The final publication is available at ACM via http://dx.doi.org/10.1145/3472456.3472471

Combining Dynamic Concurrency Throttling with Voltage and Frequency Scaling on Task-based Programming Models

Antoni Navarro*, Arthur F. Lorenzon†, Eduard Ayguadé*, and Vicenç Beltran*
*Barcelona Supercomputing Center (BSC), †Federal University of Pampa (Unipampa)
{antoni.navarro,eduard.ayguade,vbeltran}@bsc.es, {aflorenzon}@unipampa.edu.br

***Abstract*—Being on the verge of exascale performance has shifted the prioritization of performance in applications to the inclusion of power-performance efficiency as a primary objective in the High Performance Computing (HPC) community. Simultaneously, this has surfaced hardware and software efforts that employ techniques such as dynamic voltage and frequency scaling (DVFS) for core and uncore units or dynamic concurrency throttling (DCT) to exploit hardware resources efficiently, by saving energy while maintaining performance. These techniques are complementary, so they can be used together. However, employing them is not a straightforward task, as they have to be adjusted based on the workload, and it is even more complex to combine them properly. Thus, these techniques should be applied transparently by a runtime system, without relying on application developers. In this paper, we extend a task-based runtime system with an infrastructure that categorizes workloads based on their computational profile – memory-bounded, compute-bounded, or balanced. This categorization is done in an on-line manner and with a negligible overhead. With this additional information, we enhance the CPU-manager and scheduler of OmpSs-2, a task-based parallel programming model, to automatically combine DVFS and DCT techniques based on workloads. Moreover, we show that our heuristics transparently improve energy efficiency on average by 15% with no significant performance loss and either equal or surpass the energy efficiency of the best static configuration available.**

I. Introduction

High-Performance Computing (HPC) systems have become an essential tool for applications from various scientific and engineering fields. Simultaneously, task-based parallel programming models have emerged as an alternative to allow these applications to exploit the hardware's full potential. However, due to the increasing concern for energy consumption in HPC systems, optimizing performance-energy efficiency has become a primary objective. Hence, from only focusing on maximizing performance, the interest in HPC-related studies has shifted to solutions that maintain application performance while improving energy efficiency. Similarly, HPC-software developers have started offering tools or techniques with which users can either maintain or optimize performance in applications while saving energy.

A critical factor that surfaces the need for such techniques is the inherent limits in applications and systems. When exploiting thread-level parallelism (TLP), the standard practice is to expose as much parallelism as possible and use the maximum number of resources available in the system. Although this often works as expected, there are cases in which performance may suffer due to issues such as off-chip bus saturation and concurrent accesses to shared memory [1], [2]. In such scenarios, smarter resource managing techniques can (*i*) decrease energy consumption by limiting resource usage when off-chip bottlenecks are detected, and (*ii*) improve performance when limiting resources leads to having less contention over shared units.

To tackle energy-performance efficiency, two of the most widely spread techniques are dynamic concurrency throttling (DCT) and dynamic voltage and frequency scaling (DVFS). While DCT strategies artificially limit the number of active cores running an application, a DVFS system enables exploiting resources by either increasing or decreasing their operating frequency. Unfortunately, efficiently employing DCT or DVFS is not a straightforward task due to the number of variables only known at run-time. Furthermore, parallel applications may present different behaviors related to (*i*) their scalability, which affects the number of active threads; and (*ii*) application phases – compute or memory-bounded – which affect the operating frequency and voltage of hardware components. Relying on end-users to employ these techniques can lead to a non-optimal use of resources. Thus, runtime systems should be the ones to integrate such techniques.

In this paper, we propose an infrastructure that categorizes workloads based on their computational profile. This infrastructure decides whether each unit of work being executed is memory-bounded, compute-bounded, or balanced. Such categorization is done on-line, with no required warm-up, and introducing a negligible overhead in executions. Based on this additional information, we extend the resource manager and scheduler of a task-based parallel runtime library with capabilities that combine core and uncore DVFS and DCT techniques. Even though we exemplify our work using the OmpSs-2 programming model [3], its tasking model is similar to the tasking model of other programming models such as OpenMP, thus our proposals can be applied to other parallel programming models. The main contributions of this work are: (*i*) the extension of a monitoring infrastructure to include an online monitoring, task-characterization, and prediction infrastructure; and (*ii*) the efficient combination of both DVFS and DCT techniques based on predictions. When executing several well-known benchmarks and applications on two modern HPC

systems, we show that:

- Our infrastructure can categorize workloads with an average accuracy of 90% and negligible overhead in execution time (4% in the worst case).
- Combining DCT and DVFS techniques for uncore resources can lead to decreasing energy consumption by 15% on average. Furthermore, employing these techniques automatically and adaptively can even lead to an improved energy consumption compared to the best static configuration.
- In some scenarios, performance can also benefit from these techniques, by reducing resource contention or saturation.

The remainder of this paper is structured as follows. In Section II, we describe the background of dynamic techniques used to improve energy efficiency. In Section III, we give a thorough description of the implemented techniques and frameworks of this work. Then, in Section IV, we describe the methodology employed during the evaluation phase and discuss the results of our contributions. In Section V, we discuss the related work and highlight our contributions. Finally, Section VI presents the conclusions and future work.

II. Background

Categorizing application workloads is often a requirement for techniques based on tweaking hardware configurations. To make scheduling decisions, runtime libraries must first characterize the current workload based on its computational profile. In related literature, this is commonly referred to as phase tracking. In this paper, we study different DCT and core/uncore DVFS techniques. Hence, to exemplify our contributions, we require an infrastructure to extract the needed metrics to create a profile of the tasks being currently executed. This module uses hardware counters such as the number of last level cache (LLC) misses to decide how memory-bounded a task is. Furthermore, it records and normalizes timing metrics in order to profile tasks further. To avoid burdening the user with extra steps and to adapt to each application and system, our infrastructure must perform this characterization at runtime, with no off-line training phases. Task-based parallel programming runtime libraries like Nanos6 [4] offer malleability when using hardware resources. The modules in these libraries offer extensibility through policies, enabling the development of heuristics and policies to take smarter scheduling decisions when managing resources. Once workloads are categorized, techniques such as DCT and DVFS can be employed to save energy and even increase the performance of applications that might suffer from resource contention.

DCT is a technique in which the number of available active cores is modified based on heuristics. In research related to power-performance efficiency, such a technique is usually employed when an excessive amount of memory-bounded tasks is being executed. This is so that any extra processors that would have their memory transfer rates limited can be idled to avoid their energy consumption.

Another technique, based on the same principles, is DVFS, which dynamically scales core frequencies based on the computational profile of workloads. For instance, when memory-bounded tasks are detected, the frequency of the processor executing them is lowered to reduce its energy consumption while still keeping it active. Governors that take scheduling decisions for computing resources and DVFS capabilities became available through common drivers in Linux Kernel Version 2.6 [5]. Moreover, as of version 5.6 [5], several drivers enable tweaking the uncore frequency which is directly related to shared resources such as the memory controller.

These techniques are useful in scenarios such as the one illustrated in Figure 1. As shown, once the off-chip bus occupation cap is reached and more active cores are added, the gain in performance starts to decay. In such scenarios, it is common to see adverse effects on performance after a certain number of active cores, as they may generate contention in shared resources. By limiting the number of active cores upon detecting such bottlenecks, users can benefit from more energy-efficient executions that maintain performance. Furthermore, when detecting memory-bounded phases in applications, energy efficiency can be improved by scaling down core frequencies.

