Time Series based Dynamic Frequency Scaling Solution for Optimizing the CPU Energy Consumption

Tudor Cioara, Ionut Anghel, Ioan Salomie, Georgiana Copil, Daniel Moldovan, Marius Grindean
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
{tudor.cioara, ionut.anghel, ioan.salomie, georgiana.copil, daniel.moldovan, marius.grindean}@cs.utcluj.ro

Abstract— **In this paper the problem of service center servers high energy consumption is tackled by proposing a time series based CPU dynamic frequency scaling algorithm. The algorithm senses the workload changes and adapts the CPU power states thus minimizing the CPU energy consumption. Our solution analyzes the CPU workload time series for identifying the frequent workload patterns. For each frequent pattern, the corresponding dynamic frequency scaling actions are determined and associated using information about the pattern's sub-sequences trends. A workload characterization function is defined and used to identify the pattern trends. To identify the membership of the new CPU workload observations to a frequent CPU workload pattern, a sliding window based method is used. If such a match is found, the dynamic frequency scaling actions associated to the frequent pattern are executed and the pattern occurrence probability is increased.**

Keywords- time series; dynamic frequency scaling; frequent patterns; workload trend; sliding window.

I. INTRODUCTION AND RELATED WORK

The energy used by the service centers has increased drastically in the recent years, being directly related with the number of hosted servers and their workload. Service center servers computing capabilities and in consequence their power consumption will continuously increase in time. As the price and demand for energy continue to rise, service center servers power consumption must be considered as a first-class resource that is carefully controlled along with their performance.

Dynamic Power Management (DPM) is a methodology that allows for powering-down the server components to reduce the power consumption. The state of the art power management techniques can be classified into hardware and software techniques. Hardware techniques refer mainly to the design of efficient components such as low-voltage transistors, power-efficient components (CPU, DRAM, various controllers, etc.) and high conversion-rate power supplies [1]. Software techniques on the other hand, deal with power management of system hardware components (CPU, RAM, HDD, network cards) by transitioning them into one of the several different low-power states when they are idle for a long period of time [2].

According to the strategy used for deciding to trigger a power state transition for a hardware component, three types of DPM techniques are identified [3]: predictive techniques, heuristic techniques and QoS / Energy trade-offs. Predictive techniques employ simple to sophisticated predictive mechanisms to determine how long into the future the hardware component is expected to stay idle and use that knowledge to determine when to reactivate the component into a high power state [4] [5]. Heuristics techniques use quick accessible information for decision taking; they are very fast and have little overhead. However in complex systems they are loosely applicable and sometimes they can result in erratic performance degradation [6] [7]. The QoS/Energy trade-offs techniques exploit the energy saving opportunities presented by lowering the performance request levels of the running tasks.

In a service center server the main power consumers are the processors and to some extent the internal memory [8]. The introduction of the Advanced Configuration and Power Interface (ACPI) [9], an open industry standard which allows an operating system to directly control its underlying hardware, opened new directions for controlling the CPU power-saving aspects of a server like Dynamic Frequency Scaling (DFS). Most of the DFS approaches are based on the fact that, the energy consumed by the processor is a quadratic function of its operating voltage and implicitly of its frequency [10]. In [11] a DFS method based on decomposing the CPU workload into on-chip and off-chip is proposed. The on-chip workload contains the CPU clock cycles that are required to execute instructions in the CPU whereas the off-chip workload captures the number of external memory access clock cycles that are required to perform external memory transactions. Khargharia proposes a theoretical methodology and an experimental framework for autonomic power and performance management of high-performance server platforms [12]. In [13] the authors propose a DFS algorithm which detects the CPU-boundedness of a program using a regression method on the number of instruction per-second executed by the CPU in the past and then adjusting the CPU frequency accordingly. A DFS method that uses adaptive learning trees for predicting the CPU next idle periods is proposed in [14]. The learning tree nodes contain the CPU workload values and a tree path represents a workload sequence. In [15] a DFS technique based on machine learning and power policies is proposed. Techniques for predicting the best CPU frequency choices based on machine learning are also proposed in [21]. Bircher [16] performs an ample analysis of power management on recent multi-core processors using

978-1-4577-1481-8/11/$26.00 ©2011 IEEE

the functions provided by regular operating systems. In [20], a time series-based solution for managing power in mobile processors and disks for multimedia workloads is proposed. Time series statistical techniques are employed to determine the processor and I/O demands and to change the performance states. The above presented DFS approaches are conservative due to the existing concerns regarding the application performance and infrequent but inevitable workload peaks.

