OS-level Power Minimization Under Tight Performance Constraints in General Purpose Systems

Raid Ayoub*, Umit Ogras†, Eugene Gorbatov†, Yanqin Jin*, Timothy Kam †, Paul Diefenbaugh† and Tajana Rosing*
*Department of Computer Science and Engineering
University of California at San Diego, La Jolla, CA 92093-0404
{rayoub, y7jin}@cs.ucsd.edu, tajana@ucsd.edu
†Intel Corporation, Hillsboro, OR 97124
{umit.y.ogras, eugene.gorbatov, timothy.kam, paul.s.diefenbaugh}@intel.com

***Abstract*—We propose a new DVFS algorithm for enterprise systems that elevates performance as a first order control parameter and manages frequency and voltage as a function of performance requirements. We implement our algorithm on real Intel Westmere platform in Linux and demonstrate its ability to reduce the standard deviation from target performance by more than 90% over state of the art policies while reducing average power by 17%.**

***Index Terms*—Power, Performance, DVFS, Operating system, Multiprocessor.**

I. Introduction

Scaling down in technology coupled with the increasing demand of computationally intensive applications has led to a wide use of multi-core CPUs in server systems. Multiple CPU packages have been deployed in modern systems to further increase computational capacity [3]. However, running this computational infrastructure increases system power consumption. At the same time, high CPU power causes thermal hot spots which require cooling subsystems that can consume significant energy [5]. Therefore, operational power consumed for system execution and cooling has become a big concern.

In general, processor power can be broken down into dynamic and static power where static component is around 20%-40% [6]. Dynamic power is a function of clock frequency and voltage while static power is a function of the number of active components and temperature. In recent years, system designers have introduced CPU power management techniques to improve energy efficiency and thermals. Dynamic voltage and frequency scaling (DVFS) has been a mainstay of server power management for several product generations now [3], [4]. DVFS lowers processor dynamic power by scaling down its voltage and frequency thus reducing power dissipation in both CPU and cooling subsystem.

DVFS control algorithms found in todays production systems rely on heuristics that can inadvertently decrease system performance while trying to minimize processor power consumption. Both Linux and Windows, for example, implement a DVFS policy that attempts to keep processor utilization at around 90%, progressively decreasing frequency whenever utilization drops below this threshold and increasing when above [18]. Although this policy keeps system throughput constant, it may negatively impact latency sensitive applications. In addition, and perhaps more importantly, it offers no strong performance guarantees. In fact, identifying periods during which voltage and performance can be safely reduced is an active area of research [12].

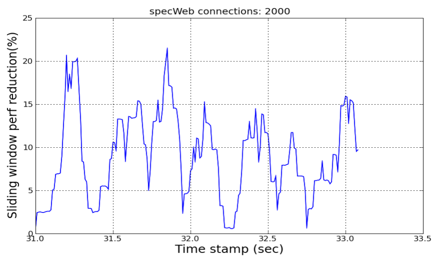

Fig. 1. Performance reduction using ondemand

Server performance remains a critical component of data center operations. Many server workloads are deployed with Service Level Agreements (SLAs) that capture workload performance requirements. Meeting these SLA requirements and providing certain level of performance guarantees lies in the heart of robust datacenter operation. The goal for power management in future generation servers will be to provide energy efficient performance rather than simply energy efficiency [2].

Figure 1 shows performance loss of *ondemand* governor, a state of the art DVFS Linux policy [2], for a representative period of executing a web server benchmark, SpecWeb2009 [10], with 2000 sessions. Each point in the graph represents average performance loss over a sliding window of 100ms. The results clearly show that Linux DVFS policy cannot constrain performance loss. In addition, performance variations from frequency scaling are very high further destabilizing system operation and user experience. Note that this behavior not only deteriorates workload quality of service (QoS) but may also be unacceptable for latency sensitive applications.

A number of DVFS techniques have been proposed for real-time applications that have hard-performance constraints [11], [19], [17]. A general idea behind this class of techniques is to pass application-level deadline information to the operating system (OS) to guide its DVFS decisions. However, using real-time mechanisms to address DVFS performance issues for data center workloads is problematic. First, most server applications and system software don't have the necessary infrastructure for deadline based scheduling and power management. Second, performance guarantees found in enterprise SLAs are statistical leading to unnecessary over-provisioning, complexity and less energy savings inherent in hard-real time systems with deterministic guarantees.

