Noname manuscript No.
(will be inserted by the editor)

Assessing the Impact of the CPU Power-Saving Modes on the Task-Parallel Solution of Sparse Linear Systems

José I. Aliaga · María Barreda · Manuel F. Dolz · Alberto F. Martín · Rafael Mayo · Enrique S. Quintana-Ortí

the date of receipt and acceptance should be inserted later

Abstract We investigate the benefits that an energy-aware implementation of the runtime in charge of the concurrent execution of ILUPACK —a sophisticated preconditioned iterative solver for sparse linear systems— produces on the time-power-energy balance of the application. Furthermore, to connect the experimental results with the theory, we propose several simple yet accurate power models that capture the variations of average power that result from the introduction of the energy-aware strategies as well as the impact of the P-states into ILUPACK's runtime, at high accuracy, on two distinct platforms based on multicore technology from AMD and Intel.

Keywords Sparse linear systems · Iterative solvers · ILUPACK · Energy-aware methods · CPU power-saving modes · High performance computing

José I. Aliaga · María Barreda · Rafael Mayo · Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain,
E-mail: {aliaga, mvaya, mayo, quintana}@uji.es

Manuel F. Dolz
Dept. of Informatics, University of Hamburg, 22.527–Hamburg, Germany,
E-mail: manuel.dolz@informatik.uni-hamburg.de

Alberto F. Martín
Centre Internacional de Mètodes Numèrics a l'Enginyeria (CIMNE), Parc Mediterrani de la Tecnologia, Universitat Politècnica de Catalunya, 08.860-Castelldefels, Spain,
UPC, Edifici C1, 08.034-Barcelona, Spain,
E-mail: amartin@cimne.upc.edu

1 Introduction

Power consumption has been identified as a crucial challenge that will need to be tackled in order to deliver a sustained EXAFLOPS[1] throughput, at an affordable cost, in high-performance computing (HPC) facilities deployed by the end of this decade [14,16,21]. In particular, although we have enjoyed considerable advances in the performance-power ratio of HPC systems during the last few years —around 5× in the MFLOPS/Watt ratio for the systems ranked in the first position of the Green500 list during the last five years [1]— the ambitious goal of building an Exascale system by 2020, which dissipates only 20 MWatts, demands even more favorable improvements rates.

A major part of the progress experienced in the energy efficiency of HPC systems in these past years has been due to the hardware, concretely, to the aggregation of low-power processors with an increasing number of cores and the adoption of hardware accelerators for HPC [3]. However, much remains to be done in order to leverage the power-saving mechanisms available in today's hardware and a holistic investigation and multidisciplinary effort has to be conducted in energy-efficient HPC, comprising applications, system software as well as hardware [4].

ILUPACK [2] is a numerical package for the solution of sparse linear systems via Krylov-based iterative methods. The software implements multilevel ILU (incomplete LU) preconditioners for general, sym-

[1] A ratio of 1 EXAFLOPS=10^{18} floating-point arithmetic operations, or flops, per second.

metric indefinite and Hermitian positive definite systems, as those arising in numerous scientific and engineering applications, in combination with inverse-based ILUs and Krylov subspace solvers.

In [5] and [6] we presented two concurrent versions of ILUPACK, for multicore architectures and distributed-memory (message-passing) platforms respectively, that efficiently exploit the task-parallelism intrinsic to the iterative solution of the sparse linear systems (including the calculation of the preconditioner). In response to the increasing energy awareness in HPC, in [7] we analyzed the energy efficiency of the task-parallel calculation of the preconditioner on an AMD-based multicore platform, leveraging the CPU power-saving modes available in this architecture [20]. In this paper we extend our previous work, with the following new contributions:

- We introduce energy-aware strategies into ILUPACK runtime and we address the characterization (modeling) of power consumption in the preconditioned solution of symmetric positive definite (s.p.d.) sparse linear systems, thus expanding our previous work [7] on the computation of the preconditioner only, to cover the complete solver.
- We collect precise information on the CPU power-saving modes, identifying the sources of the power bottlenecks and the energy gains, and relating the power consumption and attained savings observed in our detailed experimental results to the power models.
- We target a server based on the AMD Opteron 6128 processor, already analyzed in [7], but we also model and evaluate an alternative platform equipped with two Intel Xeon E5504 processors. These servers are representative of current multicore technology and both abide to the CPU power-saving modes in the ACPI (advanced configuration and power interface) specification [20]. From the points of view of power consumption and performance, these platforms present two relevant differences, though. First, all cores of the Intel processor share the same power plane and the core frequency can be only adjusted at the processor (socket) level. Instead, on the AMD server the frequency can be varied per core. Second, as our experiments will show, on the Intel platform memory bandwidth is independent of the processor operation frequency while, on the AMD server, changes to frequency also affect memory throughput.
- Finally, we refine the experimental evaluation in [7], using a more accurate wattmeter with a much higher sampling rate than that utilized in our previous study.

As one of the goals of this paper is to analyze the effect that the exploitation of the CPU power-saving mechanisms exerts on the energy efficiency of a complex scientific application, we consider only the power dissipated by the components integrated in the system motherboard (e.g., CPU and RAM chips), discarding other power sinks due, e.g., to network interface, disk, inefficiencies of the power supply unit, etc. For applications such as ILUPACK, which mainly exercise the floating-point arithmetic units of the processors and the memory, we can expect that the contribution of the discarded components is a constant that can be simply added to our models.

There are a large number of software efforts to reduce energy consumption in HPC clusters. For brevity, we next reference only a few. The work in [30] presents a characterization of energy saving techniques for clusters with two large groups: static power management (SPM) and dynamic power management (DPM). Within DPM the authors further distinguish between component-based and power-scalable load balancing (LB) techniques. Our approach, based on the exploitation of C-states, can be considered analogous to that in [11,26], however we apply it at processor level rather than at the node level. From that perspective, our techniques can be classified as LB. The authors in [31] leverage DVFS (dynamic voltage-frequency scaling) to reduce the power-energy consumption of precedence-constrained tasks running on a cluster. The heuristic-based strategy used by the authors minimizes the slack of non-critical tasks by means of reducing its frequency operation while maintaining the time-to-solution. A similar approach is applied to the execution of dense linear algebra algorithms in multicore processors in [9]. In [22], the authors leverage hierarchical genetic strategy-based grid scheduling algorithms to exploit span via DVFS and reduce energy consumption in computational grids. The simulation results show that proposed scheduling methodologies fairly reduce the energy usage and can be easily adapted to the dynamically changing grid states and various scheduling scenarios. Our strategy considers a static VFS-

based approach, in which the voltage/frequency pair is fixed at the beginning of the execution and does not vary thereafter. Our energy savings come instead from the exploitation of the C-states which, in general, lead to considerable energy savings by relying on a *race-to-halt/race-to-idle* strategy. DVFS strategies are part of future work.

The rest of the paper is structured as follows. In Section 2 we briefly review the numerical approach underlying ILUPACK and the task-parallel implementation oriented to multicore architectures from [5]. In Section 3 we introduce the setup for our experiments. The next three sections contain the main contribution of the paper: The power model in Section 4; the power-aware runtime together with a theoretical and experimental analysis of the impact of the C-states on the time-power-energy balance in Section 5; and an analogous study, from the point of view of the P-states, in Section 6. Finally, the paper is closed with a discussion of the results in Section 7.

2 Parallel ILUPACK for Multicore Processors

The approach to multilevel preconditioning in ILUPACK relies on the so-called inverse-based ILU factorizations. Unlike classical threshold-based ILUs, this approach directly bounds the size of the preconditioned error and results in increased robustness and scalability, especially for applications governed by PDEs, due to its close connection with algebraic multilevel methods [5]. Specifically, for efficient preconditioning, only a small amount of fill-in is allowed during the factorization, resulting in a modest number of floating-point arithmetic operations per nonzero entry of the sparse coefficient matrix.

