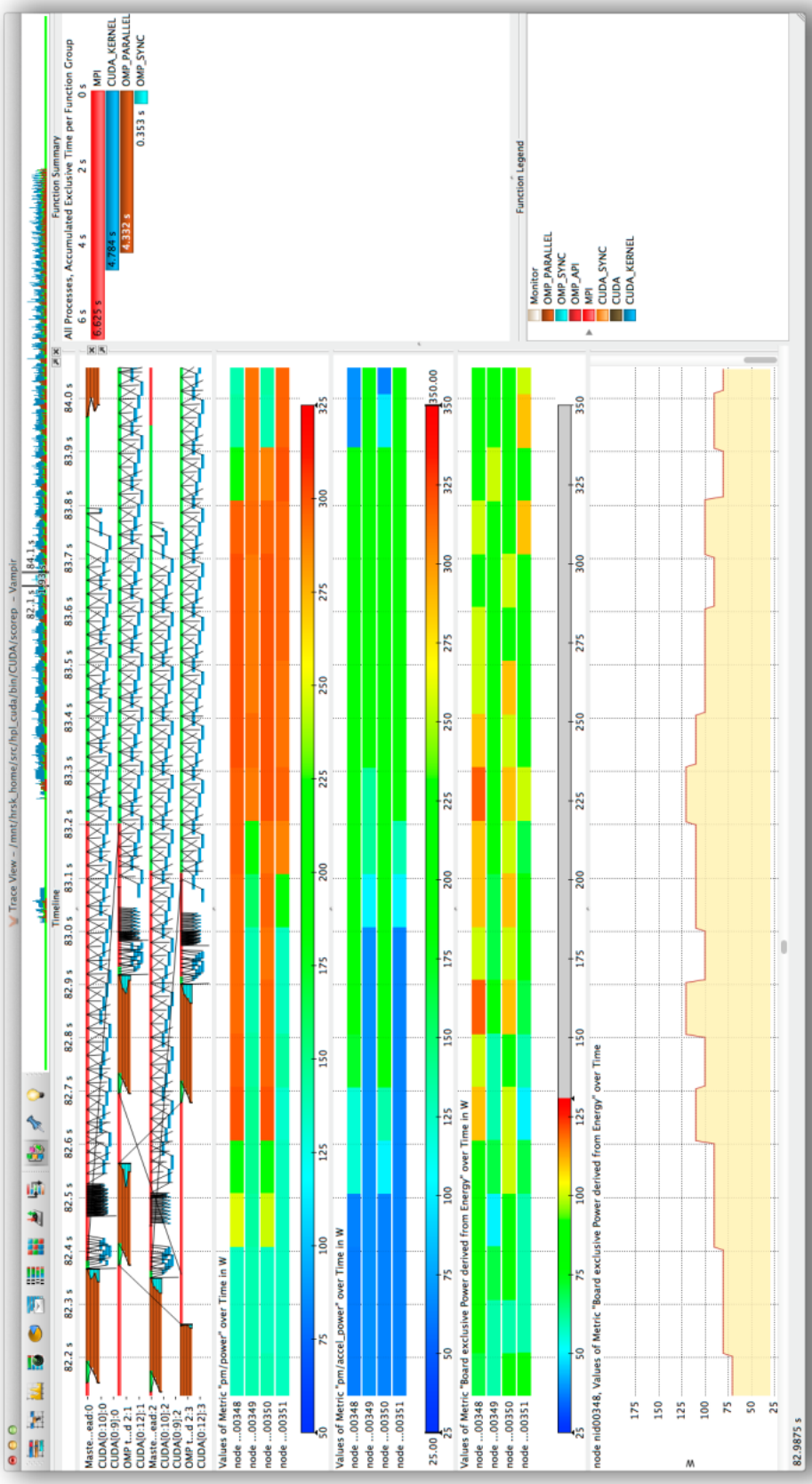

Figure 5 Colour-coded visualisation of CUDA HPL with Vampir. The topmost timeline shows the behaviour of the processes, threads, and CUDA streams over time for an interval of 2 seconds. The second timeline displays the instantaneous node power metric over time. The third timeline displays the instantaneous graphic card power metric and the lowest timeline displays the board exclusive power without the graphic card derived from the energy of the first node.

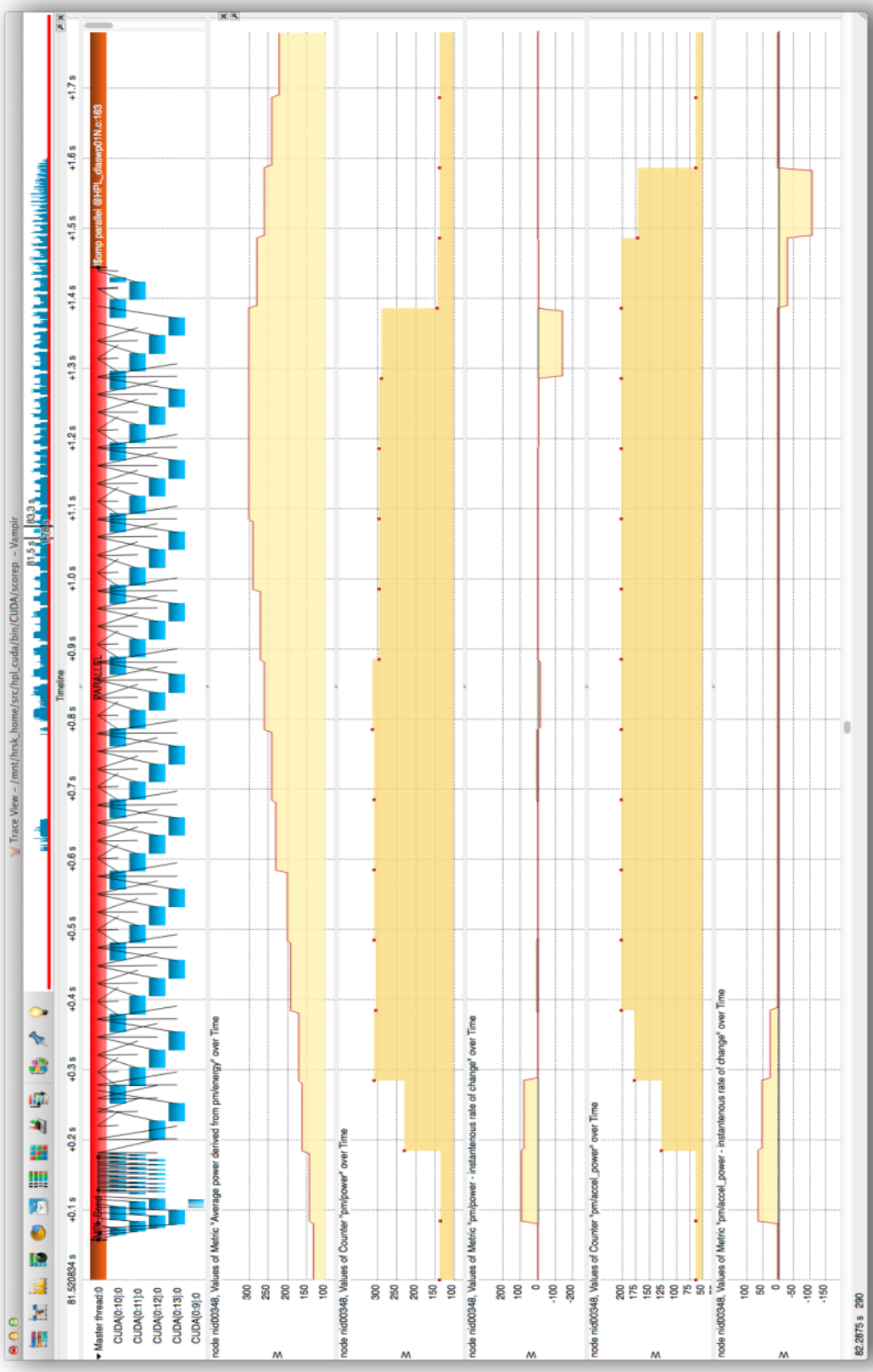

Figure 6. Colour-coded visualisation of the first node of the CUDA HPL application for an interval of 1.8s with according timelines for the events, average node power derived from the node energy metric by a backwards difference operator, instantaneous node power metric, rate of change of the node power metric, instantaneous accelerator power metric, and rate of change of the accelerator power metric.

4 Power consumption on a supercomputer architecture

In this section, we describe some energy and power consumption measurements carried out on a number of node architectures in the Cray XC30 supercomputer line.

In particular, we aim to answer a number of interesting and timely questions. All these are based around the central question, is minimising the application runtime also the most energy-efficient choice?

While many of the details we present here are specific to Cray XC30 systems, they are relevant to CRESTA partners (most of whom have access to such systems), but also reflect similar capabilities in other vendors' systems.

All codes considered here were compiled using the Cray Compilation Environment (CCE), using the latest 8.2 release of the compiler.

4.1 Idle power

The first stage in understanding the energy consumption of applications is to measure the baseline power consumption of an idle node. To measure this on each architecture, one physical CPU core on each of eight nodes (located on two physical blades) executed a ten-second sleep. The energy consumption of each node (and, where relevant, of the accelerator on each node) was measured during this interval. This was done four times (as part of the same batch job), and the mean was calculated for the resulting 32 measurements. This whole process was done separately with the CPUs of the nodes in each of the full range of available p-states. In all cases the standard error for the average was much less than 1%.

The results are shown in Figure 7.

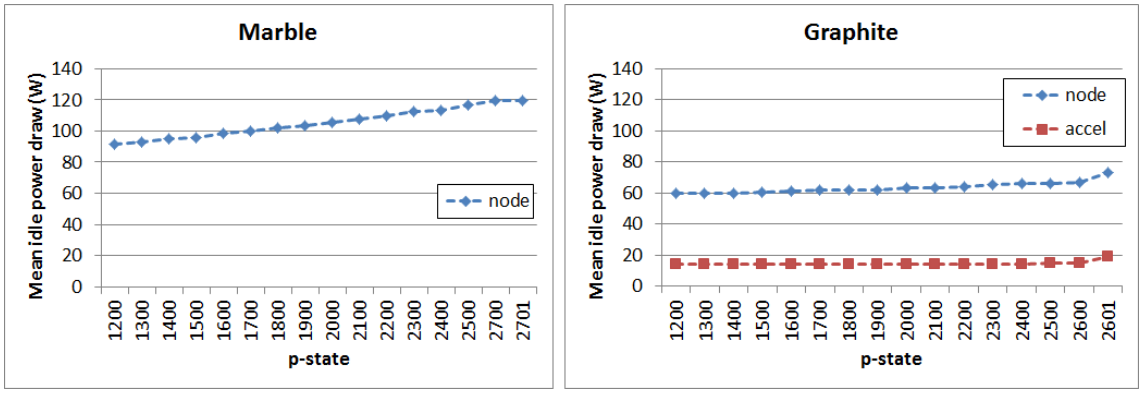

Figure 7. Idle power draw of Marble and Graphite nodes

For the CPU-based nodes, the base power increases 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.

For the GPU-accelerated nodes, we see that the base node power increases 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 accelerator; we see that the accelerator consumes a relatively constant 14.6W, with the only deviation being for the top, boost-able p-state, when the accelerator draws 18.9W. These numbers suggest that the jump in power associated with boosting is largely due to the GPU alone, with the Sandybridge CPU behaving like the Ivybridge model described above and not using the boost state.

We can attempt to compare the two CPUs at their highest common p-state, 2500. The accelerated node draws 65.9W, of which 14.6W is due to the accelerator. Subtracting, this leaves 51.3W for the idle Sandybridge CPU in this boost state. For the CPU node, the node power is 116.5W. If we divide by two, this suggests a base power of 58.3W

per Ivybridge CPU. The complication here is that the two nodes are not identical, but it is clear that any difference in power between the two is slight.

In the following subsections, we will consider the node energy consumption when running applications. Where appropriate, we will also show a "subtracted" figure that estimates the additional consumption over and above that of an idle node. To calculate the figure for the idle node, we will use the mean power figures discussed above, multiplying by the runtime of the application in question.

4.2 Choice of metrics

4.2.1 Reported and derived metrics

On execution, the NAS Parallel Benchmarks each report their execution time as well as a performance in units of Mop/s. This is an internally calculated figure based on a problem size-dependent operation count (this is hardcoded rather than measured from CPU hardware counters), divided by the runtime. In addition, we can externally measure the total energy consumption by the application.

Dividing this energy figure by the runtime gives us the mean power consumption of the application. Finally, we can construct a performance:power ratio that mirrors that used in the Green500 rankings, in units of Mop/J (which is equivalent to Mop/s per W). For this, we multiply the Mop/s figure by the runtime and then divide by the energy consumption.

As explained in the Introduction, different metrics are important in different situations and we present each of them for the benchmark codes in this Section.

4.2.2 A cost-benefit metric

We expect (and shall see below) that we can reduce the total energy consumption due to an application running on those nodes if we reduce the processor performance and so increase the application runtime.

Superficially, this appears economically (as well as environmentally) sensible, but this is based on an overly-simplistic model where HPC costs are considered solely to be due to the electricity used in running the system. In reality, the initial cost of the HPC hardware is not small, and we cannot run as many jobs over the lifetime of the system if we increase the runtime [15].

We can develop a simple model for this. Let the initial system cost be S and assume the system will be operated for T years (with the system cost depreciating at rate S/T per year). Each year, we run N (assumed similar) jobs with an annual electricity cost of C. The overall financial cost per job is therefore:

$$K_0 = \frac{S}{NT} + \frac{C}{N}$$

If we reduce the p-state to some chosen (perhaps optimal) level, we assume the runtime of all these jobs increases by a factor of R (which is greater than 1), whilst the energy consumed changes by a factor of E (which is less than 1). We will therefore run a factor of R fewer jobs per year, and the cost per job becomes:

$$K_1 = \frac{SR}{NT} + \frac{CE}{N}$$

Overall, reducing the p-state will lead to cost savings if $K_1 < K_0$. Clearly if the running costs are dominant, this favours reducing E (the energy consumption) as much as possible, regardless of the runtime penalty. If the depreciation per year dominates, however, we should make R as small as possible, minimising the runtime. In general, rearranging the inequality, we will see cost savings from throttling performance if:

$$\frac{CT}{S} > \frac{R-1}{1-E}$$

Broadly summarising this, the longer we run a system, the more we care about the costs of running it, compared to the initial outlay. The proportional change in cost by throttling performance will be:

$$\frac{K_1}{K_0} = \frac{R + E\frac{CT}{S}}{1 + \frac{CT}{S}}$$

The payback time measures how long (in years) one would need to run a system with throttled applications before it becomes economically advantageous:

$$T = \frac{R-1}{1-E} \times \frac{S}{C}$$

As a very rough figure, a typical contemporary HPC system has an annual electricity to initial system cost ratio of $C/S \sim 5\%$ and we shall see that for the results in this section, the payback time is almost always larger than the typical system lifetime of 3-5 years.

