An Integrated Framework for Power-Performance Analysis of Parallel Scientific Workloads

Sergio Barrachina, Maria Barreda, Sandra Catalán, Manuel F. Dolz,
Germán Fabregat, Rafael Mayo, Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universitat Jaume I
Castellón, Spain
Emails: {barrachi,mvaya,catalans,dolzm,fabregat,mayo,quintana}@uji.es

***Abstract*—The path towards Exascale systems will require to energetically address power consumption of future high performance computing (HPC) workloads which, in turn, urges for a better understanding of power usage. In this paper we, present an evolved framework to trace and analyze the power and energy consumption made by parallel scientific applications. The framework includes i) a flexible and extensible design that enables easy integration of different types of power measurement devices and addition of new functionality; ii) a new module that records information on processor states related to power consumption; and iii) an improved power measurement device to monitor internal direct current (DC) power consumption. This environment is thus revealed as a powerful yet easy-to-use tool to investigate and progress on the development of energy-efficient HPC applications.**

Keywords-power consumption; high performance computing; performance analyzers; scientific applications;

I. Introduction

Power consumption has traditionally been a strong constraint for mobile devices due to its impact on battery life. In recent years, the need to reduce energy expenses has also reached the market of desktop and server platforms, which are embracing "greener technologies"; and even in the HPC arena, the power wall is now recognized as a crucial challenge that the community will have to face [1], [2], [3]. Clear signs of this trend are varied, ranging from the energy efficiency regulatory requirements set by the US Environmental Protection Agency to the biannual elaboration of the Green500 list [4] and the ongoing standardization effort of this ranking.

While system power management (especially that related to the processor) has experienced considerable advances during this past period, application software has not benefited from the same degree of attention, in spite of the power harm that an ill-behaving software ingredient can infer. Indeed, tracing the use of power made by scientific applications and workloads is key to detecting energy bottlenecks and understanding power distribution. However, as of today, the number of fully integrated tools for this purpose is insufficient to satisfy a rapidly increasing demand. In this paper, we revisit the tools in [5] for power-performance analysis of parallel scientific applications, and introduce a new framework featuring stronger integration and modular design as well as several significant extensions. In particular, the paper includes the following contributions:

- We present a new release of our package `pmlib` for power measurement, portray the users' view with a detailed example, and expose the modular implementation of this software.
- We describe a new appliance to `pmlib` which collects information on processor energy states, like the C-states and P-states, offering information complementary to that in the performance and power traces.
- We enlarge the number of standard and *ad hoc* power measurement devices that can interface with `pmlib`, and review the hardware implementation of one of our own powermeters.

These new additions increase the practical utility of the framework, while maintaining its accessibility.

The rest of the paper is structured as follows. In Section II, we provide a brief overview of the framework architecture and its use. In Section III we enumerate a collection of powermeters that can interact with our framework, describing in more detail the internals of our own power device. In Section IV we review `pmlib`, the pivotal component of the framework. Finally, the paper is closed with a discussion of related work, in Section V, and a list of concluding remarks and future work in Section VI.

II. Overview of the Power Analysis Framework

Figure 1 offers a graphical representation of the framework for power-performance tracing and analysis. The starting point is a concurrent *scientific application*, instrumented with the `pmlib` software, that runs on a parallel *target platform* (e.g., a cluster, a multicore architecture, or a hybrid computer equipped with one or several graphics processor units (GPUs)) yielding a certain power consumption. Attached to the target platform there is one or several *powermeter devices* –either internal DC or external alternating current (AC)– that steadily sample power, sending this output to a *tracing server*. Calls from the application running on the target platform to the `pmlib` application programming interface (API), instruct the tracing server to