This paper proposes a new DVFS mechanism that makes performance a first order control parameter. Rather than requiring applications to specify deadlines, our approach introduces a new constraint, called performance target, expressed as a fraction of maximum system performance (e.g. 95%), as measured by the number of Instructions executed Per Second (IPS). We design a new DVFS algorithm that controls processor operating frequency and voltage based on this constraint while minimizing power and ensuring performance stability. Similar to prior work, we address typical enterprise applications that require soft guarantees [13], [14]. In contrast to prior work we provide performance guarantees using closed loop control.

The main benefit of our approach is in providing a more general solution for the soft performance guarantee problem while ensuring stability and efficiency of DVFS. To this end, we design a formal closed loop controller that dynamically changes operating frequency and voltage to meet a desired performance target. Our control framework is not limited to a single core; it is capable of managing multiple cores as well as multiple CPU packages. To evaluate our technique we implement new DVFS control algorithm in the Linux kernel and conduct detailed analysis on the latest generation server system. The results presented in the paper show that our approach has the ability to reduce the standard deviation from target performance by more than 90% over state of the art policies while reducing average power by 17%.

II. Related work

DVFS techniques can be broadly classified into multiple categories. The first class covers polices that mange power dissipation without guaranteeing QoS [7], [8], [18]. Modern Linux kernel uses $ondemand$ policy for managing DVFS [18]. The ondemand algorithm periodically calculates the CPU utilization. If the utilization is above a certain threshold, the *ondemand* set the frequency to the highest value. Alternatively, the next frequency, f_{next}, would be set to a value that equals to current frequency, f_{cur}, multiply by the utilization, U_{cur}, ($f_{next} = f_{cur} * U_{cur}$). The main problem with this policy is that it cannot constraint performance to given bounds.

The second class of DVFS techniques are designed for real-time applications that have hard deadlines [11], [19], [17]. In [11], the authors use software feedback loop mechanisms to save energy. The deadline for each time slot is provided to the OS. The OS calculates the frequency based on the current slack and the expected worst case execution time. The technique in [11] uses static timing analysis to find the execution time of the program segments. This static information is exposed to the OS to determine the frequency of the execution at run time to meet the desired deadlines. In [17], the authors address the issue of how to assign frequencies to a given set of tasks so they can meet their deadlines. Although these techniques are effective in meeting the desired performance in real-time applications, they are not so effective for soft deadlines due to their over provisioning of the resources.

The third category of DVFS techniques concern the non real-time applications that have soft performance constraints [14], [13]. In [13], the authors suggest technique for using DVFS to meet soft performance requirements for multimedia applications. These techniques use feedback control to manage DVFS. However, they are applicable to particular applications.

Our contribution can be summarized as: (1) We make performance SLAs first order parameter and allow software to specify its performance targets. (2) Rather than driving DVFS to reduce power only we drive DVFS to meet performance targets while minimizing power. (3) We show that our approach is able to meet performance targets when compared to current DVFS approaches. (4) We present detailed evaluation and analysis conducted on a real system.

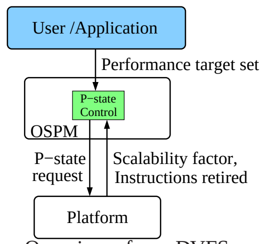

Fig. 2. Overview of our DVFS management

III. Design overview

Our DVFS (*p-states*) management is implemented at the operating system level as a kernel module. Figure 2 describes our approach. Our framework in general allows the applications to specify their respective performance target. The target performance is expressed as a fraction of maximum performance (e.g. 95%), as measured in terms of IPS. The user or the application set requests the target performance and passes this information to the operating system power manager (OSPM) where the target set can be easily modified at any point in time. Our solution is general which allows each core to run at different performance target when there is a support for changing DVFS per core. We developed a performance model that is based on frequency scalability which allows us to determine the target frequency of the individual thread. It accounts for the interference effect from other threads on the shared resources, e.g. off-core caches and memory. This model is embedded within a state-space framework which models a slack in the form of execution time to account for the limitations in frequency setting and run time variations in scalability factor. Our state-space model also handles the cases when a number of cores share a single DVFS due to hardware restrictions. The limitation is that we can assign a maximum of one performance target to the cores that share a single DVFS setting. In case the system supports multiple CPU sockets, then each socket can have different performance targets.