4.3 Application performance with Marble nodes

We begin by studying benchmark performance on the pure CPU "Marble" nodes. The node version we consider here have 24 physical cores per node. The NAS Parallel Benchmarks, however, generally require the number of MPI ranks to be a power of two. There is therefore no way to arrange running the NAS Parallel Benchmarks using all the cores on all the selected nodes. We instead opt to use eight of the available twelve cores per CPU on four nodes. For this number of cores, the appropriate problem size is CLASS=C.

4.3.1 Application performance with changes in p-state and hyperthreading

The benchmarks were compiled and run in two ways. First, we compile to use 64 ranks and run these with one rank per physical core across four nodes, using aprun options -n64 -S8 -j1. Then we compile with 128 ranks and run with two ranks per physical core, utilising the two hyperthreads available on the Intel cores. This is done with aprun options -n128 -S16 -j2.

The results are shown in Figure 8. Each row shows one of the five benchmarks considered. For each benchmark, the left graph shows application performance as reported by the benchmark (which is inversely proportional to the main calculation runtime). The graph on the right shows the combined energy consumption of the four nodes (for the entire application runtime). In each case, we show the variation in these metrics as we vary the p-state of the node between the lowest and highest settings. The two curves on each graph show the results with one ("J1", no hyperthreading) and two ("J2", with hyperthreading) ranks per core. 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 commonly seen with many codes, we see that 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. In general, it is difficult to predict if hyperthreading will benefit a particular application, and some experimentation is required.

The energy consumption graphs follow the inverse pattern; if no hyperthreading gives the best performance, it also gives the lowest energy consumption. The can be understood in terms of power. Defining the mean power consumption as the total nodal energy consumption divided by the runtime, we see that the power of the node is essentially the same whether or not hyperthreads are employed. The more efficient way of running is therefore the faster.

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 (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...). For a loaded system, however, it is not unreasonable to view these as drawing a fixed power, which we could apportion to each node as an addition to the base power draw of each CPU. This may well change the optimisation balance, and will need to be included in future studies. On the Cray XC30, this information is not easily accessible to a non-privileged user, so we do not focus on it here. Also, as we shall see, 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.

4.3.2 Optimisation and runtime and energy performances

The results so far have used the default -O2 optimisation level for CCE. An interesting question is whether this level of optimisation leads to higher energy consumption.

The short answer is "no". In Figure 9 we compare the results for each benchmark (using the better hyperthreading setting in each case) with compiler optimisation ("-O2") and without ("-O0"). It is clear that in all cases the optimisation leads to faster codes which overall consume less energy.

In Figure 10 we divide the mean energy consumed per node by the application runtime to calculate the mean power consumed by the node during application execution. It is clear from this that reducing the optimisation level does lead to the node consuming less power. This reduction, however, is not large enough to compensate for the increased runtime, leading to the higher overall energy consumption.

Future processor architectures may have a greater capacity to temporarily hibernate unused portions of silicon on the chip, which may mean that the reduction in power consumption is much greater when optimisation is reduced. But for current server-class CPUs used in HPC, this is not yet the case. The circumstance in which a reduced optimisation level is useful would be if there was a need to cap the maximal power consumption (as opposed to the total energy consumption) of the system. This would of, course, be at the price of both increased runtime and overall energy consumption.

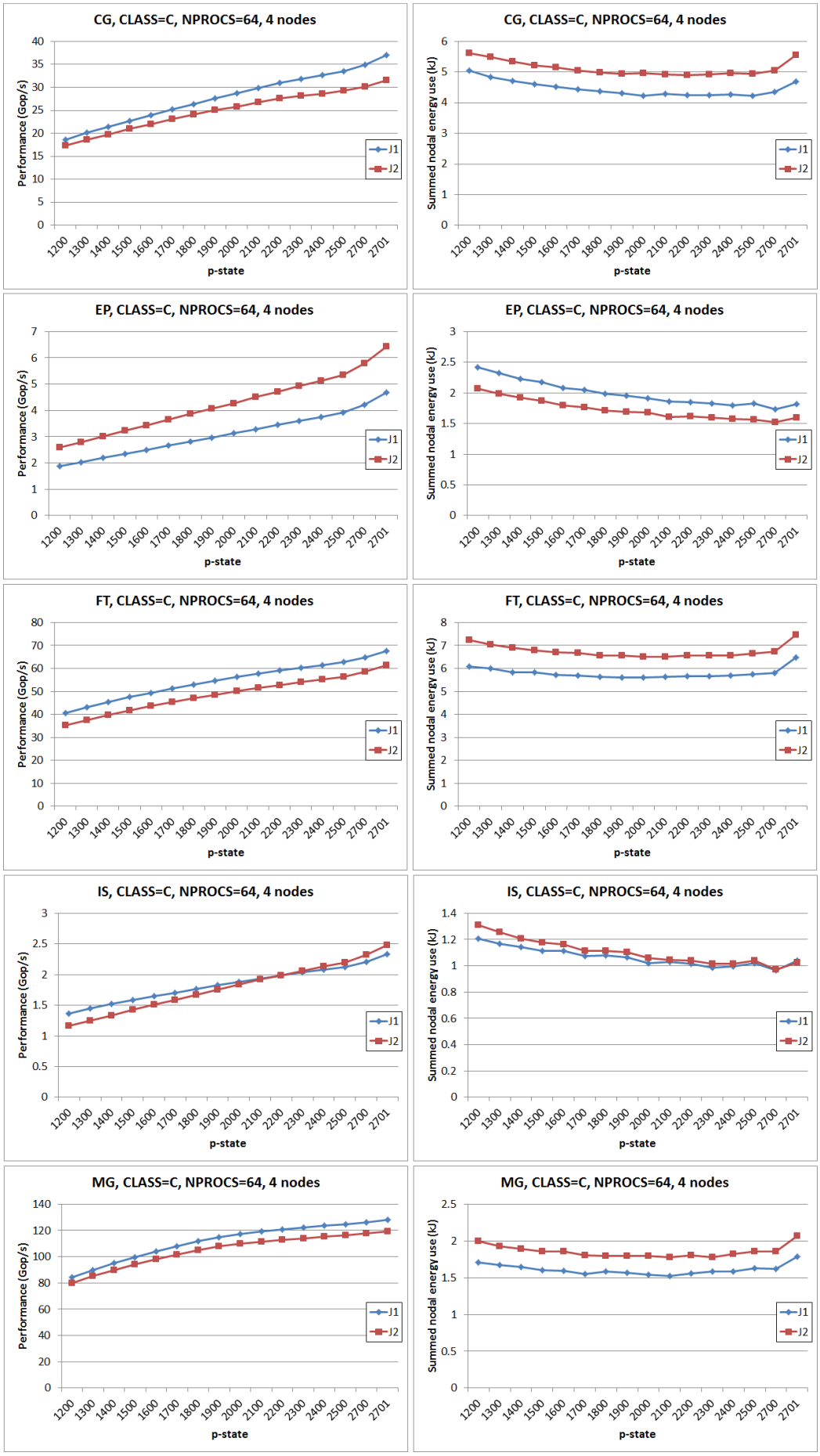

Figure 8. Running the NAS Parallel benchmarks on Marble nodes, with and without hyperthreading.

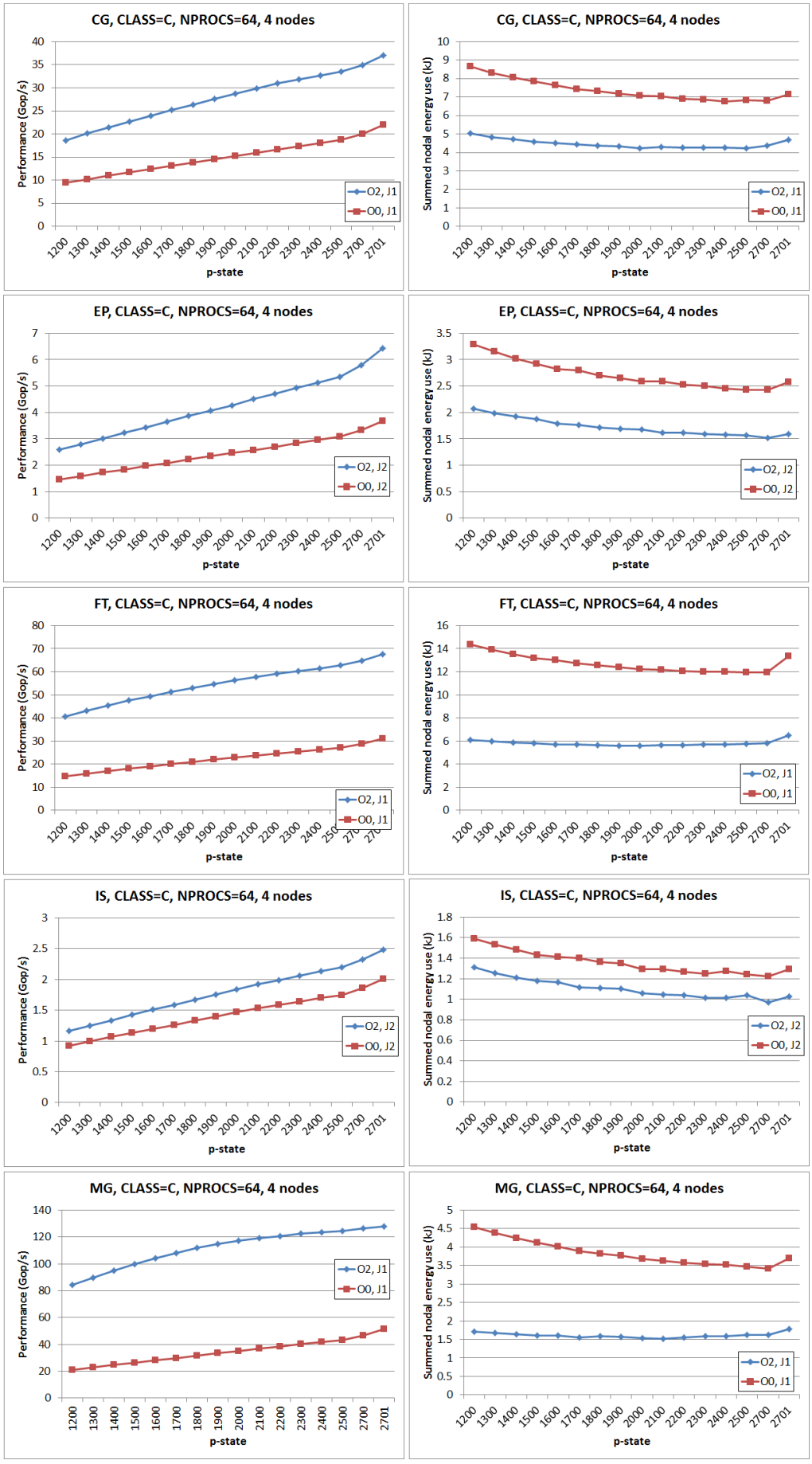

Figure 9. Effect of compiler optimisation on NPB on Marble nodes, with optimal hyperthreading.

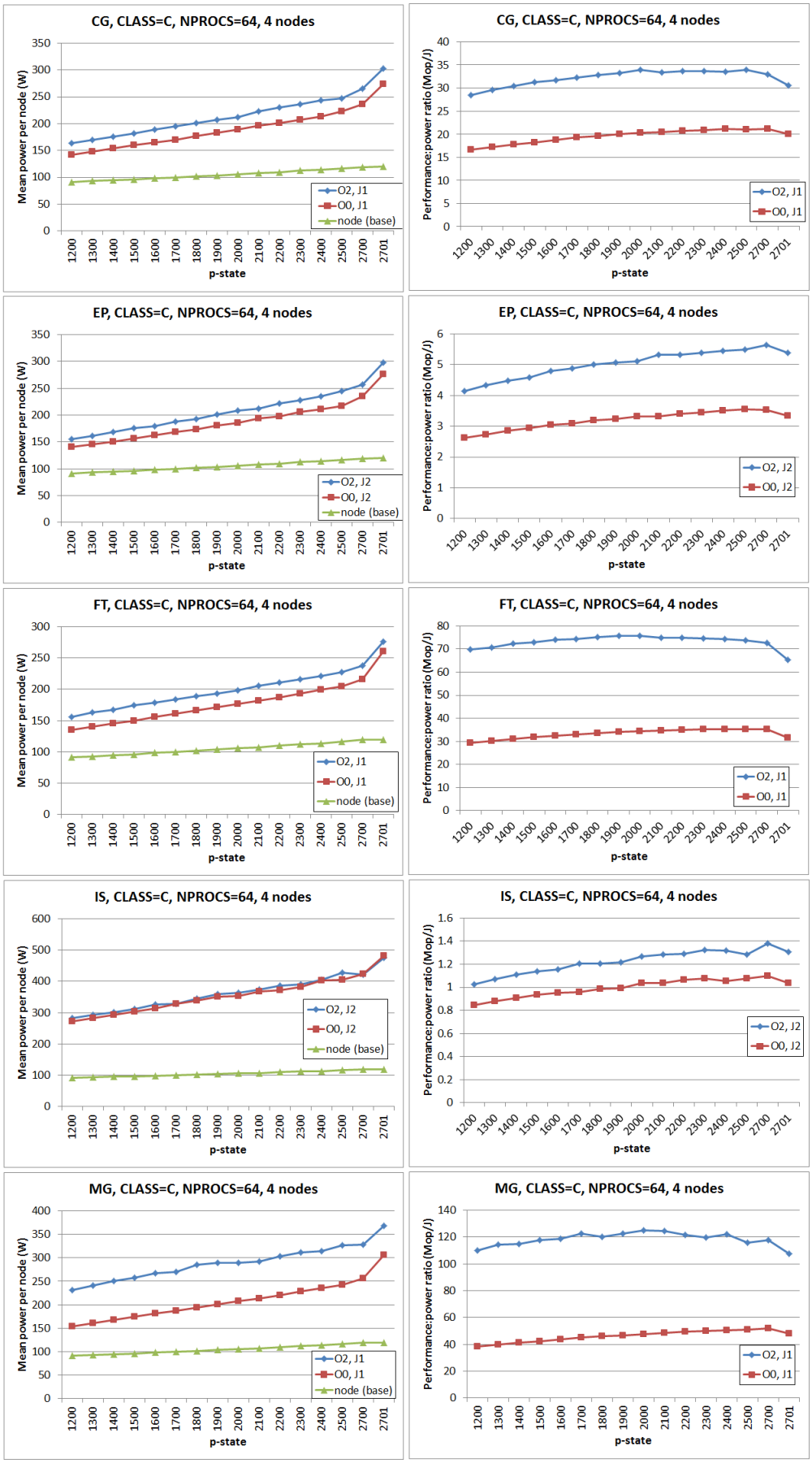

Figure 10. Mean node power consumption and performance:power ratio on Marble nodes.

