A Cool Scheduler for Multi-Core Systems Exploiting Program Phases

Zhiming Zhang and J. Morris Chang, *Senior Member*, *IEEE*

Abstract—Rapid growth of cloud computing services have led to creation of large scale enterprise data centers which consume great amounts of energy. Data centers usually have an service level agreement (SLA) between the clients and the service providers, which specify the terms and quality of service to be provided. In this paper, we consider a situation in a data center where multiple user applications are executing on a multi-core system and each application may have a specified SLA requirement. We design a voltage and frequency scheduler (the "cool" scheduler) that can be used in enterprise data centers to provide CPU energy saving under the specified SLA requirement by exploiting the applications' run-time program phases. Our design greatly improves the computation efficiency compared to other recently published works. The scheduler is built into the Linux kernel and evaluated against SPEC CPU2006 and Phoronix Test Suite on a quad-core system. Experiment result demonstrates that our cool scheduler achieves 25.8% energy saving on average with 8.7% performance loss under the given SLA requirement (10% allowed performance loss). Our design achieves 35.8% and 31.6% more energy saving compared to two of the most advanced related works.

Index Terms—Energy aware computing, dynamic voltage frequency scaling (DVFS), power-performance tradeoff, program phases, multi-core systems

1 Introduction

Energy management has now become a key issue for cloud computing service providers, focusing on the reduction of all energy related costs. Energy proportional computing has become a popular solution to provide energy savings among data centers. The basic idea of energy proportional computing is to minimize energy consumption in data centers while meeting the SLA (Service Level Agreement) requirement.

In general, SLA sets the expectations of service such as throughput and transaction response time between the customer and service provider. The transaction response time can be considered as the waiting time for a customer while the task is being processed in the data center. A data center usually has a minimum transaction response time t which is the case when the data center is operating at its maximum capability or generally the highest frequency. Assume the SLA transaction response time is set as T between the customer and the service provider, then the minimum transaction response time t of the data center must be smaller than T. Otherwise this data center can not provide service to this customer since the SLA can not be guaranteed. If t is smaller than T, there is chance for energy savings since the data center can operate on lower frequency but still meet the SLA. The goal of our work is to maximize energy saving without violating the SLA requirement by adjusting the CPU frequency based on application's run-time program phases in a multi-core system.

• *The authors are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010. E-mail: {zhiming, morris}@iastate.edu.*

Manuscript received 20 May 2012; revised 10 Nov. 2012; accepted 12 Nov. 2012; published online 29 Nov. 2012; date of current version 29 Apr. 2014. Recommended for acceptance by J. Xue.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2012.283

Processor frequency has always been a key metric of system performance and higher frequency generally means better overall system response or throughput. However high operating frequency may also lead to high potential of energy waste especially in data centers since cloud computing services usually contain many I/O and memory transactions. Our research attempts to minimize the energy waste caused by memory-related stall cycles by using the technique of dynamic voltage frequency scaling (DVFS). DVFS is widely used to provide energy efficient computing. Most modern computers support a simple workload based DVFS. When the system detects heavy workload, it will increase CPU frequency to provide high performance, and in the case of little workload, the system will decrease CPU frequency to save energy.

Moreover, consider the energy waste due to the speed gap between CPU and main memory. One ideal solution is to minimize the CPU frequency every time when the CPU is stalled by main memory access, and then switch back to high frequency after the stall is over. In this case, energy can be saved with no performance degradation. However in practice, CPU will be unavailable for about $50\ \mu s$ to $650\ \mu s$ [1], [2] during a DVFS operation. This time-span is much larger than the main memory latency which is around $100\ nano$ seconds. Thus DVFS can not be applied every time the processor is stalled by a main memory access.

A practical solution is to exploit the program phases (i.e., memory intensive phase and CPU intensive phase [3], [4]). The memory intensive phase is the time duration when the program has many memory activities. We can turn down the processor frequency during this time period to save energy but still achieve comparable performance. The CPU intensive phase is the time duration when most of the work is done on the CPU. The CPU should run on high frequency during this

time period to guarantee performance. We call the memory intensive phase and the CPU intensive phase two distinct "program phases". Recent works [5]–[8] try to reduce the memory-related energy waste by adjusting CPU frequency according to the program phases. However, all these works require high computation complexity and ignore the DVFS operation overheads which are substantial for heavy loaded data centers. Another major issue is these works are unable to precisely control the performance loss and the SLA may be violated. These major issues impede these works from being practically used in real data centers.

In this paper, we introduce a simple and effective voltage and frequency scheduler (the "cool" scheduler). We name it the cool scheduler because it has the ability to reduce CPU energy consumption and cool down the CPU. In our SLA model, the SLA defines a task execution time constraint for the CPU. We assume the system performance is dominated by the CPU without considering the changing latency of I/O devices or network accesses. Our scheduler greatly improves the computation efficiency compared with other recently published works. We first construct a simple model (the "cool" model) to calculate a desired running frequency for each thread given its program phases and SLA requirement. We verify our model against the industry standard benchmarks from SPEC CPU2006. Verification result shows our model has accurate prediction on most of the benchmark programs. After the desired operating frequency is determined for each thread, thread migration and task grouping are used to perform DVFS for a group of threads in a multi-core environment. This idea significantly reduces the number of unnecessary DVFS operations in recent works. We propose a feedback mechanism to ensure the actual performance approach closely to the SLA requirement. This allows our cool scheduler to precisely control the performance loss and maximize energy saving under the given SLA requirement.

The scheduler is built into the Linux 2.6.22.9 kernel. We evaluate our work on a desktop computer with Intel Core 2 Quad 8400 CPU against benchmarks from SPEC CPU2006 [9] and Phoronix Test Suite [10]. Experiment result demonstrates our cool scheduler achieves 25.8% energy saving on average with 8.7% performance loss under the given SLA requirement. It also demonstrates our scheduler achieves 35.8% and 31.6% more energy saving respectively compared to two of the most advanced related works. The main contributions of our work are:

- We propose a cool scheduler that can be used in enterprise data centers to provide CPU energy saving under the specified SLA requirement by exploiting the applications' run-time program phases.
- The proposed scheduler greatly improves the computation efficiency compared to two of the most advanced related works.
- The proposed scheduler significantly reduces the number of unnecessary DVFS operations which are ignored in recent works.
- The proposed scheduler can precisely control the performance loss and maximize energy saving with the SLA requirement always guaranteed.

The remaining of the paper is organized in the following sequence. Related work is given in Section 2. We provide our theoretical intuition and the cool model in Section 3. Section 4 introduces the feedback based voltage and frequency scheduling mechanism. The implementation of our design is provided in section 5. Section 6 exhibits the experiment results and Section 7 concludes this paper.

2 Related Work

A number of works have used DVFS related techniques to provide energy efficient computing, we limit our discussion to the methods that are most relevant to our work. Recent research on DVFS based energy efficient techniques can be classified into at least three groups. The first group of techniques use known task arrival times, workload, and deadlines to implement algorithms at the task level or operating system [11]–[19]. Horvath et al. [14] proposed a DVFS policy for multi-tier web server system that can minimize global energy consumption while meeting the multi-stage end-to-end delay constraint. Isci et al. [11] analyzed different policies for chip level power management under a specific power budget. These policies adjust power modes of individual cores targeting at different objectives such as prioritization of cores/benchmarks, balancing power among cores and optimizing system throughput.

The second group of techniques use compiler or application support for performing DVFS [20]–[27]. For example, in [25], the authors provide an application level power management by using the knowledge provided by the application to save energy. In [20], the authors use dynamic profiling of branch probability to characterize workload then use DVFS to maintain power-performance balance. This group of methods need additional code added to the application before it is executed on the system.

