2012 Second International Conference on Cloud and Green Computing

Energy Efficient Computing through Productivity-Aware Frequency Scaling

Lesandro Ponciano, Andrey Brito, Lívia Sampaio, Francisco Brasileiro
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Brazil
Emails: {lesandrop, andrey, livia, fubica}@lsd.ufcg.edu.br

***Abstract*—This paper proposes a new policy for dynamic frequency scaling: productivity-aware frequency scaling (PAFS). PAFS aims at optimizing energy consumptions while still satisfying performance requirements of a given application. In contrast to the commonly-used ondemand frequency scaling, PAFS may keep the processor in a power save state even in high CPU-usage situations. This will be the case as long as the application (or set of applications) for which productivity is to be preserved presents acceptable performance (e.g., as stablished by a QoS contract). Our experiments show savings of up to 23.65% in energy consumption when compared to the commonly used ondemand DFS policy with no performance degradation for the productivity metric. PAFS is, therefore, binded to a single or a set of applications running in a machine. Nevertheless, compared to previous approaches to application-specific frequency scaling, PAFS does not require modifying the application or a calibration process. PAFS requires only a productivity metric which may already be exported by an application (e.g., through a log file, such as response time or throughput in an Apache webserver) or which may be computed through a simple program or script.**

***Keywords*-power-efficient computing, green IT, quality of service, frequency scaling**

I. Introduction

Nowadays, energy efficiency is one of the key challenges in design and management of computing systems. Decreasing costs for hardware and increasing concerns about operating costs and environmental impacts have motivated a huge growth in research on energy efficient systems.

Processors are responsible for most of a system's power consumption. For example, in our DELL R410 servers, a simple CPU-intensive application (no memory, network, or disk usage) changes the total machine consumption from 103 watts (idle) to 253 watts (100% CPU utilization). Therefore, one widely-used mechanism to reduce processors' power consumption is dynamic frequency scaling (DFS) [1], [2], which consists in changing processor clock frequency in real time.

Reducing processor frequency may reduce power consumption, as the lower the processor frequency the lower its power consumption [3]. On the other hand, except in very specific cases (*e.g.*, due to contention issues, as we show later), reducing processor frequency will cause a negative impact on applications' performance. Thus, the major issue on DFS effectiveness is the policy used to decide in which frequency the processor should operate at each time. Usually, frequency scaling is performed by policies at compiler/application-level [4]–[6], operating system-level [7]–[10], or task-level [11]–[16].

Application-level policies define DFS at application design time. This approach allows optimizing both processor power consumption and application performance, but it requires the offline profilling and tuning of the application. System-level policies, in turn, do not consider application characteristics, and they change the processor frequency in response to variations on the system load. However, not knowing applications characteristics, system-level DFS will never achieve savings comparable to the ones achieveable with application-level tunning.

Finally, task-level policies are between these two approaches. This type of policy carries outs DFS taking into account some performance indicator (*e.g.*, service-level agreements – SLA, and application service level objectives – SLO). The present work focuses on task level DFS. In contrast to existing task-level approaches, ours generalizes these approaches and enable users to define custom targets with the same policy.

In this work we propose productivity-aware dynamic frequency scaling (PAFS), which is a DFS policy decoupled from both the application and the operating system. It changes processor frequency based on a *productivity metric*, which consists on one or a combination of several performance metrics (*i.e.*, a utility function [17]). PAFS does not require any offline tuning and aims at achieving energy savings and productivity without assuming any prior knowledge about application workload.

Our evaluation method is based on experimental analysis. We compare PAFS with other policies that focus on power saving, maximum system performance, and load-dependent performance. We evaluate several scenarios varying the productivity metric (response time, throughput) and the frequency scaling policy. We also evaluate CPU, I/O-intensive, and hybrid workloads.

Our main contributions are:

- We introduce productivity as a generalization for what good performance means. Furthermore, based on this principle, we propose productivity-aware frequency scaling (PAFS), a policy that aims at achieving energy savings and satisfying performance requirements without offline profiling and tuning, or assuming prior knowledge about application workloads;

- Our system enables binding savings and performance guarantees (*e.g.*, quality of service – QoS – guarantees) to keep the system in the most energy-efficient mode while meeting required performance levels (independently of the actual CPU load levels), but still is able to quickly react as situations change.
- We show that by achieving energy savings and productivity requirements, PAFS complements the range of options for frequency scaling policies: ondemand, performance, and powersave.