4.3.3 Balancing increased runtime against saved energy

We have seen that whilst reducing the p-state increases the application runtime, it can also decrease the energy consumed in completing the calculation. 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% decrease in runtime is balanced by a 15% increase in energy efficiency.

We can examine whether this is cost efficient using the model developed above. Using the cost-benefit model developed in Section 4.2.2, we have $E \sim 0.85$ and $R \sim 1.05$. We need to assume at this stage that the fractional change in nodal energy consumption is reflected in the overall energy expenditure of the system (i.e. a linear Power Usage Effectiveness (PUE) model to include network and cabinet infrastructure). In this case, we would only expect to see savings if the system were run for longer than the payback time

$$T = \frac{R-1}{1-E} \times \frac{S}{C} = 6.7 \text{ years},$$

which is 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 (and of benchmarks), there is only one instance where the payback time is less than five years. So, even though we can now demonstrate a reduction in node energy usage by downclocking the CPUs, we are not yet in an era where this is a cost-effective way to run an HPC system.

If energy prices were to rise by, say, 30%, this balance would already change. Alternatively, if the decision were to procure a long-lifetime system, it would also make reducing performance more attractive from the overall cost perspective, but these benefits may be outweighed by efficiency gains in systems available midway through this long-lifetime procurement.

4.3.4 Comparing flat MPI with a Hybrid MPI/OpenMP algorithm

So far, we have studied energy consumption as we change system parameters, but not looked at the effects of using different algorithms. As a simple example of this, we here compare a hybrid MPI/OpenMP programming model with the pure MPI examples studied in the previous sections.

For this work, we developed a hybrid MPI/OpenMP version of the MG benchmark, as this is not included in the "MultiZone" partial distribution of the NAS Parallel Benchmarks. Whilst the OpenMP scaling appears very good for the range of OMP_NUM_THREADS considered here, no particular effort was made to optimise the OpenMP directives. There was not scope within this study to hybridise all the benchmark codes, so we concentrate on MG for this section.

For the hybrid investigation of the MG benchmark, we would like to reduce the number of MPI ranks per NUMA node, introducing OpenMP threads for each rank such that the total number of processes (threads) per CPU is constant. So for this study we will continue to use 64 threads (on four nodes) to calculate the CLASS=C benchmarks. We can then compile with 64 MPI ranks and run with one thread per rank (aprun options -n64 -S8 -d1), compile with 32 ranks and use two threads per rank (-n32 -S4 -d2), with 16 ranks with 4 threads per rank (-n16 -S2 -d4) or with 8 ranks with 8 threads per rank (-n8 -S1 -d8). At this stage we have one rank per NUMA node; performance is likely to deteriorate if we reduce the number of ranks further to one per node. Aprun options -j1 and -ss were used throughout (the latter to ensure that cores on one CPU do not access memory associated with the other CPU's memory controller).

The results are shown in Figure 11, for a variety of metrics. It is clear that threading has no real effect on performance. This is not surprising; the benchmarks are run over a modest number of nodes and, at this scale, do not show any of the signs that a hybrid algorithm will be more successful.

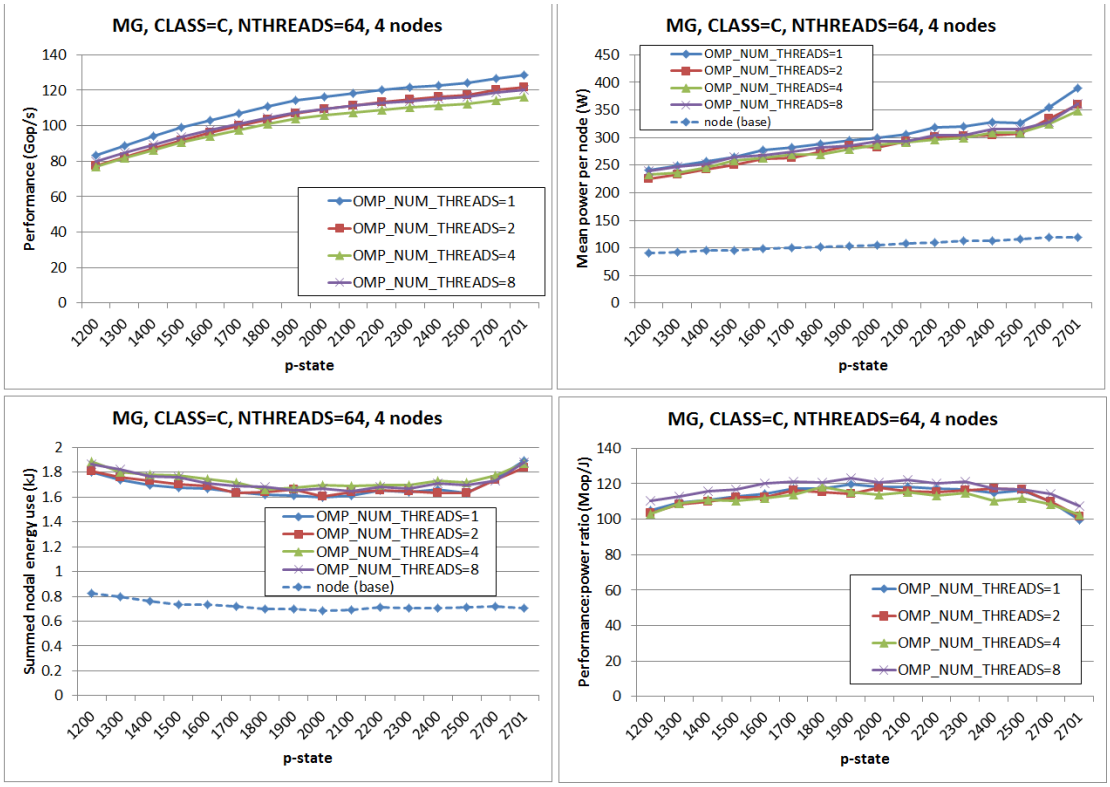

Figure 11. Moving to a hybrid MPI/OpenMP algorithm on Marble nodes.

4.4 Application performance on Graphite nodes

In this Section, we turn our attention to energy measurements on accelerated nodes containing a CPU and associated GPU.

4.4.1 Accelerator initialisation and data transfers

An interesting question to consider is to what extend do CPU clockspeed settings affect GPU performance. If there is no effect, it would make sense (from an energy perspective) to run GPU applications with the lowest possible p-state. As we see from the results in this subsection, however, the GPU performance does depend on that of the CPU. The GPU is an attached accelerator with a host-directed execution model and the performance of the CPU host does affect the GPU.

As a simple example, the GPU initialisation time (measured using the PGI pgaccelinfo utility) showed a marked increase as we vary the p-state, from 25ms at 2601 to 35ms at 1200. Whilst this is a clear example of the relationship between GPU and CPU performance, this time is, however, uninteresting for most real applications as the GPU is only initialised once during application start-up.

We also measured the effect of changing the p-state on the GPU performance, using two simple, standard benchmarks written in CUDA. A 1024x1024 double precision matrix multiplication using the CUBLAS DGEMM library showed that the GPU performance reduced by less than 1% as the p-state was changed from 2601 to 1200. A similarly-small reduction was seen in GPU memory bandwidth using a standard "triad" benchmark. By comparison, the same operations on the CPU showed much larger reductions in performance of 21% and 56%, respectively.

The performance of real applications, however, depends on the GPU performance but also (to a varying degree) on the time taken to transfer data between the separate memory spaces of the CPU and GPU. The PGI pgaccelinfo utility was used to measure the bandwidth for data transfers between CPU host and GPU accelerator. The results for 4MB data transfers (averaged over four repetitions of pgaccelinfo) are shown in Figure 12.

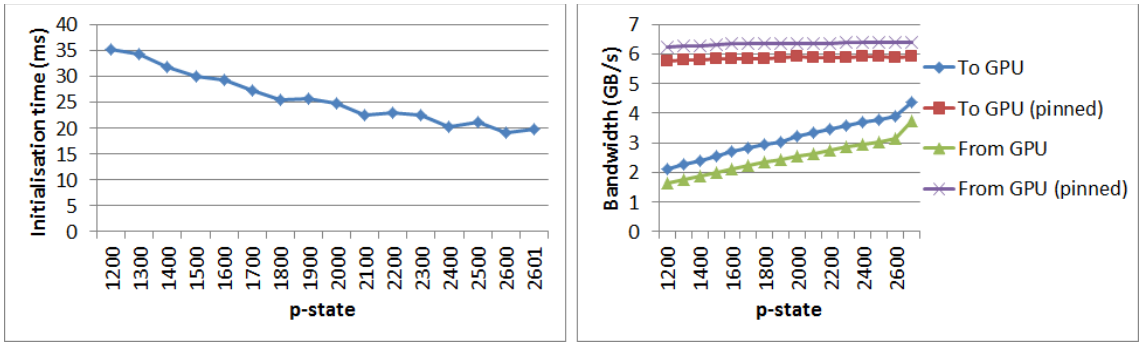

Figure 12. GPU sensitivity to CPU p-state from pgaccelinfo.

As expected, data transfers using pinned memory on the host side are faster than those using unpinned, paged memory, both for transfers from CPU to GPU and also from GPU to CPU. We might also anticipate that unpinned transfers rates might be more sensitive to the p-state that governs CPU performance. The effect is, however, quite marked: whilst pinned transfers only show a 3% reduction in bandwidth across the full range of p-states, the unpinned rates are more than a factor of 2 slower.

It is clear, then, that p-states do have a measurable effect on the performance of GPU-accelerated systems but mainly because of changes in the performance of the host CPU.

4.4.2 Using only the CPU on Graphite nodes

We first consider using only the CPU on Graphite blades. For this, we run the hybrid MPI/OpenMP version of the MG benchmark (with CLASS=C problem size) on four nodes. We vary the number of MPI ranks per node and the number of threads per rank (OMP_NUM_THREADS) to ensure there are always a total of 8 threads per node.

The results are shown in Figure 13. As with the dual CPU Marble nodes, we see that varying the p-state does change the runtime performance as well as the energy consumption, with the most energy efficient runs using an intermediate value of the p-state. OpenMP threading does not change the runtime significantly, except at the highest value of 8 threads per rank. The mean power consumption of the CPU appears insensitive to threading, so the most energy efficient value of OMP_NUM_THREADS is the fastest.

Again, to attempt to understand the trade-off in runtime for energy efficiency, we calculate the payoff time T needed to justify sacrificing runtime. If we assume the same ratio of annual energy costs to system cost $C/S = 5\%$, we again find the payoff time to be 5 years or more, which is probably too long to justify using a lower p-state to save energy. In fact, the situation is perhaps worse here; if the GPU sits idle, it draws much less power than the CPU. The cost of a GPU and a CPU are comparable, so the ratio C/S is probably closer to 2.5%, which lengthens T further.

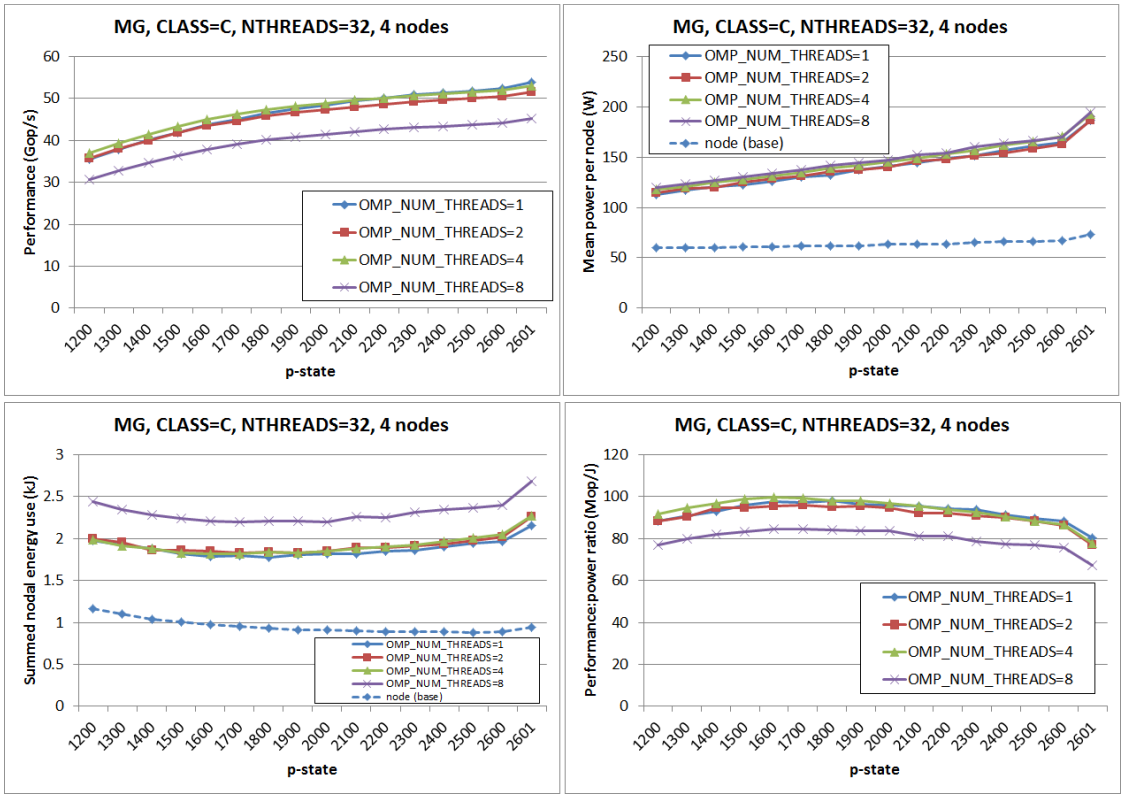

Figure 13. Running the hybrid MPI/OpenMP MG benchmark on the CPUs of Graphite nodes.