The last but not the least group of techniques use program runtime characteristics or statistics to identify the workload of a task. Then estimate and predict the optimal voltage and frequency setting [6]–[8], [28]–[36]. For example, Kotla et al. [28] use the program runtime information instruction per cycle to decide the running frequency, this method can reduce energy waste caused by memory stalls, however the scheme does not guarantee the SLA requirement. These techniques can be further classified as fine-grained or course-grained. Course-grained techniques determine the voltage and frequency setting on a task-by-task basis. Fine-grained techniques adjust the voltage and frequency setting within a task boundary and usually perform better than course-grained techniques.

Choi et al. [7] presents a fine-grained DVFS technique that minimizes energy consumption using workload decomposition which classifies workload as either on-chip or off-chip. The authors propose a regression based model to calculate the optimal running frequency for a program. Chen et al. [6] uses last level cache misses per instruction (MPI) as an indicator of energy consumption. Given the program's MPI distribution, the corresponding energy consumption and other statistics, the DVFS control problem is formulated into a multiple choice knapsack problem (MCKP) with the goal of minimizing total energy consumption.

However, both works require high computation complexity: using regression based model or solving an NP-hard MCKP. Besides, they ignore the DVFS operation overhead (invoking DVFS at every context switch or every 30 ms) which

is significant for heavy loaded data centers. Another major issue is the prediction errors in these two works. They are unable to precisely control the performance loss and the SLA requirement may be violated. To overcome these issues, we design a DVFS scheduler that has little computation complexity (O(1) compared to O(N) in [7], [6]). We use the idea of task grouping and thread migration [5] to perform DVFS for a group of threads in a multi-core environment. This significantly reduces the DVFS operation overheads. We propose a feedback mechanism to precisely control the performance loss and maximize energy saving with SLA always guaranteed.

3 Motivation and Model

Program run-time behavior can be categorized into two phases: memory-intensive phase and CPU-intensive phase [37]. In the memory intensive phase (frequent last level cache miss), the CPU spends significant amounts of time waiting for memory transactions thus wasting energy. Slowing down the CPU frequency during this time could provide energy savings while still achieve comparable performance. We use a simple experiment to illustrate this idea. We execute mcf (a benchmark program from SPEC CPU2006 used for single-depot vehicle scheduling in public mass transportation) on two different frequencies and then examine its program behaviors. Fig. 1 shows the execution behavior (MAPI vs time) of mcf when CPU is running on 1.998 GHz and 2.664 GHz respectively. MAPI is the number of Memory Access Per Instruction which can be used as an indicator of a program's memory access intensiveness. Observation shows two distinct phases: memory intensive phase ($MAPI > 0.008$) and computation intensive phase ($MAPI < 0.006$). The execution time for memory intensive phases is about the same no matter when the program is running on 1.998 GHz or 2.664 GHz. However, CPU running on 1.998 GHz causes the execution time of computation intensive phases obviously longer than when the CPU is running on 2.664 GHz.

Observations demonstrate performance drops when CPU frequency is reduced. For the same amount of frequency drop, the performance degradation depends on the program phases. This implies performance suffers less degradation at memory intensive phase for the same amount of frequency drop. This motivates us to switch down the frequency during memory intensive phases to save energy without much performance degradation. Another important observation is that the program tends to have similar run-time behaviors [38], [39] within a time span. Usual time spans for similar run-time behaviors are in seconds or tens of seconds. As we observe from the execution of "mcf" in Fig. 1, the time span for memory intensive phases is about 7 seconds, and about 20 seconds for CPU intensive phases. This feature is used to predict the program's future behavior.

3.1 Theoretical Bounds of DVFS Energy Savings

The experiment above provides the motivation to switch down CPU frequency during memory intensive phases for energy savings. In this section, we provide the theoretical bounds of DVFS energy savings when executing a program under a time constraint. Theoretically, when the program contains only memory access instructions, the program will stay in memory intensive phase throughout the execution.

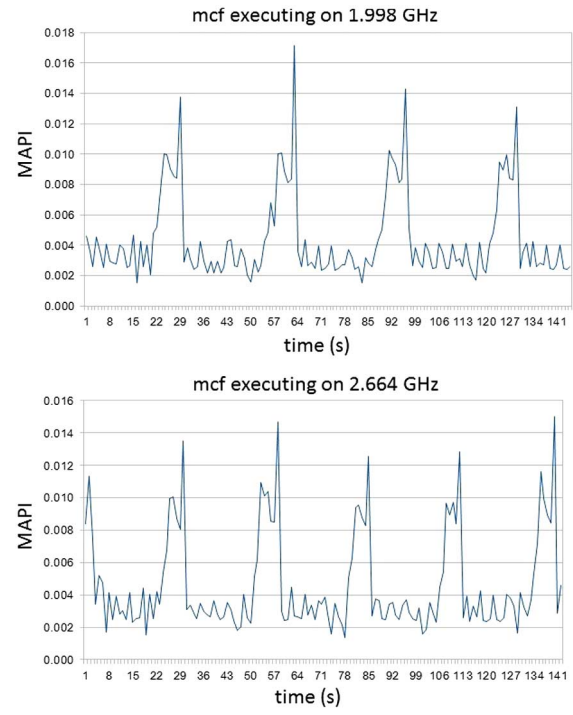

Fig. 1. Execution behavior of mcf on 1.998 GHz and 2.664 GHz.

In this case, maximum or upper bound of energy saving can be achieved since the program can be executed on the lowest frequency without performance loss (assume CPU is much faster than memory). On the other hand, when the program has no memory access, the program stays in CPU intensive phase throughout the execution. In this case, the minimum or lower bound of energy saving will be reached. These two bounds are given in the following.

Assume the SLA requirement (time constraint) for program execution is T. Consider a CPU that supports multiple operating frequencies and assume under the same time constraint, executing the program under lower frequency consumes less energy. F represents the maximum CPU frequency. T_F and P_F are the execution time and power consumption respectively when CPU is operating at F. We add a constraint that T_F is smaller than T so the time constraint can be guaranteed when CPU operates at maximum frequency. f_{min} represents the minimum CPU frequency. T_{fmin} and P_{fmin} are the execution time and power consumption when CPU is operating at f_{min}.

When the program contains only memory access instructions, it can be executed on f_{min} without performance loss and the upper bound of energy saving ΔE_U is achieved:

$$\Delta E_U = P_F T_F - P_{fmin} T_{fmin}. \quad (1)$$

When the program has no memory access, the lower bound ΔE_L is reached:

$$\Delta E_L = P_F T_F - P_{f_k} T_{f_k}, \quad (2)$$

where f_k is the CPU frequency that allows $T_{f_k} = T$, so the time constraint is strictly met. T_{f_k} and P_{f_k} are the execution time and power consumption respectively when CPU is operating at f_k. When the program contains both memory and non-memory instructions, the amount of energy saving stays between ΔE_L and ΔE_U.

3.2 Model

The energy saving capability of DVFS strategies depends on the SLA and the amount of memory accesses in a program. CPU frequency must be carefully chosen based on the distribution of the memory accesses. The scheduler must be able to identify program phases and make DVFS decisions at runtime. We propose a model that can provide the desired running frequency based on the SLA and the program phases. This model shows great computation efficiency compared to recent works [7], [6] and it can be built into OS kernel for commercial use. The program phases are mainly determined by three statistics captured at run-time using performance monitors: $MAPI, CPI_{exe}, h(f)$. We first give definitions for the behavior statistics we use.