In the remainder of this paper, before detailing the experimental setup and presenting our results (Section IV), we review the relevant background and related work (Section II), and present our PAFS policy (Section III).

II. Background and Related Work

In this section, we provide some background information on how DFS is supported by modern hardware and discuss related work on frequency scaling techniques.

A. Background

Modern processors can run at a range of clock frequencies, for example, using Intel's SpeedStep [18] and AMD's Cool 'n' Quiet [19] technologies. Dynamic frequency scaling is a mechanism that allows scaling processor frequency by software instructions. This mechanism enables the reduction on power consumption by lowering the processor frequency (with a potential negative impact on system's performance). The power consumption in a processor is a nonlinear function of the operating frequency and voltage. Nevertheless, voltage and frequency are related; it is not possible to put the processor in a high frequency state without also increasing its supply voltage. Therefore, when frequency is changed, voltage is automatically changed to match requirements.

Adjusting the frequency (and power management in general) can be made through a platform-independent interface named Advanced Configuration and Power Interface (ACPI) [20]. Regarding frequency scaling, ACPI defines power-performance states (P-states). These states vary between P_0, the highest-performance state, and P_n the lowest-performance state.

B. Related Work

Processor frequency scaling has been studied from different perspectives and with different goals. According to the level at which frequency scaling is applied and whether performance metrics are considered or not, the studies may be broadly divided into three categories: application/compiler-level [4]–[6], task-level [11]–[16], and system-level [7]–[10].

Application/compiler-level policies perform DFS at application-level or with some compiler support, focusing on a specific infrastructure, performance metrics, and/or energy saving goals. For example, the Intel Energy Checker SDK [4] is an API to help constructing green software by exporting application progress metrics and importing energy consumption measurements (from hardware meters). Thus, the energy-efficiency policies can be implemented in the application using consumption measurements imported from the meters. The advantage of this approach is that frequency scaling decisions are aware of application current and future behavior, such as loops, recursive calls, message exchanges, and deadlines, allowing a more accurate DFS. On the other hand, the developer must be aware of energy consumption at the application design [4]–[6], or use a specific compiler to help on this task [21]. In other words, in design time both performance and energy need to be considered. In our approach, energy considerations are automated in run time.

The task-level category, in turn, comprehends policies that are not coupled to the application code, but are aware of running applications. Works in this category perform DFS taking into account some performance indicator (*e.g.*, application deadlines [13], [14], service-level agreements – SLA [12], [15], and application service level objectives – SLO [16]) or prior knowledge about the characteristics of the infrastructure workload (*e.g.*, resource utilization [11]). In general, the two main differences between our DFS approach and related works are that (i) we generalize the concept of performance in a productivity metric; and (ii) we do not require any prior information on system's behavior. Furthermore, our approach allows using the same policy with different productivity metrics, which are defined according to the user requirements.

System-level strategies perform DFS at the operating system without considering any application characteristics. A number of system-level policies can be found in the literature [7]–[10]. Particularly, some of them are broadly used in today's production systems based on both Linux and Windows [7], [8], namely: *Performance*, *Powersave*, and *Ondemand*. The Performance policy focuses on maximizing the application performance by setting the CPU to run at the highest supported frequency. At the opposite side, the Powersave policy focuses on minimizing the power consumption by setting the CPU to run at the lowest supported frequency.

Lastly, the Ondemand policy adapts the CPU frequency to the current system load. According to this policy, the system load is checked periodically, and, when the load rises above a predefined threshold, the CPU is set to run at the next higher frequency. Otherwise, if the load falls below another threshold, the CPU is set to run at the next lower frequency. In this work we compare our PAFS policy with these policies.

III. Productivity-aware Frequency Scaling

As mentioned previously, the main contribution of PAFS is to empower users to dynamically control power consumption based on application's productivity needs, for example, achieving desired QoS levels, but not unnecessarily exceeding them (in contrast to system-level policies), while still not requiring applications to be constructed specifically for that purpose (in contrast to application-level policies).

The abstraction that isolates the application and the performance control is the productivity, a user-defined metric. We start by presenting the control algorithm used by PAFS to actuate in the system and, then, we discuss how meaningful metrics can be defined in a straightforward manner.