In this paper we address the problem of service center servers high energy consumption by proposing a CPU DFS algorithm capable of dynamically changing the CPU power states (p-states) and implicitly its working frequency according to the workload variations thus minimizing its energy consumption. The DFS algorithm collects data about the CPU at regular time intervals and constructs a CPU workload time series. The CPU workload time series is analyzed for identifying frequent CPU workload patterns. For each frequent pattern, the corresponding DFS actions are determined using information about the pattern's sub-sequences trends and a workload characterization function. A sliding window based method is employed to identify for new CPU workload observations the apartness to an indentified frequent CPU workload pattern. If such a match is found the DFS actions associated to the frequent pattern are executed and the pattern appearance probability is increased.

The rest of the paper is organized as follows: in section II the concept of CPU workload time series is defined and the CPU DFS algorithm initial and run time stages are detailed, section III presents some experimental energy saving results obtained for the proposed algorithm while Section IV concludes the paper.

II. TIME SERIES BASED DYNAMIC FREQUENCY SCALING

This chapter introduces a CPU Dynamic Frequency Scaling (DFS) methodology based on time series analysis and dynamically matching patterns at run-time. A time series is defined as a sequence of data values, measured at uniform and consecutive time intervals [17]. For CPU DFS, a time series describes the CPU's workload values evolution in time:

$$X = \{x_t, t = 1, \dots, N\} \quad (1)$$

where x_t is a time series observation (a CPU workload value in %), t represents the time index (the time snapshot in which the workload value has been captured) and N is the total number of observations.

A CPU time series observation represents the CPU monitored workload value and the time instance in which that value was captured ($x_t = (\mathrm{W_t}, t)$). A time interval represents the constant time difference between two consecutive workload observations. For CPU, the workload series time interval is considered one second.

The proposed DFS methodology has two main stages: (i) an *initial stage* or training stage executed offline before the algorithms deployment and continuously in background afterwards and (ii) a *run-time stage*.

In the initial stage the CPU workload values are monitored / collected for a long period of time and a CPU workload time series is constructed. The CPU workload time series is analyzed for identifying frequent CPU workload patterns and their associated DFS actions.

In the run time phase, the current workload values are captured and added to the initial CPU workload time series. A sliding window based algorithm is used to find a match between the current considered sliding window observations and the initial stage identified frequent patterns. If such a match is found, the DFS actions associated to the frequent pattern are executed and the pattern appearance probability is increased. Otherwise the sliding window observations are added to the CPU workload time series and the initial stage algorithms are used to determine the appropriate DFS actions to be executed.

A. The Initial Stage

The initial stage has two main steps: *CPU frequent workload patterns identification* and *frequent patterns DFS actions association*. In the first phase the CPU workload time series is analyzed and frequent CPU workload patterns (also small time series $p = \{p_t, t = 1, \dots\}$) are identified. In the second phase the identified frequent patterns are divided in sub-sequences with strict trends and analyzed using a workload characterization function with the goal of identifying the appropriate DFS actions to be executed to minimize the CPU's energy consumption.

1) CPU frequent workload patterns identification

To discover frequent workload patterns in the CPU workload time series a frequent sub-graph mining algorithm is applied. The CPU workload time series is modeled as a graph (see Fig. 1 for an example). Each graph vertex stores a CPU workload observation value while a graph edge represents an evolution between two consecutive CPU workload values. Each graph edge has an associated label representing the number of transitions (continuously incremented) between two CPU workload observation values.