III. Dynamic Resource Management Techniques

The primary objective of this paper is to automatically integrate and combine the usage of DCT and DVFS techniques into task-based parallel programming models. We exemplify this with Nanos6, a runtime library that implements the OmpSs-2 task-based programming model. To achieve our goals, we require the integration to be transparent to users, so the burden of employing these techniques does not lay upon them. In this section, we give a thorough description of our monitoring and prediction infrastructure, and the heuristics employed in these techniques.

A. Prediction Infrastructure

Commonly, works that focus on phase tracking – i.e., categorizing workloads – detect and classify different phases that an application is going through. However, this has room

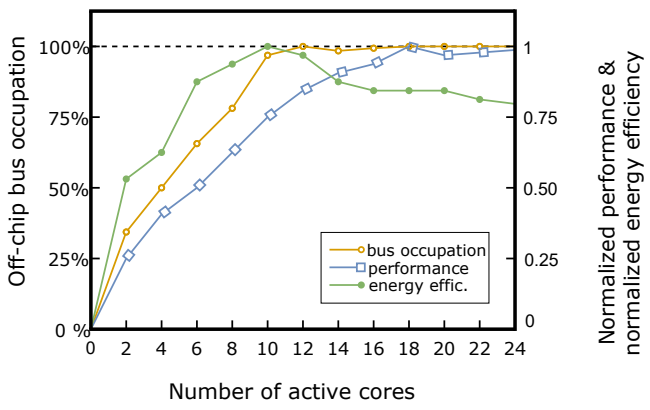

Fig. 1. Off-chip bus saturation VS. performance.

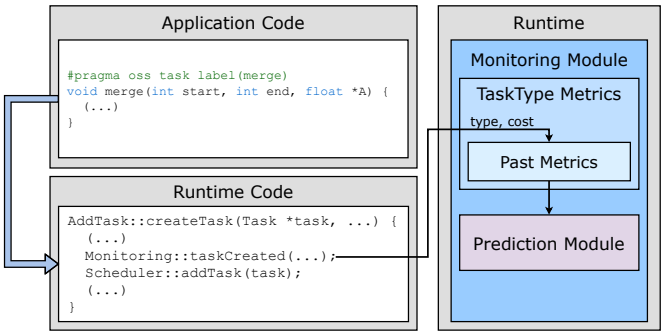

Fig. 2. Monitoring and Prediction Infrastructure.

for improvement since real applications typically combine different phases to exploit these programming models' potential fully. For instance, some applications may combine IO-intensive tasks with compute-intensive tasks to fully leverage the underlying system's potential. For this reason, and since OmpSs-2 is a task-based programming model, the categorization is done at the finest (task) level.

To categorize tasks, we extend our monitoring and prediction infrastructure [6] that collects timing and hardware counter metrics at run-time, and infers these metrics for future tasks. Inferring predictions using basic metric averages may produce adverse effects. For instance, a type of task could behave differently in different application phases depending on variabilities only known at run-time – e.g., the current status of scheduler queues, OS noise, or off-chip bus saturation. To accommodate to these variabilities, we use exponential moving averages [7], [8] to infer metric predictions.

In Figure 2, we show a view of how our infrastructure uses past metrics to infer predictions for future tasks. When a task finishes its execution, its number of LLC misses and elapsed execution time are inserted in a map. This map groups metrics with the respective task type, which is detected and registered in the runtime while the latter is being initialized. Having all of this data accessible at run-time allows inferring metrics for future tasks with the same type.

On the other hand, when a task is created and submitted to the scheduler, the monitoring module receives the task's type. With its type, the module predicts hardware counter metrics using past information from tasks of the same type, stored in the aforementioned map. All this data is fed into the prediction module, which infers timing and resource usage predictions, and accumulates normalized metrics using an exponential moving average. In the event that a task is the first of its type, all the aforementioned modules take no action so that its execution can begin as early as possible.

For a later categorization, to detect if a task x of type y is memory-bounded, its expected memory bandwidth is predicted using: (*i*) past information of LLC misses by tasks with the same type (LLC_y), (*ii*) its predicted execution time ($Time_xy$), and (*iii*) the cache line size of the system (C), as shown in the following equation:

$$MEM\_BW_{xy} = \frac{LLC_y * C}{(Time_{xy})} \quad (1)$$

Prefetching may directly impact memory bandwidth and thus metrics such as the number of LLC misses could not directly account for the entirety of the measured memory bandwidth. Nonetheless, we decided to rule out hardware counters regarding prefetching since our infrastructure is able to correctly categorize any type of task executed in our evaluation.

B. Automatic Detection of System Limits

Once the profile of tasks has been established by predicting their memory bandwidth and timing metrics, we require knowing the system's practical memory bandwidth limit prior to employing dynamic techniques. Knowing this limit allows the runtime to categorize tasks based on their profile. With this categorization, the runtime is able to detect when a task's memory requirements would surpass the system's limit. To do this transparently to software developers, we extended the Nanos6 installation phase with this capability. Once the runtime is installed in any given system, a few scalability tests are launched with several memory-bounded benchmarks. These tests are run with the same input while recording memory bandwidth metrics, with an increasing number of cores.

Once the scalability tests are finished, a script automatically detects two thresholds: (*i*) the number of cores at which any extra added core to the application's execution would not improve performance, and (*ii*) the limit reached in memory bandwidth. The tests produce results such as the ones shown in Figure 3. In this scenario, a *dotproduct* benchmark is ran with an increasing number of cores. The figure shows the application's execution time (in blue, secondary y-axis), the memory bandwidth measured by our infrastructure using Equation 1 (orange series), and the application's theoretical memory bandwidth (purple series, both in the primary y-axis), which is computed taking into account the number of memory accesses, the elapsed execution time of the application, and the cache line size of the system. As shown, after reaching 16 cores (x-axis), the benefits in performance are minimal, and the limit in memory bandwidth – approximately 95000 MB/s – is detected after 20 cores.

With these previously mentioned features, our infrastructure can accurately detect the computational profile of tasks – i.e., compute-bounded, memory-bounded, or balanced – and the moment at which an application is about to reach the limit in memory bandwidth. This provides the runtime system with the necessary information to take smarter resource management decisions.

C. DCT and DVFS Integration

Algorithm 1 shows simplified pseudo-codes of the body of worker threads and the action of polling tasks from the scheduler in Nanos6. At the start of an execution, there are as many active threads as cores in the process' mask.

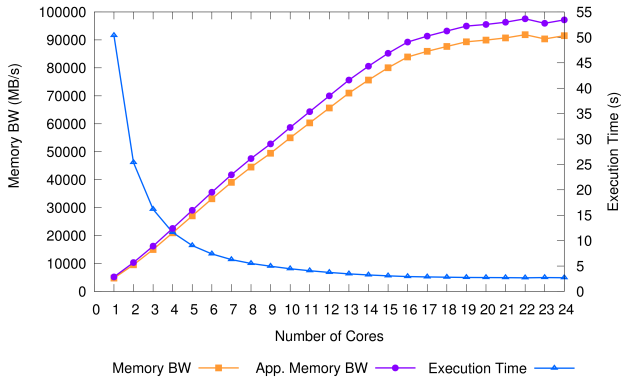

Fig. 3. Dotproduct, executed when installing Nanos6

These regularly poll the scheduler for tasks and ultimately become idle through condition variables if the scheduler does not provide tasks for them. Once a thread informs the CPU manager that it is a candidate to become idle, the CPU manager – based on its current policy – takes an informed decision to idle both the thread and the core it is attached to.