A. Optimizing system consumption

The PAFS aims at regulating system's power consumption to operate in acceptable levels of performance-consumption trade offs. Therefore, users may save energy by specifying a performance level that is no greater than what is required to achieve the required QoS level in the medium term. In this case, a *contract* is written with two parameters: one for the application productivity metric and one for the upper bound on the number of performance faults. For example, for a web server the value of the productivity metric could be related to the response time and the fault ratio would be the maximum acceptable percentage of requests for which the response time exceeds the required level.

The system starts at its highest available frequency. If productivity is already worse than the one established in the contract, the system will never be put in a lower performance state. This is an indication that the contract defined by the user does not make sense. However, if the contract is feasible the system will be satisfying the desired productivity levels (at least while operating at its maximum performance) and PAFS will be allowed to actuate.

When in operation, PAFS will periodically (with periodicity ΔT) decrease the processors clock frequency (and, consequently, power consumption) in steps (ΔF) until a performance fault happens (*e.g.*, the servicing of a request exceeds the contracted response time). At this point the control algorithm will increase the processor's clock frequency, again in steps, until it detects that response time is back to satisfactory levels. The control algorithm will then try to decrease the processor's frequency only when the number of performance faults do not exceed the percentage allowed by the contract (*i.e.*, the algorithm keeps state).

The simple approach defined above will lead to a system that is naturally unstable. In fact, instability is a necessary feature as application performance (and thus, the productivity metric) is inevitably related with the workload profile, which is normally dynamic.

Finally, to fully define the control algorithm, the values of ΔT and ΔF must be determined. Unfortunately, these parameters cannot be at the same time statically defined and optimal. Choosing ΔT is particularly hard. Different applications will have different dynamics. Consider, for example, two different applications as following. In the first application the metric is updated only every few seconds. For this application, a small ΔT will make the system react before the impact of the previous action is reflected in the metrics and the system will, consequently, take a series of inadequate actions every time a change happens. In the second application, which is subjected to erratic workloads, a large ΔT will make the system blind to request bursts, that will be badly served. As a consequence, the fault rate will become very high and, after the burst, when load is much lower, the system will be consistently (and wastingly) put in a high performance state to minimize the risk of additional faults.

For the aforementioned reason, we consider an approach that is very simple, but which has proved useful in our evaluation. The approach is the same as the one employed by the Ondemand policy in standard systems: ΔT is fixed, but may be manually changed and ΔF is the smallest step in the list of processor enabled frequencies. Later we will discuss improvements that can be made to provide an adaptive behavior to PAFS.

B. Measuring productivity

In the previous section we assumed the existence of a productivity metric. For example, consider an Apache web server [22]. This server will log every request served. Therefore, one simple way to start defining a throughput metric is using the size of the log file. The user may then write a script that periodically reads the file size and execute some computation (e.g., subtract the previous size from the current) and write the resulting value to a special file that will be periodically read by PAFS.

Writing a script is also useful in many other cases, for example, the user may consider very different types of metrics: (i) count only real requests logged; (ii) consider response time, which is also available in Apache, but needs to be read from the actual log messages (note that in this case, the user would have to specify that lower is better or simply invert the value, depending on how the contract is written); (iii) actually probe the service being ran to measure performance; (iv) consider dynamic rules (*e.g.*, if there are more gold clients in the log, performance should be favored); (v) combine any mix of parameters and metrics from the application (e.g., response time), the environment (e.g., dynamic power costs), and the machine (e.g., actual power) to compute the metric.

If QoS needs are not strict, one way of specifying a contract is by executing an experiment at the highest performance level (in order to get realistic upper bounds on productivity values) and writing a contract that considers a slightly lower QoS (to provide some maneuvering room for power saving). Through this approach, the user is stating that a performance loss is acceptable. Perhaps surprisingly, we show later that specifying a contract with a productivity level that is the same as the highest productivity level achievable in the highest performance, already enables considerable savings. This is a consequence of allowing (through an acceptable performance fault rate) the system to constantly verify if CPU speed is really a necessary resource for that metric at that time.

Similarly, if users set a productivity metric that is not related to CPU usage, such as network latency, PAFS may increase the CPU frequency to a higher level trying to achieve a better performance. However, this behavior will not increase power consumption because if the CPU remain idle, an ACPI CPU sleeping state (C-states [20]) will be automatically used.

IV. Evaluation

In the following sections we describe the testbed, describe the application scenarios we considered to evaluate our PAFS policy, and present the experimental results.