Parallelism in the computation of ILUPACK preconditioners is exposed by means of nested dissection applied to the adjacency graph representing the nonzero connectivity of the sparse coefficient matrix. Nested dissection is a partitioning heuristic which relies on the recursive separation of graphs. The graph is first split by a vertex separator into a pair of independent subgraphs and the same process is next recursively applied to each independent subgraph. The resulting hierarchy of independent subgraphs is highly amenable to parallelization. In particular, the inverse-based preconditioning approach is applied in parallel to the blocks corresponding to the independent subgraphs while those corresponding to the separators are updated. When the bulk of the former blocks has been eliminated, the updates computed in parallel within each independent subgraph are merged together, and the algorithm enters the next level in the nested dissection hierarchy. The same process is recursively applied to the separators in the next level and the algorithm proceeds bottom-up in the hierarchy until the root finally completes the parallel computation of the preconditioner; see Figure 1.

The type of parallelism described above can be expressed by a binary task dependency tree, where nodes represent concurrent tasks and arcs specify dependencies among them. The parallel execution of this tree on multi-core processors is orchestrated by a runtime which dynamically maps tasks to threads (cores) in order to improve load balance requirements during the computation of the ILU preconditioner. At execution time, thread migration is prevented using POSIX routine `sched_set_affinity`. The runtime keeps a shared queue of ready tasks (i.e., tasks with their dependencies fulfilled) which are executed by the threads in FIFO order. This queue is initialized with the tasks corresponding to the independent subgraphs. Idle threads have to wait for new ready tasks. When a given thread completes the execution of a task, its parent task is enqueued provided the sibling of the former task has been already completed as well.

The most expensive operation involved in the preconditioned iterative solution of the linear system is the application of the multilevel preconditioner, which is in turn decomposed into two steps: the multilevel forward (FS) and backward substitutions (BS). The aforementioned task dependency tree also describes the parallelism available within both computations. However, while the FS proceeds bottom-up towards the root of the tree, the BS proceeds in the opposite direction. In order to maximize data locality during the parallel multi-threaded execution of both operations, the mapping of threads to tasks resulting from the (dynamic load-balancing) computation of the preconditioner is re-used, so that each thread knows in advance which tasks it is in charge of (i.e., static mapping). The runtime uses a different task queue for each thread and substitution algorithm. For the FS, the queue of each thread is initialized with the leaves it is in charge of, and new (ready) tasks are enqueued on the corresponding queues as soon as their dependencies are ful-

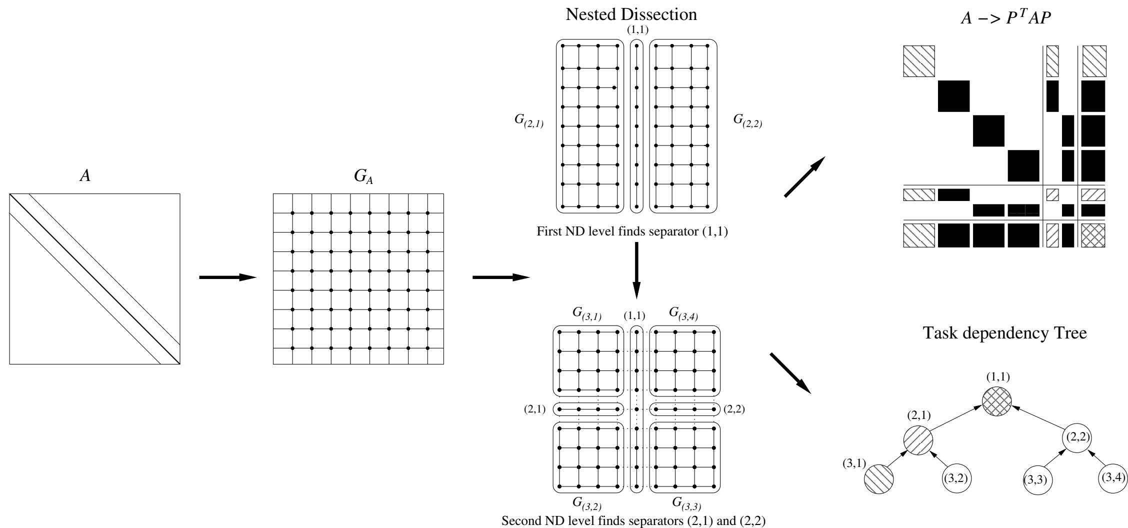

Fig. 1 Nested dissection applied to the adjacency graph associated with a sparse matrix and the corresponding task dependency tree.

filled (i.e., as soon as their children tasks are completed). For the BS, only the root task is initially included in the corresponding queue. As soon as the root task is completed, its children are enqueued on the corresponding queues, and the parallel execution (orchestrated by the runtime) proceeds top-bottom while taking care of task dependencies until the computation of the leaves is completed. The other operations involved in the preconditioned iterative solution stage (i.e., sparse matrix-vector product and vector operations) are split and mapped conformally with the FS and BS steps in order to maximize data locality. Moreover, a careful management of shared data, by maintaining consistent or inconsistent copies of the matrix and the vectors, avoids synchronization steps, except those reductions required in order to compute inner products. Further details on the mathematical foundations of the parallel algorithms, their implementation, and the runtime operation can be found in [5].

3 Environment Setup

In all our experiments, we employ a scalable symmetric positive definite sparse linear system of dimension $n = N^3$, resulting from a partial differential equation $-\Delta u = f$ in a 3D unit cube $\Omega = [0, 1]^3$ with Dirichlet boundary conditions $u = g$ on $\delta\Omega$, discretized using a uniform mesh of size $h = \frac{1}{N+1}$. We set $N = 252$, which yields the largest linear system that fitted into the main memory of the target machine, with about $16 \cdot 10^6$ unknowns and $111 \cdot 10^6$ nonzero entries in the coefficient matrix. All tests were performed using IEEE double-precision arithmetic. Execution time, power and energy are always reported in seconds (s), Watts (W) and Joules (J), respectively.

We employ two target servers in our experiments. The first platform, WT_AMD, is equipped with a single AMD Opteron 6128 processor (8 cores), 24 Gbytes of RAM, and runs the Linux Ubuntu operating system (kernel 2.6.32-220.4.1.el6.x86_64). The second server, WT_INT, contains two Intel Xeon E5504 processors (4 cores per socket), 32 Gbytes of RAM, and runs Linux Ubuntu (kernel 2.6.32-220.4.1.el6.x86_64) as well. These two types of processors are representative of current multicore technology, and adhere to the ACPI standard [20] for the CPU power-saving modes. Concretely, the AMD processor features 5 performance states (P-states P0 to P4) and three operating or power states (C-states C0, C1 and C1E). The Intel processor has 4 P-states (P0 to P3) and 4 C-states (C0, C1, C3 and C6). Information on the voltage–frequency pairs ($V_i - f_i$) associated with each P-state (P_i) is collected in Table 1. State C0 is the normal operating mode (i.e., the CPU is operative) while higher numbers correspond to deeper sleep modes, where more circuits and signals of the processor are turned off, saving more power, but requiring longer time to go back to C0 mode.

Platform	P-state, P_i	V_i	f_i	BW_i
WT_AMD	P0	1.23	2.00	30.29
	P1	1.17	1.50	24.63
	P2	1.12	1.20	20.46
	P3	1.09	1.00	17.48
	P4	1.06	0.80	14.00

Platform	P-state, P_i	V_i	f_i	BW_i
WT_INT	P0	1.04	2.00	12.72
	P1	1.01	1.87	12.58
	P2	0.98	1.73	12.61
	P3	0.95	1.60	12.55

Table 1 P-states, associated voltage–frequency pairs (V_i in Volts and f_i in GHz), and core to memory bandwidth (BW_i, in GB/sec.) measured with the `stream` benchmark.