The creation of new tasks triggers the CPU manager, as it may need to resume idle threads based on the underlying policy. Upon being resumed, threads poll the scheduler for tasks. Algorithm 1 shows a simplified pseudo-code of the interaction between threads and the scheduler. Even though the scheduler of Nanos6 [9] is more complex and detailed, we deem it irrelevant for our proposals, as our heuristics can be integrated within the scheduler while ignoring its intricacies.

To enable DCT techniques, we extended the current scheduler and resource manager of Nanos6 to support making informed decisions before resuming CPUs. One of these extensions consists of, upon a task T being created, detecting its memory bandwidth intensiveness by comparing its predicted memory bandwidth ($T_{predicted\_MEM\_BW}$) to the practical memory bandwidth limit per core (*MAX_MEM_BW*), as shown in the following equation:

$$T_{intensiveness} = \frac{T_{predicted\_MEM\_BW}}{MAX\_MEM\_BW} \quad (2)$$

To compute the practical memory bandwidth limit per core, we divide the memory bandwidth limit found with the procedures explained in Section III-B by the number of cores of the system. This gives a rough approximation of the limit per core if all of the cores in a system were to execute a memory-bounded task. Since the intensiveness parameter correlates a task's theoretical memory bandwidth to the limit per core of the system, its value is often between 0 and 1, however it may be larger than 1 as it is not upper bounded. By setting the following thresholds, this parameter informs the runtime whether the task is compute-bounded (if the value is lower than 0.20), memory-bounded (if the value is larger than 0.80), or balanced (if the value is between both thresholds). Both thresholds (compute-bounded and memory-bounded) are established in the runtime installation phase as well, as they may need to be adjusted. Nonetheless, in our work these thresholds remain immutable.

In Algorithm 2, we show the modifications needed in the Scheduler when submitting or polling for tasks. Instead of adding the task to a single queue, we separate them depending on their intensiveness. At all times, the scheduler knows the practical memory bandwidth limit (`MAX_MEM_BW`) and the current accumulated memory bandwidth (`CURR_MEM_BW`). This last parameter is updated when tasks are obtained by threads (increased by their predicted memory bandwidth) and when tasks finish their execution (decreased by their predicted memory bandwidth). When polling for tasks, the scheduler checks the current saturation status of the off-chip bus. If the current accumulated memory bandwidth has reached its predefined limit, only compute-bounded tasks and balanced tasks are returned. If there are only memory-bounded tasks, the scheduler will return none, reducing the number of active cores. In other words, we force the scheduler to enable DCT by forcing the polling thread to idle itself. We believe our categorization improves state of the art DCT techniques by not

Algorithm 1 Threads, scheduler, and resource manager

```
1: function WORKER_THREAD_BODY(...)
2:     while !_shutdown do
3:         task ← Scheduler::get_task()
4:         if task ≠ ∅ then
5:             task.execute()
6:         else                                   ▷ No tasks available
7:             CPUManager::execute_policy(IDLE)
8:         end if
9:     end while
10: end function
11:
12: function SCHEDULER::GET_TASK(...)
13:     if _queue ≠ ∅ then
14:         task ← _queue.get_task()
15:     else
16:         task ← ∅
17:     end if
18:
19:     return task
20: end function
```

Algorithm 2 DCT extension for the scheduler

```
1: function SCHEDULER::GET_TASK(...)
2:     if provider ≠ ∅ then
3:         if CURR_MEM_BW >= MAX_MEM_BW then
4:             task ← _queue_compute.get_task()
5:             if task = ∅ then
6:                 task ← _queue_balanced.get_task()
7:             end if
8:         else
9:             task ← _queue_memory.get_task()
10:            if task = ∅ then
11:                task ← _queue_balanced.get_task()
12:                if task = ∅ then
13:                    task ← _queue_compute.get_task()
14:                end if
15:            end if
16:        end if
17:    else
18:        (...) // Unmodified part of the code
19:    end if
20:
21:    return task
22: end function
```

Algorithm 3 DVFS Prediction Service

```
1: function MONITORING::UPDATE_PREDICTION(...)
2:     while !_shutdown do
3:         if intensiveness < threshold_lower then
4:             mode ← COMPUTE
5:         else
6:             if threshold_lower ≤ intensiveness ≤ threshold_upper then
7:                 mode ← BALANCED
8:             else
9:                 mode ← MEMORY
10:            end if
11:        end if
12:
13:        CPUManager::set_core_freq(mode)
14:        CPUManager::set_uncore_freq(mode)
15:
16:    end while
17: end function
```

only using hardware counter metrics such as LLC misses, but also adding the practical memory bandwidth limit per core to the scope of the heuristic. This allows compute-bounded tasks to be returned by the scheduler even if the application is in a memory-bounded phase.

Further extensions were needed to employ DVFS techniques as well. Taking advantage of the previous classification of tasks and their predicted memory bandwidth intensiveness, we accumulate the intensiveness of the current tasks in an atomic variable: `intensiveness`. As shown in Algorithm 3, we extend the monitoring infrastructure with a service that is executed every 100μs. In this service, we continuously check the accumulated `intensiveness` and compare it to a lower bound ($threshold_{lower}$) and an upper bound value ($threshold_{upper}$). As aforementioned, both of these thresholds are obtained from the runtime installation phase, as they are highly related to the underlying system's features. If the accumulated intensiveness (which is directly related to the limit of memory bandwidth per core) stays near the lower threshold, we are at a compute-bounded phase, thus we reset core frequencies to the default value and set the uncore frequency to the minimum available frequency. Otherwise, if the accumulated intensiveness is between both thresholds, we have a balanced workload, so we set every frequency to its default value. Finally, if the accumulated intensiveness surpasses the upper threshold, we are executing a memory-bounded workload, so we increase the uncore frequency to its default value and we decrease core frequencies. Nonetheless, to avoid adverse effects, upon finding a single memory-bounded task, the uncore frequency is reset to its default value.

Even though DVFS could be employed using all the available operating points, we only use the maximum, minimum, and default values already established by systems. Furthermore, upon setting specific frequencies, these are immutable. In other words, we avoid fluctuating them unless certain scenarios arise (mimicking hysteresis), such as a highly memory-bounded tasks having to be executed while the modules predicted the current workload is compute-bounded. As we further discuss in Section IV, correctly setting the previously mentioned thresholds is of critical importance, as having aggressive thresholds may jeopardize performance.

We believe our heuristics improve those in state of the art as: (*i*) we integrate recently added techniques such as uncore DVFS with other heuristics, (*ii*) the heuristics do not limit any type of workload, instead, they detect memory-bounded tasks and let other compute-bounded tasks pass through unaffected, (*iii*) the infrastructure requires no off-line application categorization nor introduces significant overhead in executions, and (*iv*) we transparently and efficiently combine both heuristics.