A. Experimental setup

Our server is equipped with two six-core Xeon 3.06 GHz processors, 24 GB of DRAM memory and two gigabit network cards. The processor supports 13 frequency steps between 1.6 GHz and 3.06 GHz (Intel's SpeedStep technology). We use the power meter embedded in the management interface (DELL iDRAC 6) to measure power consumption in real time. We run PAFS on a Linux 3.2.0-2 kernel, which supports the standard DFS techniques (Powersave, Performance, and Ondemand). PAFS controls the processor frequency using Linux's CPUFreq subsystem.

We chose to analyze the effectiveness of our policy in a web server environment because this environment presents several challenges in terms of both QoS level and energy savings [23]–[25]. We implemented three web services with different profiles regarding resource utilization: CPU-intensive, I/O-intensive, and a hybrid version. These services were implemented using the popular LAMP (**L**inux, **A**pache, **M**ySQL, and **P**HP) software bundle. This software runs in background during all experiments.

When a request is received, the CPU-intensive service generates 40 random IDs and searches for them in a small MySQL database (50,000 records). By monitoring resource usage, we confirmed that CPU was the bottleneck during the execution of this service. The I/O-intensive service performs 4 random seeks on a 8.3GB text file per request received. In this case, we confirm the disk was a bottleneck by noting that an increase in the number of seeks would cause Apache to time out on many requests. Finally, the hybrid service uniformly mixes the two types of computations.

We use the Apache Benchmark [26], which runs on a different machine, to generate the HTTP requests to these services. The Apache Benchmark can be configured regarding both the number of total requests and the number of concurrent requests. Each experiment considers $500,000$ requests with 30 concurrent requests at a time.

B. Results

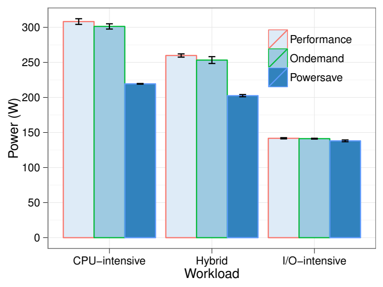
We designed our experiments to measure the effectiveness of PAFS in balancing energy savings and productivity requirements under three scenarios. The first scenario evaluates system level policies in order to put the opportunity for saving power into perspective. The second and third scenarios compare the best system level policy with PAFS when the productivity metric is defined to be throughput and response time, respectively. Each experiment in each scenario considered 20 Apache Benchmark executions in order to achieve results with a statistical error with confidence levels of 95%.

Scenario 1: In this scenario, we evaluate energy consumption and the productivity metrics with the standard power management policies: Performance, Powersave and Ondemand. This scenario enables us to analyze the trade offs between energy saving and system productivity when frequency scaling focuses on (i) maximizing only application performance (Performance policy), (ii) maximizing only server power reduction (Powersave policy), (iii) dynamically adjusting frequency based on processor usage (Ondemand). Thus, this scenario aims at putting the opportunity for saving power into perspective by taking maximum system productivity into account. The results from our first set of experiments are depicted in Figure 1.

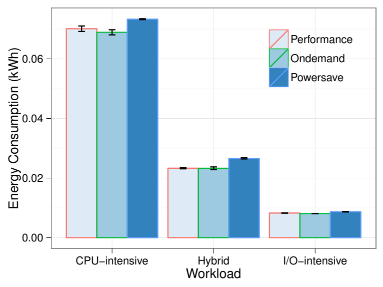
In each figure, the horizontal axis includes the three service profiles considered. Figure 1(a) depicts the average power consumed during the experiment. Figure 1(b) depicts the total energy consumed for all $500,000$ requests, which depends on the power (in Watts) and the total duration (in hours). Note that a lower power consumption does not necessarily means energy efficiency as the system may have to run for a longer period in order to process the same amount of requests. Figures 1(c) and 1(d) depict the per-request average service time and average throughput.

As we expected, the policies evaluated do not significantly affect the productivity metrics for the I/O-intensive web service. This happens because, in this service, the processing of requests requires little CPU usage. Another important observation is that in this kind of service, the Performance policy does not increase neither power consumption nor energy consumption compared to the Powersave policy. This occurs because, although set to operate at higher frequency, the processor stays idle most of the time.