From the practical point of view, the AMD and Intel servers differ in two important aspects:

– The frequency of the AMD cores can be adjusted independently while, on the Intel platform, all cores in the same processor run at the same frequency. In particular, if the cores of a processor from WT_INT operate at frequency f_i, and we instruct one of these cores to run at $f_j > f_i$ (using the Linux `cpufreq` utility), the remaining three cores in the same socket will also transition to operate at f_j. On the other hand, if the cores of a processor from WT_INT run at frequency f_i, and we instruct one core to run at frequency $f_j < f_i$, there will be no change.
– On the AMD platform, the bandwidth between the cores and the main memory varies with the processor frequency while, on the Intel platform, this bandwidth is independent of the processor frequency. To illustrate this behavior, column BW_i of Table 1 reports the bandwidth to the main memory experienced by a single core running the `stream` microbenchmark [29] at different frequencies.
  We note that the bandwidth-frequency dependence is a design decision specific of each processor type: more recent processors as, e.g., the Intel Xeon E52670 “Sandy Bridge” seems to follow AMD 6128’s strategy and reduce the bandwidth with the processor frequency [18]; on the other hand, some other processors like the AMD 6274 “Interlagos” apparently abandon this approach [27].

In our experiments, power samples were obtained from the 12-Volt lines connecting the power supply unit to the motherboard of the target platform[2], using an internal wattmeter composed of a National Instruments (NI) analog input module (9205) plugged into a NI chassis (cDAQ-9178) and a board of current transducers (LEM HXS 20-NP). The wattmeter is connected via an Ethernet link to a separate power tracing server that runs a daemon application to collect power samples form the internal wattmeter. The measurement application is built by calling routines from the `pmlib` library [8,13]. We set the sampling rate to 1 kSamples/sec., which is high enough to obtain reliable measures for the power model and remaining experiments.

Our multithreaded implementation of ILUPACK is built on top of the OpenMP interface available with Intel `icc` (version 12.1.3) on both platforms. Performance (core activity) traces were captured using the `Extrae+Paraver` (versions 2.2.1+4.3.4) tracing environment [25]. Traces of CPU power modes were recorded using the Linux interface to read and write model-specific registers (MSRs) and dumped into `Paraver`-compatible files for interactive visualization and analysis.

4 The Power Model and the CPU Power-Saving Modes

We open this section by revisiting the following simple model from [10] for the total (aggregate) power dissipated by an application at a given instant of time t:

$$P^T = P^Y + P^P = P^Y + P^U + P^C, \quad (1)$$

where P^P is the power dissipated by the CPU processor(s) and P^Y is the power dissipated by the remaining components (system power corresponding, e.g, to DDR RAM chips, motherboard, etc.). Furthermore, P^U is the power dissipated by the *uncore* [19] elements of the processor (e.g., last-level cache, memory controllers, core interconnect, etc.); and P^C is the power for the cores (including the in-core cache

[2] On a separate experiment [15], it was determined that the aggregate power supplied by the 3-Volt and 5-Volt lines during the execution of ILUPACK on these two platforms remains practically constant and, furthermore, it is negligible compared with that measured from the 12-Volt lines.

Platform	P-state, P_i	P_i^Y	α_i	$\beta_{burn,i}$	$\beta_{busy,i}$	$\beta_{gemm,i}$	P_i^U
WT_AMD	P0	84.83	140.94	14.70	13.50	13.82	56.11
	P1	77.85	137.47	7.71	6.10	10.01	59.62
	P2	73.83	131.35	5.48	4.58	7.40	57.52
	P3	71.86	127.87	4.12	3.33	5.46	56.01
	P4	72.46	124.89	3.10	2.28	4.27	52.43
WT_INT	P0	33.43	64.44	9.48	7.10	11.12	31.01
	P1		63.38	8.19	6.16	9.84	29.95
	P2		64.10	7.33	5.43	9.03	30.67
	P3		64.72	6.34	4.64	7.81	31.29

Table 2 Parameters for the simple power model for the cpu*burn*, *busy* and *gemm* benchmarks (kernels) on WT_AMD and WT_INT. Note that $P_i^U = \alpha_i - P^Y$ and $P_{k,i}^C(c) = P_{k,i}^{C1} \cdot c = \beta_{k,i} \cdot c$, with i, k and c denoting, respectively, the kernel type, P-state and number of cores.

levels, floating-point units, branch logic, etc.). While our power models refer to the power dissipated at a given instant of time t, in most experiments next, we will report instead the average power for the application, as this allows us to compile the information for the complete execution in a single figure. Furthermore, this easily connects the results with the total energy consumption.

Given a platform with all cores in state P_i, for simplicity we will assume that P_i^Y and P_i^U (i.e., the system and uncore powers in state P_i) remain constant during the execution of the application. In practice, starting from an idle (cold) platform, these two factors grow with the system temperature till their addition reaches a plateau [12]. To avoid this effect, we assume that there is a continuous workload to run in the platform and, in order to mimic this situation, all our tests will be performed on a "hot" system, with this state reached by initially warming the cores with an execution of the same kernel or application for a given period of time. Also, we will assume that P_i^U is independent of the application that runs in the platform. In practice, this is not the case, but our results will show that the errors introduced by this simplification are small, and can be easily accommodated into the model.

To obtain practical values for the power model, we proceed as follows. For simplicity, let us assume that all c active cores of the platform run the same type of task (kernel) k, in the same state P_i, during all the execution time. In this scenario, we can easily consider that the total power at instant t equals the average power. For P_i^Y, we thus simply set all the cores of the platform to each P-state using `cpufreq`, and then measure the power with the platform idle for 30 seconds and average the results: between 71.86 and 84.83 W, depending on the state P_i, for WT_AMD; and 33.43 W for all P-states in WT_INT; see column P_i^Y in Table 2.

The estimation of P_i^U and P_i^C is more elaborated, as it is difficult to separate these two components, and the second depends also on the application that is being run. In order to achieve this, let us start by refining (1), to capture the total power for the execution of c copies of task k, with the active cores in state P_i:

$$P_{k,i}^T(c) = P_i^Y + P_i^U + P_{k,i}^C(c) \approx P_i^Y + P_i^U + P_{k,i}^{C1} \cdot c, \quad (2)$$

where $P_{k,i}^{C1}$ denotes the power dissipated by a single core in state P_i running task k.

To estimate the missing parameters in (2), P_i^U and $P_{k,i}^{C1}$, we will leverage three compute-intensive kernels: the *cpuburn* microbenchmark[3], a simple *busy*-wait test consisting of a "`while (1) ;`" loop, and the general dense matrix-matrix product (*gemm*) routine implemented as part of Intel MKL operating with double-precision data. Specifically, we executed these tests for 60 seconds and averaged the power draw, for an increasing number of cores c (all in the same P-state) of the machines (8 cores of both WT_AMD and WT_INT), while the remaining cores remain in an inactive C-state.

Applying linear regression to the data obtained from this experimental evaluation, we obtained linear models for the total power of the form

$$P_{k,i}^T(c) = \alpha_{k,i} + \beta_{k,i} \cdot c, \quad (3)$$

[3] `http://manpages.ubuntu.com/manpages/precise/man1/cpuburn.1.html`.