4.4.3 Using only the GPU on Graphite blades

Given the host-based execution model, we cannot use just the GPU on a Graphite blade and at least one core of the CPU must be used to control the GPU. By just using the GPU, we here refer to 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 and MPI send/receive buffers. The CPU is still responsible for launching computational kernels to the GPU and for synchronisation.

We ported the MG benchmark to the GPU in this way using the directive-based OpenACC programming model. We then ran this on 4 Graphite nodes using one MPI rank per node.

The results are shown in Figure 14. Once more, we see that lower p-states result in lessened performance. We have already seen that lower p-states lead to slower CPU/GPU data transfers. Profiling the OpenACC code with CrayPAT verifies this effect, but also shows that reduced p-states affect the kernel performance. The difference to the CPU case is that we cannot reduce the energy efficiency by increasing the runtime.

4.4.4 Using both CPU and GPU on Graphite blades

To make best use of accelerated nodes, we would really like to use both CPU and GPU for 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 GROMACS. 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.

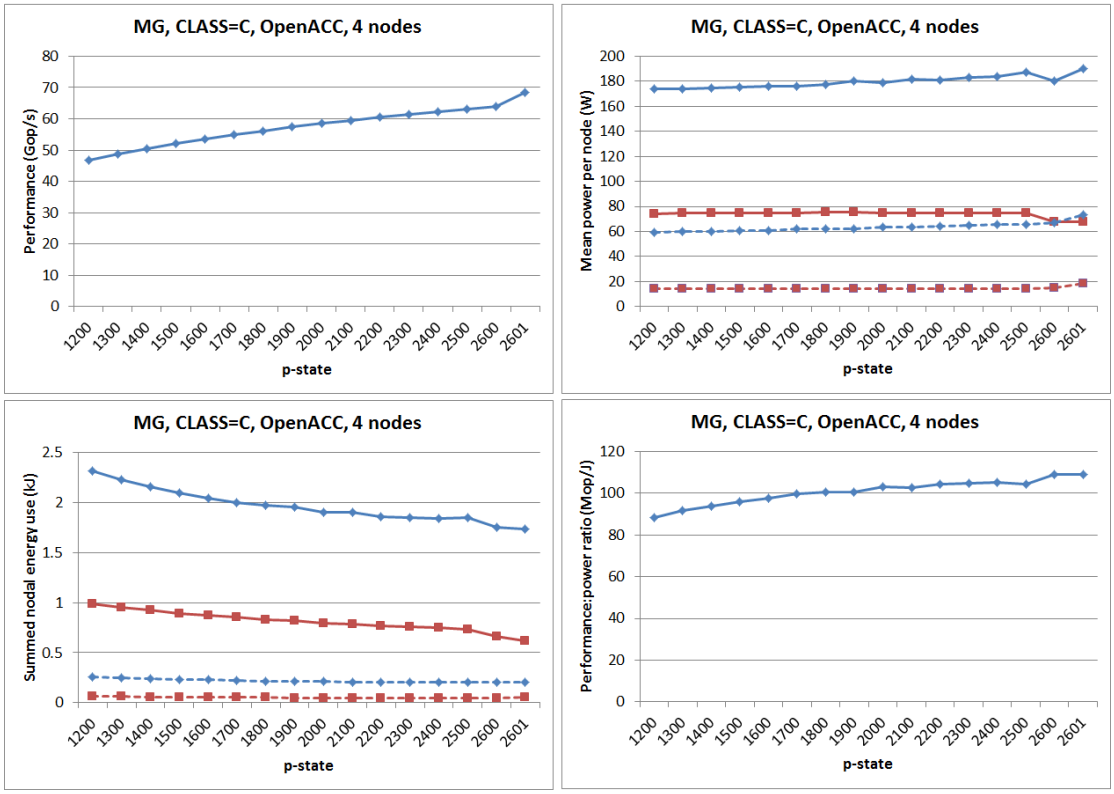

Figure 14. An OpenACC version running on Graphite nodes. Blue diamonds show total node energy or power use and red squares show that due to the accelerator. Dashed lines show the base consumption for an idle system.

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. For this, we need the extended MPMD capabilities described earlier to allow multiple binaries to execute on the same Cray node.

The complication here is that we need the MPI ranks to be load balanced, but (for simplicity) they will probably each calculate local portions of the global problem that are the same size. To use the GPU will require one CPU core to act as "host" so, if we run with one MPI rank per core, we will sacrifice one CPU rank to gain a GPU rank, but all we will gain by this is load imbalance, and we are unlikely to see improvements in energy efficiency.

A better approach is to colocate the GPU rank with one or more hybrid MPI/OpenMP ranks on the CPU. The problem size per rank can then be increased, but the CPU rank execution time can be adjusted by threading.

We can try this using the versions of MG developed so far, again using 4 nodes. We choose a number of MPI ranks between 8 (two per node) and 32 (8 per node) and compile both the MPI/OpenACC and MPI/OpenMP versions of the code. We then run the code with that number of ranks, but arranging that one rank per node is the GPU version of the code and the remaining are the threaded CPU versions. We then run this colocated version of the code with an increasing number of threads per CPU rank (ensuring that we do not have more than one thread on each of the 8 physical CPU cores). For instance, with a total of 16 MPI ranks, we will have 4 ranks per node. Of these, one will be a GPU rank with a dedicated core on the CPU. The remaining three ranks on the node are CPU ranks and we can run these with either 1 or 2 OpenMP threads per rank, using a total of either 4 or 7 cores on the CPU.

Having done this, we take the results for the number of threads per rank which gives the best performance. For 8 ranks total (2 per node), it is 6 (rather than 7) threads per rank. For 16 ranks total, it is two threads per rank. With 32 ranks in total, we are back to flat MPI.

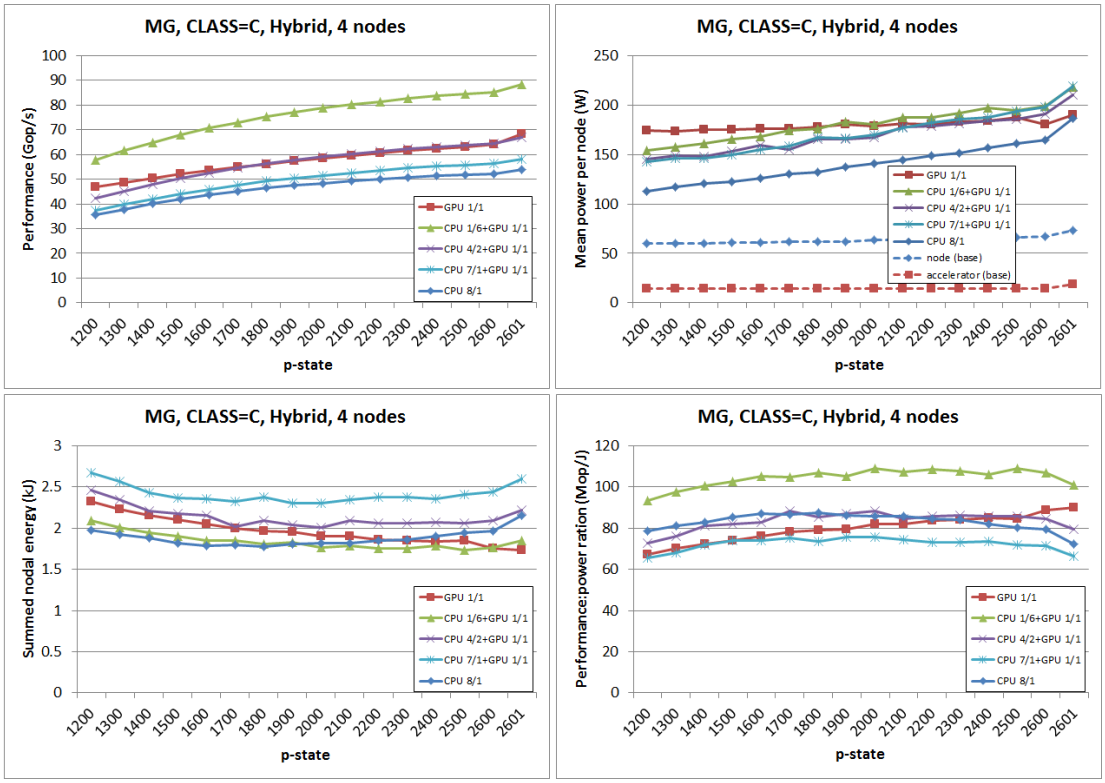

Figure 15. Hybrid CPU/GPU performance of the MG benchmark. Dashed lines show the base consumption for an idle system (blue diamonds for the total node, red squares for the accelerator).

The results for these hybrid configurations are compared with the pure CPU and pure GPU results in Figure 15. The legend here describes the number of ranks per node and the number of threads per rank for the CPU and GPU portions of the application. The latter is always "GPU 1/1". We see the best hybrid performance comes with using 1 CPU rank per node (with 6 threads per rank), which performs 30% better than the pure GPU version and more than 60% better than the pure CPU version. More striking is the fact that this hybrid performance boost did not lead to increased energy consumption by the node.

We argued previously that we did not expect to see much benefit if the CPU part of the application were not threaded (e.g. with OpenMP), as we would merely generate load imbalance by adding a faster GPU rank. We see, however, that this is not the case if we compare the performance of "CPU 7/1 + GPU 1/1" with "CPU 8/1". The likely reason for this is that for a load balanced application, system jitter on each node means that each MPI rank introduces a little delay into the overall runtime. If one rank is much faster, any jitter there will not contribute to the overall runtime. As we might expect, however, it costs a lot more energy to run one of these 8 ranks on the GPU.

In conclusion, then, a good hybrid use of the CPU and GPU can increase application performance significantly for little or no increase in application energy consumption. To do this without significant application restructuring, however, the application must be separately compiled for CPU and GPU targets (potentially from different source codes), and the CPU version should be OpenMP threaded.

4.5 Conclusions of this section

In this section, we have shown how application energy consumption can be measured on Cray XC30 systems. We have studied how both runtime and energy consumption of an application is changed as the CPU clockspeed is varied via the p-state. We have also presented a simple financial model that quantifies whether lower energy consumption is worth the concomitant increase in application runtime, based both on the initial cost and energy consumption of a typical HPC system.

Using the NAS Parallel Benchmark codes, we found no evidence that a hybrid MPI/OpenMP algorithm is more energy efficient than a pure MPI model for small numbers of dual-CPU nodes. On accelerated nodes, the same result was seen when just using the CPU. For higher p-states at least, it was, however, more energy efficient (as well as faster) to run the calculation just on the GPUs of the node. The best performance, however, came when both CPU and GPU were used for the calculation. For this approach to work best, however, the CPU version of the code should be hybrid MPI/OpenMP rather than flat MPI.

If energy prices were to rise by, say, 30%, this balance would already change. Alternatively, if the decision were to procure a long-lifetime system, it would also make reducing performance more attractive from the overall cost perspective.

Most of the results in this section focused on the MG benchmark, and it is tempting to try and use the performance results to compare CPU and GPU performance. To do so would be misleading. Whilst the MG code is stencil-based (which makes it well-suited to GPU architectures), it iterates over a number of different grid resolutions. On the coarsest grids, the local problem size can be very small (typically 4^3=64 grid points, plus uncomputed halos). This is far too small to get good performance from a GPU, which requires a large oversubscription of threads to cores to hide memory latency.

MultiGrid methods are employed as an optimised algorithm to speed overall convergence. For a GPU the algorithm should be revisited, perhaps removing some of the inefficient coarse-graining levels to improve overall performance. Given that current Nvidia Kepler K20x GPUs do not clock down individual cores when idle, removing inefficient small grids is also likely to increase energy efficiency (whichever precise metric is used for this).

This emphasises that not only is algorithm choice important for understanding energy efficiency, the choices are also influenced by the underlying architecture. Different algorithms may have different formal, mathematical speeds of convergence, but this must be balanced against the computational costs, which can vary widely across different architectures.

5 Power consumption of processor and memory

As shown in the previous section, simply switching between the different p states of the processor before executing of the entire application is not (currently) economically rational. We also saw that the performance and energy consumption depends not only on the number of active cores and the frequency, but also on the type of the running applications (compare results for the MG and IS benchmarks; Multi-Grid and Integer Sort, random memory access). We believe that a more flexible strategy for choosing the optimal frequency and the number of cores will lead to more efficient energy consumption than shown above. This would require a lot of work in the application and required more background information about the underlying hardware (i.e. power model) and algorithms (i.e. performance model).

The objective of the work presented in this section is to identify and understand the energy consumption of processors and memory for simple kernel operations. Another important goal is to develop the methodology by which the developers and users could estimate the energy consumption of the different algorithms on different systems with minimal effort and satisfying accuracy. As shown in the section 2.3 the frequency in which the power measurement can be done on the large systems (e.g. Cray XC30) is almost not sufficient to identify the power consumption of the single small algorithms. We believe that a better approach is the colocation of the measured power and the interpolated one, which could be calculated with help of a power model and collected information by the profiling tool (e.g. Score-P).

In this deliverable we present the results of the work, which was done by HLRS in the framework of the Cresta project. More details to this work can be found in Ref. [16].

The methodology for the measurement of the different components of the computer systems are referenced briefly in the deliverable, as long as it is the part of the work, which was done in the framework of the ExaSolvers project that has received funding from the German Research Foundation (DFG) [20]. More details to the measurement methods can be also found in Ref.[16].

5.1 Power consumption of computational node

The electric power (P, in Watts) dissipated in a device in a DC circuit is given by the product of applied voltage (V, in Volts) and the electrical current (I, in Amps). In order to determine the electric power of the processor and memory, we have measured V and I at the power connectors on the motherboard. The high precision shunts and A/D converter used provide high accurate measurements with a tolerance of ±1%. An additional device (based on the precision current transformers) allows us to record the power consumption in an AC circuit with the same A/D converter.

To review the assertion that the processor and the memory consume a substantial part of the energy, we implemented a test bed. We use this to measure the electrical power not only of whole computational node but also of its hardware components. A workstation with an Intel ® Xeon ® Processor E5-2687W was selected for the analysis. The technical description of the workstation is in Figure 16.

Components	Description
Motherboard	Supermicro ® X9SRA Single Socket R (LGA 2011)
Processor	Sandy Bridge E5-2687W (8 cores, 20 M L3-Cache, 3.10 GHz)
Memory	4 x Kingston ® Server Premier 4GB Module - DDR3 1600MHz ECC
Video	Simple graphic card (HD5450, 1GB DDR-3, passive cooling)
Power supply	Antec ® EarthWatts EA-450 Platinum 450W
Hard disc	Toshiba ® DT 01ACA100 (1 TB, 72000 rpm, 32M Buffer)
CPU-Fan	Be Quiet! DARK ROCK 2, 135 mm

Chassis Fans	(x2) 200mm Fans (x1) 120mm Fans
OS	Scientific Linux release 6.3 (Carbon), kernel version 2.6.32

Figure 16 Technical description of the workstation

The hardware components are divided into the six groups *Motherboard*, *CPU+RAM*, *Power supply*, *Hard disk* and *Fans*.