This set of target performance is the input to our Multi-Input Multi-Out (MIMO) feedback control algorithm that adjusts the frequency and voltage settings of the cores in a way that ensures the target performance level is satisfied. The proposed controller estimates two parameters (per core) from the platform using hardware performance counters. One is a scalability factor and the other is instructions retired. Scalability factor determines the portion of the execution time that is scalable with frequency. The controller makes DVFS decisions on a regular basis so that it can respond to run time changes in workload behavior. The controller sampling period is in the order of tens of milliseconds. We show that the overhead of this technique is negligible since the time for controller's computation time and DVFS switching is in the range of microseconds.

IV. DVFS control methodology

A. Performance model

In this section we develop the relations between OS level performance and clock frequency. When a thread is executing the thread is in *active* state, and when it is waiting in the task queue of the operating system it is in *idle* state. During the active mode the thread can be either executing or stalling while waiting for some on-core or off-core resource. In general, the latency of on-core stalls scales with frequency while the latency of off-core stalls does not. This indicates that the active time, T_{act} period of time when the thread is in active state, can be broken into frequency scalable and unscalable segments. We define the *scalability factor*, S_F, as the ratio between scalable time, $T_{scalable}$, to active time as:

$$S_F = \frac{T_{scalable}}{T_{act}} \tag{1}$$

Suppose frequency changes from f_1 to f_2. As a result, active time changes from $T_{act}(f_1)$ to $T_{act}(f_2)$:

$$T_{act}(f_2) = T_{act}(f_1) + \Delta_{T_{act}} \tag{2}$$

$$\Delta_{T_{act}} = S_F T_{act}(f_1)(\frac{f_1}{f_2} - 1) \tag{3}$$

where $\Delta_{T_{act}}$ represents the change in scalable time due to the change in frequency. We do not consider the unscalable time since it does not change with frequency.

To measure performance we use the commonly used metric which is the number of instructions, N_I, executed per second $IPS = \frac{N_I}{T_{act}}$. This metric is applicable to single and multiple threads. For multiple threads it is equivalent to throughput. The upper bound for the instructions execution rate, IPS_{max}, can be determined by setting the clock frequency of the active cores to the highest. The objective is to run a given set of jobs at IPS that is fraction of the highest one as follows:

$$IPS_{ref} = \beta IPS_{max} \tag{4}$$

where β is performance fraction factor (0 to 1) and IPS_{ref} is the target IPS. The value of β is the target performance ratio. It is provided by the user or application and can be easily changed (see Figure 2). The lower bound for β is determined based on the minimum and maximum frequencies supported by the CPU (f_{min}, f_{max}) and average value of scalability factor $\bar{S_F}$. The value of β_{min} equals to $1 + \bar{S_F}(\frac{f_{min}}{f_{max}} - 1)$. We use the IPS_{ref} in our approach as a reference input for the DVFS algorithm. The value of IPS_{ref} may vary at run time depending on the jobs run time profile. Calculating IPS_{ref} reference become straightforward if we know the value of IPS_{max}. The challenge is how to extrapolate the value of IPS_{max} when the cores execute at arbitrary clock frequencies. To relate the target IPS of each task to IPS_{max}, we use Equation (3) and obtain:

$$IPS_{ref} = IPS_{max} \frac{1}{1 + S_F(\frac{f_{max}}{f_{ref}} - 1)} \tag{5}$$

Finally, using (4) and (5), the target frequency can be computed as follows:

$$f_{ref} = \frac{f_{max} S_F}{\frac{1}{\beta} + (S_F - 1)} \tag{6}$$

B. State-space slack model

Our performance model given in (3) shows that a slack in the form of execution time, $\Delta_{T_{act}}$, would occur if the clock frequency is set to value that is different from the target one. The convergence of the performance to the target depends on the slack's convergence to zero. Before we address the convergence problem we need to develop a model for the slack. We start with a simple case where we have a single core executing a single thread. Ideally, at the end of each control interval, k, we need to calculate a new target frequency, f_{ref}, and then set the operating frequency to that value to meet the desired QoS. However, deviation from the desired QoS may occur due to errors in scalability factor or requesting unavailable frequencies. The source of errors in scalability factor come from the fact that we need to rely on scalability factor prediction for the next period (period between k and k+1). In this work, we assume that the next value of scalability factor equals to the current one that is already measured by performance counters. The frequency selection related errors occur because only limited set of frequencies are exposed to the operating system. Such deviations can be modeled as a slack in execution time which can be either positive (frequency is lower than the target) or negative (frequency is higher than the target). The interesting feature of the slack is that it is accumulative and can be modeled naturally in a state-space form. Modeling the slack problem in state-space form is desirable since it allow us to use the robust formal methods in state-space control.