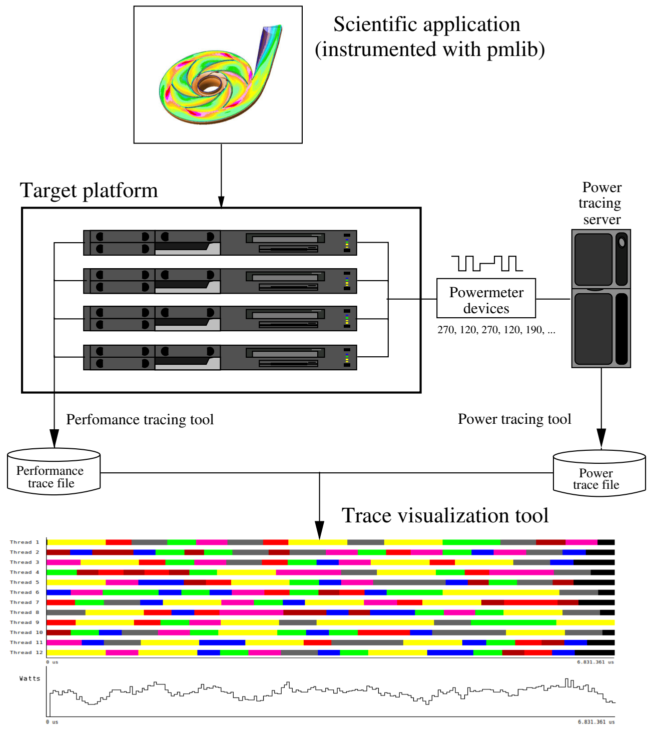


Figure 1. Collecting traces at runtime and visualization of power-performance data.

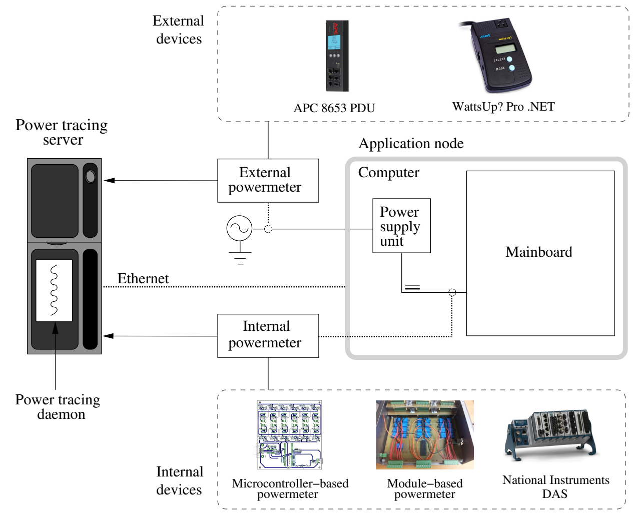


Figure 2. Single-node application system and sampling points for external and internal powermeters.

start/stop collecting the data captured by the powermeters, dump the samples in a given format into a disk file (*power trace*), query different properties of the powermeters, etc. Upon completion of the application's execution, the power trace can be inspected, optionally hand-in-hand with a performance trace, using some *visualization tool*. Our current setting allows a smooth integration of the framework power-related traces and the performance traces obtained with `Extrae`. The resulting combined traces can be visualized with `Paraver` [6]. Nevertheless, the modular design of the framework can easily accommodate other tracing tools like, e.g., TAU, Vampir, etc.

III. Hardware Power Sampling Devices

The `pmlib` package interacts with a number of power sampling devices, including i) external commercial products, such as *APC 8653* Power Distribution Unit (PDU) and *WattsUp? Pro .Net*, which are directly attached to the wires that connect the electric socket to the computer Power Supply Unit (PSU), thus measuring external AC for the full platform; and our own internal DC powermeter designs, consisting of an appropriate choice of current transducers that produce data for ii) a commercial data acquisition system (DAS) from *National Instruments* (NI) and, alternatively, iii) our own designs that use a microcontroller to sample transducer data. These devices are described in more detail next, and the connection points are illustrated in Figure 2.

i) External AC powermeters. The APC 8653 PDU has 24 outlets and operates at a sampling rate of 1 Hz, employing the Simple Network Management Protocol (SNMP) to communicate with the tracing server via Ethernet. The WattsUp? Pro .Net also works at 1 Hz and returns samples to the server through an Universal Serial Bus (USB) 2.0 line.

ii) Powermeter using NI DAS. Our own measurement tools have been developed taking into account that they had to measure currents ranging from 1 to 15 A, without introducing significant voltage drops. The selected transducer was the *LEM HXS 20-NP* Hall effect current sensor. The device exhibits high accuracy and linearity, and a very low internal resistance, while being able to measure current in the required ranges.

A set of our designs include several channels with each one comprising a transducer that is connected to one of the power lines leaving from the PSU. Our final system is a modular design, based on stackable 8-channel components that share power and reference voltage, for a total of 32 current channels.