The different computational loads were exerted by three types of tasks:

• *No load*: No load on the workstation, except that produced by the OS (so-called "jitter")

• *Add*: Summing the elements of two arrays and storing the result in a third array

• *Copy files*: Copying a large file from one directory to another

We changed not only the type of the task and its size to activate the different hierarchy levels of memory, but also the frequency of the processor. The processor was set either to the lowest or the highest of the possible frequencies. The power used by the various components of the workstation for different problems is shown in Figure 17. The energy consumption of the processor and memory depends on the number of active cores as well as the frequency. Another factor is the active hierarchy level of memory. As can be seen, the processor together with memory modules are the most significant power consumers. The power consumption of the high performance network components will be considered by us in future work.

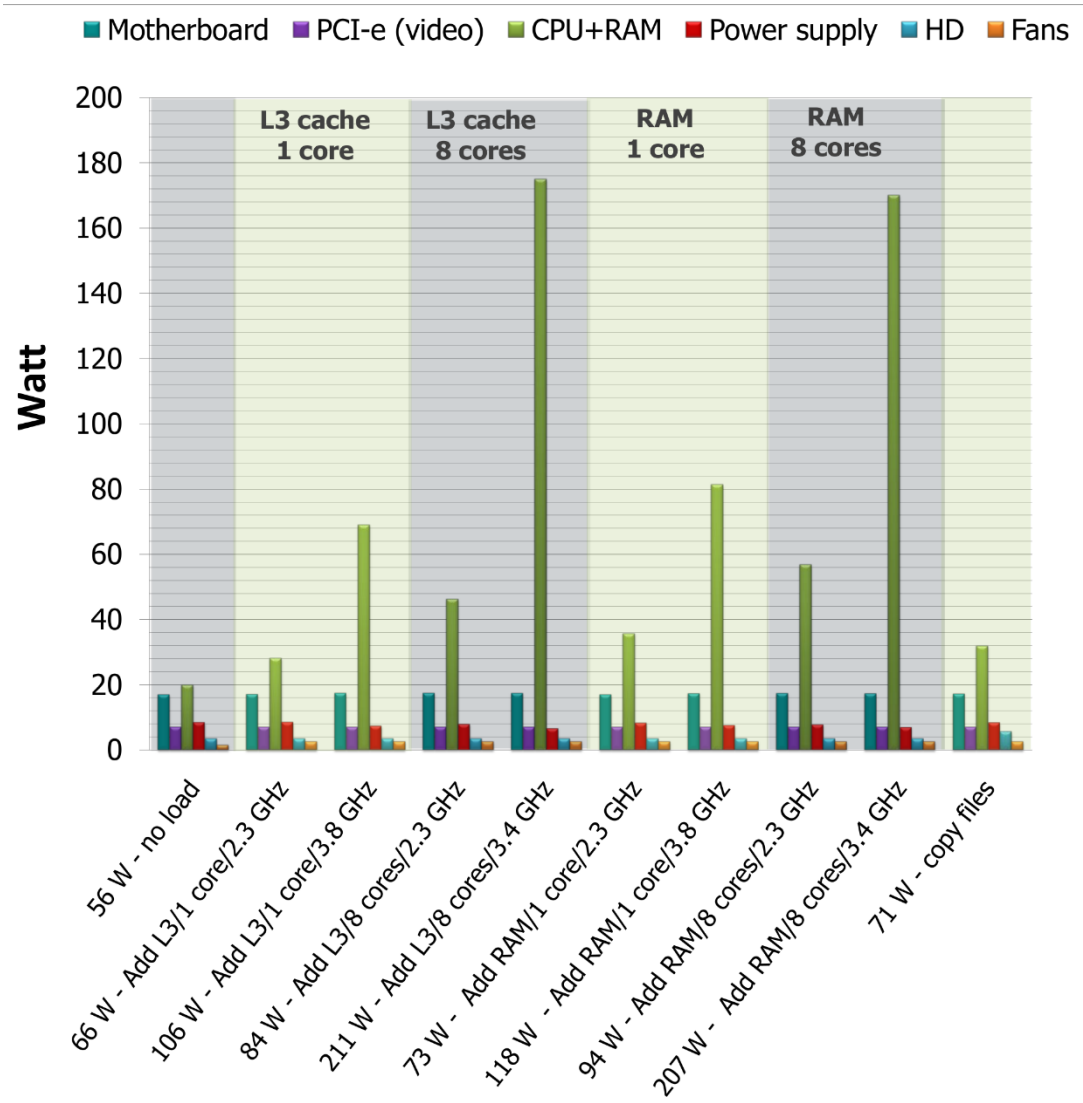

Figure 17 Electric power of the workstation components (see below) under various loads

5.2 Power consumption of the kernel operations

5.2.1 Kernel operations

We address four kernel operations (among many others that we have studied):

• *Add*: Sums of the elements of two arrays and stores the result in a third array:

$$a_i = b_i + c_i;\ 0 \leq i < length$$

• *Dot product*: Computational of the scalar product of two arrays:

$$dot \mathrel{+}= b_i * c_i;\ 0 \leq i < length$$

• *Load AVX*: Load the values from cache or memory to the CPU AVX registers by using the stream instruction mm256_load_pd:

• *Store AVX*: Store the values from CPU AVX registers to cache or memory by using the stream instruction mm256_store_pd:

By varying the size of the arrays, the different hierarchy levels of memory (L1, L2, L3 or RAM) can be activated. The AVX instructions for floating point operations in double precision and loop unrolling were used in the implementation of the kernel operations. The kernel operations were parallelized with OpenMP.

5.2.2 Power approximation

According to the Intel White Paper for the PentiumM® processor [17] that supports Enhanced Intel SpeedStep® Technology, the power consumption of the processor (P) is approximately

$$\boldsymbol{P = CV^2 f} \qquad \textbf{(1)}$$

where *C* is the capacitance, *V* is the actual voltage and *f* is the actual frequency. This formula refers to an older, simpler CPU, and direct use of this formula is now more complicated. A modern processor changes not only its voltage and frequency, but also the capacitance depending on the executable instructions and hardware configuration. In addition, the various components of the processor operate at different frequencies. We thus decided to check on the suitability of a new formula for the approximation of its electric power depending on the frequency for the set of simple kernel operations:

$$\boldsymbol{Power(f)_{lp} = a_{0lp} + a_{3lp} * f^{\rho}} \qquad \textbf{(2)}$$

Where the constants a_{0lp} and a_{3lp} are to calculate for the different hierarchy levels of memory (index l), number of active cores (index p) and selected kernel operation. The constants a_{0lp} and a_{3lp} are greater than zero and the exponent ρ is greater than one. The constant a_0 should be equal to the consumption of the processor's components, which are independent of the variable frequency. Note, the calculations at the different hierarchy levels of memory may well affect both constants. The term $a_{3lp} * f^{\rho}$ reflects the changes in the power consumption depending on the actual frequency. The absolute error ε_{abs} is the sum of the two maximum differences (negative and positive) between measured and approximated power over all cores and four levels of the memory hierarchy.

5.2.3 Power of kernel operations

The best approximation (within the meaning of the minimal absolute error ε_{abs}) for the kernel *Add*, can be achieved with the exponent ρ=2.42. Figure 18 shows the measured and approximated power of processor and memory for all hierarchy levels of memory, different numbers of active cores and frequencies.

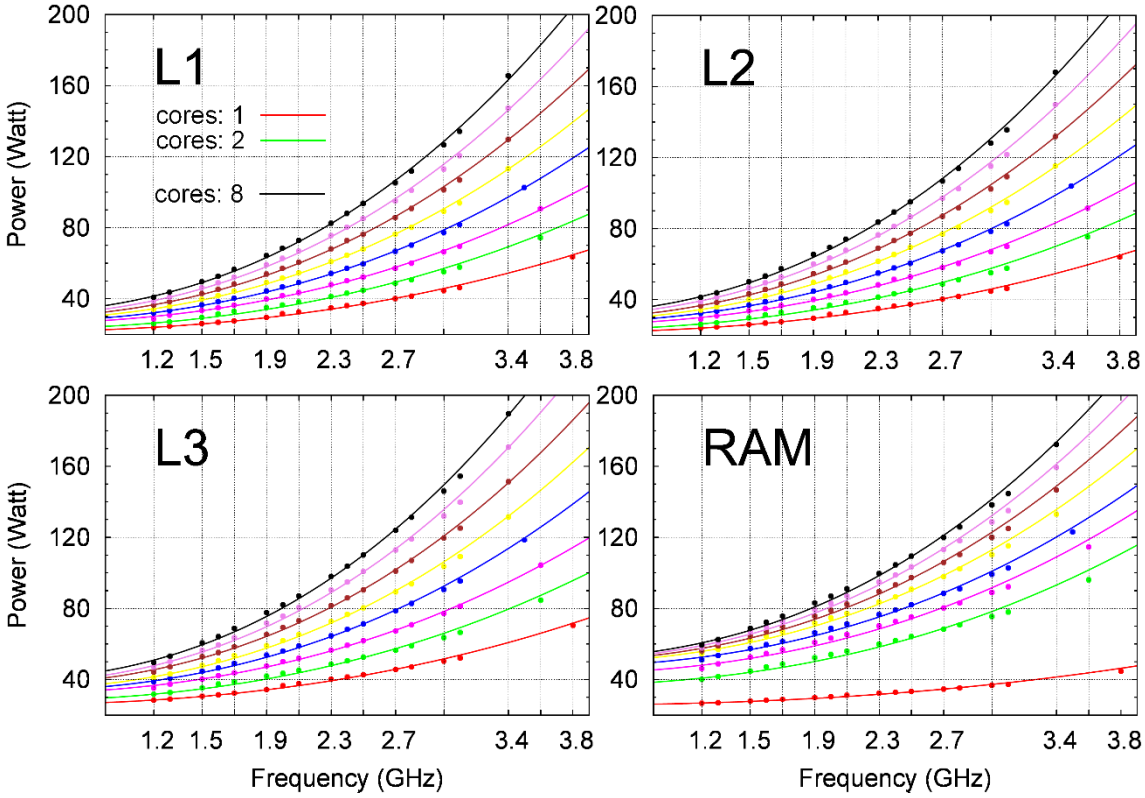

Figure 18 Electric power of kernel operation Add on the Intel processor E5-2687W, when the data is in L1, L2, L3 caches and in RAM

The measured values are marked with the dots. The curves shows the approximated power according to the formula $Power(f)_{lp} = a_{0lp} + a_{3lp} * f^{\rho}$ with ρ =2.42 for one (red) up to eight (black) cores. The other curves colors can be easily assigned to the number of active cores. The power consumption increases with the increase of the frequency and number of active cores. For some l and p and the kernel operations *Add* and *Load* the constants a_0 and a_3 are listed in Table 1.[1]

Cores	Mem	Add	α_0	α_3	ρ	$\tilde{\varepsilon_{abs}}$	Load	α_0	α_3	ρ	$\tilde{\varepsilon_{abs}}$
1	L1	-	21.3	1.7	2.42	5.23	-	21.3	1.4	2.51	6.92
8	L1	-	30.8	6.8	2.42	5.23	-	29.9	5.1	2.51	6.92
1	L2	-	21.3	1.7	2.42	5.30	-	21.4	1.5	2.51	7.25
8	L2	-	30.8	7.0	2.42	5.30	-	30.5	5.6	2.51	7.25
1	L3	-	25.6	1.8	2.42	6.47	-	24.2	1.46	2.51	7.13
8	L3	-	38.8	7.8	2.42	6.47	-	38.0	6.2	2.51	7.13
1	RAM	-	25.5	0.8	2.42	7.89	-	25.2	0.62	2.51	9.69
8	RAM	-	50.8	3.4	2.42	7.89	-	44.1	6.0	2.51	9.69

Table 1 Power approximation parameters a_0 and a_3 for several cases

As expected, the constant a_0 is about the same for both kernels. The difference is only considerably if the data are in RAM. Because the throughput of instructions of kernel *Load* is higher, the power consumption increases faster. This fact is indicated by ρ.

[1] The $\tilde{\varepsilon_{abs}}$ indicates the absolute error only over a corresponding memory level (for all i and p; l is fixed) in contrast to the ε_{abs} that indicates the error over all memory levels: $\varepsilon_{abs} \geq \tilde{\varepsilon_{abs}}$ (for all i , l and p).

5.2.4 Performance of kernel operations

In this section we briefly discuss a performance approximation model which fits the data well and is more suitable for our purpose than models described, for example, in [18] and [19].

We define the performance ("Elements per second [10^9]") as the rate calculation of array elements, measured in units of 10^9 per second. The performance of the cache increases generally linearly with the frequency. The linear approximation of the performance as function of frequency shows good approximation if the data is held in cache:

$$\boldsymbol{Perf(f)_{lp}} = \boldsymbol{\gamma_{olp}} + \boldsymbol{\gamma_{1lp}} * \boldsymbol{f} \qquad \textbf{(3)}$$

But if the data are stored in RAM, the error is too large. The asymmetrical utilization of memory channels relative to the number of cores contributes additional complexity to the problem, and existing models do not include this accurately. The roofline model [5] provides realistic expectations of performance that can be achieved with regard to the memory bandwidth limitation and arithmetic instructions throughput. But the above mentioned aspects are not considered sufficiently accurate. The same can be said about the ECM performance model described in [6]. The formula we select for the performance and its important properties are as follows:

$$\boldsymbol{Perf(f)_{lp}} = \frac{\boldsymbol{\gamma_{max\,lp}}}{\boldsymbol{\gamma_{lp}} + \frac{\boldsymbol{1-\gamma_{lp}}}{\boldsymbol{f}\boldsymbol{\gamma_{lp}}}} \qquad \textbf{(4)}$$

$$\boldsymbol{\lim_{f\to 0} Perf(f) = 0} \qquad \textbf{(5)}$$

$$\boldsymbol{\lim_{f\to\infty} Perf(f) = \frac{\gamma_{max}}{\gamma};\ \gamma \neq 0} \qquad \textbf{(6)}$$

Figure 19 shows the performance of the kernel operation *Add* if the data are in RAM. For this diagram we used the metric virtual CPU frequency instead of CPU frequency. The virtual frequency depends on the number of active cores and their frequency. For example if two cores are used for the calculation and each runs with 1.2 GHz, the virtual frequency is equal to 2.4 GHz. The diagram shows that the maximal bandwidth is achieved with five cores. However, the difference to the performance achieved with three and six core is within the measurement accuracy.