$$\begin{aligned} s(k+1) &= s(k) + \Delta_s \\ &= s(k) + \frac{S_F(k)T_{act}}{f_{ref}(k)}(f(k) - f_{ref}(k)) \\ &= s(k) + \gamma(k)u(k) \end{aligned} \tag{7}$$

where $s(k)$ represents the slack accumulated until k'th period and $\Delta_s(k)$ is the slack for the period between k and $k+1$, which is computed using (3). The parameter T_{act} represents the time duration of the active state for the period between k and $k+1$ and $u(k) = f(k) - f_{ref}(k)$. The value of $\Delta_s(k)$ can be either zero, positive or negative quantity. For example, the value of $\Delta_s(k)$ is zero if $f(k)$ is set to f_{ref}.

Now we extend our modeling to multiple cores where each core executes a job. This means that we need to have a control over the slack of the active cores. Let's first assume the number of cores is N. We define the state of the slack with a vector of N states, at time k, as: $S(k) = [s_1(k), s_2(k), ..., s_N(k)]^T$. In case there is a DVFS support per core, the state-space formulation of $S(k)$ can be written as:

$$S(k+1) = S(k) + [\Gamma]_{N\times N}[U(k)]_{N\times 1} \tag{8}$$

The $[\Gamma]_{N\times N}$ matrix is a diagonal matrix where the diagonal element, Γ_{ii} corresponds to $\gamma(k)$ of the i^{th} slack. The matrix $[U(k)]_{N\times 1}$ is the control input where the i^{th} entry represents $f_i(k) - f_{ref_i}(k)$.

Controllability of the state space model in (8):
Using the analysis given in [15], [16], one can show that the system given in (8) is controllable under the condition where each core has its own DVFS. In practice, hardware restrictions may force multiple cores to share the same clock frequency, an issue that makes the system more difficult to control. This means that we need to reduce the number of states in the slack space to be equal to the number of controllable clock frequencies.

C. State space reduction

In order to reduce the number of slack states while meeting the required performance SLAs we utilize the superposition property in the performance relation (4). Let's start first with writing the general formulas for the IPS_{ref} and IPS_{max} when there are N cores each is executing a task, that is: $IPS_{ref} = \sum_{i=1}^{N} IPS_{ref_i}$ and $IPS_{max} = \sum_{i=1}^{N} IPS_{max_i}$ Next, we need to incorporate the slack variables in the performance model. To do so, we rewrite the performance equation (4) in terms of *absolute number of instructions executed* as follows:

$$\sum_{i=1}^{N} IPS_{ref_i}(T_{act} + s_i) = \beta \sum_{i=1}^{N} IPS_{max_i} T_{act} \tag{9}$$

The right side of the equation counts the target number of committed instructions over the period of time, T_{act}. In ideal case when all slack equal to zero, the right side of this equation should equal to $\sum_{i=1}^{N} IPS_{ref_i} T_{act}$. We include the effect of slack, s_i, by adding it to the execution period T_{act} as shown on the left side of the above equation. This means that if the slack of a particular core is positive then we execute more instructions than the case of zero slack and vice versa. We note that no slack is included in the right side of the equation since IPS_{max_i} represents the case when all cores execute at the highest clock frequency. Next, we need to find the condition that needs to hold for the slack values in order to meet the desired performance. To do so, we first rewrite this equation to separate the slack as follows:

$$\sum_{i=1}^{N} IPS_{ref_i} T_{act} + \sum_{i=1}^{N} s_i IPS_{ref_i} = \beta \sum_{i=1}^{N} IPS_{max_i} T_{act} \tag{10}$$