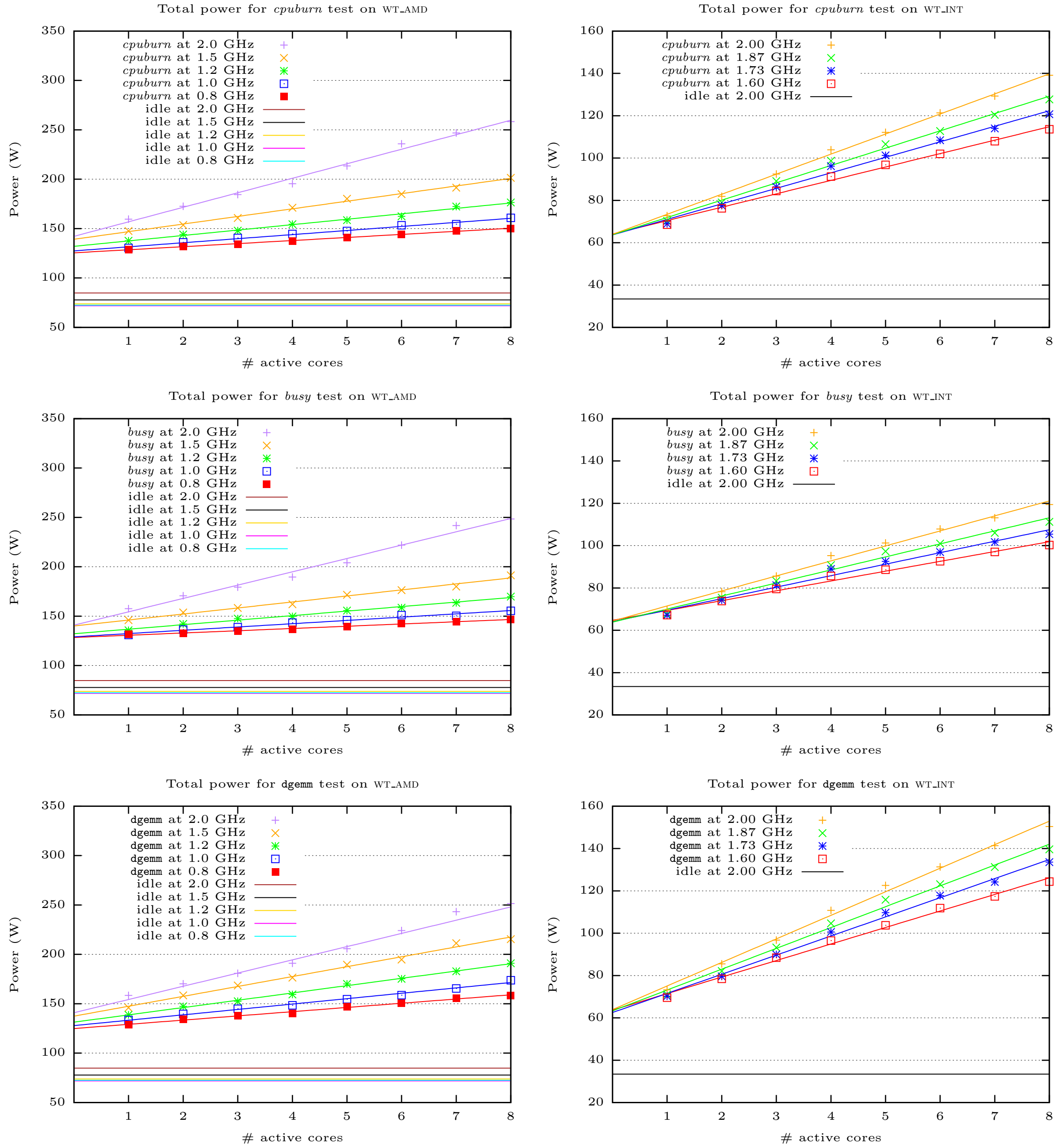

Fig. 2 Power dissipated as a function of the number of active cores for kernels cpu*burn*, *busy* and *gemm* (top, middle and bottom, respectively) on WT_AMD (left) and WT_INT (right).

with the values for $\alpha_{k,i}$ and $\beta_{k,i}$ in the corresponding columns of Table 2, and the relation between the models and the experimental data graphically captured in Figure 2.

These regression models show quite a perfect fit with the experimental data, offering the same (rough) value $\alpha_{k,i}$ for all three kernel types (the largest variation between the three was 2.11 % and the average difference 0.61 %.) Therefore, in the following we use $\alpha_i - P_i^Y$ as an estimation for P_i^U (see Table 2), the uncore power dissipated by a socket in state P_i (which agrees with our assumption that the uncore power is independent of the kernel type); and we set $P_{k,i}^{C}(c) = \beta_{k,i} \cdot c$ so that $P_{k,i}^{C1} = \beta_{k,i}$.

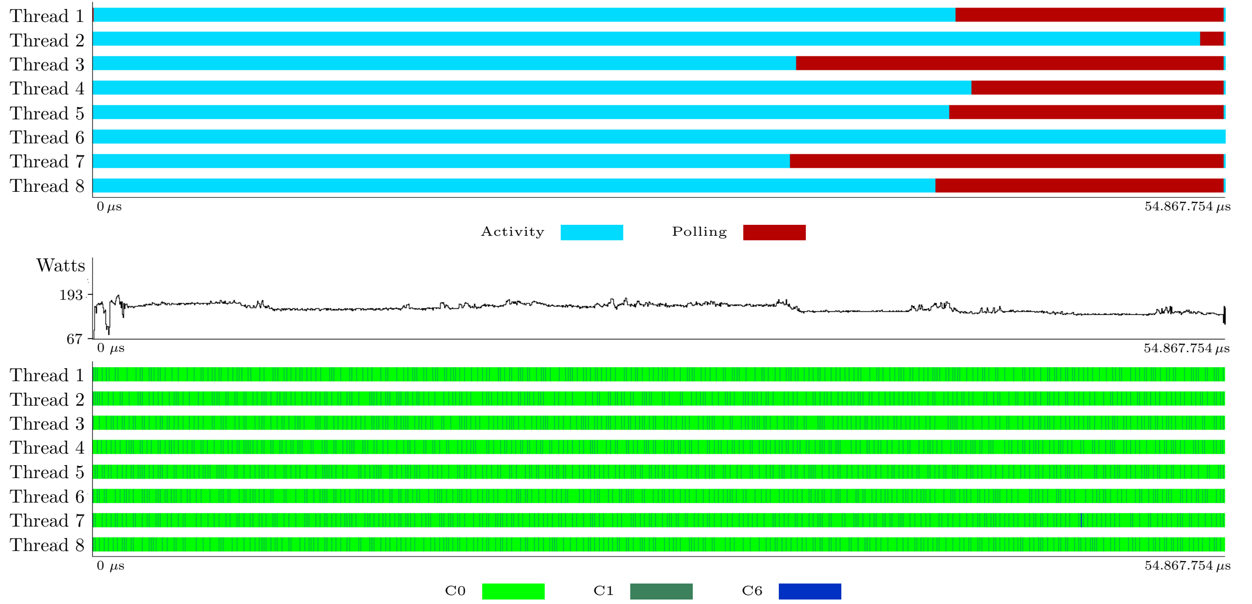

Fig. 3 Traces of core activity, power and C-states (top, middle and bottom, respectively) during the computation of the ILU preconditioner using the performance-oriented, power-oblivious runtime, with all cores of WT_INT in state P0.