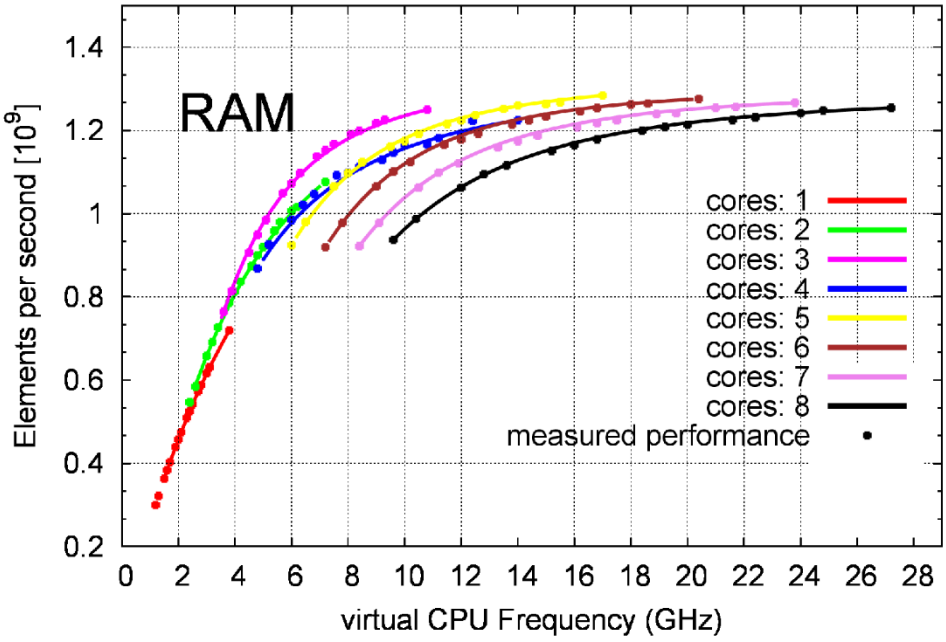

Figure 19 Performance approximation of operation Add on the Intel processor E5-2687W according to the formula, if the data are in RAM

5.3 Energy consumption of kernel operations

Once we have calculated the power and the performance of the kernel operations, we can calculate the dependence between energy consumption of the processor and memory modules and the achieved performance. Note the performance increases due to the increase of frequency.

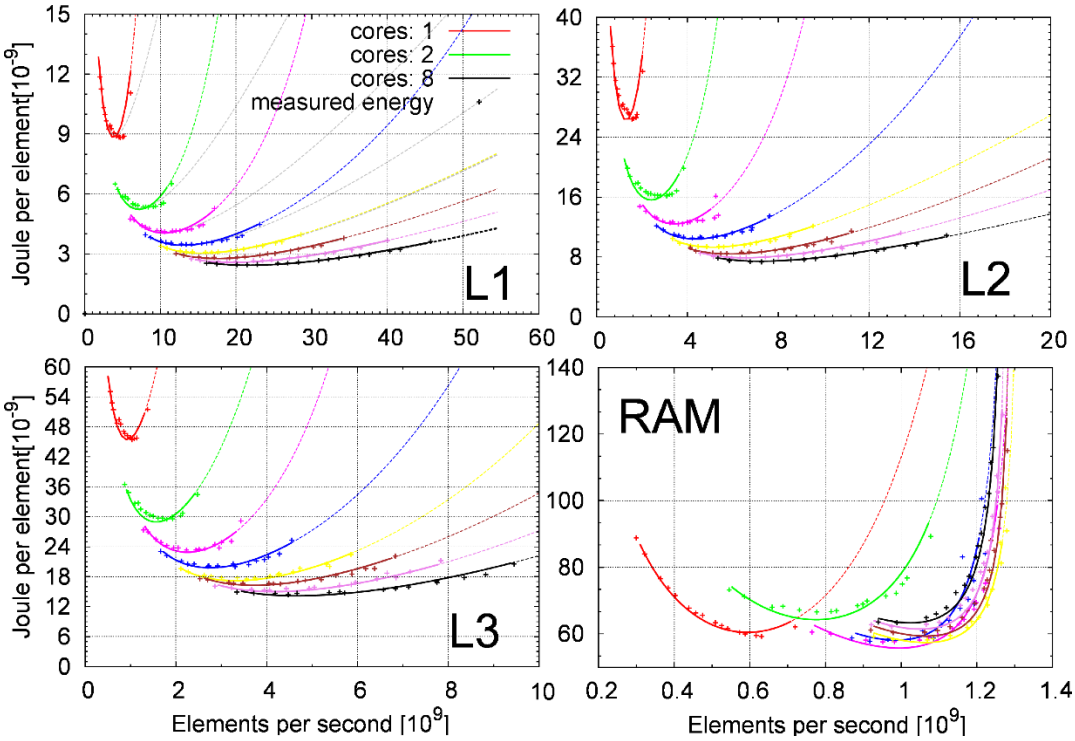

Figure 20 Curves shows how many nanojoules are consumed by the processor and memory modules for the calculation of one element a[i]=b[i]+c[i] with the corresponding performance.

Figure 20 shows the energy consumption depending on the performance for all hierarchy levels of memory. The axes are scaled differently. What is remarkable is a big difference between different cases:

• If the memory is not active there are big advantages of increasing of frequency and activation of more cores.

• If the memory is active, there are no apparent advantages of increasing of frequency to the highest possible. The performance increases very slow and the energy consumption rises very quickly. The next significant difference is that the calculation on all eight cores is not optimal, neither for the performance nor for the consumption of energy.

Figure 21 shows the consumption of operation *Add* with one and eight active cores for all hierarchy levels of memory. The X and Y axis are in base 2 logarithmic scale. The difference between the left and right diagrams is that the consumption of the rest of the system was considered by adding of 30W to the constant term a_0. In doing so, not only was the energy amount increased for the calculation of one element, but also the steepness of the curves was changed. The minimum values (marked with vertical short lines) of the approximation were shifted slightly to the right. This means that the optimum frequency (within the meaning of the energy) became higher.

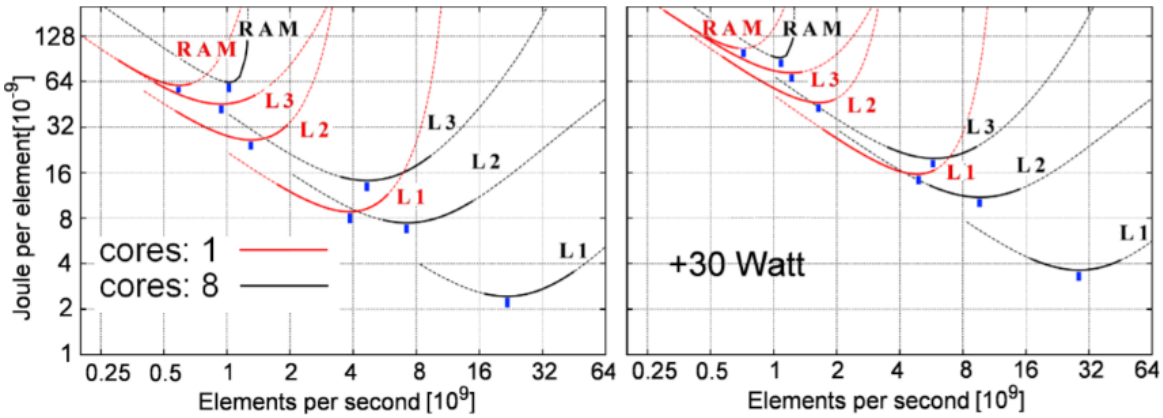

Figure 21 The left diagram shows the energy consumption in nanojoules of the processor and memory modules for computing of one element of the operation Add. The right diagram shows the energy consumption of the processor, memory modules and the rest of the system (+30 Watt).

5.4 Comparing kernel operations

Now we can compare the energy consumption of four different kernels that are described in section 5.2.1. Figure 22 shows how many nanoJoules are needed to carry out these operations. The most expensive and slowest is the operation *Add*. The other operations consume less energy and are faster. In the case of the calculation in level 3 cache, the operations *Store* and *Load* don't differ greatly. In case of the calculation in memory, however, the operation *Store* needs much more energy and is slow; before the data can be stored it must be read from the memory. The consequence is that the store access requires two expensive streams, from and to the memory. In case of the calculation in memory the different kernel operations have different benefits from the increasing of the frequency. While the optimum frequency of *Add* is very low, the optimum frequency of the others operation is higher. It shows that different algorithms have their own optimum frequency.

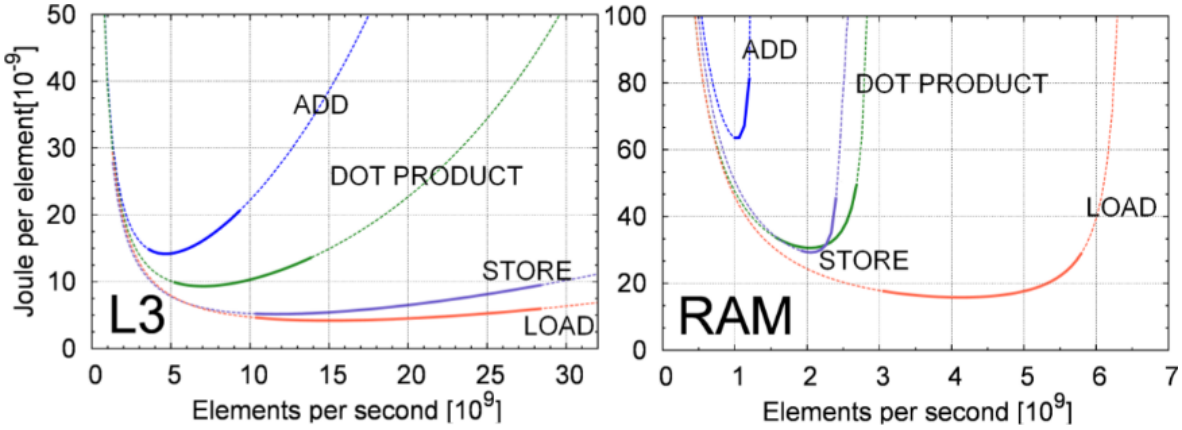

Figure 22 Energy consumption in nanojoules of kernel operations Add, Dot product, Load, Store on the Intel processor E5-2687W, when the data are in L3 caches and in RAM and all eight cores are active.

5.5 Conclusion and Outlook

As shown in the previous sections the power model $\boldsymbol{Power(f)_{lp} = a_{0lp} + a_{3lp} * f^{\rho}}$ can predict the power consumption of the simple kernel operations with high accuracy. Also we demonstrate that a more flexible strategy for choosing the optimal frequency and the number of cores leads in many cases to a larger reduction in the energy consumption than was seen in Section 4. The kernel operations are very low-level and simple. In further studies, we will consider more complex examples and extend the power and performance models as needed.

It is clear that dynamic switching of processor frequency can lead to large reduction in energy consumption, and this could therefore be one of the key features needed when developing exascale architectures. Most current architectures do not offer sufficient control of processor frequency during execution of individual applications. Whilst this control could be utilised directly by the application programmer, the results of this section indicate that the optimal frequency choice depends on different low-level operations. It may therefore be more practical for the variation to be controlled within the OS or the underlying hardware. Both approaches would require the internal use of performance models to choose the correct frequencies depending on the operations, and the work in this Section is clearly a first step in this direction.

6 Power consumption on a low-power architecture

In this section, we take a more speculative view of how an energy-efficient exascale processor might look, and investigate the performance and energy usage of the standard set of benchmarks used in Section 4 on hardware that is more common in the embedded, mobile world than in HPC. This processor is extremely similar to that being used within BSC's Tibidabo exascale prototype machine as part of the EU Mont-Blanc project.

6.1 Low-power hardware

To measure power consumption of HPC applications on an example of a low-power architecture, CRESTA was given access to an ARM-based ODROID XU+E system via the EC FP7-funded Adept[2] project. The ODROID system is based on a Samsung Exynos 5 Octa System-on-Chip (SoC) with a "big.LITTLE" architecture that focuses on power usage optimisation by combining a high-performance CPU with a simpler high-efficiency CPU. In the case of the ODROID system used in this work the big CPU is a quad-core ARM Cortex A15 running at 1.6 GHz. The little CPU is a 1.2 GHz quad-core ARM Cortex A7, a much simple in-order processor. Each SoC has 2GB of 32-bit dual-channel 800 MHz LPDDR3 memory. The disk is a 16GB Class 10 MicroSD card. The SoC also has a GPU on board, however this is only programmable with OpenCL 1.0 and was not used for the work presented here.

The ODROID XU+E has built-in power monitoring through four separate current sensors. These measure the power consumption of the A15, the A7, the GPU and the memory in real time. Figure 23 shows the detailed block diagram for the ODROID XU+E system.

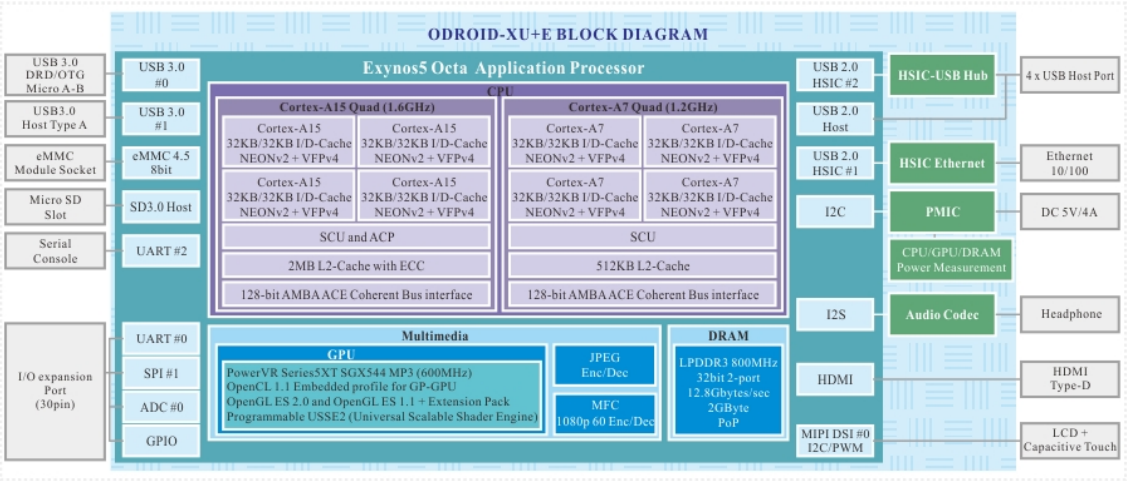

Figure 23: ODROID XU+E block diagram.