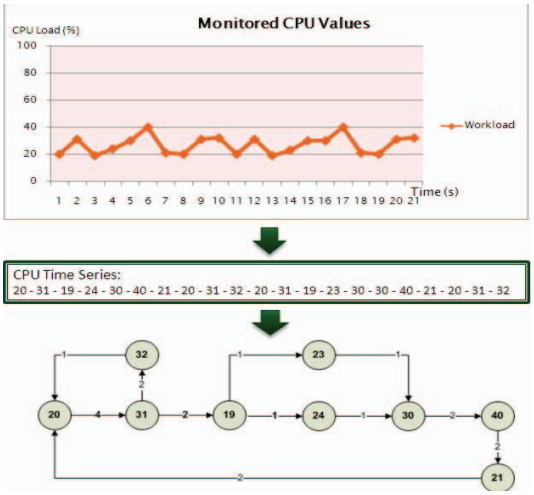

Figure 1. Constructing the CPU observations time series and the associated transition graph

To construct the CPU workload time series graph, time series observations are considered one by one in the order of their appearance. The first workload time series observation represents the starting vertex of the graph. Let us consider X_t a time series observation that is represented in the graph as

vertex A. For the next time series observation X_{t+1} a new graph vertex B must be created.

Two main possibilities are identified:

- If a vertex already exists in the graph and corresponds to a workload observation similar to X_{t+1} (having the same workload value), no new vertex will be created. If an edge connecting node A and node B exists, then its transition number value is incremented by 1, else a new one ($A \rightarrow B$) with the transition number value 1 is created.
- Otherwise (no graph vertex with the same workload value as X_{t+1} exists), a new node B and the edge connecting node A and node B is created. The edge transition number value is set to 1.

These steps are repeated for every CPU workload observation in the time series. A frequent workload pattern is represented by subgraphs of different dimensions (minimum two vertex) with a high occurrence probability.

The occurrence probability or rank of a pattern is equal with the minimum transition number value assigned to an edge part of the pattern sub-graph. This means that the occurrence probability of a graph cannot exceed the occurrence probability of its subgraphs:

$$\forall X, Y \in G, X \subseteq Y \rightarrow P(Y) \leq P(X) \quad (2)$$

To identify the frequent sub-patterns from the CPU workload time series, an adapted version of the *apriori-like* frequent sub-graph mining algorithm [18] is used. The adapted version of the *apriori-like* method has three main steps which are repeated until no new CPU workload subgraphs are generated: *candidate generation*, *candidate pruning* and *candidate elimination*. In the following sub-sections by *(k)-graph* we refer to a graph that has *k* vertices.

a) Candidate generation

To generate the candidate CPU workload patterns subgraphs the incremental vertex growing method is used. At each increment *k*, pairs of *(k-1)*-subgraphs are considered and eventually merged to create a *(k)*-subgraph. Two *(k-1)*-subgraphs can be merged if and only if they share a *(k-2)*-subgraph (also called core subgraph). The *(k)*-subgraph is obtained by attaching to the core *(k-2)*-subgraph the two additional vertices and edges from the considered *(k-1)*-subgraphs.

Fig. 2 presents an example of the incremental vertex growing method for *(2)*-subgraphs and *(3)*-subgraphs. Two *(2)*-subgraphs can be merged if and only if share a common vertex, while two *(3)*-subgraphs can be merged if they share two common vertexes.

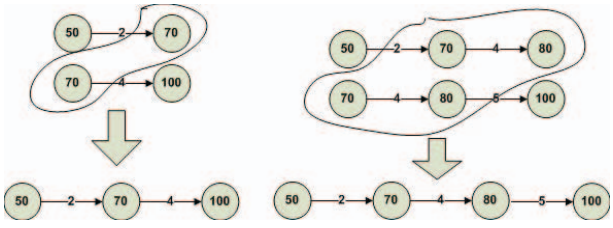

Figure 2. Candidate generation example

To determine the common core subgraph of two graphs, the graphs' adjacency matrices are used. The common core can be computed by removing the last line and column of an adjacency matrix. If the two core adjacency matrices are identical, the two subgraphs can be merged.

b) Candidate pruning and elimination

In this step all the generated *(k)*-graphs representing CPU workload patterns candidates are investigated to determine subgraphs with low occurrence probability (smaller than a defined threshold). If such a subgraph is identified, the *(k)*-graph representing CPU workload pattern is removed from the candidates set. This step is successively repeated for every *k*. This ensures the fact that infrequent CPU workload sequences are always removed from the generated workload patterns. Also, in the next iterations, the number of the generated *(k+1)*-subgraphs will be reduced by considering the anti-monotone property of the occurrence probability function: a *(k+1)*-subgraph is frequent if the two generated *(k)*-subgraphs are frequent.