To meet the QoS condition, the second sum in the left side of this equation, $\sum_{i=1}^{N} s_i IPS_{ref_i}$, has to be equal to zero. To generalize this, we can state that each subset of cores, g, in the CPU package that share a single clock frequency, must satisfy the following condition in order to meet the target performance SLAs:

$$\sum_{i \in g} s_i IPS_{ref_i} = 0 \tag{11}$$

Using equation (11) and (7), the state-space formulation for the instruction count slack for a subset of cores that shares a single clock frequency can be reduced to a single state as follows:

$$\begin{aligned} s_I(k+1) &= s_I(k) + f(k) \sum_{i \in g} \lambda_i(k) - \sum_{i \in g} \lambda_i(k) f_{ref_i}(k) \\ &= s_I(k) + f(k)\Lambda - \Theta \end{aligned} \tag{12}$$

where $f(k)$ is the input clock frequency, $s_I(k)$ is the slack in number of instruction execution, g is the set of cores that shares same frequency and same β, and $\lambda_i(k) = IPS_{ref_i}(k) \frac{S_{F_i}(k) T_{act}(k)}{f_{ref_i}(k)}$. The value of $IPS_{ref_i}(k)$ can be computed as follows:

$$IPS_{ref_i}(k) = IPS_i(k) \frac{1}{1 + S_{F_i}(\frac{f(k)}{f_{ref_i}(k)} - 1)} \tag{13}$$

The value of f_{ref_i} is computed using (6) which is determined based on S_{F_i} and the desired β_i. The formulation given in (12) shows that it is possible to merge the slack states of a group of cores into a single state that is controlled by a single frequency $f(k)$. This indicates that it is possible to make the number of states in the slack vector equals to the number of controllable frequencies, hence the system can be fully controllable. Let us define the state vector with M states as, $S_I(k) = [s_{I_1}(k), s_{I_2}(k), ..., s_{I_M}(k)]^T$. The state-space formulation of $S_I(k)$ can be written as:

$$S_I(k+1) = S_I(k) + [A]_{M \times M}[F(k)]_{M \times 1} - [B]_{M \times 1} \tag{14}$$

where $[F(k)]_{M \times 1}$ is the vector of input clock frequencies, $[A]_{M \times M}$ is a diagonal matrix where the element A_{ii} corresponds to the value of Λ in the s_{I_i}. For the vector B, the B_i element represents the value of Θ in the s_{I_i}. Since $[A]_{M \times M}$ is a diagonal matrix, we can treat each state independently an issue that simplifies the controller design. In the following section we focus on the design of the state-space controller to be used for adjusting the input frequency in away that uses all of the available slack while ensuring stability.

D. State-Space controller

In this section we address the design of a feedback controller that dynamically manages the controllable frequencies to achieves the desired performance constraints. The relation (14) shows that the individual slack variables, s_{I_i}, that belong to the set of cores which share the same clock frequency can be controlled independently, an issue that simplifies the controller design. The objective of the controller is to converge the slack variable to zero. We achieve that by using the control law which is the linear feedback of states under control. The instruction slack equation given in (12) has the term Θ which is not controllable by the input frequency. We resolve that through simple algebraic manipulation as shown below:

$$\begin{aligned} \Lambda_i f_i(k) - \Theta_i &= -G_i \Lambda_i s_{I_i}(k) \\ f_i(k) &= -G_i s_{I_i}(k) + \frac{\Theta_i}{\Lambda_i} \end{aligned} \tag{15}$$

where G_i is the state feedback gain of the i^{th} slack and $f_i(k)$ is the frequency setting that is required to meet our objectives. Next, we need to find the value of G_i that will place the eigenvalue of the closed loop system within the unit circle to converge the slack to zero. To calculate the desired gain we need to obtain the characteristic equation of the system which has to be in z-domain [15]. The characteristic equation of this system is $|z - 1 + \Lambda_i G_i| = 0$. Using the characteristic equation, we can calculate the feedback gain as, $G_i = \frac{1-z}{\Lambda_i}$, where $1 > z \geq 0$. The controller's transient time constant, τ, is another important metric that needs to be calculated. The controller represents a simple first order system. Using first order system analysis, the transient time of the controller can be computed as, $\tau = \frac{-T_s}{ln(1-\Lambda_i G_i)}$, where T_s is the controlling interval. The controller response speed depends on the location of eigenvalue and T_s. In general, the response speed increases as the value of the eigenvalues become closer to the origin of the unit circle or the value of T_s is reduced. However, the changes in the input clock frequency are expected to increase with the reduction in the response time.

Power reduction: The controller delivers power reduction when the value of $\beta < 1.0$. This is because the controller reduce the frequency to what is just needed to meet the desired performance target. The controller ensure minimal frequency since it incorporates the scalability factor information in its performance model which allows it to reduce the frequency when the application scalability factor is below one, thus minimizing the power consumption. The power reduction ratio is normally higher than $1 - \beta$ and it increases inversely to the scalability factor.