IV. Evaluation

In this section, we present the evaluation of our DCT and DVFS heuristics with several benchmarks and on two different multi-core systems. Evaluating DCT techniques does not require any special permission or mode, as the runtimes we evaluated provide malleability in the number of active resources. Nonetheless, to assess the benefits of DVFS heuristics – more specifically, uncore DVFS – we required a recent Linux kernel version (5.6) e.g., to enable drivers that allow employing such techniques. Thus, DVFS results are only showcased for one of the two systems.

A. Experimental Setup

The experiments were performed on two different multi-core systems, as shown in Table I. In the same table, we also show the operating system, the kernel version – i.e., Kernel DVFS shows the version used to evaluate core and uncore DVFS techniques, and Kernel DCT shows the version used to evaluate DCT techniques – and the compilers used in each system. We present all results as the arithmetic mean of five runs for all metrics. Furthermore, we present raw performance and energy consumption metrics to quantify the benefits in energy efficiency while keeping tabs on the overhead introduced in application performance. To retrieve energy consumption metrics, due to the lack of physical energy meters in the used platforms, we used the Intel Running Average Power Limit library [10]. All our experiments had a standard error of less than 2%, with the exception of the HPCCG benchmark when using our combined heuristic, which had a standard error of 5%.

The set of experiments we used includes well-known benchmarks such as Cholesky, Dotproduct, MultiSAXPY, STREAM, NBody, NQueens, and the Gauss-Seidel solver for the **Heat**

TABLE I
Architectures used in our experimental setup

Codename	**MN4**	**SSF**
Processor	Intel Xeon Platinum 8160	Intel Xeon E5-2690 v4
Architecture	Skylake	Broadwell
CPU Freq.	1.0GHz to 2.10GHz	1.20GHz to 3.50GHz
Uncore Freq.	–	1.20GHz to 2.70GHz
# of Sockets	2	2
Threads / Core	1	2
# of Cores	24	28 (14 x 2)
Memory	96 GB	128 GB
OS	SUSE 12.2	SUSE 12.2
Kernel DCT	4.4.12	4.13.10
Kernel DVFS	–	5.8.13
Intel Compiler	19.1.0.166	19.1.0.166
GNU Compiler	9.2.0	9.2.0

TABLE II
OVERHEAD AND AVERAGE ACCURACY OF TIME AND HARDWARE COUNTER PREDICTIONS OF THE MONITORING INFRASTRUCTURE

		Cholesky	Dotproduct	Heat	HPCCG	LULESH	MiniAMR	Saxpy	NBody	Stream
MN4	*overhead*	0.17%	1.00%	3.29%	0.72%	3.36%	0.23%	2.90%	0.16%	0.09%
	acc. time	93.83%	87.91%	91.82%	82.39%	86.70%	91.39%	94.75%	99.82%	78.69%
	acc. hwc	88.97%	99.77%	99.38%	82.24%	74.54%	87.64%	99.83%	48.06%	99.07%
SSF	*overhead*	1.99%	2.03%	2.29%	4.87%	3.17%	0.29%	0.20%	0.48%	0.07%
	acc. time	94.85%	82.13%	87.32%	77.64%	82.19%	88.23%	82.41%	91.02%	95.61%
	acc. hwc	93.01%	91.59%	91.27%	72.02%	71.57%	84.66%	94.55%	45.95%	94.05%

equation simulation. Furthermore, we also use larger applications such as High-Performance Computing Conjugate Gradients [11] (**HPCCG**[1]), **LULESH**[1] [13], and **miniAMR** [14]. The benchmarks of this list can be categorized as (*i*) purely memory-bounded, such as Heat, Dotproduct, MultiSAXPY, and STREAM, (*ii*) purely compute-bounded, such as NBody and NQueens, and (*iii*) balanced, such as Cholesky, miniAMR, HPCCG, and LULESH. Furthermore, all of these benchmarks have been parallelized using tasks, as their task-based versions [15], [16], [17] offer competitive or better performance than the fork-join OpenMP counterpart.

Finally, the evaluation is partitioned into two phases. In the first phase, we measure the overhead and accuracy of the extended modeling and prediction infrastructure of Nanos6. The second phase targets the comparison of all techniques and runtimes – i.e., the vendor implementation of OpenMP in each system (Intel OpenMP - IOMP), the OmpSs-2 baseline version (OSS), and the OmpSs-2 version modified to extend the prediction infrastructure and to include our techniques – showing their respective performance and energy metrics (DCT/DVFS).

B. Overhead and Prediction Accuracy

In Table II, we show for each benchmark and system the overhead of our monitoring infrastructure implemented in Nanos6. Table II also showcases the accuracy of timing and hardware counter predictions, where the values indicate, in average, how close predictions are to the measured metrics. These results are computed using the arithmetic mean of the accuracy of predictions of all task instances. As shown, all the accuracies are close to 90% on average, except for the LULESH, HPCCG, and NBody scenarios. We attribute this to the variability in task granularities and execution times.

For the NBody scenario, as the execution of iterations advance, data may not be found in cache, depending on the application's phase. This produces peaks in L3 cache miss predictions, which explains the lower accuracy values for this application. These inaccuracies could be problematic, as we base the computation of the memory bandwidth prediction on them. Nonetheless, we deemed the low accuracy in this scenario irrelevant, as NBody is not a memory-bounded application. Furthermore, our infrastructure was able to correctly categorize every type of task from all the applications used in our evaluation.

[1]HPCCG and LULESH are implemented using multidependences, available in OpenMP 5.0 [12]. As the Intel 2020.0 compiler does not support them, the *IOMP* results for both of these are using either experimental versions of Intel 2021 or CLANG 12

C. Evaluation of DCT Techniques

In this section, we present and discuss the performance and energy results of our DCT heuristics, and we compare them to the best static configurations per benchmark found. When analyzing each benchmark, we noticed that for most memory-bounded benchmarks the best static configuration when using DCT is to idle at most three to four processors in each system. We base this on the fact that, when looking at the performance to energy ratio plots like the ones shown in Figures 1 and 3, the peak ratio is obtained when idling that number of processors. This is specifically observed in benchmarks such as MultiSAXPY and Dotproduct (DOTP). If the thresholds in heuristics are not adjusted per system – either manually or in an automatic way like ours – idling more processors would have adverse effects on performance, and not idling enough would prevent unlocking more potential in energy consumption improvements.

Figure 4 shows the comparison of performance for the two systems and for all runtimes and DCT heuristics. To offer a fair comparison between versions, we executed each benchmark with all the possible combinations of number of active cores in OmpSs-2. With these experiments, we found the best static configuration that minimizes energy consumption while having no adverse effects on performance. We use the results of these configurations as the baselines of our evaluation (represented by horizontal black lines in each plot), normalizing the results of each runtime to these values.