- MAPI: Memory Access Per Instruction, to determine the memory access intensiveness of a thread [40].

$$\text{MAPI} = \frac{Bus\_Trans\_Mem}{Instr\_Exe}, \tag{3}$$

  where Bus_Trans_Mem is the number of main memory accesses and $Instr\_Exe$ is the number of instructions executed.
- CPI_{exe}: cycle per instruction when CPU pipeline not stalled by memory transactions.
- $IC(f)$: instruction count, total number of instruction executed in one second at CPU frequency f.
- Δm: latency of the main memory.
- $h(f)$: number of cycles when CPU is halted while operating at frequency f.
- $o(f)$: stall cycles caused by reasons other than memory access while CPU running at f.
- α: memory latency overlap rate. This factor represents the out-of-order execution before CPU gets stalled by a memory access.

The cycle usage for a CPU operating on frequency f within a second can be expressed as:

$$f = \text{IC}(f) \times \text{CPI}_{\text{exe}} + \alpha \times MAPI \times \text{IC}(f) \times \Delta m \times f + h(f) + o(f), \tag{4}$$

where $IC(f) \times CPI_{exe}$ is the number of cycles while the CPU is not stalled by memory transactions neither halted. $\alpha \times MAPI \times IC(f) \times \Delta m \times f$ is the number of stall cycles due to main memory access. Notice that quantities on both sides of eq. (4) are in cycles/sec. $h(f)$ represents the number of cycles when CPU is halted while operating at frequency f. The CPU gets halted when there is no work to be done, the CPU starts running an idle thread (HLT instructions) and enters its idle state. CPU stall happens when the CPU is still executing program instructions but waiting for the operand or data (usually because of the latency of memory) to be available.

In a recently published model [7], the authors ignore the effect of out-of-order execution and memory level parallelism [41] in superscaler processors which leads to prediction errors. In our model we define α to represent this effect and enhance the accuracy of our model. The value of α is determined by the processor issue rate, re-order buffer size and system memory latency. In general, most of the stall cycles are caused by main memory access, thus we can ignore the $o(f)$ (e.g., L1 cache miss and branch miss prediction related stalls) in eq. (4) with little impact on the accuracy of eq. (4). Eq. (4) is rewritten into:

$$f \approx \text{IC}(f) \times \text{CPI}_{\text{exe}} + \alpha \times MAPI \times \text{IC}(f) \times \Delta m \times f + h(f). \tag{5}$$

The instruction count IC(f) can be derived from eq. (5):

$$\text{IC}(f) \approx \frac{f - h(f)}{\alpha \times MAPI \times \Delta m \times f + \text{CPI}_{\text{exe}}}. \tag{6}$$

In our performance model, we consider instruction count in a given interval of time as the performance measure of a thread [28]. Thus, the performance loss for CPU running on frequency f compared to CPU running on the highest frequency F can be defined as:

$$\delta = \frac{\text{IC}(F) - \text{IC}(f)}{\text{IC}(F)}. \tag{7}$$

When the SLA requirement is given as a percentage of the maximum system performance, the required performance loss can be calculated as:

$$\delta = 1 - SLA. \tag{8}$$

Consider a time-sharing multi-tasking system, each thread is given a time slice to execute on the CPU. Let f^{n-1} be the frequency level of a thread t's $(n-1)$th execution, its program behaviors CPI_{exe}, $MAPI$, $h(f)$ are monitored by the CPU during t's $(n-1)$th execution. Experiment demonstrates that the number of halted cycles $h(f)$ depends on CPU frequency:

$$h(f_1) \approx \frac{f_1}{f_2} \times h(f_2), \tag{9}$$

where f_1 and f_2 are two different CPU frequencies. After collecting all the program behavior statistics, combine eq. (6), (7), (9) and obtain eq. (10), which is the equation that provides the desired operating frequency for t's nth execution:

$$f^n_{target} = \frac{IC(F) \times (1-\delta) \times CPI_{exe}}{1 - IC(F)(1-\delta)\alpha \times MAPI \times \Delta m - \frac{h(f^{n-1})}{f^{n-1}}}, \tag{10}$$

where $IC(F)$ can be derived from eq. (6).

$$\text{IC}(F) \approx \frac{F - \frac{F}{f^{n-1}} h(f^{n-1})}{\alpha \times MAPI \times \Delta m \times F + \text{CPI}_{\text{exe}}}. \tag{11}$$

Eq. (10) is our proposed model that provides the desired operating frequency f_{target} for a thread given its program phases and SLA requirement (i.e., target performance loss δ). The computation complexity of our model is O(1) compared to O(N) in two recent works [7], [6].

3.3 Model Evaluation

The accuracy of our model is evaluated against benchmark programs from SPEC CPU2006. System configuration is given in Section 6. The basic evaluation idea is to compare the value derived from the proposed model with the actual value from

TABLE 1
Model Evaluation Result

Program	MAPI	CPI_{exe}	Error
gcc	0.0014	0.576	0.8%
astar	0.0009	0.701	7.8%
bzip2	0.006	0.553	2.3%
perlbench	0.0015	0.572	1.4%
gobmk	0.00141	0.581	3.5%
hmmer	0.000466	0.472	0.6%
sjeng	0.002	0.612	2.7%
libquantum	0.0018	0.588	1.5%
h264ref	0.0015	0.549	0.7%
omnetpp	0.0014	0.565	0.7%
xalancbmk	0.002	0.676	1.5%
mcf	0.013	0.543	1.3%

the performance monitors. First, capture the behavior statistics used in the model while running the benchmark programs on the highest frequency F. Second, calculate the performance loss δ assuming the operating frequency is set to f which is different from F. Finally, compare the calculated performance loss with the actual performance loss to get the accuracy of our model. Eq. (12) shows how to calculate the error:

$$error = \frac{\delta_{prediction} - \delta_{actual}}{\delta_{actual}}. \tag{12}$$

Evaluation result is demonstrated in Table 1. $MAPI$, CPI_{exe}, and the error for each program is shown in the table. The benchmark programs are executed on $F = 2.664$ GHz, $f_1 = 2.333$ GHz, and $f_2 = 1.998$ GHz each three times. The average performance loss is calculated for comparison. The error rate ranges from 0.5% to 7.8% and 2.1% on average. The result demonstrates our model can make accurate prediction on most of the benchmark programs. The error in our model mainly comes from the inaccurate estimation of the effect of out-of-order execution and memory level parallelism that vary at run-time. Another reason for the error are the stalls caused by, e.g., data dependency, branch miss prediction etc. We assume most of the stalls come from memory transactions and ignore other stalls in the model. $astar$ (a program derived from a portable 2D path-finding library [9]) suffers a 7.8% prediction. This is because for $astar$, most of the stalls are caused by data dependencies and branch miss predictions instead of memory transactions (MAPI is less than 0.001). One more reason for the error comes from the estimation of the halted cycle $h(f)$. The actual value deviates slightly from our estimation eq. (9).

4 Feedback Based Voltage and Frequency Scheduler

The cool model has been derived and provides the desired running frequency for a thread given the SLA requirement and the program phases. In this section, we introduce our cool scheduler. Fig. 2 demonstrates the architecture of our scheduler. This is a situation where multiple user applications are running on a multi-core server and each application has a specified SLA requirement. The proposed Cool Scheduler works in four steps:

Step 1: Statistics Collection. At the user application level, each App is given an SLA requirement. At the hardware level,