The DAS is composed of the NI9205 module and the NIcDAQ-9178 chassis. The module features 32 16-bit resolution analog-to-digital (AD) channels which can sample data at 7,000 Hz. In principle, the `LabView` software from NI runs in the tracing server, reading the data captured by the DAS from a USB 2.0 port in the chassis. For convenience, we have developed our own daemon/software to interact with the chassis, without the need of LabView, enabling a better integration of the device with `pmlib`.

iii) Microntroller-based powermeters. Our initial designs [5] featured 10 and 25 channels and a Peripheral Interface Controller (PIC) 18 microcontroller from *Microchip*, to perform AD conversion. Each channel consisted of the aforementioned HXS 20-NP transducer and a 10-bit resolution AD channel in the microcontroller. All the channels shared

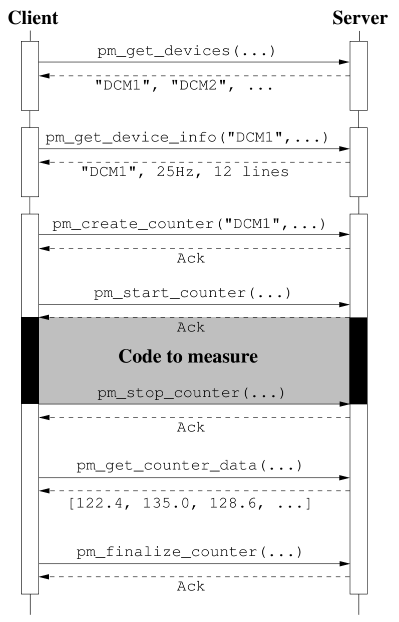


Figure 3. Diagram of the communication between client (running a scientific application) and the (pmlib) server.

a reference voltage of 2.5 V generated by the transducers. Data was sent to the host computer over an asynchronous RS232 port, and the sampling rate was therefore limited by the speed of the communications link (115,200 bauds in the selected microcontroller).

IV. Software

The pmlib software package is developed and maintained by our research group to investigate power usage of HPC applications. The current implementation of this package provides an interface to utilize all the above-mentioned powermeters and a number of tracing tools. We next portray the interface of pmlib using a practical example (user's view), offer a few key implementation details (developer's view), and describe the functionality of a new module to gather information about power-related states of the processor cores. We close this section by illustrating the kind of information provided by the framework using a simple parallel application.

A. User's view

Power measurement is controlled from the application using a collection of routines that allows the user to query information on the power measurement units, create counters associated to a device where power data is stored, start/interrupt/continue/terminate power sampling, etc. All this information is managed by the pmlib server, which is in charge of obtaining these data from the devices and returning the appropriate answers, via the interface of the pmlib routines, to the invoking application (client). This client-server interaction is exposed in Figure 3.

```
int main (int argc, char *argv[])
{
    server_t  server1,  server2;
    counter_t counter1, counter2;
    line_t    lines1,   lines2;
    device_t  disp;
    char      **list;
    int       i, num_devices,
              freq1=0, freq2=0, aggr1=1, aggr2=1;
    // ... Some other variables...

    // Initializes the servers' structures
    pm_set_server("150.128.82.30", 6526, &server1);
    pm_set_server("127.0.0.1",     6526, &server2);

    // Query on #devices connected to server1,
    // and obtain handles. Then, output information,
    // e.g., for device[0]
    pm_get_devices(server1, &list, &num_devices);
    pm_get_device_info(server1, list[0], &disp);
    printf("Name:␣%s\n", disp.name);
    printf("Max␣freq:␣%d\n", disp.max_frecuency);
    printf("Number␣of␣lines:␣%d\n", disp.n_lines);

    // Selects the lines to measure
    pm_set_lines("0-11", &lines1);
    pm_set_lines("0-31", &lines2);

    // Creates a counter for powermeter DCMeter1
    pm_create_counter("DCMeter1", lines1, !aggr1,
                      freq1, server1, &counter1);

    // Creates a counter for C-states
    pm_create_counter("Cstates", lines2, !aggr2,
                      freq2, server2, &counter2);

    // Starts to collect samples: power, C-states
    pm_start_counter(&counter1);
    pm_start_counter(&counter2);
    // Sampled application code fragment
    dgemm( &transa, &transb, &m, &n, &k,
           &alpha,  &A[k*lda+i], &lda,
                    &B[j*ldb+k], &ldb,
           &beta,   &C[j*ldc+i], &ldc );
    // Stops to collect samples
    pm_stop_counter(&counter1);
    pm_stop_counter(&counter2);

    // ...  Some other nonsampled   ...
    // ... aplication code fragment ...

    // Continue to collect samples: only power
    pm_continue_counter(&counter1);
    // Sampled application code fragment
    dsyrk(&transa, &transb, &m, &n,
          &alpha,  &A[k*lda+i], &lda,
          &beta,   &C[i*ldc+i], &ldc);
    //Stops to collect samples
    pm_stop_counter(&counter1);

    // Dumps collected data onto memory
    pm_get_counter_data(&counter1);
    pm_get_counter_data(&counter2);

    // Prints power data in Paraver format
    pm_print_data_paraver("out.prv", counter1,
                          lines1, 0, "us");
    // Prints c-states data in Paraver format
    pm_print_data_paraver_cstates("cstates.prv",
                                  counter2, lines2,
                                  0, "us");

    //Finalizes the counters
    pm_finalize_counter(&counter1);
    pm_finalize_counter(&counter2);
    return 0;
}
```