As shown, the performance of OmpSs-2 using DCT (OSS-DCT) is, for most benchmarks, similar to the performance of the best static configuration. It is important to note that HPCCG and LULESH results for the IOMP variants are included for completeness, but should be viewed with skepticism, as these results are provided by experimental versions of compilers, as these applications had features not included in the Intel 2020.0 compiler OpenMP runtime. In the case of LULESH, there is a slight slowdown in performance of up to 6% for both systems, and for DOTP and SAXPY (memory-bounded benchmarks), there are slowdowns of 16% and 8% in MN4, respectively. We attribute these slowdowns to the aggressiveness of DCT in memory-bounded applications. On

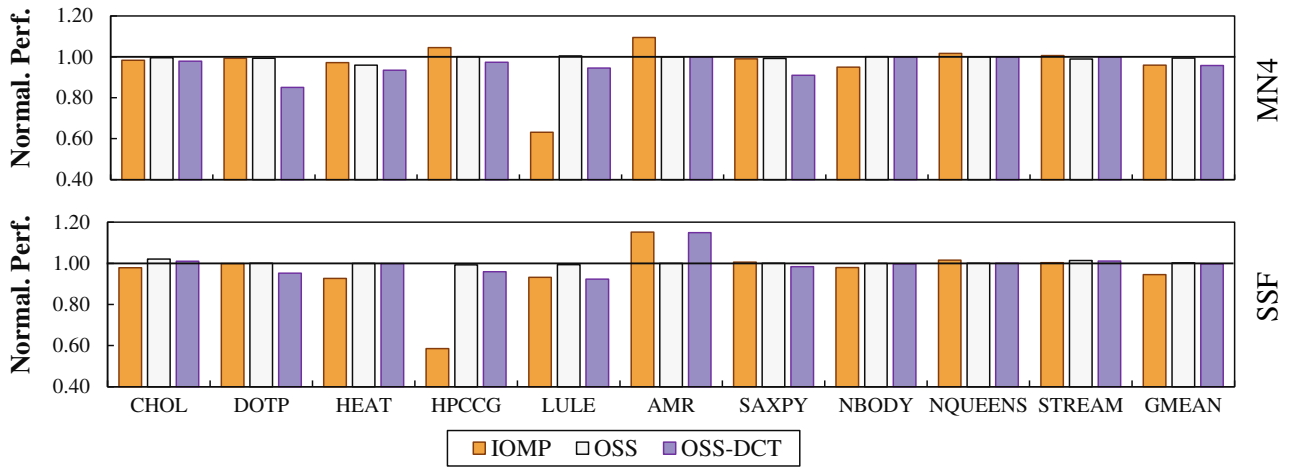

Fig. 4. Performance results normalized to the best static configuration (↑ values = ↑ performance).

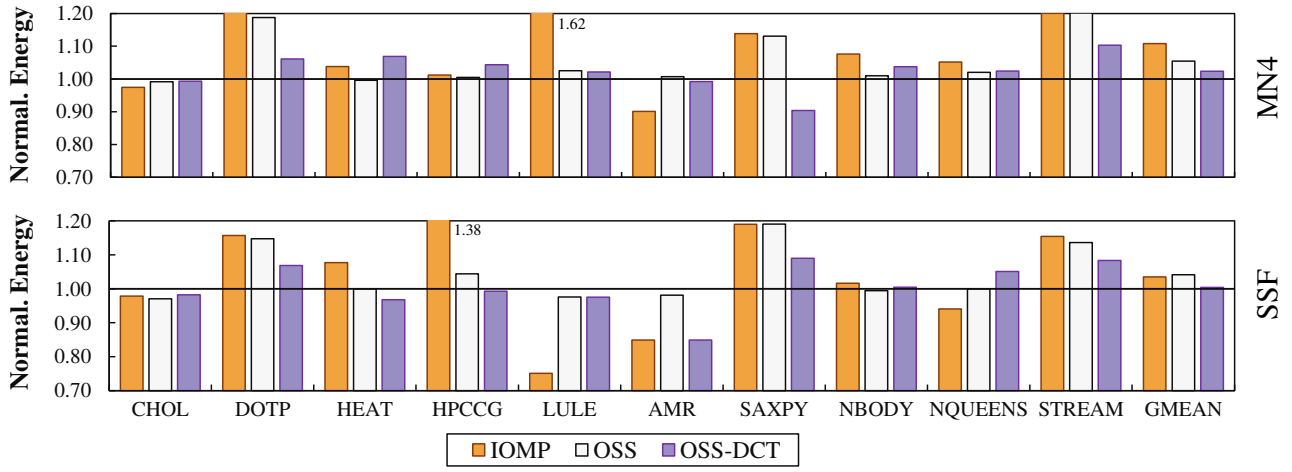

Fig. 5. Energy results normalized to the best static configuration (↓ values = ↑ energy efficiency).

the other hand, for miniAMR (SSF) we notice a performance improvement of 19%. This last improvement is caused by the dynamism of our heuristic which, instead of being bounded to a static number of active cores, may decide to use a different number of cores for each application phase. This specifically helps in the miniAMR scenario, which presents memory-bounded phases in which having all active cores executing memory-bounded tasks ends up hurting performance.

In Figure 5, we show the energy results when employing DCT techniques. These show raw energy consumption data normalized to the energy consumption of the best static configuration. Thus, unlike the performance results, the lower the value, the better the result. In both systems, and for most benchmarks with memory-bounded kernels – DOTP, HEAT, HPCCG, AMR, SAXPY, STREAM – we see energy consumption reductions when compared to the baseline OmpSs-2 scenario. The remainder benchmarks have either the same or a slightly lower energy consumption.

D. Evaluation of DVFS Techniques

To evaluate our DVFS techniques, we required the *intel-uncore-frequency* module, recently added in Linux kernel version 5.6. Thus, we could only analyze our heuristics in the SSF system, as we had privileges to install a more recent version. In Figure 6 we show the performance results of these techniques, replicating the layout of Figure 4. For these results, we compare the vendor OpenMP implementation (IOMP), the baseline OmpSs-2 execution (OSS), a version of OmpSs-2 using core and uncore DVFS techniques (DVFS (uncore + core)), and a version of OmpSs-2 using the uncore DVFS technique in combination with the DCT technique (DVFS (uncore) + DCT). Thus, for memory-bounded phases we use DCT, while for compute-bounded phases we use DVFS. We normalize all the results to the best static configuration, having tested all the possible number of cores (as in the DCT figures) and every available frequency operating point as well.

The last heuristic of our experiments (DVFS (uncore) + DCT) leaves core DVFS out of the scope. Comparing DVFS

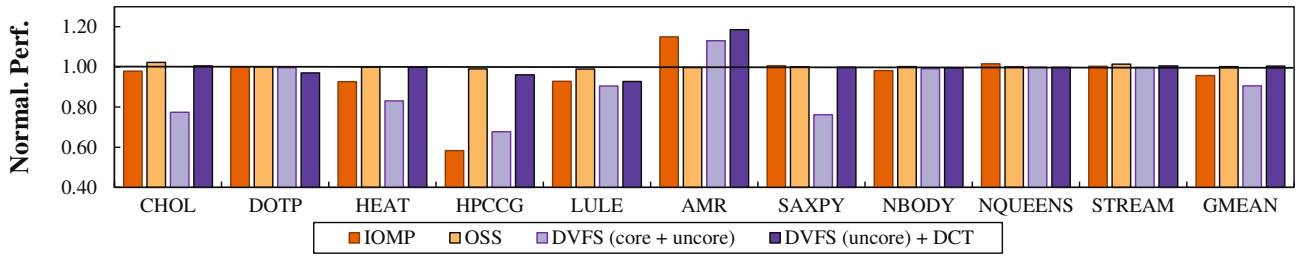

Fig. 6. Performance results normalized to the best static configuration in SSF (↑ values = ↑ performance).