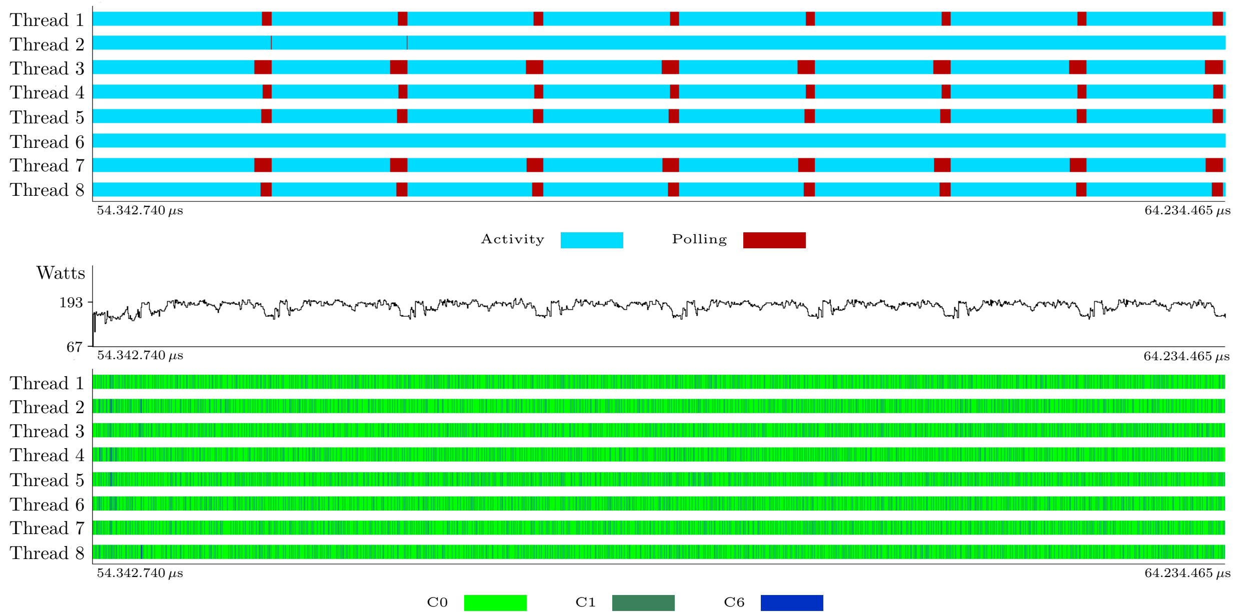

Fig. 4 Traces of core activity, power and C-states (top, middle and bottom, respectively) during (part of) the iterative solution stage using the performance-oriented, power-oblivious runtime, with all cores of WT_INT in state P0.

5 Leveraging the C-States in ILUPACK

We next investigate the exploitation of the C-states made by two implementations of the runtime underlying ILUPACK, and relate their costs to the previous power model. In the experiments in this section, we employ all the cores of the target platforms; and we set the Linux governor to `ondemand` operating all the active cores in the same state P0 during all the execution (i.e., we do not allow voltage-frequency changes).

In Section 2 we exposed that the task-parallel calculation of the preconditioner in ILUPACK is organized as a directed task graph, with the struc-

ture of a binary tree and bottom-up dependencies, from the nodes (tasks) at each level to those in the level immediately above it. The subsequent iterative process basically requires the solution of (lower and upper) triangular linear systems per iteration, with tasks that are also organized as binary task-trees, with bottom-up (lower triangular system) or top-down (upper triangular system) dependencies. In any case, when the tasks of these trees are dynamically mapped to a multicore platform by the runtime, the execution should result in periods of time during which certain cores are idle, depending on the number of tasks of the tree, their computational complexity, the number of cores of the system, etc. It is basically these idle periods that we could expect that the operating system leverages, by promoting the corresponding cores into a power-saving C-state (sleep mode).

Figure 3 presents the execution trace, power consumption, and C-states observed during the computation of the ILUPACK preconditioner, using the original (power-oblivious) runtime, with all cores of WT_INT in state P0. Surprisingly, the results are quite different from what we had expected: Idle periods do not show a transition of the corresponding core to a power-saving C-state and the associated reduction of the power rate. Figure 4 reports an analogous behavior for the (preconditioned) iterative solution stage on WT_INT (and similar results were also obtained for both stages on WT_AMD). A closer inspection of the runtime that leverages the task-parallelism in ILUPACK reveals the reason for these surprising results. Concretely, in the original implementation of ILUPACK runtime, upon encountering no tasks ready to be executed, "idle" threads simply perform a "busy-wait" (polling) on a condition variable, till a new task is available. This strategy thus prevents the operating system from promoting the associated cores into a power-saving C-state because the threads are not actually idle (but doing useless work). This performance-oriented decision is far from uncommon, being adopted in runtimes like OmpSs (SMPSs) [28] or `libflame`+SuperMatrix [17] as well. Furthermore, the same performance-oriented but power-oblivious behavior appears, for example, when a synchronous GPU kernel is invoked with the default operation mode of CUDA [24] (the CPU remains in an active polling, waiting for the GPU to finish), or with the polling mode of certain MPI implementations (e.g., MVAPICH [23]).

As an alternative to the previous power-hungry strategy, we developed a power-aware version of the runtime underlying ILUPACK, which applies an "idle-wait" (blocking) whenever a thread does not encounter a task ready for execution and, thus, becomes inactive. (Note that setting the necessary conditions for the operating system to promote the cores into a power-saving C-state is as much as we can do, since we cannot explicitly enforce the transition from the application code.) As in the original version of the runtime, upon completing the execution of a task, a thread updates the corresponding dependencies identifying those tasks, if any, that have become ready for execution. However, in the power-aware runtime, the thread also ensures that the number of active (non-blocked threads) is, at least, equal to the number of ready tasks, releasing blocked threads if needed. The effect of idle-wait on the power trace and use of the C-states of WT_INT is illustrated in Figure 5, for the calculation of the preconditioner, and Figure 6, for the iterative solution stage. Compared with the performance-oriented (but power-hungry) implementation of the runtime (see Figures 3 and 4), the new runtime effectively allows inactive cores to enter a power-saving C-state, thus yielding the sought-after power reduction.

The pending question, however, is whether the adoption of the power-aware runtime comes with a performance penalty which may blur the energy benefits, as in most cases the key factor is energy instead of power. Table 3 compares the execution time, average power, and energy consumption of the two runtimes, showing that fortunately this is not the case for the computation of the preconditioner and iterative solution, on any of the two target platforms when operating in state P0. Consider for example platform WT_AMD: For a minimal increase in the total execution time, from 286.28 s to 287.91 s, we observe reductions in the (average) power from 240.17 W to 227.16 W for the preconditioner; and from 269.27 W to 230.80 W for the solver. The outcome is a decrease of the total energy from 75,163.74 J to 67,758.84 J (-10.88 %). The power reductions attained by the power-aware implementation with respect to the power-oblivious case are given in percentage in the columns labeled as "Experimental" of Table 4 (averaged for 10 repeated executions). Combined with the negligible impact of the runtime on the execution time, these power figures also justify similar energy savings.