Overhead: The computational overhead at each controlling tick involves calculating the new frequency using (15) and updating the slack states (12). These ordinary computations can be done in no more than few microseconds. The overhead of DVFS switching is also in the range of microseconds [1]. On the other hand, the period between consecutive controlling ticks is in the range of tens of milliseconds. This indicates that the overhead of our controlling mechanism is negligible.

TABLE I
SPEC2K BENCHMARKS CHARACTERISTICS

Benchmark	IPC	Characteristics
bzip2	1.30	CPU bound
perl	2.05	CPU bound
gcc	1.38	CPU bound
mcf	0.31	Memory bound
equake	0.65	Memory bound
swim	0.57	Memory bound

V. EVALUATION

A. Methodology

We evaluate our approach using a state of the art test bed, a 32nm Intel hexa-core dual socket Westmere Xeon with a total DRAM/DDR3 memory of 12GB. Our Wsetmere processor supports 6 operating frequencies (1.6, 1.733, 1.867, 2.0, 2.133, 2.267 GHz). The hardware design of Westmere processor allows only a single frequency for all cores in a socket at any point in time. To accommodate this restriction we use our state-reduction method to get a single slack for an entire set of cores in the CPU socket. We implemented our state-space controller in latest Linux kernel (2.6.33.2) in place of the *ondemand* policy for managing the p-states. We use the CPU's hardware performance counters to estimate the scalability factor and IPS. To measure these metrics we need to determine the scalable time active time and number of retired instructions. We use one counter for each of these metrics (e.g. *unhalted core cycles* counter for measuring the active time. We use an eigenvalue of 0.1 for the controller gain so it can converge in about one scheduling tick or so. The sampling interval of our controller is set to 20ms. Please refer to section IV.D for discussion on the controller sensitivity to the values of eigenvalue and sampling interval. We use this setup and evaluate our controller by executing batch class (throughput critical) workloads. For a more comprehensive analysis of our technique, we evaluate it by executing service class workloads (e.g. web servers). Running service class workloads requires cluster of multiple physical machines. Owing the limited availability of physical machines, we simulated our controller and used real-life traces from actual measurements in the simulations.

We use representative benchmarks from the Spec2K suite for the batch class workloads (see Table I)[9]. A set of benchmarks with various levels of CPU intensity are selected to emulate real life applications. We run each benchmark in the work set till its completion. For service class workloads we use SpecWeb benchmarks that are used commonly for web server performance evaluation [10].

We compare our technique against the default *ondemand* policy in Linux (described in section II). However, for batch class of jobs with high utilization the *ondemnad* would set the frequency merely to maximum, hence it cannot constrain performance to given bounds. We also evaluate our method against a performance aware policy that is suitable for high utilization jobs, Fixed Frequency Performance Aware, FFPA, that sets the frequency, f, to a value that is based on the given performance target as, $f = \beta f_{max}$. The frequency is rounded up to the closest value available to prevent performance violations.

B. Evaluation

Figure 3 shows the robustness of our controller to meet the required SLAs performance. We run workloads on a single CPU socket that span from CPU intensive to memory intensive with single as well as multiple threads. We select representative performance targets to show that our