Figure 4. Example of use of pmlib.

Figure 4 displays a detailed example illustrating the use of `pmlib`. The code first declares the most important variables. Next, two server structures are initialized with their respective Internet Protocol (IP) addresses and the port that will be used for the communication with both servers. Here, the first server returns power samples, and it is located in a separated machine to avoid interfering with the parallel application. C-states are recorded using the second server, which is placed in the same machine where the parallel application runs, so that, it can query the files of this machine containing the requested data on C-states. The invocation to the function `pm_get_devices` establishes a communication with the server, to obtain a list with the names of connected powermeters. With this information, the call to `pm_get_device_info`, with one of the detected powermeters, returns more specific information on this device.

With the next two calls to `pm_set_lines`, we select the lines to measure (distinct powermeters may have different numbers of lines). Next, we also call function `pm_create_counter` twice, to create one counter associated with the `DCMeter1` powermeter and a second one that is bound to the C-states. The measurement is initiated and terminated from the application via routines `pm_start_counter` and `pm_stop_counter`, respectively. In this case, we measure the power and record the C-states during the execution of kernel `dgemm`. The sampling process is momentarily interrupted then, by invoking `pm_stop_counter`, and continued later, with `pm_continue_counter`, to record only power samples for kernel `dsyrk`. Finally, routine `pm_get_counter_data` saves the collected data onto the corresponding counter structure; this information is printed in one of the available formats (in the example, `Paraver` format); and the counters are destroyed using routine `pm_finalize_counter`.

B. *Developer's view*

The `pmlib` software is developed in Python and consists of two modules: the *settings* file and the *server*. Figure 5 depicts how the server works. The daemon starts by initially reading the settings file, which contains configuration information on the powermeters available in the system. Afterwards, a new thread is created per powermeter in order to manage and receive data from these devices. The server then creates as many counters (i.e., new thread instances) as required by the clients.

The main threaded classes implemented by the server are:

- `Device`. This class reads the data collected by a specific powermeter and stores all the active counters that measurement.
- `Counter`. This class manages all the operations performed on a counter. It is stored in the `Device` object it is associated with, and contains the data acquired while the counter is running.
- `Info`. This class comprises information about the devices and their configuration.