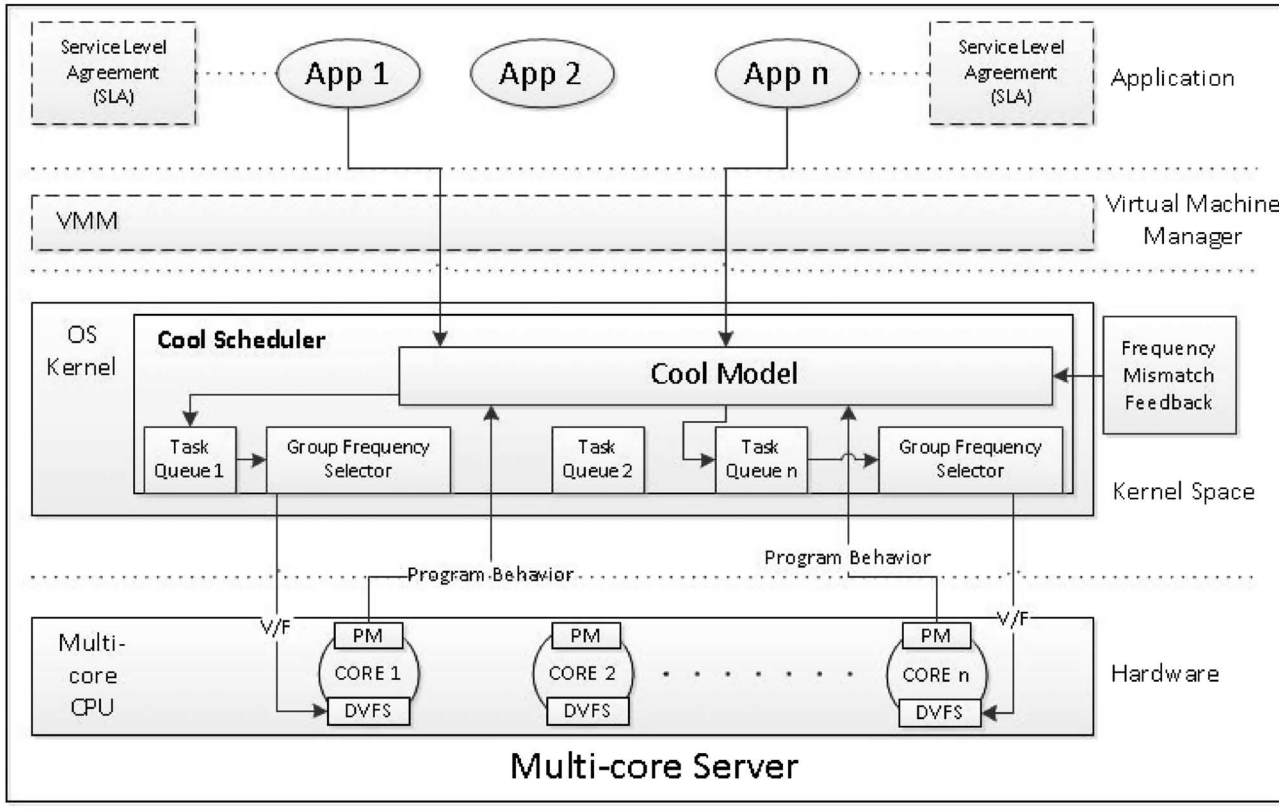

Fig. 2. Scheduler architecture.

the CPU performance monitors (PMs) keep monitoring the Apps' Program Behavior.

Step 2: Desired Frequency Calculation. All the statistics collected in Step 1 along with the Frequency Mismatch Feedback will be sent to the Cool Model. The model uses the statistics to calculate the desired operating frequency for each thread.

Step 3: Task Grouping. The Group Frequency Selector groups the threads with the same target frequency onto the same CPU core using thread migration.

Step 4: Apply DVFS to each CPU core. After the task grouping is complete, the group frequency is determined by the Group Frequency Selector and applied to each CPU core accordingly.

The goal of Step 3 is to minimize the overhead caused by DVFS operations. Past works [7], [6] ignore the DVFS operation overhead and apply DVFS to each thread every tens of milliseconds. Typically, the DVFS overhead accounts for 1–3% of the task execution time. However, this is a substantial portion of the overall performance degradation. Assume the DVFS overhead ranges from 150 to 250 microseconds (measured in Section 6), and DVFS is applied to each thread every 10s of milliseconds. This implies that if the overall performance degradation is 10%, the overhead of DVFS transitions accounts for up to 25% of the degradation (ideally, the performance degradation should only be caused by slowing down CPU frequencies during memory intensive phases). This is why we claim DVFS operation overheads are substantial and unnecessary DVFS operations need to be reduced. We use the idea of task grouping and thread migration to group the threads that have the same target frequency onto the same CPU core, and then apply DVFS to this core. This process is done by the Group Frequency Selector. The task grouping method can significantly reduce the number of unnecessary DVFS operations.

In Step 4, the target frequency is applied to each core. However, a challenge is that modern CPUs do not provide continuous frequency levels thus a thread might have to execute on a frequency different from its target frequency. This situation is called a frequency mismatch. When frequency mismatch happens, the actual performance loss for this thread will deviate from its target performance loss and this might lead to SLA violation. We propose a feedback mechanism to guarantee the SLA. It also ensures the actual performance approach closely to the SLA. Through this way, our cool scheduler can precisely control the performance loss and maximize energy saving under the given SLA requirement. Continuous feedbacks are provided to each thread after each execution. There are two types of feedbacks in this mechanism: feedback due to frequency mismatch denoted as δ_{freq}, and feedback Due to DVFS overhead denoted as δ_{DVFS}.

4.1 Frequency Mismatch Feedback

Frequency mismatch happens when a thread is being executed on a frequency different from its target frequency. A continuous feedback δ_{freq} is provided to each thread in case of frequency mismatch. First, definitions used in this feedback mechanism are given as follows:

- $f^i_{target}(t)$: the f_{target} of the thread in its ith execution.
- $\delta^i_{target}(t)$: the *target performance loss* including feedback of thread t in its ith execution.

$$\delta^i_{target}(\mathrm{t}) = \frac{\mathrm{IC}^i(F(\mathrm{t})) - \mathrm{IC}(f^i_{target}(\mathrm{t}))}{\mathrm{IC}^i(F(\mathrm{t}))}. \tag{13}$$

- $\delta_{overall}(t)$: overall *target performance loss* for thread t which is preset by a user ($\delta = 1 - SLA$).
- $\delta^i_{operating}(t)$: actual performance loss for thread t in its ith execution.

$$\delta^i_{operating}(\mathrm{t}) = \frac{IC(F(\mathrm{t})) - IC(f^i_{operating}(\mathrm{t}))}{IC(F(\mathrm{t}))}. \tag{14}$$

- $IC^i(F(t))$: Instruction count while thread t running on F (highest frequency) in its ith execution.
- $f^i_{operating}(t)$: thread t's actual operating frequency which might be different from its f_{target}.
- $\Delta IC^i(t)$: Instruction count offset due to the difference between the $f^i_{target}(t)$ and the $f^i_{operating}(t)$.

$$\Delta IC^i(\mathrm{t}) = IC(f^i_{target}(\mathrm{t})) - IC(f^i_{operating}(\mathrm{t})). \tag{15}$$

- $\delta^i_{freq}(t)$: system feedback due to frequency mismatch from thread t's ith execution.