2) Frequent pattern DFS actions association

The identified frequent CPU workload patterns are analyzed and the DFS actions or plans of actions associated to the workload pattern are determined. To associate a CPU DFS action to a frequent workload pattern three main steps are used: (i) the frequent workload patterns are divided into elementary sub-sequences that have strict ascendant or descendent trends, (ii) for each elementary CPU workload sub-sequences, a characterization function is used to determine the sub-sequence which would trigger a change of the CPU p-state and (iii) based on the previous steps results, the CPU's DFS actions associated to the frequent workload pattern are established.

a) CPU frequent pattern's sub-sequences

To indentify the CPU workload pattern elementary sub-sequences with strict ascendant or descendent trends, the time series least square trend estimation method is used (see Fig. 3 for an example). The time series least square trend estimation method is based on determining for the pattern's consecutive CPU workload observations $x_t = (\mathrm{W_t}, t)$ the values of parameters a and b for which the following relation is minimized:

$$\sum_t [(a * t + b) - x_t]^2 \quad (3)$$

The trend (as a value) is represented by the slope of the trend line ($a * t + b$). In our case, an elementary sub-sequence has an ascending trend if the determined trend line which fits the entire pattern's observations has a positive slope and descendant otherwise.

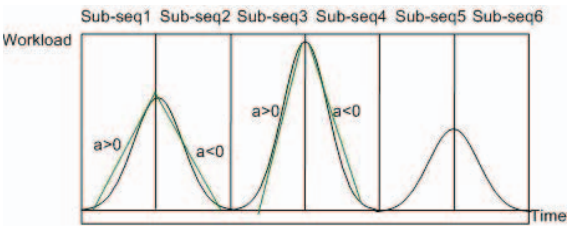

Figure 3. Dividing a CPU frequent pattern in sub-sequences

To divide the CPU workload frequent pattern into elementary sub-sequences, the frequent pattern first observation is considered. The frequent workload pattern's observations are then iterated one by one and are added to the default sub-sequence. For each new observation that is added the default sub-sequence, the trend is estimated using the least square trend estimation method. The objective is to find the straight line (also called trend line) which fits all the CPU observations part of the default sub-sequence. If such a trend line can be determined, the default sub-sequence observations have a strict trend and the iteration continues. Otherwise, if a trend line cannot be determined, the default sub-sequence observations trend is modeled by a non-linear function. This means that the last added observation is an inflexion point. The default sub-sequence observations without the inflexion point observation is saved as a strict trend sub-sequence and is re-initialized using the inflexion point as its first observation. The iteration will continue until the entire frequent pattern's observations are considered.

b) Workload characterization function

The goal of the workload characterization function is to establish, for a frequent workload pattern sub-sequence with a clear trend, the potential to trigger a CPU p-state change. The values of p-states depend on the CPU type, but usually $P_{state-0}$ is always the highest-performance state (highest frequency, highest power consumption), while $P_{state-1}$ to $P_{state-n}$ are successive lower-performance states. The minimum and maximum workload values for which the CPU operates in a certain p-state are defined by its technical datasheet (we note them with: $T^{min}_{P_{state-i}}$ and $T^{max}_{P_{state-i}}$). The workload characterization function is constructed in such a way that t time index correlates highly with the occurrence of a CPU p-state transition as in (4).

$$C(g(t), P_i) = \begin{cases} p^+, & g(t) \geq T^{max}_{P_i} \\ p^0 & T^{min}_{P_i} \leq g(t) \leq T^{max}_{P_i} \\ p^- & g(t) \leq T^{min}_{P_i} \end{cases} \quad (4)$$

$$g(t) = g(x_1, x_2, \dots, x_t) = x_1 + \sum_{t=2}^{N} x_t - x_{t-1}$$

where x_1 is the workload observation inflexion point (first element of the sub-sequence) and $x_2, \dots, x_t$ are the CPU observed workload values part of a frequent pattern sub-sequence (seq). A frequent pattern sub-sequence with strict trend can be in one of the following classes: p^+ - may trigger a CPU p-state change form an inferior p-state to a superior p-state, p^- - may trigger a CPU p-state change form a superior p-state to an inferior p-state and p^0 - doesn't trigger any CPU p-state changes.