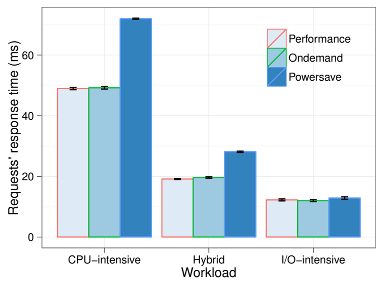
We then analyze the Performance and Powersave policies in executions with the CPU-intensive and the hybrid services. Our experiments show that Powersave reduces the average power consumption, but has a huge impact in performance (Figures 1(c) and 1(d)). This loss in performance is not compensated by the decrease in consumption as seen in



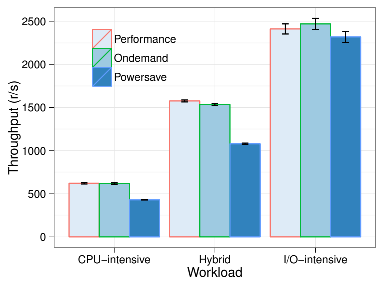
(a) Power consumption



(b) Energy consumption



(c) Requests' response time



(d) Throughput

Figure 1. Impact of the Performance, Powersave, and Ondemand policies on server's total energy consumption and average power consumption, requests' response time, and system's throughput.

Figure 1(b): the energy costs are higher if executing in Powersave mode. On the other extreme, the Performance policy does not achieve better results than Ondemand considering the response time and throughput metrics.

Overall, the Ondemand policy is the most energy efficient policy. It achieves energy consumption equivalent or lower than that achieved by the Performance and Powersave policies, while mirroring the throughput and requests' response time of the Performance policy. In the next scenario, we compare the Ondemand with our PAFS policy.

Scenario 2: In this scenario, we analyze the effectiveness of the PAFS policy by using system throughput as productivity metric. In order to evaluate different levels of productivity requirements, we considered the results of our previous experiments with the Performance policy and collected the achieved productivity values. These values then served as references for generating other realistic values for the productivity metric. The target throughput values in the contract were then defined to be equivalent to 100%, 97%, 95% and 90% of the best performance case. The values for the contracts are depicted in Table I.

As shown in Figure 2, for the I/O-intensive web service, the frequency scaling policies do not significantly affect power and energy consumption. Nevertheless, for hybrid and CPU-intensive web services, PAFS(100) consistently exhibits a lower power consumption than the Ondemand policy and at most an equivalent total energy consumption.

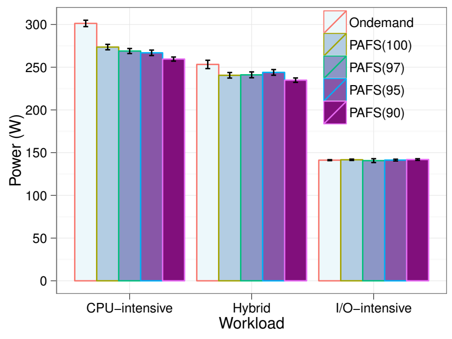
Table I
THROUGHPUT TARGET VALUES FOR THE PAFS CONTRACTS (WITH A 5% ALLOWED FAULT RATE).

Reference	I/O-intensive	CPU-intensive	Hybrid
PAFS(100)	2410.87	622.24	1576.21
PAFS(97)	2338.54	603.57	1528.92
PAFS(95)	2290.33	591.13	1497.40
PAFS(90)	2169.78	560.02	1418.59

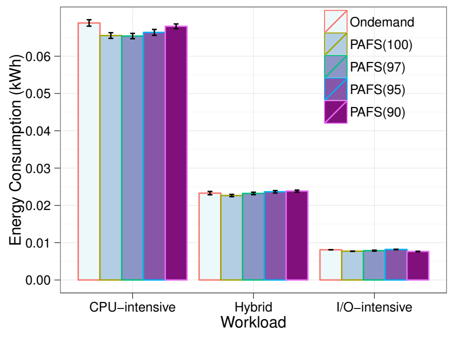
For the CPU-intensive web services, we can clearly note the effect of a QoS reduction from 100% to 90% on the consumed power and energy. Reducing the QoS tends to reduce the power consumption and throughput (Figure 3), but may increase the total energy consumed.

We also observed that increasing the CPUs frequency can reduce the productivity of the main application in some situations. This can be seen more clearly in the I/O-intensive experiment (Figure 3). In this case, even PAFS(90) generated a higher throughput than the Ondemand policy. Cases where higher performance is obtained with a lower CPU frequency have also been observed in related work [6].