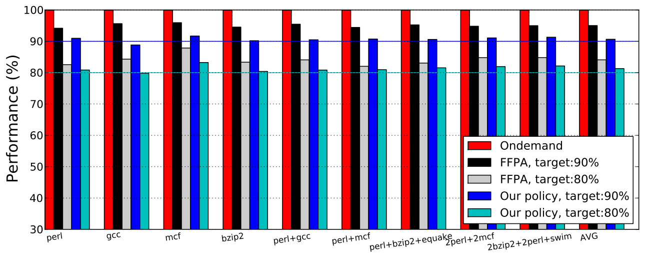

Fig. 3. Performance SLAs with a single CPU socket

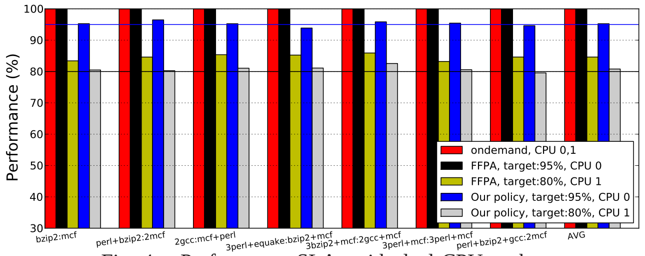

Fig. 4. Performance SLAs with dual CPU sockets

controller is not limited to a particular value. The results clearly show that our controller is capable of meeting the desired objectives. In general, controlling the p-states when running multiple threads is more challenging compared to single thread execution because of interferences in the last level cache. However, our controller performs quite well across these varied cases. For instance, the results obtained for the case of *perl* (cpu intensive) and $\{$*2perl+2mcf*$\}$ (*mcf* is memory intensive) are equally good even though the latter run has a substantial increase in the number of last level cache conflicts. The results also show that our policy outperforms FFPA since FFPA cannot exploit the workload dynamics in terms of frequency scalings. For the case of default Linux policy, *ondemand*, the results show it is ineffective in controlling the performance of this class of high utilization jobs as it would merely increase the frequency to maximum. To accurately evaluate our technique we calculated the standard deviation from performance target using the results in Figure 3. The reduction in average standard deviation over the FFPA and *ondemand* policies is as high as 72% and 91% respectively.

In Figure 4 we show that the capability of our controller to meet the target SLAs is not limited to a single CPU package. In this experiment we execute workload on the two CPU sockets in the system (CPU0, CPU1), where the workload includes a mix of CPU and memory intensive applications. We set the performance targets of the workloads of the two CPU sockets to be different, (95% and 80%). We use the notation (:) to separate the workload of the two sockets. The results show that our controller is able to meet the performance target in all cases and outperform both FFPA and *ondemand*. Our controller is able to perform equally well when there is a single thread is executing in each socket, $\{$*bzip2:mcf*$\}$, as well as when there is multiple threads in each one, $\{$*perl+bzip2+gcc:2mcf*$\}$. We also calculated the reduction in average standard deviation over the FFPA and *ondemand*; it reaches, 80% and 92% respectively.

We will now discuss the performance of our controller compared to the *ondemand* policy in managing the p-states for service class workloads (using SpecWeb traces). The results of applying *ondemand* policy are shown in our earlier Figure 1 where the performances are averaged over a sliding window of 100ms. The results show that the *ondemand* policy can cause large variations in performance which can lead to

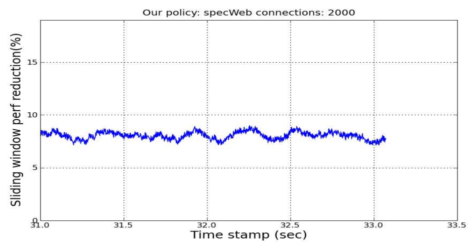

Fig. 5. Running SpecWeb with our controller

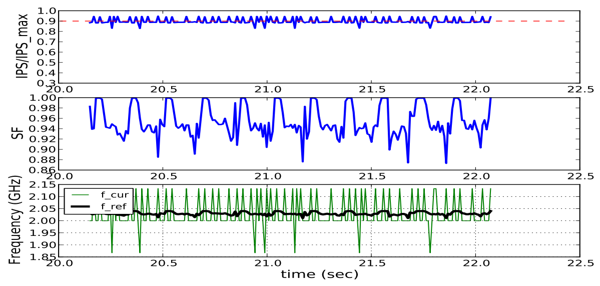

Fig. 6. Controller behavior with executing bzip2

performance violations. To compare it with our policy we use the performance hit from the *ondemand* policy and set a corresponding desired performance level for our controller. Figure 5 shows the results of applying our policy. The results show that the performance variation is reduced by 92% which is significant.

Next we give results pertaining to the run time behavior of our controller. In these experiments, we set the desired performance to 90%. Figure 6 shows results of executing a single thread of *bzip2* on Westmere processor. The top portion of the graph shows how the ratio between current and the maximum IPS changes over time. The middle part of the graph depicts the scalability factor while the bottom portion illustrates the current and target frequencies. The results clearly show that the controller is able to meet the target performance in spite of large variations in scalability factor. The bottom part shows that the frequency assignment may have some fluctuation between the two adjacent frequencies (2GHz and 2.13GHz). These fluctuations occur due to the lack of enough frequencies to cover the range that is required by the controller. In spite of this limitation in the hardware, out controller is able to meet the target performance.