c) Determining the CPU's DFS actions

In this section an algorithm for selecting and constructing the DFS actions plans that need to be executed for a certain frequent workload pattern is presented. To design an algorithm for improving the CPU's energy efficiency the following considerations must be kept in mind: (i) to improve the CPU's energy consumption, DFS actions should be taken to keep as much as possible (and if possible) the CPU in low power consumption p-states and (ii) all the DFS actions introduce a power consumption and performance penalty. As a consequence, the algorithm must filter (ignore in the DFS actions selection process) the frequent pattern isolated workload spikes because they will induce a high energy consumption payload with a small performance gain.

The algorithm analyzes the frequent patterns pairs of successive sub-sequences and using their trend related information, decides on three types of DFS actions: scale-up CPU to a higher performance p-state, scale-down CPU to a lower performance state and no CPU scaling. A CPU state transition to a higher performance p-state (↑ in the decision Table 1) is selected for execution for the following pairs of elementary sub-sequences part of the frequent workload pattern: $p^+ \rightarrow p^+$ and $p^+ \rightarrow p^0$. A CPU state transition to a lower performance p-state (↓ in the decision Table 1) is selected to be executed for the following pairs of successions of frequent workload pattern elementary sub-sequences: $p^- \rightarrow p^-$ and $p^- \rightarrow p^0$. No state transition action (↔ in the decision Table 1) is selected for the following succession of sub-sequences $p^0 \rightarrow p^0$, $p^0 \rightarrow p^-$, $p^0 \rightarrow p^+$.

TABLE I. DFS ACTIONS DECISION TABLE

Current pattern \ Next pattern	p^+	p^0	p^-
p^+	↑	↑	∘
p^0	↔	↔	↔
p^-	∘	↓	↓

The pairs of sub-sequences $p^+ \rightarrow p^-$ and $p^- \rightarrow p^+$ usually signal a workload spike so their simple analysis is not enough to decide on the appropriate DFS action (unidentified action notated with ∘ in the decision Table 1). In this case the current sub-sequence length and its associated threshold (T_l) is used as a decision criteria. For the $p^+ \rightarrow p^-$ pair of sub-sequences the CPU will be scaled up to a higher p-state if and only if the size of the p^+ sub-sequence is higher than T_l otherwise no scaling action will be selected. For $p^- \rightarrow p^+$ the CPU will be scaled down to a lower p-state if and only if the size of the p^- sub-sequence is higher than T_l otherwise no scaling action will be selected.

There are some particular cases in which the frequent pattern sub-sequences cannot be grouped in pairs: (i) the frequent pattern has only one sub-sequence and (ii) the current considered sub-sequence is the last one in the pattern. In this situations, the selected DFS actions correspond to the current sub-section classification if the sub-section length is higher than the predefined threshold T_l.

T_l threshold is empirically selected by means of experiments. By varying T_l the performance / energy consumption of the CPU is directly influenced as follows: a smaller value for T_l implies that the emphasis will be on performance while a higher T_l implies that the emphasis will be on energy consumption.

B. The Run-Time Stage

At run-time, the CPU is monitored, observations regarding its workload values are gathered at regular time intervals, and

added to the initial stage CPU workload time series. The main goal of this stage is to identify a match between the current collected workload observations and the frequent workload patterns determined in the initial stage (see Fig. 4 algorithm lines 8-12). If such a match is found the DFS action or plans of actions associated to the frequent pattern is taken (line 16) and the pattern occurrence probability is increased by one. Otherwise it means that a new workload pattern is found and the initial stage algorithms (workload characterization and pattern DFS action selection) are run in background to select the proper DFS actions to be executed and associated for the new workload pattern. The new workload pattern occurrence probability is set to one (see Fig. 4 algorithm line 15).