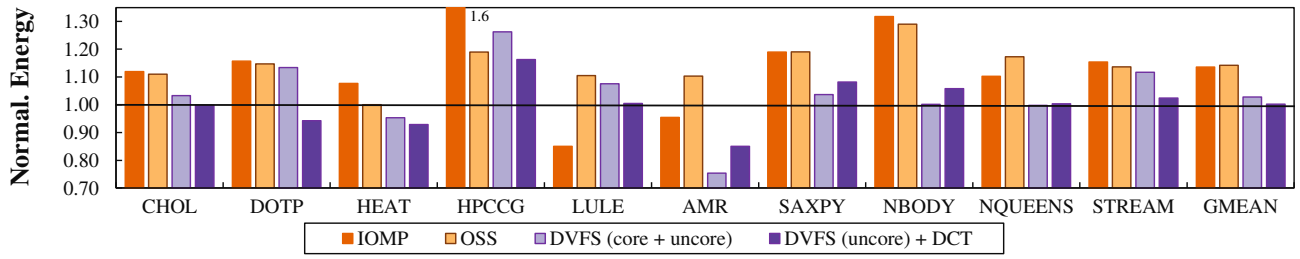

Fig. 7. Energy results normalized to the best static configuration in SSF (↓ values = ↑ energy efficiency).

techniques – to adjust core frequencies – to DCT techniques, we obtained better results when using DCT. When analyzing the behavior of DVFS for cores, there were application phases in which some compute-bounded kernels had to be executed in memory-bounded phases. In these cases, the performance of the application was negatively affected. With that in mind, our final heuristic uses DCT techniques to limit core activity and DVFS techniques to scale down the frequency of off-chip resources.

The overall results for all benchmarks show that performance is similar across all versions. When employing DVFS for both core and uncore units, we see that for HPCCG, MultiSAXPY, Heat, and Cholesky, the performance worsens. Since some of these benchmarks have memory-bounded and compute-bounded phases that intertwine between them, when decreasing the frequency of cores at a memory-bounded phase the performance suffers while executing compute-bounded kernels. For this reason, we include the last series in these plots, where we use DCT for core units and DVFS for uncore units, combining the best of both heuristics in our final heuristic. In miniAMR we notice an increase of performance due to the previously explained reasons for the DCT heuristic. On the other hand, for the combination of DCT and DVFS (uncore) heuristics, we notice how the slowdowns for both Cholesky and MultiSAXPY disappear, as the negative effects of DVFS core are avoided, and for Heat and miniAMR we see improvements in performance. These are the mirrored improvements from the DCT heuristic combined with the benefits from using DVFS uncore.

Once assured that our final heuristic leaves performance unhindered, we include Figure 7, which shows the energy results of our DVFS heuristics. To ease readability we also include Figure 8, which summarizes the energy consumption reductions (percentage) of the DVFS (uncore) + DCT heuristic (Dynamic Heuristic) and best static configuration (Best Static Config) when we compare them to a default Nanos6 execution using its default CPU manager policy. The default policy idles cores as soon as there are no tasks left to execute, and resumes CPUs when new tasks are added to the scheduler queues. This figure showcases a comparison that identified how our combined heuristic performs in comparison to the best static configuration. When obtaining the best static configurations by tweaking frequencies, we noticed that using DVFS uncore can benefit: (*i*) purely compute-bounded benchmarks by decreasing the frequency, but also (*ii*) memory-bounded phases in benchmarks by maximizing the frequency. In this last scenario, energy consumption is reduced by maintaining a static frequency in highly memory-bounded phases. As shown in the figure, for all benchmarks the energy consumption of the combination of heuristics is always lower than the energy consumption of the OmpSs-2 baseline. As seen in Figure 8, for compute-bounded benchmarks, we notice a considerable reduction of energy consumption due to

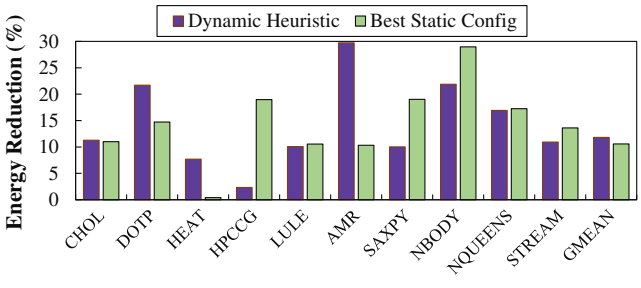

Fig. 8. Energy reduction improvements per benchmark

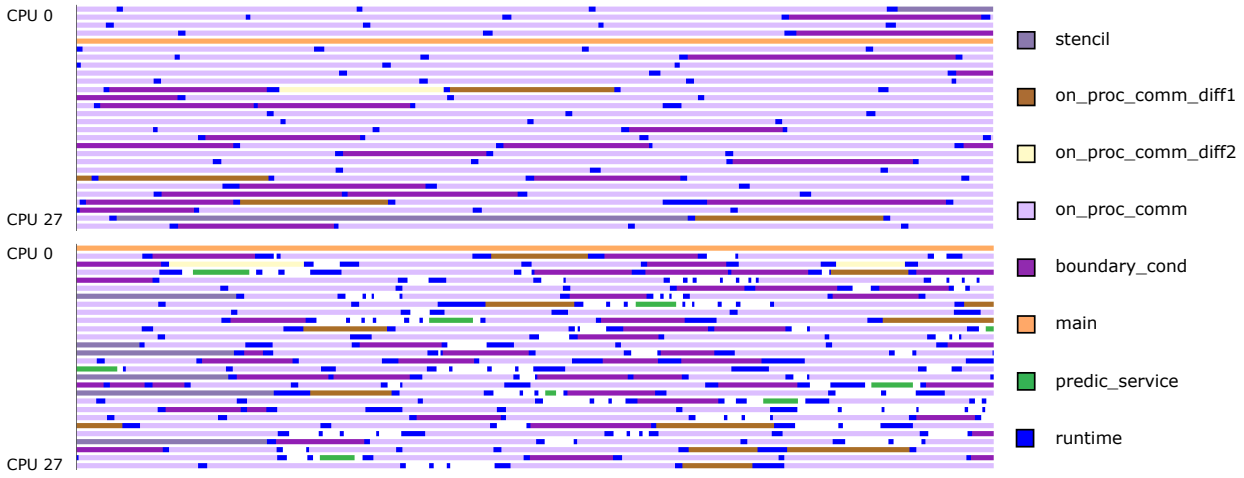

Fig. 9. Zoomed miniAMR Traces – Baseline (Top) vs. DVFS+DCT (Bottom)