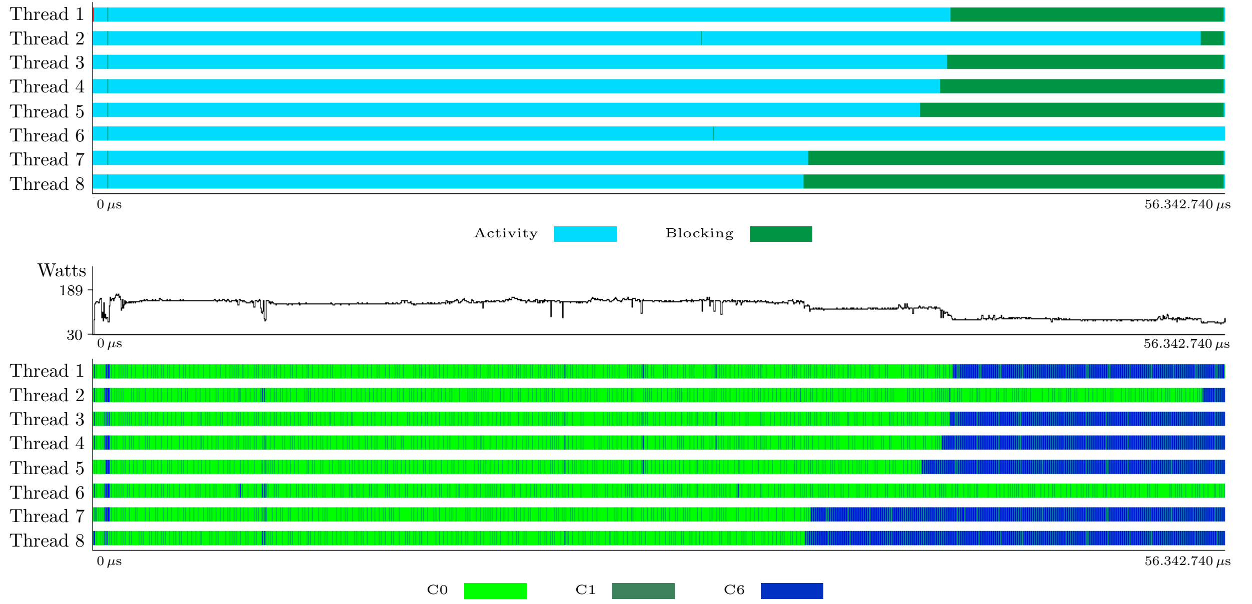

Fig. 5 Traces of core activity, power and C-states (top, middle and bottom, respectively) during the computation of the ILU preconditioner using the power-aware runtime, with all cores of WT_INT in state P0.

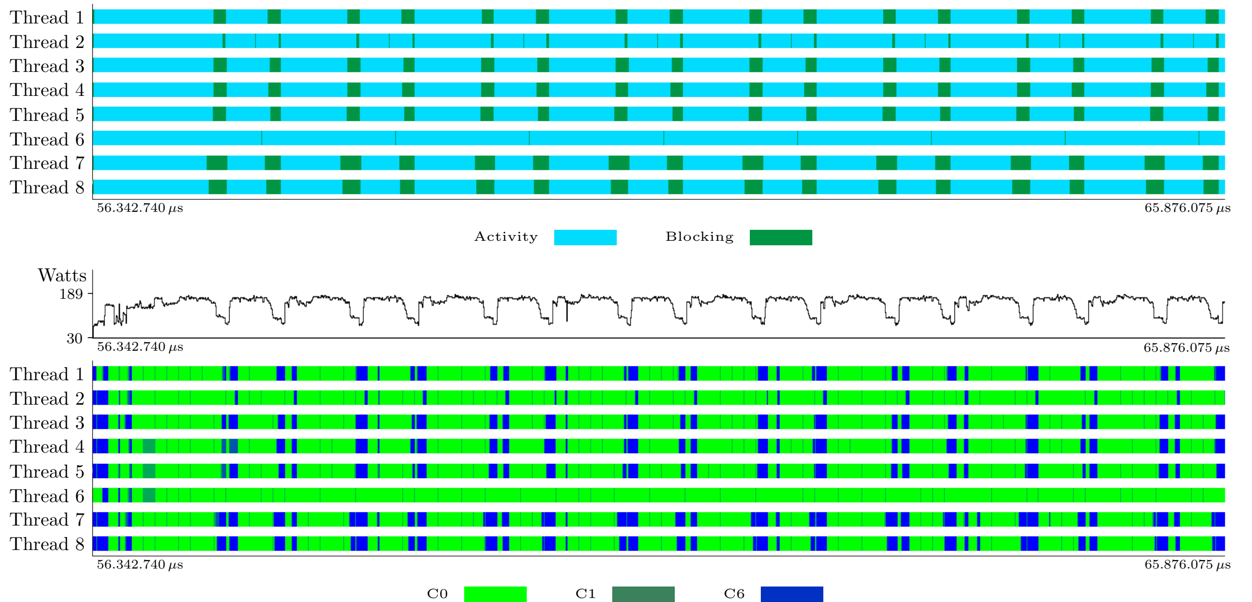

Fig. 6 Traces of core activity, power and C-states (top, middle and bottom, respectively) during (part of) the iterative solution stage using the power-aware runtime, with all cores of WT_INT in state P0.

Let us now relate the power-energy reductions attained by the reimplementation of the runtime that leverages the CPU C-states to the power model of the previous section. For this purpose, we need to *i)* account the periods of "idle" time during the execution of ILUPACK, with both the original and energy-aware variants, as well as *ii)* assess the impact of replacing a busy-wait (polling) for an idle-wait (blocking). In order to tackle *i)*, we follow a pragmatic approach, and simply execute the actual codes and measure the actual idle and computation times in our case, e.g., using the tracing framework `Extrae+Paraver`, while collecting power samples with the internal wattmeter and `pmlib` library.

Platform	Runtime type	Preconditioner			Iterative solver			Total	
		Time	Power	Energy	Time	Power	Energy	Time	Energy
WT_AMD	Oblivious	66.11	240.17	15,878.82	220.17	269.27	59,284.92	286.28	75,163.74
	Aware	66.52	227.16	15,112.06	221.39	237.80	52,646.41	287.91	67,758.84
WT_INT	Oblivious	54.01	137.58	7,431.67	146.67	162.34	23,809.87	200.68	31,241.69
	Aware	54.47	125.70	6,847.47	148.15	150.17	22,247.93	202.62	29,095.40

Table 3 Execution time, average power and energy of the power-oblivious and power-aware implementations of the runtime (top and bottom, respectively) with all cores operating in state P0.

For *ii)*, we use the data in Table 2 for P_0^Y, P_0^U; and estimate $P_{ilu,0}^{C1} = \beta_{ilu,0}$ using a procedure for ILUPACK analogous to that exposed for the three benchmark kernels combined with linear regression. Finally, we assume that a core promoted to a sleep state does not dissipate any core power.

Consider $P_{pilu,0}^T(c)$ and $P_{bilu,0}^T(c)$ denote, respectively, the total power dissipated during the execution of ILUPACK, using the power-oblivious (polling *pilu*) and power-aware runtimes (blocking *bilu*), with c cores in state P0. Since now some cores may be inactive during a certain part of the execution, we need to adapt (2), which now becomes

$$\begin{aligned} P_{pilu,0}^T(c) = f_{pilu,0} \cdot (P_i^Y + P_i^U + P_{ilu,0}^{C1} \cdot c) \\ + (1 - f_{pilu,0}) \\ \cdot\ (P_i^Y + P_i^U + P_{polling,0}^{C1} \cdot c). \end{aligned} \quad (4)$$

The first term of the addition captures the cost of the cores performing useful work during the computation of ILUPACK (like (2)), and appears multiplied by $f_{pilu,0}$, which corresponds to the ratio of the total time that this computation occupies. Thus, the second part of the addition represents the remaining fraction of the total time, $(1 - f_{pilu,0})$, and captures the power dissipation of the cores performing polling. In our evaluation, we set $P_{polling,0}^{C1} = P_{busy,0}^{C1}$, as the underlying procedures are similar.

On the other hand, for $P_{bilu,0}^T(c)$, we have

$$\begin{aligned} P_{bilu,0}^T(c) = f_{bilu,0} \cdot (P_i^Y + P_i^U + P_{ilu,0}^{C1} \cdot c) \\ + (1 - f_{pilu,0}) \cdot (P_i^Y + P_i^U), \end{aligned} \quad (5)$$

as we assumed that a core in blocking mode wastes no power (i.e, $P_{blocking,0}^{C1} = 0$).

Table 4 compares the values of the theoretical ratios $P_{bilu,0}^T(c)/P_{pilu,0}^T(c)$ with the experimental data (averaged for 10 different executions), showing a very close matching between the two, below 2 %, WT_INT and slightly larger, about 4 % for WT_AMD. These results confirm the benefits of the power-aware runtime, but also the accuracy of the power model. For all other frequencies, as we will see next, the model always predicted the power-ratio with an error below 2 %.