The other important feature of our technique is its capability to deliver CPU power reduction while ensuring performance guarantee. Figure 7 shows the power reduction comparing *ondemand* and FFPA and our technique for running workload on two sockets where both sockets have the same performance target. We study the savings over various values of performance targets. The power reduction with using our controller is the highest when compared to *ondemand* policy since *ondemand* keeps the frequency at maximum. The power reduction is expected to increases with the reduction in performance target. Our technique also outperforms FFPA policy in most cases. The savings in the case of performance target being 94% come primarily from utilizing frequency scalability in the application as there is no rounding up in the frequency setting of FFPA (rounding up frequency causes extra power dissipation). The savings for memory intensive workload e.g. ({*swim:swim+mcf*}) is higher than the case of cpu intensive ones (e.g. {*bzip2:bzip2*}) since memory intensive applications exhibit lower frequency scaling. The savings against FFPA increase further for the cases of performance targets of 90% and

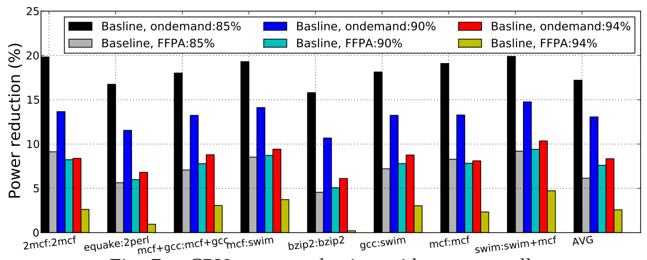

Fig. 7. CPU power reduction with our controller

85% since their target frequency is rounded up to match the available frequency. Unlike FFSA, our controller dynamically sets the frequency to just what is needed while meeting the target performance.

VI. Conclusion

This paper presented a formal approach to control the p-states of the cores in a CMP system to achieve the target performance while minimizing the power consumption. The algorithm is implemented in Linux Kernel and tested on a state of the art test bed, a 32nm Intel hexa-core dual socket Westmere Xeon. Extensive measurements using Spec2K and SpecWeb show that the algorithm delivers the target performance successfully, it reduces the standard deviation from target performance by more than 90% over state of the art policies while reducing average power by 17%.

VII. Acknowledgement

This work has been funded by NSF Project GreenLight grant 0821155, NSF SHF grant 0916127, NSF ERC CIAN, NSF Variability, NSF Flash Gorden, CNS, NSF IRNC, Translight/Starlight, Intel, Oracle, Google, Microsoft, MuSyC.

References

[1] http://www.intel.com/design/mobile/datashts/.
[2] Intel. energy-efficient performance for the data center, http://www.intel.com/it/pdf/energy-efficient-perf-for-the-data-center.pdf.
[3] www.sun.com/servers/x64/x4270/.
[4] Intel unwraps dual-core xeon server processor. In *PCWorld*, 2005.
[5] R. Ayoub and T. Rosing. Cool and save: cooling aware dynamic workload scheduling in multi-socket cpu systems. In *ASP-DAC*, pages 891–896, 2010.
[6] S. Borkar. Low power design challenges for the decade (invited talk). In *ASP-DAC '01*, pages 293–296, 2001.
[7] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *ISLPED*, pages 174–179, 2004.
[8] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISLPED*, pages 38–43, 2007.
[9] http://www.spec.org/cpu2000/.
[10] http://www.spec.org/web2005/.
[11] S. Lee and T. Sakurai. Run-time power control scheme using software feedback loop for low-power real-time application. In *ASP-DAC*, pages 381–386, 2000.
[12] B. Lin, A. Mallik, P. A. Dinda, G. Memik, and R. P. Dick. Power reduction through measurement and modeling of users and cpus: summary. pages 363–364, 2007.
[13] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *CASES*, pages 156–163, 2002.
[14] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys*, pages 237–250, 2010.
[15] K. Ogata. *Discrete-Time control Systems*. Prentice-Hall, 1995.
[16] U. Ogras, R. Marculescu, and D. Marculescu. Variation-adaptive feedback control for networks-on-chip with multiple clock domains. In *DAC*, pages 614–619, 2008.
[17] T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *ISSS*, pages 24–28, 1999.
[18] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present, and future. *Linux Symposium*, 2:223–238, 2006.
[19] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *DAC*, pages 438–443, 2001.