We designed a simple experiment to better investigate this effect. This experiment considers two simple programs. The first program executes an I/O-intensive computation; it updates random positions in a big file. The second program executes a hybrid computation (including both CPU and I/O-intensive operations). We then define productivity as the



(a) Power consumption



(b) Energy consumption

Figure 2. Impact of the Ondemand and PAFS policies on server's energy and power consumption (productivity defined as throughput).

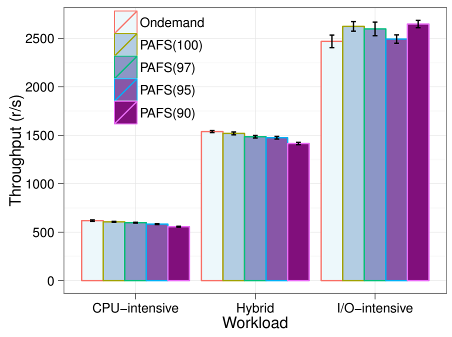


Figure 3. Comparison between the Ondemand and the PAFS policies with different contracts for throughput.

number of updates the first, purely I/O-intensive, program is able to perform per second.

These two programs run concurrently in the same server we used in the previous experiments. We did a set of executions using the lowest and the highest CPU frequencies available in the machine, namely, 1.60 and 3.06 GHz. The results are shown in Figure 4. As previously, the graphs indicate the average of the productivity metric with 95% confidence intervals.

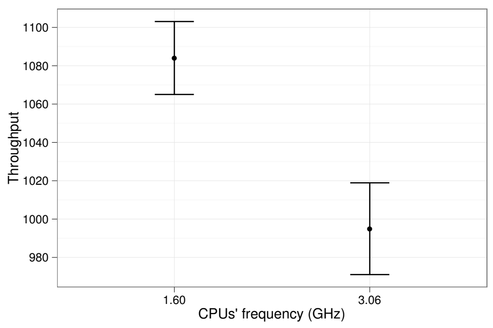


Figure 4. Effect of changing CPU frequencies in the productivity of an I/O intensive application in an environment with high disk contention.

The effect shown in Figure 4 is clear, a lower CPU frequency may lead to a higher productivity in the main application. The explanation, although counterintuitive, is simple. First, the performance of the purely I/O-intensive application depends solely on the performance of the disk and our productivity metric considers only this application. Second, for its computations the hybrid application needs to do some (not negligible) processing before issuing disk requests.

When CPU frequency is higher, the second application can do its processing faster and, thus, issues more disk requests per second. Likewise, if the CPU frequency is lower the processing of the hybrid application takes more time, increasing the time between disk requests, and, consequently, reducing the number of disk requests issues per second. Consequently, with less disk requests from the other application, the purely I/O-intensive application performs better, which is reflected in its metric.

Scenario 3: This scenario differs from the previous only regarding the productivity metric. We now analyze the effectiveness of the PAFS policy when using requests' response time as productivity metric. The values for the contracts are depicted in Table II.

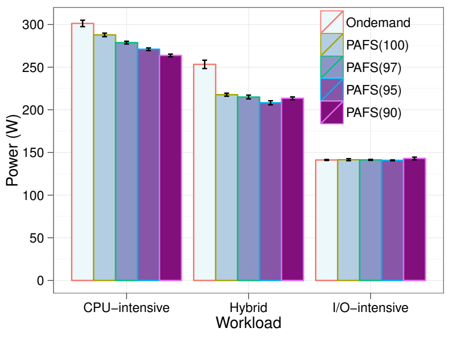
Table II
RESPONSE TIME TARGET VALUES FOR THE PAFS CONTRACTS (WITH A 5% ALLOWED FAULT RATE).

Reference	I/O-intensive	CPU-intensive	Hybrid
PAFS(100)	12.23	48.94	19.13
PAFS(97)	12.60	50.41	19.70
PAFS(95)	12.84	51.39	20.09
PAFS(90)	13.45	53.83	21.04

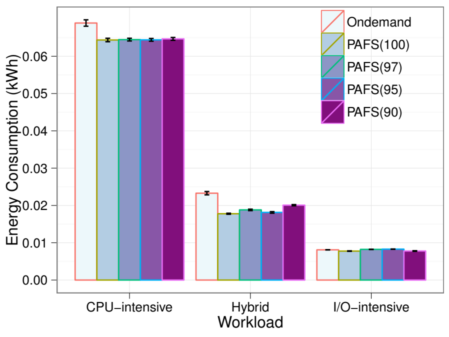
The results of our third set of experiments are depicted in Figures 5 and 6. Note that the benefit in terms of both energy and power savings (Figure 5) are proportionally greater than the performance degradation (Figure 6). In particular when the reference is defined as PAFS(100), PAFS achieves the required productivity (Figure 6), but it also reduces power consumption (Figure 5(a)) and total energy consumption (Figure 5(b)) in comparison to the Ondemand policy. This effect is caused by the preventive behavior of Ondemand and the reactive behavior of PAFS.