6 Impact of the P-states on ILUPACK

In this section we evaluate the effect of the different P-states available for each processor on the performance-power-energy trade-off of ILUPACK. For that purpose, we set the Linux governor mode to `userspace`, and operate all the cores of the platforms in the same P-state. In the following, we always employ the power-aware version of the runtime. Therefore, we assume that, when idle, a core will remain in one of the deep power-saving C-states (C1 or higher), consuming a negligible amount of power.

The general consensus is that, for a memory-bound computation, some benefit may result from operating the system cores at low frequencies. The reason is that, although there exists a linear dependence between the core performance and the frequency, the effect on the execution time of a memory-bound algorithm should be minor because the key for this type of computation is not core performance but memory bandwidth. On the other hand, for current multicore technology, a reduction of frequency is associated with a decrease of voltage (see Table 1) and, because of the relation between static power to V^2 and dynamic power to $V^2 \cdot f$, in principle we can expect a significant reduction of the power draw. However, the balance between these two factors, time and power, on the energy efficiency is delicate, and other elements also play a role. Whether these variations of time and power yield a loss or a gain for ILU-

Platform	Preconditioner		Iterative solver	
	Theoretical	Experimental	Theoretical	Experimental
WT_AMD	89.58	93.88	91.47	87.36
WT_INT	90.11	90.64	93.65	92.05

Table 4 Expected and observed (theoretical and experimental, respectively) power ratios, in %, between the power-aware implementation of the runtime and the power-oblivious one, with all cores running in state P0.

Platform	P-state, P_i	Preconditioner			Iterative solver		
		Time	Power	Energy	Time	Power	Energy
WT_AMD	P0	66.52	227.17	15,112.06	221.39	237.80	52,656.41
	P1	81.56	197.77	16,131.31	252.54	207.98	52,525.50
	P2	97.16	172.68	16,778.25	288.31	187.14	53,954.38
	P3	113.25	160.16	18,138.44	326.11	176.10	57,426.43
	P4	137.62	151.52	20,852.36	284.34	167.36	64,321.79
WT_INT	P0	54.47	125.70	6,847.47	148.15	150.17	22,247.94
	P1	57.31	119.23	6,833.73	147.16	145.37	21,392.83
	P2	60.65	114.16	6,924.03	151.35	140.75	21,302.70
	P3	65.29	108.95	7,114.07	164.85	132.35	21,819.16

Table 5 Execution time, power and energy of the power-aware implementation of the runtime, with all cores in state P_i.

Platform	P_i/P0	Δf_i	ΔBW_i	Preconditioner			Iterative solver		
				ΔTime	ΔPower	ΔEnergy	ΔTime	ΔPower	ΔEnergy
WT_AMD	P1/P0	−25.00	−18.68	22.60	−12.94	6.74	14.07	−12.54	−0.22
	P2/P0	−40.00	−32.45	46.06	−23.99	11.02	30.23	−21.30	2.48
	P3/P0	−50.00	−42.29	70.24	−29.50	20.02	47.30	−25.94	9.07
	P4/P0	−60.00	−53.78	106.88	−33.30	37.98	73.60	−29.62	22.17
WT_INT	P1/P0	−6.50	−1.10	5.21	−5.15	−0.20	−0.67	−3.20	−3.84
	P2/P0	−13.50	−0.86	11.35	−9.18	1.12	2.16	−6.27	−4.25
	P3/P0	−20.00	−1.33	19.86	−13.33	3.89	11.27	−11.86	−1.93

Table 6 Variations of frequency, bandwidth, execution time, power and energy ratios (%), of the power-aware implementation of the runtime, between state P_i and state P0.

PACK from the point of view of energy efficiency is thus the question to investigate in this section.

Table 5 reports the impact of the P-states on the time, (average) power consumption and energy efficiency of the two stages of ILUPACK, calculation of the preconditioner and iterative solution, on both platforms. To help with the analysis of these results, Table 6 offers the variation ratios of bandwidth and the results that are experienced when moving from state P0 to state P_i, calculated as $100{\cdot}(M_i{-}M_0)/M_0$, where M_0 and M_i denote, respectively, the values of the magnitudes (parameters or results) in states P0 and P_i.

The first aspect to notice is that the presumed independence between execution time and core frequency does not hold on WT_AMD. This should not be a surprise as our experiment in Table 1 already revealed that there is a strong connection between the core frequency and the memory bandwidth on this platform (see also column ΔBW_i in Table 6). The combined decreases of frequency and memory bandwidth when moving from P0 to a higher P-state (between −25 % and −60 % for the former and from −18.68 to −53.78 % for the latter) explain the increases of execution time for both the preconditioning stage (22.60–106.88 %) and the iterative solver (12.54–73.60 %) in this platform. The behavior of

WT_INT is quite different, which is partially explained because now the reduction of frequency does not bring a decrease of memory bandwidth. Still, for the preconditioner, the reduction of frequency when moving from P0 to a higher P-state (−6.50 % for P1, −13.50 % for P2 and −20.00 % for P3), basically matches the increase of execution time for this stage (5.21, 11.35 and 19.86 %, respectively). We can take this as an indicator that the computation of the preconditioner (or, at least, parts of it) is not such a memory-bound computation as one could, in principle, presume. The results are different for the iterative solve. In this case, there is no significant difference in the execution time when running the stage in states P0 or P1, but the time increase when moving from P0 to P2/P3 is 2.16/11.27 %, which is still lower than what could be explained by the reduction of frequency alone.

From the performance point of view, the major conclusion of this analysis is that the best solution is to always run ILUPACK with all the cores operating at the highest frequency (i.e., in state P0), though in some cases —in particular, the iterative solver executed in frequencies P0, P1 and P2—, the differences are small on WT_INT.

Performance is crucial and, under some circumstances, energy efficiency is also vital. From that point of view, a reduction of power is beneficial only if it does not yield an increase of execution time that blurs the positive effects on energy consumption. For the particular case of ILUPACK, the results in Table 5 show that, on WT_AMD, the most energy efficient solution is to execute the preconditioner with all cores in state P0 but the iterative solver in state P1. For WT_INT, however, using states P1, P2 or P3 for the iterative solver results in small significant energy savings, from −1.93 to −4.25 %.

Let us connect again the power variations attained with the different P-states and the models for total power. For this purpose, we relate $P^T_{bilu,i}(c)$ and $P^T_{bilu,0}(c)$, using

$$P^T_{bilu,i}(c) = f_{bilu,i} \cdot (P^Y_i + P^U_i + P^{C1}_{ilu,i} \cdot c) + (1 - f_{bilu,i}) \cdot (P^Y_i + P^U_i), \tag{6}$$

and the experimental data. Table 7 reports the accuracy of our model to capture the experimental behavior due to the variations of the P-state on ILUPACK, with an error at most 3.08 % for WT_AMD and even smaller for WT_INT.

7 Concluding Remarks

We can list two main contributions for this work: *i)* the implementation of an energy-aware runtime for the complete preconditioner+iterative solve process in ILUPACK; and *ii)* the elaboration and experimental characterization of simple models for the (average) power that explain/justify the variations observed for the new energy-aware runtime and the effect of the different P-states for this particular application and two different multicore architectures.

The introduction of the energy-aware runtime results from the experimental observation that, in an energy-oblivious execution of the original runtime for ILUPACK, idle threads with no useful task to execute simply poll till new work is available. As a result, these threads dissipate a significant amount of power in current processors, for no practical performance benefit for the particular case of ILUPACK. Our energy-aware implementation replaces this behavior with a more power-friendly implementation, that blocks idle threads till new work is available. This requires a careful reorganization of the underlying runtime, to avoid deadlocks and ensure a rapid response that does not impair performance. In our experiments, we observed savings in the energy usage between 7 and 13 %, at practically no cost from the performance point of view, which are clearly connected to the impact of the C-states by our power model.