Algorithm: Run-time_CPU_TimeSeries_DVS

1 **Input:** $x_t = (W_t, t)$ – CPU current workload value observation
$X = \{x_1, x_2, \dots x_{t-1}\}$ – current CPU workload time series
$FREQ_{Patterns}$ - initial stage identified frequent CPU workload patterns and their associated actions
2 **Output:** $DVS_{actions}$ – dynamic voltage scaling action that must be executed
3 **Begin**
4 $SL = \{x_t\}$
5 $incompleteMatchPatterns = null$
6 **While** $((SL.lenght \leq MAX_{LENGHT}(FREQ_{Patterns}) \&\& (exactMatchPattern == null))$
7 $SL = SL \cup \{x_{t-1}\}$
8 **Foreach** $pattern \in FREQ_{Patterns}$
9 **If** $((corr(SL, pattern) \leq T_c) \&\& (SL.lenght == pattern.lenght))$ **then**
10 $exactMatchPattern = pattern$
11 **Else** $incompleteMatchPatterns = incompleteMatchPatterns \cup pattern$
12 **EndForeach**
13 $t = t - 1$
14 **EndWhile**
15 **If** $(exactMatchPattern == null)$ **then** $DVS_{actions} = Determine\_DVS\_actions(SL)$
16 **Else** $DVS_{actions} = getAssociatedActions(exactMatchPattern)$
17 **Return** $DVS_{actions}$
18 **End**

Figure 4. The CPU time series based DFS algorithm – run time stage

To identify the frequent workload patterns in which the current CPU observations may be placed, a dynamic sliding window based algorithm is used. The sliding window is a small time series which dimension is gradually increased by varying the number of past observations together with the current ones. The sliding window dimension cannot exceed the maximum size of a frequent pattern identified in the initial phase. To evaluate the degree of matching between the sliding window CPU workload observations and the initial stage determined patterns we define a correlation factor as:

$$corr(SL, p) = \sum_{t=1}^{N_{sl}} (sl_t - p_t)^2 \quad (5)$$

where SL represents the sliding window time series, p is an initial phase identified frequent pattern, sl_t represents a sliding window observation, p_t is a pattern observation and N_{sl} represents the sliding window number of observations. If the correlation factor value is smaller than a predefined correlation threshold T_c, a match between the sliding window and the initial phase pattern is found otherwise no matching is found.

The match between the sliding window and the initial phase pattern can be classified in: exact matching and incomplete matching.

When dealing with exact matching the sliding window and the matching CPU workload pattern have the same number of observations. It can be easily shown that if there is an exact matching between the current sliding window and an initial stage frequent pattern, this matching is unique. In this case the taken DFS actions correspond to the matching pattern class.

On the other hand when dealing with an incomplete match, the pattern number of observations is greater than the sliding window number of observations. An incomplete match represents only a partial mapping of the sliding window observations onto the starting observations of a CPU workload pattern.

An incomplete matching is not unique meaning that it can be more than one initial stage frequent patterns with an incomplete matching for the current sliding window. In this case new collected CPU observations are added to the sliding window until an exact match with one frequent pattern indentified in the initial stage or no match is found. If no mach is found, the sliding window observations form a new workload pattern and the DFS actions that need to be executed for this pattern are selected using the initial stage algorithms.

III. Case Study and Results

To test and validate the time series based DFS algorithm an IBM server with Intel i7 3GHz processor and 6 GB of memory is used. The Intel i7 processor provides 14 power states (p-states), each state having different associated frequencies and implicitly different power consumption values (see Table 2).

TABLE II. The processor p-states

P-State	*Frequency (GHz)*	*Frequency (%)*	*Power Cons. (W=J/s)*
0	2.93	100	95
1	2.79	95.22	83.41
2	2.66	90.78	72.87
3	2.53	86.34	63.33
4	2.39	81.56	54.75
5	2.26	77.13	47.06
6	2.13	72.69	40.24
7	2.00	68.25	34.21
8	1.86	63.48	28.94
9	1.73	59.04	24.36
10	1.60	54.60	20.44
11	1.46	49.82	17.13
12	1.33	45.39	14.36
13	1.20	40.95	12.09

For testing purposes, CentOS 5.5 [22] is installed as host operating system on the server. The CPU usage information is monitored and collected using cpufreq-utils [23] kernel utility which also allows for controlling the CPU p-state / frequency. On top of this infrastructure the time series based DFS algorithm is deployed.