On the one hand, the Ondemand policy increases CPU frequency preventively, without observing if this change is justified by the main application (the one used to defined productivity) or was triggered by other less important processes running in background. On the other hand, PAFS increases frequency only reactively, after observing an actual deficit on productivity.

As expected, the results in Figure 6 show that, in CPU-intensive service profiles, PAFS reduces requests' response time when the user reduces the response time in the QoS requirement (contract). For example, PAFS(100) requires a response time that is 100% as good as the response time when in performance mode. Similarly, PAFS(90) is a slightly degraded level, meaning a 10% acceptable increase in response time.



(a) Power consumption



(b) Energy consumption

Figure 5. Impact of the Ondemand and PAFS policies on server's energy and power consumption (productivity defined as response time).

In hybrid services, there is a performance degradation when reference is changed from PAFS(100) to PAFS(97) (*i.e.*, QoS is reduced). However, there is no degradation of response time when reference is changed from PAFS(97) to PAFS(90). This is because the CPU frequency is already reduced to its minimum and requests' response time are affected only by factors that PAFS does not influence, such as disk accesses. For the same reason, when the application is dominated by I/O, the policies do not significantly affect the considered metrics. Finally, in this case, comparing PAFS(100) and Ondemand, PAFS(100) exhibits a lower response time and a 23.65% savings in energy consumed.

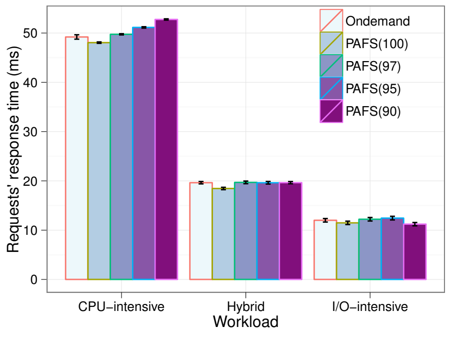


Figure 6. Comparison between the Ondemand and the PAFS policies with different contracts for requests' response time.

V. Conclusion

In this paper, we propose PAFS, an approach to generalize task-level DFS policies by enabling a single policy to be used with a user-defined productivity metric. PAFS is based on productivity, which can be defined according to user needs, and requires neither offline tuning nor prior information on the system's or applications' workload.

Our evaluation aims at comparing PAFS with commonly used system-level policies (Performance, Powersave, and Ondemand), which are broadly used in today's production systems. We extensively evaluate PAFS considering different web services profiles (I/O-intensive, CPU-intensive, and hybrid), different productivity metrics (response time and throughput) and four different QoS contracts.

Our results show that PAFS policy exhibits a consistently better energy consumption values in comparison to the Ondemand system-level policy, reaching up to 23.65% energy savings with no noticeable losses on the productivity metric. This is a consequence of PAFS having an reactive behavior instead of the preventive behavior found in Ondemand. Ondemand's preventive behavior will increase frequency anytime any running application requires CPU. Reactive behavior increases CPU frequency only after observing a relation between frequency and productivity *and* only if this increase in productivity is needed (as specified in the QoS contract).

Our findings may be extended in various directions. Among those, given the ability to define arbitrarily complex metrics, we want to evaluate how helpful complex metrics can be and how to help users defining them. Also, adaptive control techniques could be used to dynamically define the actuation interval (ΔT) and frequency steps (ΔF). A more sophisticated control algorithm could bring additional savings and increased robustness.

Acknowledgment

Authors would like to thank the financial support from CNPq (grant 305858/2010-6) and CAPES. F. Brasileiro is a CNPq researcher.

References

[1] J. Balladini, R. Suppi, D. Rexachs, and E. Luque, "Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems," in *9th IEEE/ACS International Conference on Computer Systems and Application*, 2011, pp. 16 –21.

[2] S. Albers, "Energy-efficient algorithms," *Communications of the ACM*, vol. 53, no. 5, pp. 86 – 96, 2010.