In summary, the approach adopted in this paper is based on the exploitation of idle periods during the concurrent execution of a task-parallel version of ILUPACK for multithreaded architectures. By replacing the busy-waits of the runtime in charge of execution with idle-waits, we favor the introduction of *race-to-idle*, which in turn allows the operating system to promote the hardware into a more energy-efficient C-state. We believe that the same technique can be applied to other task-parallel scientific codes and, as part of ongoing work, we are currently embedding this approach into a general runtime like OmpSs, modified to embrace idle-wait, so that we can evaluate the performance-power-energy trade-offs for other task-parallel applications.

Busy-wait and idle-wait are analogous to well-known concepts of operating systems like spinlock and mutex, respectively. The reason that idle-wait is beneficial for ILUPACK is that the number of changes between busy and idle periods is moderate

Platform	P_i/P0	Preconditioner		Iterative solver	
		Theoretical	Experimental	Theoretical	Experimental
WT_AMD	P1/P0	88.05	85.48	84.17	87.64
	P2/P0	78.96	76.38	76.56	79.65
	P3/P0	73.50	70.78	71.60	74.85
	P4/P0	69.73	66.65	68.77	70.80
WT_INT	P1/P0	95.62	95.54	96.47	96.47
	P2/P0	90.84	90.80	91.25	91.84
	P3/P0	87.21	86.94	85.96	87.77

Table 7 Expected and observed (theoretical and experimental, respectively) power ratios (%), of the power-aware implementation of the runtime, between state P_i and state P0.

and the duration of these periods is "long enough". This depends on a number of factors including not only the number and duration of the periods but, e.g., also the costs in time and energy of blocking/releasing the threads, the costs in time and energy of the changes between different C-states, etc.

In theory, there exists a linear relation between performance and frequency, which could be expected to be even sublinear (at least on the Intel processor) for a presumedly memory-bound computation like ILUPACK, and a quadratic/cubic relation between energy and voltage-frequency. However, the analysis of the time-power-energy trade-off when the cores operate in a certain P-states (voltage-frequency pair), with the energy-aware version of the runtime, reveals the high impact of idle and, to a minor degree, uncore power which clearly favor shorter execution time over lower power dissipation rates. This is also contrasted to and accurately captured by our power model.

Acknowledgments

The researchers from the Universidad Jaume I were supported by project CICYT TIN2011-23283 of the *Ministerio de Ciencia e Innovación* and FEDER and the FPU program of the *Ministerio de Educación, Cultura y Deporte*. A. F. Martín was partially funded by the UPC postdoctoral grants under the programme "BKC5-Atracció i Fidelització de talent al BKC".

References

1. The Green500 list. Available at `http://www.green500.org`.
2. ILUPack. Available at `http://www.icm.tu-bs.de/~bolle/ilupack/`.
3. The Top500 list, 2010. Available at `http://www.top500.org`.
4. Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53:86–96, May 2010.
5. José I. Aliaga, Matthias Bollhöfer, Alberto F. Martín, and Enrique S. Quintana-Ortí. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Computing*, 37(3):183–202, 2011.
6. José I. Aliaga, Matthias Bollhöfer, Alberto F. Martín, and Enrique S. Quintana-Ortí. Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In *State of the Art in Scientific and Parallel Computing – PARA 2010*, number 7133 in Lecture Notes in Computer Science, pages 162–172. 2012.
7. José I. Aliaga, Manuel F. Dolz, Alberto F. Martín, Rafael Mayo, and Enrique S. Quintana-Ortí. Leveraging task-parallelism in energy-efficient ilu preconditioners. In *2nd International Conference on ICT as Key Technology against Global Warming – ICT-GLOW*, number 7453 in Lecture Notes in Computer Science, pages 55–63. 2012.
8. P. Alonso, R. M. Badia, J. Labarta, M. Barreda, M. F. Dolz, R. Mayo, E. S. Quintana-Ortí, and Ruymán Reyes. Tools for power-energy modelling and analysis of parallel scientific applications. In *41st Int. Conf. on Parallel Processing – ICPP*, pages 420–429, 2012.
9. P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Energy-efficient execution of dense linear algebra algorithms on multi-core processors, *Cluster Computing*, 16(3): 497-509, 2013.
10. P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - Research and Development*, 2012. Available online.
11. M. F. Dolz, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí. EnergySaving Cluster Roll: Power Saving System for Clusters. *Architecture of Computing Systems - ARCS 2010*, number 5974 in Lecture Notes in Computer Science, pages 162–173, 2010.
12. AnandTech Forums. Power-consumption scaling with clockspeed and Vcc for the i7-2600K. `http://forums.anandtech.com/showthread.php?t=2195927`, 2011.
13. S. Barrachina, M. Barreda, S. Catalán, M. F. Dolz, G. Fabregat, R. Mayo, and E. S. Quintana-Ortí. An integrated framework for power-performance analysis

of parallel scientific workloads. In *3rd Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2013.
14. M. Duranton and *et al.* The HiPEAC vision for advanced computing in horizon 2020, 2013.
15. M. El Mehdi Diouri, M. F. Dolz, O. Glück, L. Lefèvre, P. Alonso, S. Catalán, R. Mayo, and E. S. Quintana-Ortí. Solving some mysteries in power monitoring of servers: take care of your wattmeters! In *Proc. Energy Efficiency in Large Scale Distributed Systems conference – EE-LSDS 2013*, 2013. To appear.
16. Wu-chun Feng, Xizhou Feng, and Rong Ge. Green supercomputing comes of age. *IT Professional*, 10(1):17 –23, jan.-feb. 2008.
17. FLAME project home page. `http://www.cs.utexas.edu/users/flame/`.
18. S. Gunther, A. Deval, and T. Burton. Energy-efficient computing: Power-management system on the Intel Nehalem family of processors. *Intel Technology Journal*, 15(1), 211.
19. D. L. Hill, T. Huff, S. Kulick, and R. Safranek. The Uncore: A modular approach to feeding the high-performance cores. *Intel Technology Journal*, 14(3), 2010.
20. HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., and Toshiba Corp. Advanced configuration and power interface specification, revision 5.0, 2011.
21. J. Dongarra et al. The international ExaScale software project roadmap. *Int. J. of High Performance Computing & Applications*, 25(1):3–60.
22. J. Kolodziej, S. U. Khan, L. Wang, A. Byrski, N. Min-Allah, S. A. Madani, Hierarchical genetic-based grid scheduling with energy optimization, *Cluster Computing*, 16(3): 591-609, 2013.
23. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. `http://mvapich.cse.ohio-state.edu/`.
24. NVIDIA Corporation. *CUDA toolkit 4.0 readiness for CUDA applications*, 4.0 edition, March 2011.
25. Paraver: the flexible analysis tool. `http://www.cepba.upc.es/paraver`.
26. E. Pinheiro, R. Bianchini, E. V. Carrera, T. Heath. Dynamic cluster reconguration for power and performance. In: Proc. of Workshop on Compilers and Operating Systems for Low Power, pp. 7593, 2003.
27. R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors. In *2012 USENIX conference on Power-Aware Computing and Systems*, 2012.
28. SMP superscalar project home page. `http://www.bsc.es/plantillaG.php?cat_id=385`.
29. The STREAM benchmark: Computer memory bandwidth. `http://www.streambench.org/`.
30. G. L. Valentini, W. Lassonde, S. U. Khan, U. Samee, N. Min-Allah, S.. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej, H. Li, A. Y. Zomaya, C. Xu, P. Balaji and A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, and P. Bouvry, An overview of energy efficiency techniques in cluster computing systems. *Cluster Computing*, 16(1):3–15, 2013.
31. L. Wang, S. U. Khan, D. Chen, J. Kolodziej, R. Ranjan, C. Xu and A. Y. Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Generation Comp. Syst.*, 29(7):1661–1670, 2013.