When the actual performance loss deviates from the *target performance loss* during thread t's ith execution, there will be an instruction count offset $\Delta IC^i(t)$ and this offset should be taken into account to determine the $f^{i+1}_{target}(t)$. The number of instructions that must be executed in the $(i+1)$th execution in order to achieve the overall *target performance loss* $\delta_{overall}(t)$ is:

$$IC(f^{i+1}_{target}(\mathrm{t})) = IC(f^{i+1}_{target}(\mathrm{t})) + \Delta IC^i(\mathrm{t}). \tag{16}$$

$IC(f^{i+1}_{target}(\mathrm{t}))$ on the right hand side is the original number of instructions that need to be executed in the $(i+1)$th execution. $IC(f^{i+1}_{target}(\mathrm{t}))$ on the left hand side is the actual number of instructions that need to be executed while taking account of the system feedback $\Delta IC^i(\mathrm{t})$ due to frequency mismatch. Then use eq. (17) to calculate the *feedback* due to frequency mismatch, which should be added to the $\delta_{overall}(t)$ to get the value of $\delta^i_{target}(t)$:

$$\delta^i_{freq}(\mathrm{t}) = \frac{IC(f^i_{target}(\mathrm{t})) - IC(f^i_{operating}(\mathrm{t}))}{IC(F(\mathrm{t}))}. \tag{17}$$

Notice that $\delta^i_{freq}(t)$ can also be expressed as the difference between actual performance loss and *target performance loss* $\delta^i_{target}(t)$.

$$\delta^i_{freq}(\mathrm{t}) = \delta^i_{operating}(\mathrm{t}) - \delta^i_{target}(\mathrm{t}). \tag{18}$$

4.2 Feedback Due to DVFS Overhead

To minimize the deviation of a thread's actual performance loss from its *target performance loss*, the overhead of DVFS transitions also need to be taken into consideration. During the DVFS operation, the processor becomes unavailable for

10 μs to 650 μs [2], [1]. For heavy loaded data centers with large number of threads, DVFS overhead can degrade a thread's performance thus should not be ignored. We introduce the feedback δ_{DVFS} that takes DVFS overhead into account while calculating the f_{target}. To clearly illustrate this feedback idea, we demonstrate how to calculate the feedback due to DVFS overhead in context of the Linux 2.6 task scheduler. We first introduce the active array and the expired array in the Linux 2.6 scheduler. The active array [42] has all the threads with remaining timeslices. The expired array contains all the threads that have exhausted their timeslices but are not terminated yet. Their timeslices will be recalculated for the next execution. When the active array becomes empty, i.e., all the threads have exhausted their timeslices, the two arrays are swapped and the expired array becomes the active array and vice versa.

When a thread is in the active array, its performance is affected by the DVFS overheads of the threads with higher priorities. Assume thread t is given a timeslice $T^i_{exe}(t)$ for its ith execution. Assume there have been N_a DVFS operations before t starts its ith execution. The CPU unavailable time due to DVFS transition is denoted as $T_{unavailable}$. The first component of the DVFS overhead feedback $\delta^i_{DVFS\_active}(t)$ is calculated using eq. (19).

$$\delta^i_{DVFS\_active}(\mathrm{t}) = \frac{N_a(\mathrm{t}) \times T_{unavailable}}{T^i_{exe}(\mathrm{t})}. \tag{19}$$

On the other hand, when the thread enters the expired array after its ith execution, it has to wait for the threads in the active array to finish. During this time, assume there have been N_e DVFS operations and thread t is given a timeslice $T^i_{exe}(t)$. The second component of the DVFS overhead feedback $\delta^i_{DVFS\_expired}(t)$ is calculated using eq. (20).

$$\delta^i_{DVFS\_expired}(\mathrm{t}) = \frac{N_e(\mathrm{t}) \times T_{unavailable}}{T^i_{exe}(\mathrm{t})}. \tag{20}$$

Notice that the *feedback* due to DVFS overhead $\delta^i_{DVFS}(t)$ is equal to the first component $\delta^i_{DVFS\_active}(t)$ when the thread remains in the active array after its execution. However, if the thread enters the expired array after its execution, $\delta^i_{DVFS}(t)$ is equal to the sum of the two components as in eq. (21). This is because it has to wait for the remaining threads in the active array to finish before it can be put back to the active array again. And once it enters the active array, it also has to wait for the threads with higher priorities to finish.

$$\delta^i_{DVFS}(\mathrm{t}) = \delta^i_{DVFS\_active}(\mathrm{t}) + \delta^i_{DVFS\_expired}(\mathrm{t}). \tag{21}$$

Algorithm 1 shows how to calculate the *feedback* component due to DVFS overhead.

Algorithm 1. The *feedback* due to DVFS overhead

Require: A thread t

Ensure: *feedback* component due to DVFS switches

1. $N_a(t) \leftarrow$ Number of DVFS switches during the time thread t staying in the active array and before its start of execution.
2. $T_{exe}(t) \leftarrow$ Timeslice thread t is allocated for execution. $\delta^i_{DVFS}(t) \leftarrow \frac{N_a(t) \times T_{unavailable}}{T_{exe}(t)}$
3. **if** (context switch) **then**
4. **if** (thread t enters the expired array) **then**
5. $N_e(t) \leftarrow$ Number of DVFS switches during the time interval of thread t staying in the expired array.
6. $T_{exe}(t) \leftarrow$ Timeslice thread t is allocated for execution.
7. **else**
8. $N_e(t) \leftarrow 0$.
9. $T_{exe}(t) \leftarrow 0$
10. **end if**
11. **end if**
12. $\delta^i_{DVFS}(t) \leftarrow \delta^i_{DVFS}(t) + \frac{N_e(t) \times T_{unavailable}}{T_{exe}(t)}$

4.3 Total Feedback

After calculating both the frequency mismatch feedback and the DVFS overhead feedback. We can get the total feedback from a thread t's ith execution:

$$\delta^i(t) = \delta^i_{freq}(t) + \delta^i_{DVFS}(t). \tag{22}$$

After each execution, add this feedback to thread t's next execution $(i+1)$th to determine its target performance loss:

$$\delta^{i+1}_{target}(\mathrm{t}) = \delta_{overall}(\mathrm{t}) - \delta^i(t). \tag{23}$$

Then put $\delta^{i+1}_{target}(t)$ into the model eq. (10) to calculate $f^{i+1}_{target}(t)$.

5 Implementation

The cool scheduler is built into the Linux kernel 2.6.22.9. Most of our modifications are on the task scheduler without interfering its original functions. Fig. 2 demonstrates our scheduler architecture. In this section, we demonstrate how to capture each thread's behavior at runtime, how to do thread migration and finally how to apply DVFS to each CPU core.

5.1 Data Collection for Each Thread

Performance monitors (PMs) [34]–[36] are used to capture the program behavior metrics in the proposed model eq. (10). Intel Core 2 processors have five performance counters per core [40], [43]. Performance monitor one (PM1) and performance monitor two (PM2) are fully programmable. PM1 and PM2 can count 116 and 115 different types of events respectively. The other three counters can each count one fixed type of event (for counter 3: INSTR_RETIRED.ANY, 4: CPU_CLK_UNHALTED.CORE, and 5: CPU_CLK_UNHALTED.REF). In the implementation, performance monitor one (PM1) records the number of memory accesses. Performance monitor two (PM2) records the number of stalled cycles. PM3 and PM4 record the number of instructions executed and the number of unhalted cycles respectively. Parameters in the

TABLE 2
Mapping of f_{target} to a Phase and CPU Frequency for Intel Core 2 Quad 8400

Phase	f_{target} Range	Mapped CPU Frequency
1	(0,2.16] GHz	1.998 GHz
2	(2.16,2.50] GHz	2.333 GHz
3	(2.50,2.66] GHz	2.664 GHz

model are then calculated based on the PM values, e.g., $\mathrm{MAPI} = \mathrm{PM1}/\mathrm{PM3}, CPI_{exe} = (\mathrm{PM4} - \mathrm{PM2})/\mathrm{PM3}$. The PMs start their data collection for each thread after every system call *context_switch(previous thread, next thread)* [42]. The PMs are reset at the next context switch to collect data for a new thread.

5.2 Thread Migration and DVFS Operation

After f_{target} is determined by the model, it will be mapped to the closest frequency that is supported by the CPU. This mapping strategy allows the program to be executed closest to the most energy efficient way under the frequency mismatch condition. Table 2 shows the mapping of a range of f_{target} to frequencies that are supported by Intel Core 2 Quad 8400. We give each CPU core a phase number to represent its operating frequency. The total number of phases for a given CPU is determined by the number of different frequencies supported by that CPU. Table 2 shows that Intel Core 2 Quad 8400 has 3 phases since it supports 3 different frequencies. The phase number is also given to a thread after frequency mapping. For example, if a thread has $f_{target} = 2.4$ GHz, it will be mapped to the closest 2.333 GHz and given a phase number which is 2 in this case.