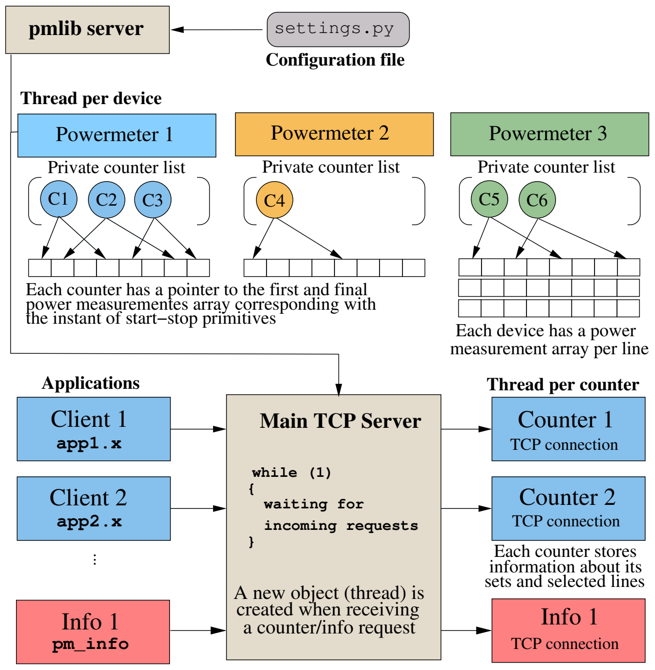


Figure 5. Internal workings of the `pmlib` server.

Figure 5 shows that the server can receive two types of *requests*: a query on information about a device or an operation on a counter. In the first case, the server creates an `Info` object to obtain the required data from the settings file, and sends this back to the client.

If the operation is a request to create a counter, the server allocates a `Counter` object, which will manage all subsequent operations on it, and stores the structure in the appropriate `Device` object. After the creation of a counter, the client should invoke `pm_start_counter` to instruct the server to start recording samples, and `pm_stop_counter` to stop doing it. The client can also use `pm_continue_counter` to restart the recording process, and force the server to record samples from other fragments of the application code in the same counter, generating different `sets` of data. Finally, all collected data can be retrieved by invoking `pm_get_counter_data`.

C. *A module to detect power-related states*

Most current processors, from those designed for mobile devices to desktop and HPC servers, adhere now to the the Advanced Configuration and Power Interface (ACPI) specification [7], which defines an open standard for device configuration and power management from the operating system.

For our power monitoring purposes, the ACPI specification defines a series of *processor states* (or power modes), collectively known as C-states, that are valid on a per-core basis. For example, the C-states available in one of our power-monitored platforms (a server with two Intel Xeon E5504 processors) are:

- **C0**. The core is in operating state (i.e., active or executing instructions).
- **C1**. The core is inactive, but can return to an executing state essentially instantaneously.
- **C3**. The core maintains all software-visible state, but may take longer to wake up.
- **C6**. The core does not need to keep its cache coherent, but maintains other state. Some architectures have variations on the C6 state that differ in how long it takes to wake the processor.

A core in state C0 can be in one of several performance states, referred to as P-states. These modes are architecture-dependent, though P0 always corresponds to the highest-performance state, with P1 to Pn being successively lower-performance modes. In practice, the P-states differ in the operation voltage-frequency pair, with P0 being always bind to the highest pair.

Our power framework obtains a trace of the C- and P-states of each core. For example, in order to obtain information on the C-states, a daemon integrated into the power framework accesses the file `/sys/devices/system/cpu/cpuX/cpuidle/stateY/time`, for each cpu `X` and state `Y`, with a user-configured frequency. The daemon reads values from this file corresponding to the total time (in microseconds) spent in a certain state. This value is then subtracted from the previous read, normalized, and stored together with a timestamp in a file with a user-selected format.

Note that the state-recording daemon necessarily has to run on the target application and, thus, it introduces a certain overhead (in terms of execution time as well as power consumption) that, depending on the software that is being monitored, can become nonnegligible. To avoid this effect, the user is advised to experimentally adjust the sampling frequency of this daemon with care.

Figure 6 offers a graphical example of the information that can be collected with our power-tracing framework, when combined with the performance tracer `Extrae` and the visualization tool `Paraver`. The view there corresponds to the execution of a synthetic parallel benchmark that randomly issues three types of computational kernels: `dgemm` (matrix-matrix product), `dtrsm` (triangular system solve), and `sleep`. The test was run using 8 threads on a platform equipped with two Intel Xeon E5504 cores at 2.00 GHz, with 24 GBytes of RAM. The performance trace in the top plot displays task activity per core; the second plot corresponds to the aggregated power dissipated by the mainboard of the machine, captured with the NI powermeter operating at 1 KHz; the C-states trace in the third plot represents the variations that cores experience between processor states C0, C1, C3 and C6 (with a sampling frequency of 10 Hz). The final part reports the same information contained in the performance and C-states traces, but in numerical format.

V. Related Work

An excellent survey on hardware, software, and hybrid tools for power profiling is given in [8]. We next briefly review some of these efforts, in particular, those proposing solutions related to our framework.