To evaluate the energy efficiency capabilities of our time series based DFS algorithm we have generated a highly intensive test case workload with frequent spikes for which the CPU usage varies between 40% and 100 %. Super PI CPU benchmarking software [24] is used to generate such workload. Super PI CPU benchmark is based on computing 2^n digits of π (n has a maximum value of 25). We have randomly started and executed for a certain amount of time several PI digits generators, thus obtaining highly intensive workloads.

Using a power meter (ISO-TECH IPM3005 [25]) we have measured the test case IBM server instant power consumption (W=J/s). The IBM server energy consumption was determined as the total power consumed in the time period needed to execute the generated workload. As it can be seen in Fig. 5, for the same executed workload the IBM server average power consumption was about 99.9 W when managed with our time series based DFS algorithm and about 112.6 W for the CentOS OnDemand frequency governor algorithm. This results in an estimated energy consumption of about 81418 J (22.6 Wh) for the DFS algorithm and 91796 J (25.5 Wh) for CentOS OnDemand. The energy efficiency increase is about of 12.8 % for the time series based DFS algorithm compared with CentOS OnDemand.

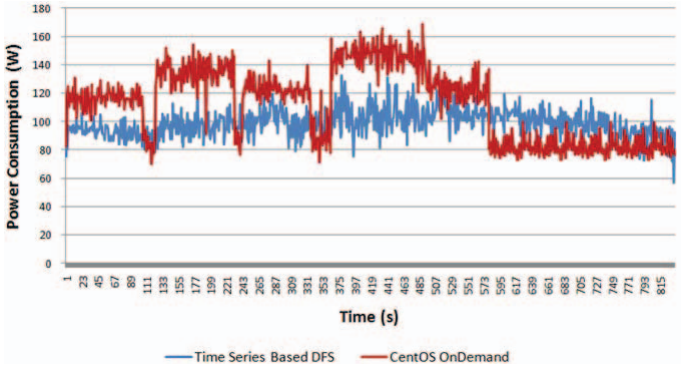

Figure 5. Server power consumption chart for time series based DFS and CentOS OnDemand

The higher energy consumption of the CentOS OnDemand is justified by the fact that the CPU is over-provisioned and kept constantly in P-0, the highest power consumption state, regardless the workload intesitiy oscillations until the entire test case workload is executed (see Fig.6).

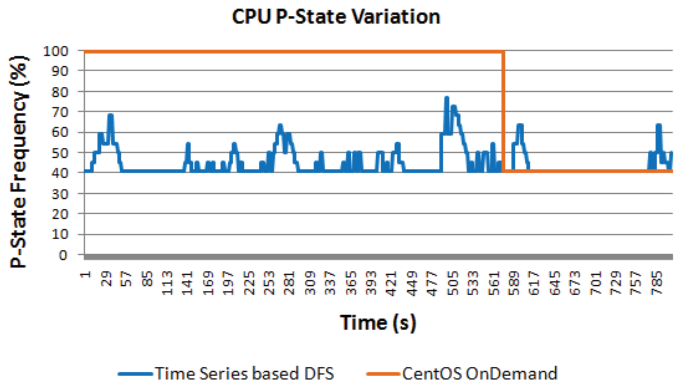

Figure 6. CPU frequency and P-State variation for time series based DFS and CentOS OnDemand

On the other hand, the time series based DFS algorithm manages to filter the workload spikes and varies the CPU p-states between p-state 5 and p-state 13. The energy efficiency increase has a performance penalty of about 25%. As a consequence, the same workload execution will be performed faster when CentOS OnDemand governor is used.

IV. CONCLUSIONS

This paper proposes a time series based DFS algorithm for energy efficient management of service center servers CPU power states. Time series and data mining specific concepts / processes like observations, trends, time series characterization or frequent patterns identification are employed to determine the DFS actions that must be executed to accommodate and run the incoming CPU workload with minimum energy consumption. The test case results are promising showing that the proposed algorithm manages to save approximate 12.8% of energy when running a highly intensity workload compared with the CentOS frequency governor. This is due to the time series based algorithm capability of adapting the CPU power states according to the incoming workload spikes.