In our design, we use thread migration to cluster threads with the same phase and put them into the same group. For example, the threads in Table 3 are clustered into three groups based on their phases: {1,2,3,9},{4,5,6,7},{8,10} with the numbers representing thread IDs. These three groups are then put on different CPU cores for execution. By using this method of task grouping, DVFS can be applied to a whole group of threads instead of each single thread. As a result, the unnecessary DVFS operations can be significantly reduced. Thread migration is part of the task grouping mechanism and facilitates the group frequency selection. It is operated every time before the active array and the expired array swaps. The proposed thread migration strategy does not interfere with the original load balancing function. For each migration, the migrator checks the expired array of every CPU core k and considers this group as the *source group*. The expired array of any other CPU core l is considered as the *destination group*.

TABLE 3
Thread Frequency Mapping Example

Thread ID	f_{target}(GHz)	Mapped CPU Frequency (GHz)	Phase No.
1	1.5	1.998	1
2	1.8	1.998	1
3	2.1	1.998	1
4	2.2	2.333	2
5	2.4	2.333	2
6	2.3	2.333	2
7	2.3	2.333	2
8	2.6	2.664	3
9	1.7	1.998	1
10	2.6	2.664	3

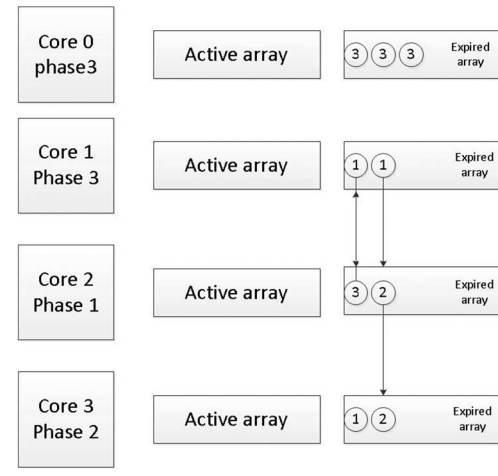

Fig. 3. Thread Migration Policy.

A simple bidirectional thread exchange is made between the source group and the destination group. An example is given in Fig. 3. Assume CPU core 1 is in phase 3 and it contains two threads both in phase 1. Since these threads' phases are different from the CPU core phase, these two threads in phase 1 will be moved to core 2 which is also in phase 1. On the other hand, the thread in phase 3 on core 2 is moved to core 1. For the migration between core 2 and core 3, the idea is the same. Algorithm 2 demonstrates the thread migration policy.

After the group frequency is determined, it is applied to the corresponding CPU core. For Intel processors, the CPU has a *p-state* to represent a frequency and voltage operating state. CPU frequency is adjusted by writing a corresponding value of *p-state* to the $IA32\_PERF\_CTL$ register, which is one of the Model Specific Registers (MSRs) [40], [43]. After writing the *p-state*, the CPU will be unavailable for a short period of time due to voltage transitions. The CPU starts operating on the new frequency when the transition completes.

Algorithm 2. The Thread Migration Strategy

Require: The expired array E_k of each core k

Ensure: Move threads to the group where their phase is the same with the group phase.

1. **for** each possible CPU core k **do**
2. $f_{group}(k) \leftarrow$ the group frequency of core k
3. **for** each other CPU core l **do**
4. $f_{group}(l) \leftarrow$ the group frequency of CPU l
5. $S \leftarrow \{a \,\text{where}\, a \in E_k \,\&\, f(a) = f_{group}(l)\}$
6. $D \leftarrow \{b \,\text{where}\, b \in E_l \,\&\, f(b) = f_{group}(k)\}$
7. **while** $S \neq \emptyset \,\&\, D \neq \emptyset$ **do**
8. $p \leftarrow s \,\text{where}\, s \in S$
9. $q \leftarrow d \,\text{where}\, d \in D$
10. **if** p is not movable **then**
11. $S \leftarrow S - \{p\}$
12. **end if**

```
13.          if q is not movable then
14.                D ← D − {q}
15.          end if
16.          if p and q are movable then
17.                D ← D − {q}
18.                E_k ← E_k + {q}
19.                S ← S − {p}
20.                E_l ← E_l + {p}
21.           end if
22.         end while
23.      end for
24.  end for
```

6 Experiment Result

The proposed scheduler is evaluated on a desktop computer with Intel Core 2 Q8400 processor, FSB 1333 MHz and 4 GB DDR2 memory. Table 4 shows the available frequency and voltage levels for Intel Core 2 Quad Q8400 Processor. Benchmark programs used for evaluation are from SPEC CPU2006 and Phoronix Test Suite. We evaluate the scheduler in the following categories: (i) Performance (SLA requirement). (ii) Energy Consumption. (iii) Energy Delay Product (EDP). (iv) No. of DVFS operations. LMbench [44] is used to measure the main memory latency and the time overhead of the DVFS operation in our system. Measurement results show the main memory latency is 160 cycles and the time overhead for each DVFS operation ranges from 150 to 250 μs. The memory latency overlap α is set to 0.8 in our system configuration. This is because for the CPU we use in our experiment (Intel Core 2 Q8400), the instruction issue rate is 4 with a 128 entry ROB. It takes 32 cycles for the reorder buffer to be full and stall the CPU pipeline. In our system configuration, the memory latency is 160 cycles and when a memory access happens, the CPU can keep executing for 32 cycles before the ROB gets full. Thus $32/160 = 20\%$ of the memory latency is actually hidden by the out of order execution and α is set to $1 - 0.2 = 0.8$.

How to measure the CPU energy consumption (including dynamic and leakage) is a challenge given that the CPU power is always changing due to DVFS control. Our approach is to use a current clamp (Fluke i30 [6], [45]) on the 12 V CPU power supply cables. The output of the current clamp is sampled every 10 ms using an Agilent 34410A digital multimeter [46]. Then we download the data from the multimeter and calculate the total CPU energy consumption. In this experiment, we compare our design with two past works [7], [6], where the allowed performance loss is 10%. Thus we set the *target performance loss* δ to 10%. We assume the system performance is dominated by the CPU without considering the changing latency of I/O devices or network accesses. The goal of this experiment is to demonstrate that our cool scheduler can provide the most energy savings and best system efficiency under a given SLA requirement. Our design is compared with three other configurations: (1) the system always operating at the highest frequency 2.664 GHz. (2) The DVFS policy from Choi [7]. and (3) the DVFS policy from Chen [6]. The DVFS policies from Choi and Chen are two of the most advanced related works. In [7], Choi proposed a fine-grained DVFS policy that classifies workload as either on-chip or off-chip. It uses a regression model to calculate the optimal running frequency for a program. In [6], Chen formulated the DVFS problem into a multiple-choice knapsack problem (MCKP). It also exploits the program's runtime information and periodically solves the MCKP to provide DVFS control.

TABLE 4
Supported Frequency and Voltage for Intel Core 2 Quad 8400 Processor

Level	frequency	voltage
1	2.664 GHz	1.288 V
2	2.333 GHz	1.175 V
3	1.998 GHz	1.080 V

6.1 Evaluation Result with SPEC CPU2006

Fig. 4 demonstrates the experiment result for the four system configurations. *Highest Frequency* represents the system always operating at highest frequency. *Choi* represents the DVFS policy from Choi [7]. *Chen* represents the DVFS policy from Chen [6]. *Cool Scheduler* represents our proposed DVFS policy. The results of performance, energy consumption and EDP from *Choi, Chen and Cool Scheduler* are normalized to the results from the *Highest Frequency*. For the the number of DVFS operations, all the results are normalized to *Choi* since it uses per thread DVFS which invokes DVFS at every context switch.