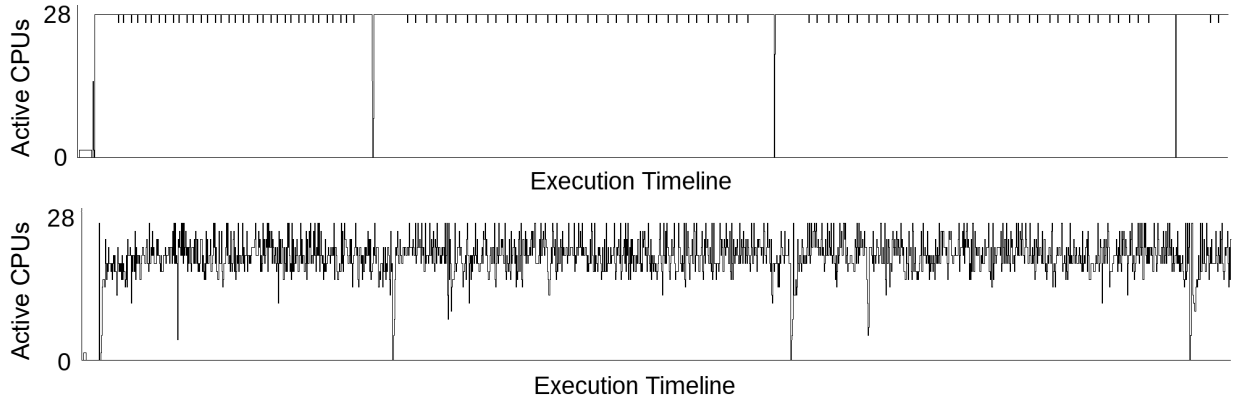

Fig. 10. Active CPUs over time – Baseline (Top) vs. DVFS+DCT (Bottom)

uncore DVFS, more specifically 22% and 17% for NBody and NQueens, respectively. In other benchmarks where compute and memory-bounded phases are balanced (such as Cholesky, Lulesh and HPCCG), DVFS uncore also enables a reduction of the energy consumption, making it identical to the best static configuration by reducing it by 11% and 10% for Cholesky and Lulesh, or bringing it closer for HPCCG, with a reduction of 3%. Lastly, for memory-bounded benchmarks we notice a considerable reduction of energy consumption when compared to the baseline OmpSs-2 version. Specifically the energy consumption reductions are 21% for Dotproduct, 8% for Heat, 30% for miniAMR, 10% for MultiSAXPY, and 11% for STREAM. When comparing both columns, we notice how, in some scenarios (CHOL, DOTP, HEAT, and AMR), the energy reduction of our heuristic surpasses that of the best static configuration. Furthermore, for some other scenarios (LULESH, NBODY, NQUEENS, and STREAM) the energy reduction is close to the best static configuration. Only the HPCCG and SAXPY benchmarks present energy reductions that, although improving the baseline, cannot meet the results obtained by the best static configuration. In both HPCCG and LULESH, we notice how the effect of cross-NUMA accesses hinders our heuristics. Nonetheless, even in those scenarios our final heuristic reduces the energy consumption while leaving performance almost unhindered (10% in the worst case, LULESH).

Even though some these reductions may only account for 10% globally, it is important to notice that in almost every scenario, the energy consumption obtained using these heuristic meets the one reported by the best static configuration. This shows that in average, our heuristics are able to match the energy consumption of the best static configuration, and as previously shown, even improve it in some scenarios. To further illustrate the benefits of our heuristics and analyze where the improvements stem from, in the next section, we include some execution traces showcasing the most important features when comparing our heuristics to the OmpSs-2 baseline scenario in the miniAMR application.

E. In-depth Execution Analysis

To thoroughly analyze where our heuristics' potential comes from, we extracted execution traces of the miniAMR executions in the SSF system, using the baseline OmpSs-2

version and our extended version using the DCT heuristic in combination with uncore DVFS. In these traces, we see at all times which tasks each core is executing, as each execution line corresponds to one of the 28 cores. Dark blue areas (Runtime) represent threads going into the scheduler to poll for tasks, areas that are left blank are areas where cores barely consume energy as they are idle, and the rest of the colors correspond to specific task types from the application, labeled in the legend. The only color missing in the baseline trace is green, which corresponds to the prediction policy service.

In Figure 9 we show a zoomed-in view of part of the execution. In the upper view (baseline), we notice how as soon as threads poll for tasks, they obtain one and begin executing it. However, in the bottom view, we see that when DCT is enabled the scheduler forces cores to become idle and instead execute the prediction policy.

In Figure 10, we showcase a histogram of a portion of the trace, which displays the number of active cores at each time step over the whole execution timeline for both versions, OmpSs-2 (top) and OmpSs-2 using DCT and DVFS (bottom). As shown, for the OmpSs-2 baseline scenario the number of active cores is almost always the maximum available, except when reaching checksum tasks (three times in the trace). On the other hand, in the OmpSs-2+DCT+DVFS scenario the number of active cores keeps adapting to the decisions of our DCT heuristic and the application's current workload at each time step. This enables the performance enhancement and the energy reduction shown in previous figures.

V. Related Work

Categorizing workloads and automatically detecting off-chip saturation limits are topics that have been widely explored in previous work. This has enabled the creation of benchmarks designed explicitly to detect such limits. Concurrently, research that focuses on power-performance efficiency has used these previous topics to showcase benefits in energy with trade-offs in performance.

On the one hand, works such as [18], [19], [20], [21] specialize in detecting workload properties and tracking recognizable patterns, also known as phase analysis. Sherwood et al. [18] propose a model that captures basic-block information of programs and classifies them into different categories w.r.t. a set of past information that is used to predict future behaviors in each program phase. Song et al. [19] also focus on profiling by studying the impact of computation and communication among threads to predict performance and power metrics later. Jayakumar et al. [20] present a prediction framework that matches execution signatures for performance predictions of HPC applications using a single small-scale application execution. Finally, Witkowski et al. [21] use multivariate linear regression to build a model that estimates the power consumption of HPC environments. These works establish the ground for accurate prediction models that enable workload categorization.

On the other hand, dynamic techniques that use phase analysis to improve performance, energy efficiency, or both, are proposed in [22], [23], [24], [25], [26], [27], [28], [29], [30]. Some of these, such as [22], [23], [24], [25], rely on an off-line training phase to later categorize applications for their techniques and algorithms.

Li et al. [22] propose an energy-saving library for hybrid MPI / OpenMP applications. Chetsa et al. [23] present a principal component analysis phase to characterize applications according to their workload behavior. Later, at run-time, per-core frequencies are adapted to save energy. Similarly, Curtis-Maury et al. [24] describe a resource manager that relies on off-line training to optimize the performance and power utilization of HPC systems. Patki et al. [25] propose a framework that also relies on an off-line phase to predict the application behavior to adapt its execution at run-time.

All of these previous works use either DVFS or DCT to improve energy efficiency while prioritizing performance. Nonetheless, they are based on off-line training phases. On the other hand, there have also been numerous efforts [26], [27], [28], [29], [30] that combine the detection of patterns in an on-line manner with DVFS and DCT.

Conductor [26] is a runtime system that dynamically selects the ideal number of threads and per-core frequency to improve performance under a power constraint. However, users must modify source codes to insert specific functions in order to help the runtime. Isci et al. [27] present an on-line phase prediction technique to optimize energy efficiency through DVFS. However, their proposals do not showcase the impact on performance.

LIMO [28] is a dynamic system that monitors applications at run-time and adapts their execution accordingly. However, it requires hardware modifications. Similarly, Porterfield et al. [29] propose an adaptive run-time system that automatically adjusts the number of threads based on on-line measurements of system resource usage. However, they focus on OpenMP-based fork-join applications. Nornir [30] is an algorithm that studies different configurations per application to derive the most performant variants and the variants with less energy consumption. Their targets differ from our objectives, as our approach is to find the most energy-efficient variant that introduces no overhead in performance. Curtis-Maury et al. [31] expand on previous work by proposing an on-line prediction system that combines both DCT and DVFS at the core level for OpenMP applications, in order to optimize energy consumption.