PowerMon2 [9] is a hardware device which, coupled between the computer's PSU and mainboard, samples the power running through the DC lines, offering a basic software interface.

PowerPack [10] employs a commercial DC powermeter from NI, much like ours, connected to the lines leaving from the PSU. This package also performs a number of tests with the purpose of identifying which lines feed different components such as disks, memory, network interface controllers (NICs), processors, etc. This information is then offered to the user who can gain insights on where and how applications consume power. PowerPack exhibits a user-friendly interface, and targets applications running on single-node platforms, though PowerPack's information can be "manually" aggregated for parallel Message Passing Interface (MPI) applications.

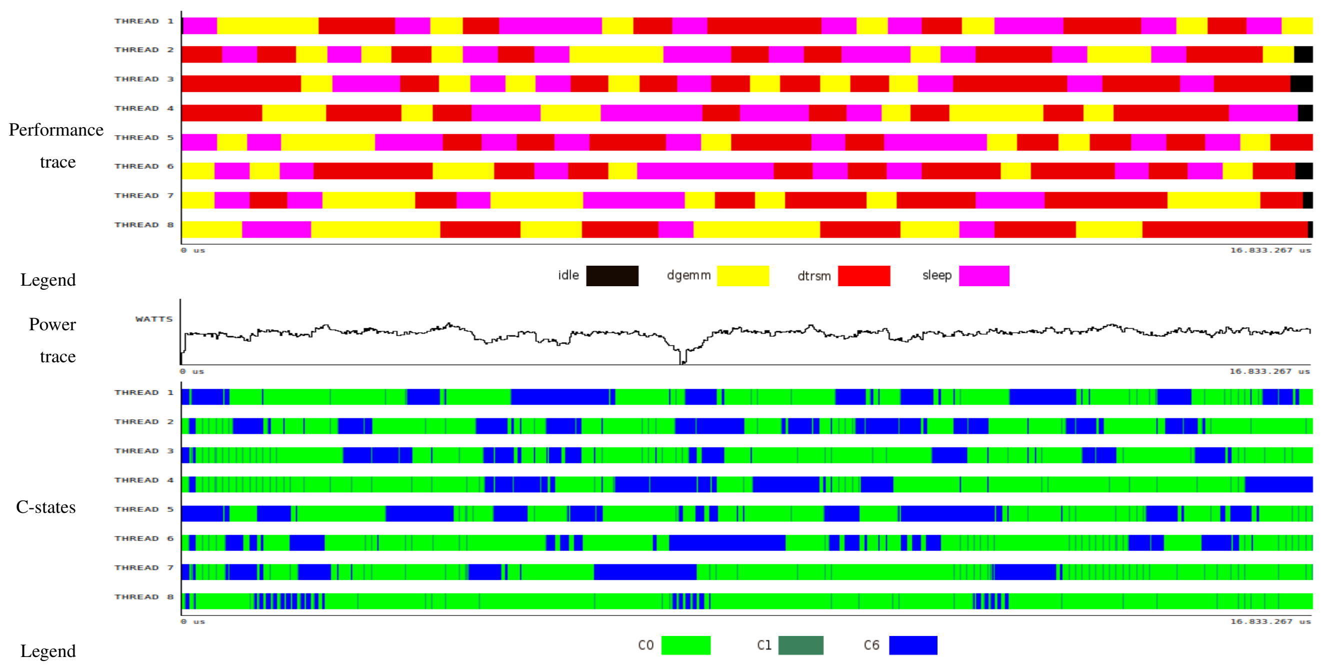
HDTrace [11] is a package to trace and simulate the power-performance footprint of MPI programs on a cluster. This software supports MPICH2 and the parallel file system Parallel Virtual File System (PVFS). Recently, it has also been extended to identify power hot spots in these applications, using information from commercial AC powermeters.

Compared with these other efforts, we believe that our project provides a more complete framework with stronger integration and a better modular design.

VI. Conclusions

We have presented a power-tracing framework composed of internal/external powermeters, a power tracing modular package, power-related modules, etc., that is easily integrable with standard performance tracing and visualization tools. The framework offers highly useful information on power usage of scientific workloads running on a variety of parallel platforms, from MPI applications operating on a moderate-scale cluster to multi-threaded codes that execute on a multicore+GPU platform.