Fig. 4(a) demonstrates the performance for each system configuration. We assume the 100% performance is achieved when the CPU is operating on the highest available frequency. Thus $Highest\,Frequency$ always has zero performance loss. The performance of other configurations are normalized to $Highest\,Frequency$. $Choi$ shows 12.8% performance loss on average with the highest performance loss 17.0% on $omnetpp$ and the lowest performance loss 7.9% on $hmmer$. $Chen$ shows 9.3% performance loss on average with the highest performance loss 13.2% on $gobmk$ and the lowest performance loss 5.4% on $libquatum$. Notice that there are SLA ($\delta = 10\%$) violations in both $Choi$ and $Chen$'s work. This is because of the prediction errors in their models and they do not have a mechanism to guarantee the SLA requirement. The heavy computation overhead and frequent DVFS operations also degrade the actual performance and compromise their energy saving capabilities. The performance loss for each benchmark program shows great deviation from the SLA requirement (ranging from 5.4% to 17.0%). Our proposed cool scheduler shows 8.7% performance loss on average with the highest performance loss 9.7% on $xalancbmk$ and lowest performance loss 6.1% on $sjeng$. This result proves our scheduler can successfully guarantee the SLA requirement. It also shows the actual performance loss (8.7%) can approach closely to the target performance loss (10%). This allows our proposed scheduler to precisely control the performance loss and maximize energy saving under the given SLA requirement.

Fig. 4(b) demonstrates the normalized energy consumption for each configuration. $Choi$ achieves 19.0% energy saving on average with the most energy saving 23.5% on

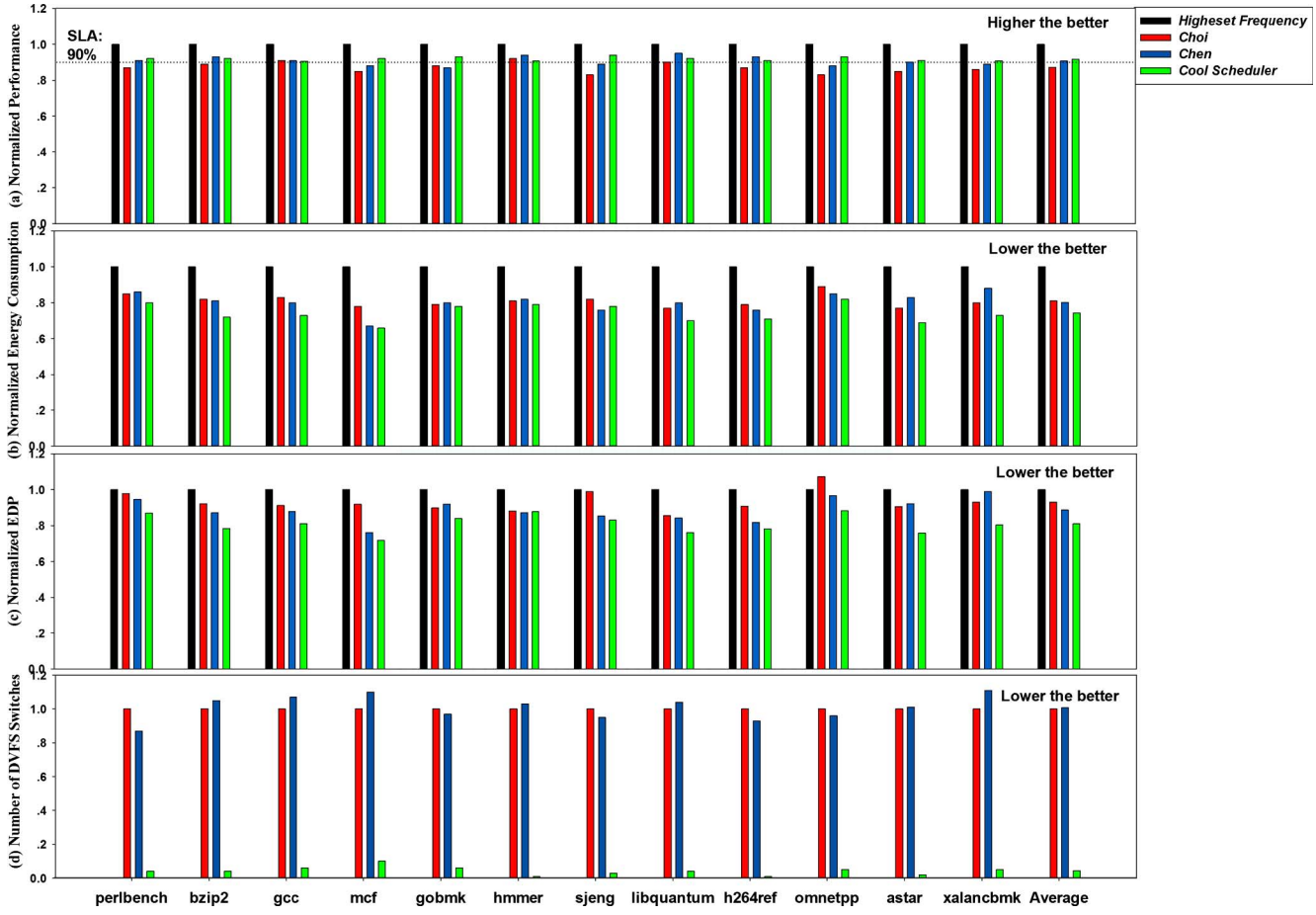

Fig. 4. Results for SPEC CPU2006 benchmarks ($\delta = 10\%$): (a) Performance. (b) Energy consumption. (c) Energy Delay Product (EDP). (d) No. of DVFS switches.

mcf and the least energy saving 11.8% on $omnetpp$. $Chen$ achieves 19.7% energy saving on average with the most energy saving 32.3% on mcf and the least energy saving 12.4% on $xalancbmk$. Our cool scheduler achieves 25.8% energy saving on average with the most energy saving 34.5% on mcf and the least energy saving 12.8% on $xalancbmk$. The result shows our scheduler achieves 35.8% and 31.6% more energy saving compared to $Choi$ and $Chen$ respectively. The heavy computation overhead and frequent DVFS operations significantly degrade the performance and compromise the energy saving capabilities in $Choi$ and $Chen$'s work. This result proves our scheduler can provide the most energy saving under the given SLA requirement compared to two of the most advanced related works. All three configurations achieve highest energy saving on mcf, this is because mcf has the most L2 cache misses thus providing the most energy saving opportunities among all benchmark programs.

Fig. 4(c) demonstrates the normalized energy delay product (EDP) for each configuration. EDP measures the overall performance and system efficiency. The result shows all three configurations ($Choi, Chen, Cool\ Scheduler$) can improve the system efficiency compared to $Highest\ Frequency$. The average normalized EDP for $Choi$ and $Chen$ are 93.1% and 88.7% respectively. Our $Cool\ Scheduler$ has the lowest average normalized EDP 80.9%. This result proves our scheduler has the best system efficiency among all the configurations.

Fig. 4(d) demonstrates the normalized number of DVFS operations invoked in each configuration. This number is zero for $Highest\ Frequency$. $Choi$ calculates the optimal frequency and invokes DVFS at every context switch. $Chen$ solves the MCKP problem every second and invokes DVFS every 30 ms. It shows a slightly increment in the total number of DVFS operations compared to $Choi$. Our proposed scheduler uses task grouping to group the threads in the same phase onto the same CPU core, and then invoke DVFS to this group instead of each single thread. The result shows our scheduler reduce the number of DVFS operations by 24 times compared to $Choi$. This proves our scheduler can significantly reduce the unnecessary DVFS operations in related works.