6.2 Power measurements

The goal of this part of the work was to be able to quantify the performance versus power usage trade-off when using low-power hardware. We use the same five kernel benchmarks from the NAS Parallel Benchmark suite as were used in Section 4: CG, FT, MG, IS and EP. Due to the small amount of memory on the ODROID system, we only used problem size class A.

6.2.1 Benchmark and system setup

Both optimised (-O3) and unoptimised (-O0) versions of the benchmarks were compiled using GCC 4.7.3 and OpenMPI 1.6.4. All benchmarks were run using 4 MPI processes and repeated 10 times. The numbers reported here are the average values

[2] www.adept-project.eu

for those 10 runs. The energy usage was calculated by multiplying the average power consumption by the benchmark's runtime.

By default the operating system will choose which CPU to use in order to optimise performance and power usage. This is typical of embedded systems with heterogeneous workloads. For HPC workloads, however, this most likely results in only the A15 CPU being used, although this behaviour is not predictable. In order to fix the CPU usage, we set the CPU governor either to "performance" or "powersave". In "performance" mode the clock speed is set to 1.6 GHz – this forces sole use of the A15 CPU. In "powersave" mode, on the other hand, the clock speed is reduced to 250 MHz, resulting in only the A7 CPU being active. The energy usage reported here is always the sum of the energy usage for A15, A7 and memory.

6.2.2 CG benchmark

The Class A problem for the CG benchmark defines a sparse matrix of 14,000 rows with 11 non-zero elements and runs for 15 iterations. Figure 24 shows the wall time and energy usage for the four different run configurations of this benchmark. As expected, turning on optimisation improves the run time for both CPUs, and the A7 CPU is between 5 and 8 times slower than the more powerful A15 CPU. Optimisation improves the run time on the A7 by a factor of 2 and reduces energy usage by 80%; on the A15 the run time improvement is only 40%, however interestingly energy usage is reduced by ~65% (see Table 2).

In the case of the CG benchmark, the sweet spot for combined performance and power efficiency is to use the A15 CPU with optimised code. Although the energy usage is more than 3 times that of the A7 CPU in this case, the run time is more than 5 times better.

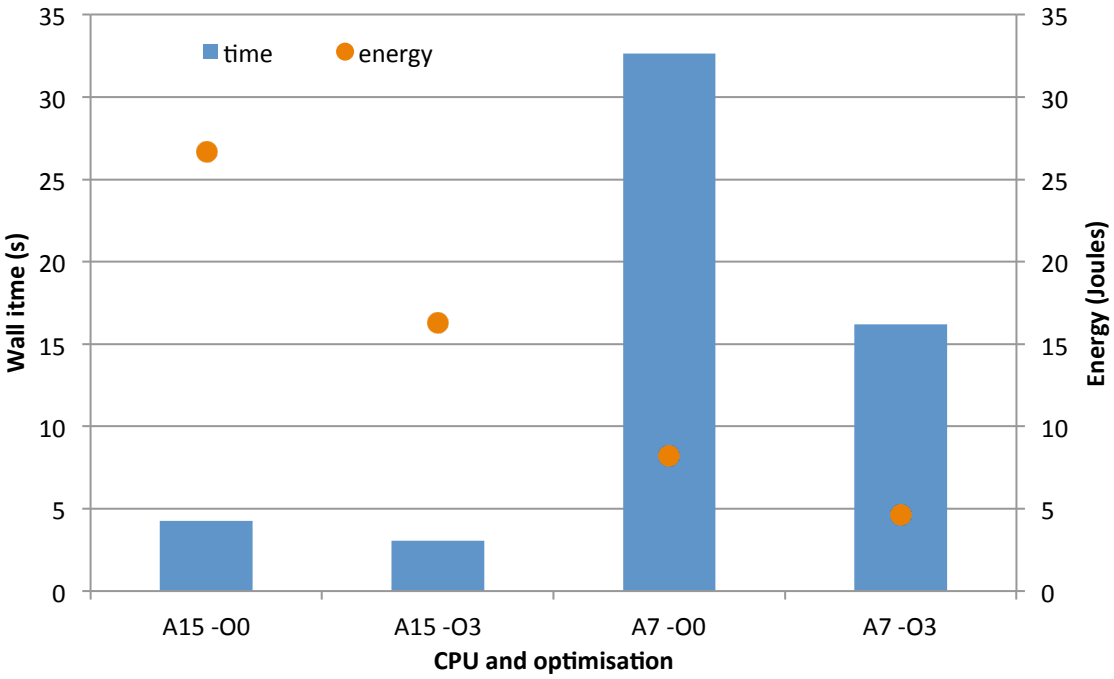

Figure 24: Comparison of wall time and energy usage for the Class A CG benchmark.

	"-O3" A7 v A15	"-O0" A7 v A15	"-O0" A15 v "-O3" A15	"-O0" A7 v "-O3" A7
Slowdown	5.29	7.64	1.39	2.01
Energy reduction	3.52	3.26	0.61	0.56

Table 2: Slowdown and energy reduction factors for CG when comparing different run configurations.

6.2.3 FT benchmark

The Class A problem size for the FT benchmark, which solves a 3D partial differential equation, uses a 256 x 256 x 128 grid and runs for 6 iterations. Figure 25 shows the wall time and energy usage for the different run configurations, and Table 3 shows the slowdown and energy reduction factors. As with the previous benchmark, using the high-performance CPU with optimised code gives the best overall run time. By comparison, the optimised benchmark runs 4.5 times slower on the A7 CPU and uses just under 4 times less energy. This again makes the optimised run on the A15 CPU the most efficient in terms of both performance and energy use, however the optimised A7 run is not far behind and reduces total energy considerably.

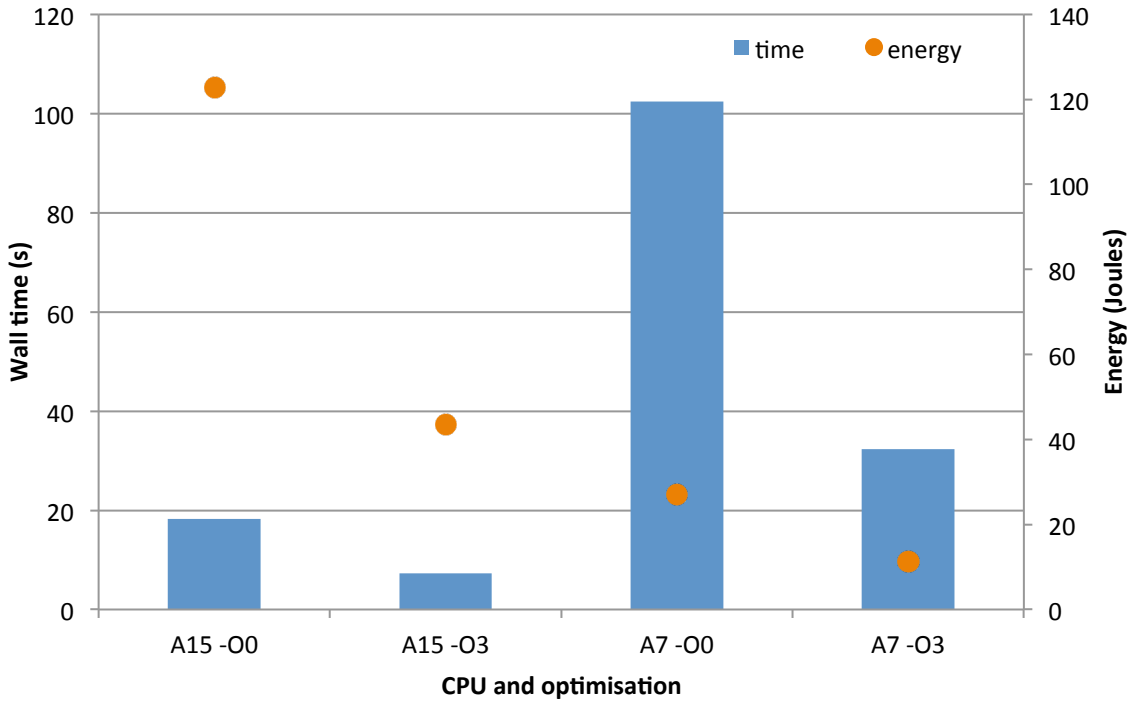

Figure 25: Comparison of wall time and energy usage for the Class A FT benchmark.

	"-O3" A7 v A15	"-O0" A7 v A15	"-O0" A15 v "-O3" A15	"-O0" A7 v "-O3" A7
Slowdown	4.47	5.61	2.52	3.16
Energy reduction	3.86	4.54	0.36	0.42

Table 3: Slowdown and energy reduction factors for FT when comparing different run configurations.

6.2.4 MG benchmark

The MG benchmark represents a simplified multigrid kernel that uses a 256^3 grid over 4 iterations for its Class A problem. Figure 26 and Table 4 show the results of this benchmark and looking at the graph it becomes immediately clear that the code benefits significantly from optimisation on the A7 CPU: run times is improved by almost a factor of 6, while for the A15 this is closer to a factor of 2.5. In fact, when comparing to the optimised run on the A15, the A7 has the upper hand in terms of combined performance and energy efficiency for this particular benchmark. The optimised benchmark takes ~25s to complete on the A7, consuming only 7.6J; on the A15 it completes in 6.2s, but consuming 31.7J.

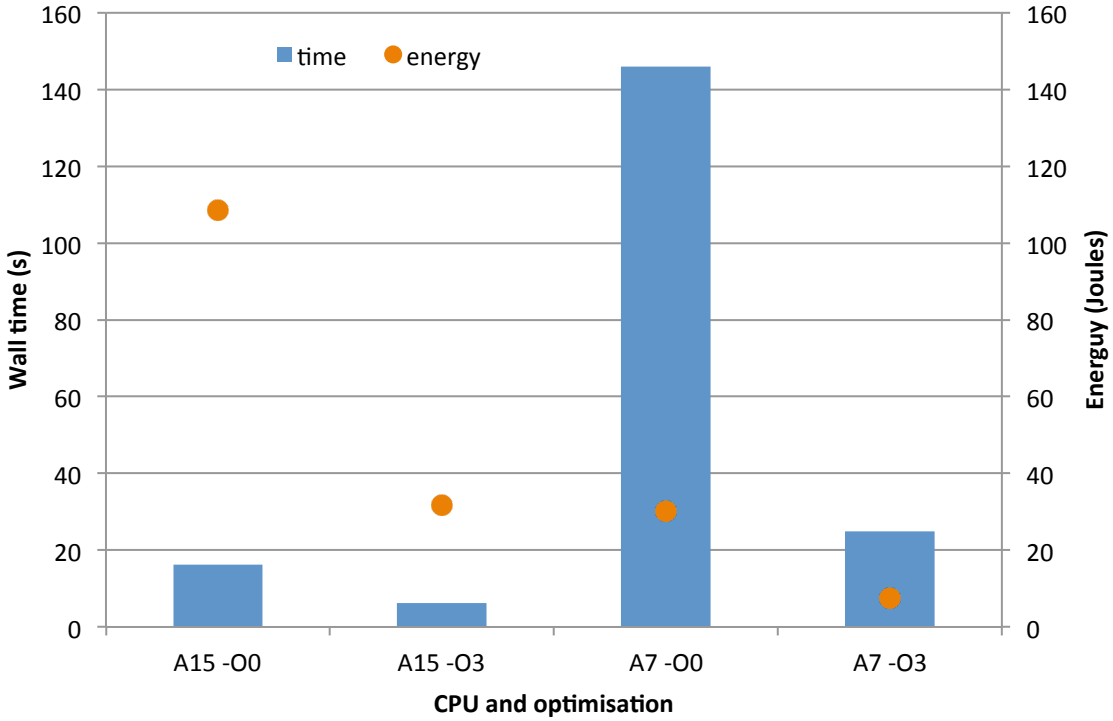

Figure 26: Comparison of wall time and energy usage for the Class A MG benchmark.

	"-O3" A7 v A15	"-O0" A7 v A15	"-O0" A15 v "-O3" A15	"-O0" A7 v "-O3" A7
Slowdown	4.02	9.09	2.60	5.86
Energy reduction	4.15	3.61	0.29	0.25

Table 4: Slowdown and energy reduction factors for MG when comparing different run configurations.

6.2.5 IS benchmark

The IS benchmark performs a large integer sort and tests both the speed of integer computation and communication. For the Class A problem size, the IS benchmark sorts 2^{23} keys in parallel and maximum key value of 2^{19}.

Figure 27 and Table 5 shows the performance of the IS benchmark on the two ARM CPUs. It can be seen that the optimised code on the simple A7 core provides by far the best combined performance and power efficiency. The runtime may be slower by almost a factor of 3 as compare to the A15, but the energy usage is reduced by a factor of 4.75.

The behaviour is not unexpected given the different CPU architectures: the A15 is a high-performance CPU with a focus to providing optimal float-point performance; the A7 on the other hand is a much simpler, highly energy-efficient CPU. The IS benchmark only performs integer operations and the added complexity of the A15 CPU increases the energy usage of each operation. The A7 on the other hand can handle these simple operations with almost the same speed as the A15, but with a lower energy cost per operation.