[3] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *11th international joint conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2009, pp. 157–168.

[4] Intel Corporation, "Intel energy checker sdk," http://software.intel.com/en-us/articles/intel-energy-checker-sdk/, Online June 2012.

[5] J. Jelschen, M. Gottschalk, M. Josefiok, C. Pitu, and A. Winter, "Towards applying reengineering services to energy-efficient applications," in *16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 353 –358.

[6] T. Wirtz and R. Ge, "Improving mapreduce energy efficiency for computation intensive workloads," *International Green Computing Conference and Workshops*, vol. 0, pp. 1–8, 2011.

[7] Canonical Ltd, "Power management in ubuntu," https://wiki.ubuntu.com/power-management-in-Ubuntu, Online June 2012.

[8] V. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proceedings of Linux Symposium*, vol. 2, 2006, pp. 223 – 238.

[9] R. Ayoub, U. Ogras, E. Gorbatov, Y. Jin, T. Kam, P. Diefenbaugh, and T. Rosing, "Os-level power minimization under tight performance constraints in general purpose systems," in *2011 International Symposium on Low Power Electronics and Design*, 2011, pp. 321 –326.

[10] G. Dhiman and T. Rosing, "System-level power management using online learning," *IEEE Trans Comput-aided Des Integr Circuits Syst*, vol. 28, no. 5, pp. 676 –689, 2009.

[11] S. Luiz, A. Perkusich, A. Lima, J. Silva, and H. Almeida, "System identification and energy-aware processor utilization control," *IEEE Trans. Consum. Electron*, vol. 58, no. 1, pp. 32 –37, 2012.

[12] C. Jiang, J. Wan, X. You, and Y. Zhao, "Power aware job scheduling in multi-processor system with service level agreements constraints," *JCP*, vol. 5, no. 8, pp. 1193–1203, 2010.

[13] K. H. Kim, A. Beloglazov, and R. Buyya, "Power-aware provisioning of virtual machines for real-time cloud services," *Concurrency Computat. Pract. Exper.*, vol. 23, no. 13, pp. 1491–1505, 2011.

[14] S. Albers, F. Müller, and S. Schmelzer, "Speed scaling on parallel processors," in *19th annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 289–298.

[15] V. Anagnostopoulou, M. Dimitrov, and K. Doshi, "Sla-guided energy savings for enterprise servers," in *2012 IEEE International Symposium on Performance Analysis of Systems and Software*, 2012, pp. 120 –121.

[16] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *2nd ACM Symposium on Cloud Computing*. New York, NY, USA: ACM, 2011, pp. 5:1–5:14.

[17] J. Wilkes, *Utility Functions, Prices, and Negotiation*. John Wiley and Sons, Inc., 2009, pp. 67–88.

[18] Intel Corporation, "Enhanced intel speed-step technology - how to document," http://www.intel.com/cd/channel/reseller/asmo-na/eng/203838.htm, Online June 2012.

[19] Advanced Micro Devices (AMD), "Amd family 10h desktop processor. power and thermal data sheet," 2012, http://www.amd.com/us/products/technologies/cool-n-quiet/Pages/cool-n-quiet.aspx, Online June 2012.

[20] ACPI, "Advanced configuration and power interface specification," 2012, http://www.acpi.info/spec.htm Online May 2012.

[21] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," *SIGPLAN Not.*, vol. 38, no. 5, pp. 38–48, 2003.

[22] The Apache Software Foundation, "Apache benchmark tool," http://httpd.apache.org/ Online May 2012.

[23] R. Bolla, R. Bruschi, F. Davoli, and F. Cucchietti, "Energy efficiency in the future internet: A survey of existing approaches and trends in energy-aware fixed network infrastructures," *IEEE Communications Surveys Tutorials*, vol. 13, no. 2, pp. 223 – 244, 2011.

[24] T. Kim, Y. Lee, and Y. Lee, "Energy measurement of web service," in *3rd International Conference on Future Energy Systems*, 2012, pp. 27:1–27:8.

[25] K. Le, O. Bilgir, R. Bianchini, M. Martonosi, and T. D. Nguyen, "Managing the cost, energy consumption, and carbon footprint of internet services," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 1, pp. 357–358, 2010.

[26] The Apache Software Foundation, "Apache http server," 2012, http://httpd.apache.org/docs/2.0/programs/ab.html Online May 2012.