ACKNOWLEDGMENT

This work has been supported by the European Commission within the FP7 GAMES project [19].

REFERENCES

[1] V. L. Prabha, E. Monie, “Hardware architecture of reinforcement learning scheme for dynamic power management in embedded systems”, EURASIP Journal on Embedded Systems, 2007.

[2] Y.-H. Lu, T. Simunic, G. De Micheli, “Software controlled power management”, Proceedings of the Seventh International Workshop on Hardware/Software Codesign, 1999.

[3] B. Khargharia, “Adaptive Power and Performance Management of Computing Systems”, PhD Thesys, University of Arizona, 2008.

[4] V. Delaluz, M. Kandemir et al., “Hardware and Software Techniques for Controlling DRAM Power Modes”, IEEE Trans. Computers, vol. 50, pp. 1154-1173, 2001.

[5] A. R. Lebeck, X. Fan, H. Zeng and C. Ellis, “Power Aware Page Allocation”, In ASPLOS, pp. 105-116, 2000.

[6] C. Hsu and U. Kremer, “The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction”, ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 38-48, 2003.

[7] H. Huang, P. Pillai and K. Shin, “Design and Implementation of Power-Aware Virtual Memory”, USENIX Technical Conf., pp.57-70, 2003.

[8] C. Lefurgy, X. Wang, M. Ware, “Server-level Power Control”, International Conference on Autonomic Computing (ICAC), 2007.

[9] ACPI - Advanced Configuration and Power Interface, http://www.acpi.info

[10] C.M. Krishna and Yann-Hang Lee, “Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems”, IEEE Transactions on computers, vol. 52, no. 12, 2003.

[11] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram, “Off-chip Latency-Driven Dynamic Voltage and Frequency Scaling for an MPEG Decoding”, AC ’04: Proceedings of the 41st annual conference on Design automation, 544-549, 2004.

[12] B. Khargharia, S. Hariri, M. Yousif, “Autonomic power and performance management for computing systems”, Cluster Computing, Vol. 11 (2), 167 – 181, ISSN:1386-7857, 2008.

[13] Chung-Hsing Hsu and Wu-Chun Feng, “Effective Dynamic Voltage Scaling through CPU-Boundedness Detection”, Workshop on Power Aware Computing Systems, 135--149, 2004.

[14] E. Chung, L. Benini, G. De Micheli, “Dynamic Power Management Using Adaptive Learning Tree”, In: IEEE/ACM Int. Conf. on Computer-aided design, pp. 274-279, 1999.

[15] Gaurav Dhiman, Tajana Simunic Rosing, *Dynamic Power Management Using Machine Learning*, ICCAD '06, November 5-9, 2006, San Jose.

[16] L. Bircher, L. John, “Analysis of Dynamic Power Management on Multi-CoreProcessors”, In: Proc. of the 22nd annual international conference on Supercomputing, pp. 327-338, 2008.

[17] Richard J. Povinelli, “Time Series Data Mining: Identifying Temporal Patterns for Characterization and Prediction of Time Series Events”, Milwaukee, Wisconsin, December, 1999.

[18] Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda, “An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data”, Lecture Notes in Computer Science, Volume 1910/2000, 13-23, 2000.

[19] GAMES FP7 Research Project, http://www.green-datacenters.eu/

[20] Xiaotao Liu, Prashant Shenoy, Weibo Gong, “A Time Series based Approach for Power Management in Mobile Processors and Disks”, Proc. of the 14th Int. workshop on Network and operating systems support for digital audio and video, ISBN:1-58113-801-6, 2004.

[21] Michael Moeng, Rami Melhem, “Applying statistical machine learning to multicore voltage & frequency scaling”, Proc. of the 7th ACM Int. conference on Computing frontiers, ISBN: 978-1-4503-0044-5, 2010.

[22] CentOS Enterprise Linux Distribution, http://www.centos.org/.

[23] cpufreq utility, http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html.

[24] Super Pi benchmark, http://www.super-computing.org/

[25] ISO-TECH IPM3005, http://www.iso-techonline.com/products/iso-tech-electrical-installation-testers.html