We are currently using this framework to develop more energy-efficient HPC linear algebra libraries, which leverage idle periods during the execution using dynamic frequency-voltage scaling and avoiding busy-waits. In the future, we plan to extend this work to other numerical codes and scientific applications in general, by integrating it into a practical runtime task-scheduler.



	idle	dgemm	dtrsm	sleep
THREAD 1	0,14 %	26,41 %	37,77 %	35,69 %
THREAD 2	1,78 %	28,32 %	34,22 %	35,69 %
THREAD 3	2,03 %	19,62 %	54,56 %	23,79 %
THREAD 4	1,41 %	28,02 %	40,83 %	29,74 %
THREAD 5	0,10 %	22,32 %	38,87 %	38,70 %
THREAD 6	1,66 %	19,54 %	46,09 %	32,71 %
THREAD 7	0,98 %	36,23 %	38,00 %	23,79 %
THREAD 8	0,57 %	45,19 %	42,34 %	11,89 %
Total	8,68 %	225,65 %	333,68 %	231,00 %
Average	1,08 %	28,21 %	41,71 %	28,00 %
Maximum	2,03 %	45,19 %	54,56 %	38,70 %
Minimum	0,10 %	19,54 %	34,22 %	11,89 %
StDev	0,70 %	8,21 %	5,84 %	8,25 %
Avg/Max	0,53	0,62	0,76	0,75

	C0	C1	C6
THREAD 1	64,28 %	0,17 %	35,55 %
THREAD 2	64,47 %	0,15 %	35,38 %
THREAD 3	76,92 %	0,16 %	22,92 %
THREAD 4	70,54 %	0,12 %	29,34 %
THREAD 5	63,19 %	0,14 %	36,67 %
THREAD 6	71,83 %	0,13 %	28,04 %
THREAD 7	75,56 %	0,21 %	24,23 %
THREAD 8	92,92 %	0,04 %	7,04 %
Total	579,71 %	1,12 %	219,17 %
Average	72,46 %	0,14 %	27,40 %
Maximum	92,92 %	0,21 %	36,67 %
Minimum	63,19 %	0,04 %	7,04 %
StDev	9,14 %	0,04 %	9,12 %
Avg/Max	0,78	0,67	0,75

Figure 6. Example of performance and power traces captured by `Extrae` and the proposed power framework, visualized with `Paraver`.

Acknowledgments

This work was supported by project TIN2011-23283 of the *Ministerio de Ciencia e Innovación* and FEDER.

References

[1] J. Dongarra et al., "The international exascale software project roadmap," Int. J. of High Performance Computing & Applications, vol. 25, no. 1, 2011, pp. 3–60.

[2] M. D. et al., "The HiPEAC vision," http://www.hipeac.net/roadmap, [retrieved: January, 2013].

[3] W. Feng, X. Feng, and R. Ge, "Green supercomputing comes of age," IT Professional, vol. 10, no. 1, 2008, pp. 17 –23.

[4] "The Green500 list," http://www.green500.org, [retrieved: January, 2013].

[5] P. Alonso et al., "Tools for power and energy analysis of parallel scientific applications," in Proc. of 41st International Conference on Parallel Processing–ICPP, Sep. 2012, pp. 420–429.

[6] "Paraver: the flexible analysis tool," http://www.cepba.upc.es/paraver, [retrieved: January, 2013].

[7] HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., Toshiba Corp., "Advanced Configuration and Power Interface Specification," 2010.

[8] S. Wang, H. Chen, and W. Shi, "SPAN: A software power analyzer for multicore computer systems," Sustainable Computing: Informatics and Systems, vol. 1, no. 1, 2011, pp. 23–34.

[9] D. Bedard, A. Porterfield, R. Fowler, and M. Y. Lim, "PowerMon 2: Fine-grained, integrated power measurement," RENCI, North Carolina, Tech. Rep. TR-09-04, 2009.

[10] R. Ge, X. Feng, S. Song, H. Chang, D. Li, and K. W. Cameron, "Powerpack: Energy profiling and analysis of high-performance systems and applications," IEEE Transactions on Parallel and Distributed Systems, vol. 99, 2009, pp. 658–671.

[11] J. Kunkel, "HDTrace-a tracing and simulation environment of application and system interaction," Dept. of Informatics, Sci. Comp., Universität Hamburg, Tech. Rep. 2, 2011.