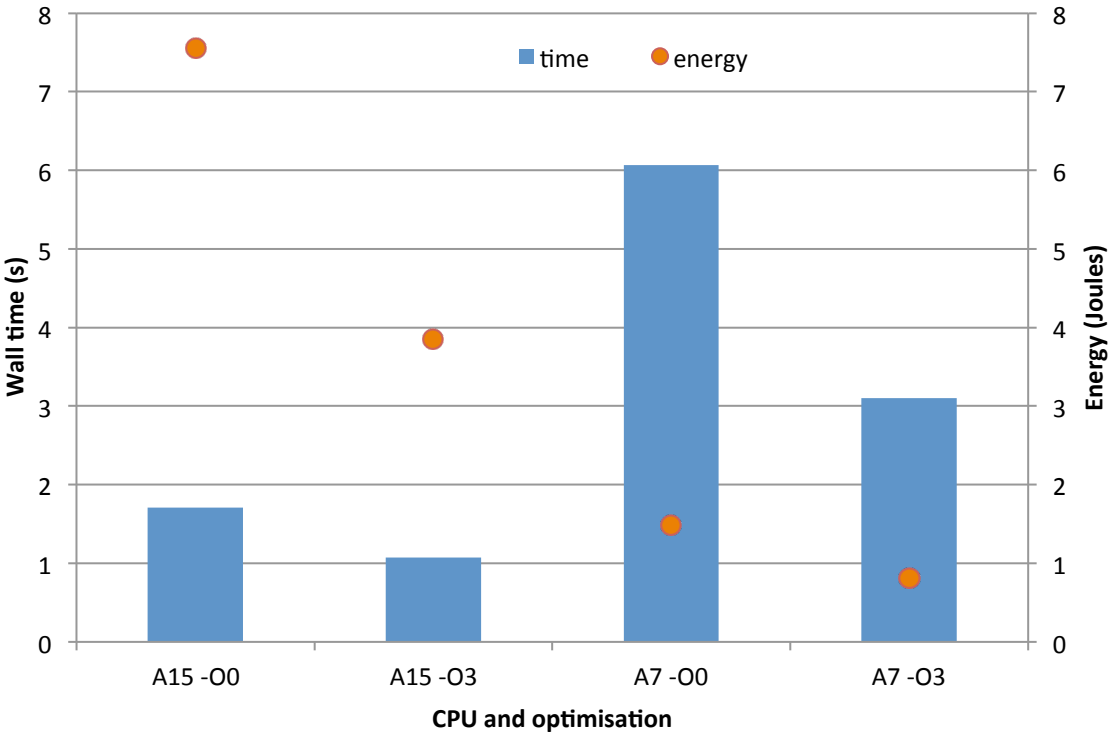

Figure 27: Comparison of wall time and energy usage for the Class A IS benchmark.

	"-O3" A7 v A15	"-O0" A7 v A15	"-O0" A15 v "-O3" A15	"-O0" A7 v "-O3" A7
Slowdown	2.89	3.55	1.60	1.96
Energy reduction	4.75	5.08	0.51	0.55

Table 5: Slowdown and energy reduction factors for IS when comparing different run configurations.

6.2.6 EP benchmark

This final benchmark is an "embarrassingly parallel" kernel that generates and tabulates 2^{28} pairs of random numbers for the Class A problem. It is focused on measuring floating-point performance.

Figure 28 and Table 6 show the runtimes and energy usage for the EP benchmark. There is a significant performance difference between the two types of CPU, which is largely due to the superior float-point performance the A15 can offer. However the problem is again not complex enough for the A15 CPU to outpace the A7 in energy usage as well. In this case, where the trade-off between speed and energy usage are almost even, the choice of hardware very much depends on whether time-to-solution is more important than total energy usage, and vice versa.

It is also interesting to observe that enabling compiler optimisation for this type of embarrassingly parallel benchmark does not give the same levels of improvement (in either runtime or energy usage) as seen in the previous benchmarks.

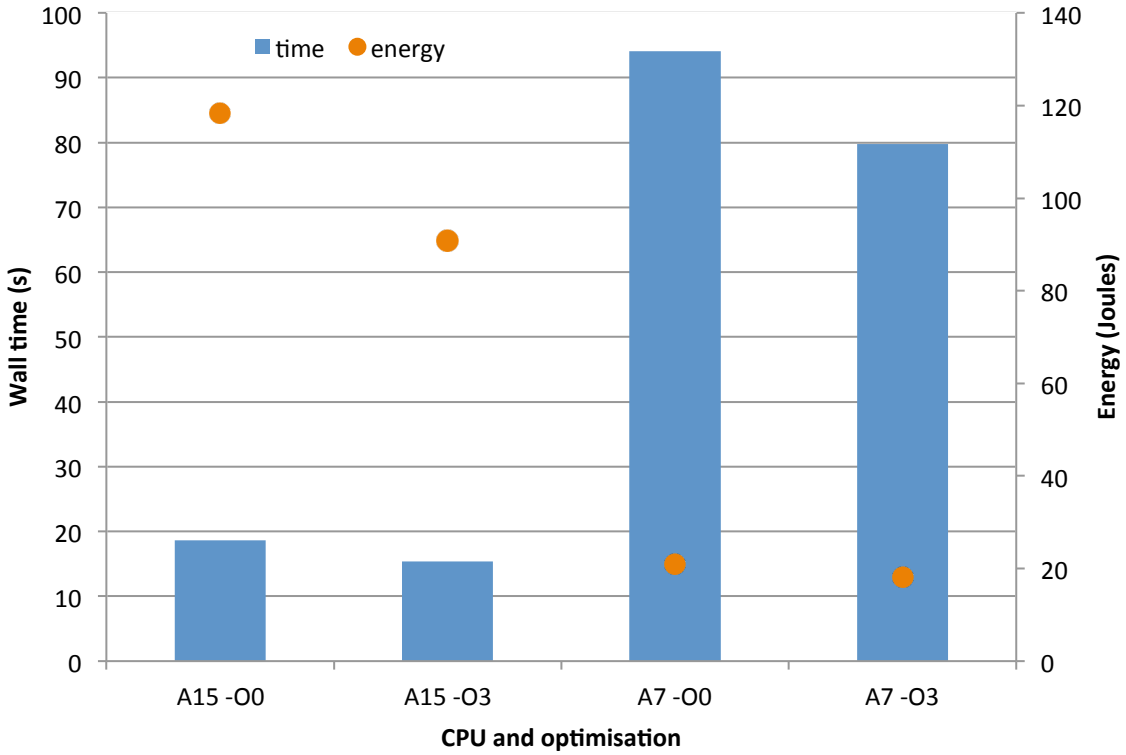

Figure 28: Comparison of wall time and energy usage for the Class A EP benchmark.

	"-O3" A7 v A15	"-O0" A7 v A15	"-O0" A15 v "-O3" A15	"-O0" A7 v "-O3" A7
Slowdown	5.19	5.06	1.21	1.18
Energy reduction	5.02	5.64	0.77	0.86

Table 6: Slowdown and energy reduction factors for EP when comparing different run configurations.

6.3 Conclusions

The pure clock-speed ratio between the two CPU types is 1.6 GHz / 250 MHz = 6.4, which can be taken as a base value for the difference in run time for compute-bound codes. This does of course not take into account the different CPU architectures (i.e. A15 multi-issue pipeline with out-of-order execution, A7 simple in-order execution), but offers a guideline nonetheless.

The CG and EP benchmarks are closest to this number in terms of runtime, and for the IS benchmark the difference is closer to a factor of 3. For three out of the five benchmarks, using the simpler, more energy-efficient A7 CPU is a viable option if the goal is to optimise for energy usage rather than time to solution. This is especially true in the case of the IS benchmark, where the capabilities of the A15 CPU effectively waste energy without being able to increase the performance to a sufficient level.

We can also quantify the energy efficiency of the different executions by looking at the number of operations that are performed per Joule. Table 7 shows the efficiency of the CG, MG and FT benchmarks on both A15 and A7 in terms of millions of operations (Mop) per Joule of energy spent. It can be seen from the table that using optimised code on the A7 processor increases the number of operations that can be executed per Joule by a factor of between 3.5 (for CG) to just over 4 (for MG). It can also be observed that using unoptimised code significantly decreases the number of operations than can be executed per Joule.

	A15 –O3	A15 –O0	A7 –O3	A7 –O0
CG	92.107 Mop/J	56.172 Mop/J	323.626 Mop/J	182.557 Mop/J
MG	184.635 Mop/J	51.437 Mop/J	745.623 Mop/J	174.483 Mop/J
FT	163.596 Mop/J	58.157 Mop/J	629.182 Mop/J	264.031 Mop/J

Table 7: Comparison of efficiency in terms of millions of operations (Mop) per Joule of energy for the CG, MG and FT benchmarks on each of the two ARM CPUs.

7 Summary and conclusions

In this Deliverable we have explored how power and energy measurements can be made on a variety of computing architectures, with a view to understanding how future exascale systems and applications can meet the expected and stringent power and energy constraints for such systems. In particular, the US DOE has put a limit of 20MW power consumption for such a system. Even allowing for a typical application to yield only 10% of the peak performance of the system, this still requires a performance:power ratio of $(10^{18}x10^{-6})/(20x10^{6})x10\%=5000$ Mflops/W.

The results in this deliverable show that current benchmark performance is still well below this target, both on server-class CPUs or GPUs in a system like the Cray XC30 or on "lower-power" CPUs aimed at the mobile and embedded markets. As we might expect, the lowest-power A7 CPU does best in this regard, yielding around 750 Mop/J (which is comparable to Mflops/W). Clearly much work is needed at the architectural level to improve this. We must also bear in mind that processor systems aimed at the mobile and embedded markets often do not include features that would be considered essential in an HPC environment, like efficient double precision arithmetic and ECC error correction in the memory systems. Adding such features is likely to reduce the power efficiency of such systems. The A15 processor was much closer in the performance:power ratio to the Sandybridge and Ivybridge processors included in the Cray XC30. Again, the A15 is a simplified processor compared to the Intel Xeons and the relative closeness of the performance:power ratio demonstrates that modest simplification of current processor designs is not going to be sufficient to reach the exascale requirements. A more radical approach is probably required.

The results of measuring the power consumption of low-level operations (Section 5) is informative in this regard. Having a fine-grained control of processor frequency can produce marked increases in energy and power efficiency; for instance, the power for add and load operations increased as something between the square and the cube of the frequency. Clearly this is an area which should be explored further for exascale systems. Given the difference in optimal frequencies for very simple, low-level operations, it seems likely that the clockspeed would need to be governed by the systemware (e.g. the OS or low-level runtime system) or in the hardware itself. Developer-inserted control, for instance through API calls inserted in the application, is likely to be too coarse-grained to give the same benefits. Performance models, such as those introduced in this Deliverable, will be very important in implementing this approach.

We have also explored how different algorithms (which we interpret here to include compiler and runtime choices as well as different programming models and source code) affect the performance and energy consumption of applications. Compiler optimisation is important: whilst the CPU works harder with optimised code, this increased power consumption is more than compensated by the decreased runtime. We probably do not expect this to change in the future. Even with an increased ability to idle ("darken") unused portions of a CPU, there will always be a certain, base power associated with a node and simply increasing the runtime through broad-spectrum suppression of all optimisation is likely to result in an increased base energy consumption that swamps the desired effect.

What we have not explored here is whether reducing the level of specific optimisation features (rather than just the global optimisation level) can benefit particular applications. This should be revisited as we approach the exascale, particularly in the light of the results we present here on the power consumption of different low-level operations, which may favour a more fine-grained approach to optimisation.

The results from the accelerated XC30 nodes are interesting. Accelerators like GPUs can improve performance and decrease power consumption. Current GPUs, however, require a host CPU processor and, even if this is idle, its power consumption is significant. Our best performance came from using both (in a balanced manner) for the

same calculation. This is not easy, from an application standpoint. The other point that must be borne in mind is that the MultiGrid algorithm suited to the CPU does not work well on the GPU. This example makes it clear that algorithmic work must be done in parallel with exascale hardware design, building on the co-design model used in CRESTA.

Finally, if application developers are to contribute to this energy-efficient co-design process, they require an informed view of the integrated application performance that includes information on the energy and power consumption of individual phases and computational kernels. Whereas exploratory work as presented in this Deliverable can be done with simple, single-purpose and stand-alone benchmarks, it is often difficult to extract such exemplars from real applications. A whole-application tracing framework removes the need for this.

The work in this Deliverable lays out a framework for doing this, and demonstrates the results of implementing this in the widely-used Score-P and Vampir packages. Tools like these will be a keystone in providing the feedback needed to drive the co-design cycle. Understanding and interpreting the information provided is, however, still challenging. Therefore, a more detailed investigation of the provided and derived counters in the context of usefulness and quality has to be done in the future.

Clearly, there are limitations in using a generic benchmark suite like the NAS Parallel Benchmarks, rather than the CRESTA applications (either directly or as captured in the CRESTA benchmark suite), and also in using a restricted number of nodes. Future work should look to remedying this, as time and machine availability permit.

In conclusion, then, we appear to have reached a point where not only is energy consumption important in HPC, but we are now at a point 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.

8 References

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

[2] D3.6.2, CRESTA project.

[3] EU energy, transport and GHG emissions trends to 2050 reference scenario 2013, http://ec.europa.eu/energy/observatory/trends_2030/doc/trends_to_2050_update_2013.pdf [Accessed 26.Feb.14]

[4] The Green 500 list, http://www.green500.org .

[5] The Top 500 list, http://www.top500.org .

[6] The Mont Blanc project, http://www.montblanc-project.eu

[7] N. Rajovic et al., "Are mobile processors ready for HPC?", https://www.montblanc-project.eu/sites/default/files/publications/Are%20mobile%20processors%20ready%20for%20HPC.pdf

[8] http://www.montblanc-project.eu/sites/default/files/publications/armhpc-sc.pdf

[9] http://www.samsung.com/global/business/semiconductor/minisite/Exynos/data/Enjoy_the_Ultimate_WQXGA_Solution_with_Exynos_5_Dual_WP.pdf

[10] Mont Blanc press release, http://www.montblanc-project.eu/press-corner/news/mont-blanc-project-selects-samsung-exynos-5-processor-3 [Nov. 2012]

[11] D. Brodowski and N. Golde, "Linux CPUFreq Governors", https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt [Accessed 24.Feb.2014].

[12] The HPL CUDA package, https://github.com/avidday/hpl-cuda [Accessed 28.Feb.14].

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

[14] CRESTA deliverable D2.4.1 "Alternative use of fat nodes".

[15] This idea was first suggested by M. Bull, private communication.

[16] D. Khabi, U. Kuester, “Power consumption of kernel operation”, Proceedings of the joint Workshop on Sustained Simulation Performance (University of Stuttgart (HLRS) and Tohoku University, 2013), edited by W. Bez, E. Focht, H. Kobayashi, Y. Kovalenko, and M. M. Resch, Springer International, 2013, S. 27-45.

[17] Intel Corporation, “Enhanced Intel SpeedStep Technology for the Intel PentiumM Processor”, ftp://download.intel.com/design/network/papers/30117401.pdf : s.n., 2004.

[18] S. Williams, A. Waterman, D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” Communications of the ACM, 2009, Bde. Vol. 52, No. 4, pp 65-76. doi:10.1145/1498765.1498785.

[19] Markus Wittmann, Georg Hager, Thomas Zeiser, Gerhard Wellein, “An analysis of energy optimized lattice-Boltzmann CFD simulations from the chip to the highly parallel level.” April,2013. arXiv:1304.7664.

[20] Project EXASOLVERS - Extreme scale solvers for coupled problems (German Priority Programme 1648 Software for Exascale Computing): http://www.sppexa.de/general-information/projects.html