Distinctively, there have also been other works that tackle the same objectives from a different standpoint. For instance, the authors of [32] create specifically-designed cores that target energy-efficiency rather than performance. They achieve their objectives by using reconfigurability from the hardware point of view, rather than the conventional software approach. M. Etinski et al. [33] propose a job scheduling policy to maximize performance under a given power budget. Their work shows benefits in performance and energy. However, to achieve accurate predictions, users must define the impact of frequency scaling per job.

There have also been studies [34], [35] that take advantage

of uncore frequency scaling. S. Vaibhav et al. [34] propose power and performance models to use core/uncore frequency scaling aiming to reduce energy consumption. However, it relies on an offline phase to build the models and does not employ DCT techniques. Similarly, N. Gholkar et al. [35] present UPSCavenger, a runtime technique that dynamically detects application phases and selects the ideal uncore frequency. However, it does not employ DVFS for core frequencies nor DCT. Hence, to the best of our knowledge, our work is the first that proposes an on-line prediction infrastructure with negligible overhead that enables DCT techniques combined with core and uncore DVFS techniques.

VI. Conclusions & Future Work

Transparently employing DCT and DVFS techniques can be a hurdle without the proper information available at run-time. Furthermore, the recent addition of DVFS-enabling drivers in off-chip resources and co-processors increases the difficulty as all these techniques should be combined. Current studies either focus on a single technique or try different techniques in a single type of resource. Moreover, some even require off-line training phases. This paper proposes an on-line modeling and prediction infrastructure that categorizes workloads at run-time with negligible overhead in application performance. We use this information to dynamically modulate the kind of workload allowed onto a CPU and the number of active CPUs (e.g., combine DVFS and DCT techniques at the uncore and core level). Our analysis shows that combining these techniques can provide up to 15% better energy efficiency on average, meet the energy efficiency of the best static configuration available, and sometimes improve performance in applications. Furthermore, we observed that using DCT for memory-bounded phases provides more benefits than using per-core DVFS. Finally, our findings can be summarized as an effective heuristic, which combines: (*i*) a DCT heuristic for memory-bounded phases, and (*ii*) an uncore DVFS heuristic for compute-bounded phases.

One of the contributions of this work is the seamless usage of DCT and DVFS techniques for core and uncore units. However, heterogeneous systems could benefit from runtime systems that employ these techniques at the accelerator level. Thus, we leave employing these techniques in accelerators as future work. Furthermore, we plan to extend our techniques by scheduling tasks from different processes based on their category. This would be specifically beneficial for hybrid applications in multi-node scenarios. Nonetheless, targeting executions with multiple nodes will require an extension of our infrastructure to identify communication tasks. Moreover, DVFS for uncore units would need re-consideration, as it could impact other elements such as network interfaces. This research has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement N.955606 (DEEP-SEA), and is supported by the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB), and by the Generalitat de Catalunya (2017-SGR-1414). This work was also supported by Project HPC-EUROPA3, with the support of the EC Research Innovation Action under the H2020 Programme.

References

[1] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 277–286, 2008.

[2] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*. NY, USA: ACM, 2012, pp. 223–234.

[3] Barcelona Supercomputing Center, "OmpSs-2 Specification," accessed: 2020-10-19. [Online]. Available: https://pm.bsc.es/ompss-2-docs/spec/

[4] ——, "Nanos6 Official Gib Repository," accessed: 2020-10-19. [Online]. Available: https://github.com/bsc-pm/nanos6

[5] I. Linux Kernel Organization. The linux kernel archives. [Online]. Available: https://www.kernel.org/

[6] A. Navarro, S. Mateo, J. M. Perez, V. Beltran, and E. Ayguadé, "Adaptive and architecture-independent task granularity for recursive applications," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer Int. Publishing, 2017, pp. 169–182.

[7] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Boston: Addison-Wesley, 1997, p. 232.

[8] T. Chan, G. Golub, and R. LeVeque, "Algorithms for computing the sample variance: Analysis and recommendations," *The American Statistician*, vol. 37, pp. 242–247, 1983.

[9] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, "Advanced synchronization techniques for task-based runtime systems," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 334–347.

[10] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *ISLPED*. NY, USA: ACM, 2010, p. 189–194.

[11] M. A. Heroux, "Mantevo 3.0 overview." 1 2015. [Online]. Available: https://www.osti.gov/biblio/1513939

[12] OpenMP Architecture Review Board, "OpenMP API. Version 5.0," November 2018, accessed: 2020-10-19. [Online]. Available: https://www.openmp.org/

[13] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[14] A. Sasidharan and M. Snir, "Miniamr - a miniapp for adaptive mesh refinement," Tech. Rep., 2016.

[15] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, and J. Labarta, "Integrating blocking and non-blocking mpi primitives with task-based programming models," *Parallel Computing*, vol. 85, pp. 153 – 166, 2019.

[16] M. Maroñas, X. Teruel, J. M. Bull, E. Ayguadé, and V. Beltran, "Evaluating worksharing tasks on distributed environments," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 69–80.

[17] K. Sala, A. Rico, and V. Beltran, "Towards data-flow parallelization for adaptive mesh refinement applications," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 314–325.

[18] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Int. Symp. on Computer Architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, p. 336–349.

[19] S. L. Song, K. Barker, and D. Kerbyson, "Unified performance and power modeling of scientific workloads," in *Int. Workshop on Energy Efficient Supercomputing*, ser. E2SC '13. NY, USA: ACM, 2013.

[20] A. Jayakumar, P. Murali, and S. Vadhiyar, "Matching application signatures for performance predictions using a single execution," in *IEEE Int. Parallel and Distributed Processing Symposium*, 2015, pp. 1161–1170.

[21] M. Witkowski, A. Oleksiak, T. Piontek, and J. Weglarz, "Practical power consumption estimation for real life hpc applications," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 208 – 217, 2013, including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.

[22] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid mpi/openmp power-aware computing," in *IEEE Int. Symp. on Parallel and Distributed Processing*, 2010, pp. 1–12.

[23] G. L. T. Chetsa, L. Lefevre, J. Pierson, P. Stolf, and G. D. Costa, "Application-agnostic framework for improving the energy efficiency of multiple hpc subsystems," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 62–69.

[24] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 157–166.

[25] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. de Supinski, "Practical resource management in power-constrained, high performance computing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 121–132.

[26] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained hpc applications," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer Int. Publishing, 2015, pp. 394–408.

[27] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *IEEE/ACM Int. Symp. on Microarchitecture*, 2006, pp. 359–370.

[28] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (LIMO): Controlled parallelism for improved efficiency," in *CASES*, USA, 2012, pp. 141–150.

[29] A. K. Porterfield, S. L. Olivier, S. Bhalachandra, and J. F. Prins, "Power measurement and concurrency throttling for energy reduction in openmp programs," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, May 2013, pp. 884–891.

[30] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, Dec. 2016.

[31] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 250–259.

[32] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, p. 205–218.

[33] M. Etinski, J. Corbalan, J. Labarta, and M. Valero, "Optimizing job performance under a given power constraint in hpc centers," in *Int. Conf. on Green Computing*, 2010, pp. 257–267.

[34] S. Vaibhav, S. Masha, W. Bryce, and G. Mark, "Core and uncore joint frequency scaling strategy," *Journal of Computer and Communications*, vol. 6, no. 12, Dec. 2018.

[35] N. Gholkar, F. Mueller, and B. Rountree, "Uncore power scavenger: A runtime for uncore power conservation on hpc systems," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019.