6.2 Evaluation Result with Phoronix Test Suite

The proposed scheduler is further evaluated with multi-threaded benchmarks provided by Phoronix Test Suite [10]. The benchmarks include: Apache, SQLite, 7-Zip, FFmpeg, Stream, Java 2D and PHP. The results are demonstrated in Fig. 5. We still compare our design with two past works [7], [6]. Fig. 5(a) demonstrates the normalized performance in each configuration. $Choi$ has a performance loss ranging from 6.6% to 19.2% and 12.7% on average. $Chen$ has a performance loss ranging from 7.2% to 14.0% and 10.4% on average. Both $Choi$ and $Chen$ have shown SLA violations. Our proposed scheduler has a performance loss ranging from 7.3% to 9.2% and 8.5% on average. This result further demonstrates our

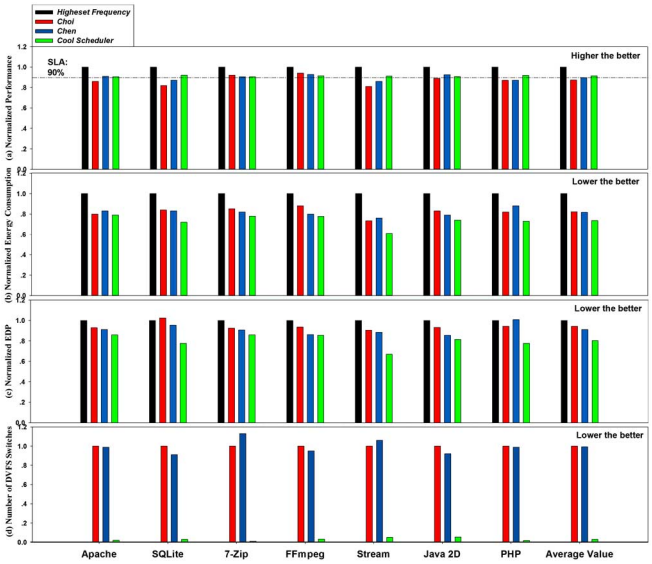

Fig. 5. Results for Phoronix Test Suite ($\delta = 10\%$): (a) Performance. (b) Energy consumption. (c) Energy Delay Product (EDP). (d) No. of DVFS switches.

scheduler can successfully guarantee the SLA requirement and keep the actual performance loss very close to the target performance loss (10%).

Fig. 5(b) demonstrates the normalized energy consumption for each configuration. $Choi$ achieves energy savings ranging from 12.0% to 26.7% and 17.8% on average. $Chen$ achieves energy savings ranging from 12.9% to 24.5% and 18.4% on average. Our scheduler achieves energy savings ranging from 20.4% to 38.8% and 26.5% on average. Results show our scheduler achieves 44.0% and 48.9% more energy savings than $Choi$ and $Chen$ respectively. Fig. 5(c) demonstrates the normalized energy delay product (EDP). Compared to $Highest\,Frequency$, $Choi$ has 94.2% and $Chen$ has 91.1% on average. Our scheduler achieves the lowest value: 80.4% on average. This result further proves our scheduler achieves the best system efficiency among all the configurations. Fig. 5(d) demonstrates our scheduler can significantly reduce the number of unnecessary DVFS switches (by more than 30 times) compared to related works.

6.3 Design Overhead and Improvement Breakdown

A major improvement of our design is the reduction of system overhead: computation complexity and unnecessary DVFS transitions. Past works have been using regression based model or by solving NP-hard multiple choice knapsack problem (MCKP) to determine the optimal operating frequency. Our implementation demonstrates solving the regression model and MCKP problem within the OS could extend the program execution time by 3% to 7%. Past works also chose to apply DVFS to each thread every tens of milliseconds. These DVFS overheads further accounts for 1–3% of the total program execution time. In summary, our implementation demonstrates past works has 6–10% of overall system overhead. This implies that if the actual performance degradation for program execution is 10%, over 60% of the degradation is caused by system overhead which demonstrates significant system inefficiency. In our design, the computation overhead accounts for less than 0.4% of the program execution time. Our design uses thread migration and task grouping to reduce the number of DVFS transitions. Experiment demonstrates the thread migration overhead ranges from tens to hundreds of micro-seconds. This overhead is negligible compared to the number of unnecessary DVFS transitions we have reduced. With the cost of model computation, system feedback and thread migration all taken into consideration, our design only adds a total of 1.3% overhead to the program execution.

Unlike the two related works, our scheduler can always guarantee the SLA requirement because of the continuous system feedback. Our scheduler can also significantly reduce the number of unnecessary DVFS switches because of our thread migration and task grouping strategy. In this experiment, we turn off the system feedback, thread migration and task grouping functions in our design to further evaluate our scheduler against benchmarks from Phoronix Test Suite. DVFS decisions are made for each thread at every context switch. We compare this new scheduler with $Choi$, $Chen$ and our original scheduler. Results demonstrate this new configuration achieves 9.5% performance loss on average. It achieves 23.1% energy savings on average, which is still 29.8% and 25.5% more than $Choi$ and $Chen$ respectively. However, without thread migration and task grouping, this configuration has about the same amount of DVFS switches as $Choi$ and $Chen$. Moreover, it has 14.7% less energy savings compared to our original scheduler. Most importantly, without the continuous feedback, the scheduler is unable to guarantee the SLA requirement and performance loss ranges from 5.8% to 12.6%.

To summarize, the experiment result demonstrates our proposed voltage and frequency scheduler provides the most energy saving and the best system efficiency under the given SLA requirement compared to two of the most advanced related works. The success is due to three categories of improvement: (1) our scheduler greatly improves the computation efficiency (O(1) compared to O(N) in $Choi$ and $Chen$). (2) Task grouping significantly reduces the number of unnecessary DVFS operations. (3) The feedback mechanism allows the actual performance loss to approach closely to the target performance loss, thus maximizing energy saving opportunities under the SLA requirement.

7 Conclusion

This paper presents a voltage and frequency scheduler that can be used in enterprise data centers to provide CPU energy saving under the SLA requirement. The scheduler dynamically adjusts the CPU voltage and frequency level exploiting the run-time program phases. Our design demonstrates significant reduction on the computation overhead and the number of unnecessary DVFS transitions compared to two recently published works. The scheduler is built into the Linux 2.6.22.9 kernel and evaluated against benchmark programs from SPEC CPU2006 and Phoronix Test Suite. Experiment result demonstrates our cool scheduler achieves 25.8% energy saving on average with 8.7% performance loss under the given SLA requirement (10%).

Acknowledgments

The authors would like to thank the reviewers for their valuable comments and feedbacks that have tremendously

helped in improving the quality of this paper. We would also like to thank our colleagues and the ECE department for their valuable support on our research.

References

[1] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2008, pp. 123–134.

[2] Intel Corp., *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, White Paper*. Santa Clara, CA: Intel Corp., Mar. 2004.

[3] C. Isci, A. Buyuktosunoglu, and M. Martonosi, "Long-term workload phases: Duration predictions and applications to DVFS," *IEEE Micro*, vol. 25, no. 5, pp. 39–51, Sept./Oct. 2005.

[4] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proc. 30th Ann. Int. Symp. Comput. Archit.*, Jun. 2003, pp. 336–347.

[5] M.-K. Huang, J.-M. Chang, and W.-M. Chen, "Grouping-based dynamic power management for multi-threaded programs in chip-multiprocessors," in *Proc. Int. Conf. Comput. Sci. Eng.*, Aug. 2009, vol. 2, pp. 56–63.

[6] X. Chen, C. Xu, and R. Dick, "Memory access aware on-line voltage control for performance and energy optimization," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2010, pp. 365–372.

[7] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 24, no. 1, pp. 18–28, Jan. 2005.

[8] J. Kim, S. Yoo, and C.-M. Kyung, "Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 30, no. 1, pp. 110–123, Jan. 2011.

[9] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sep. 2006.

[10] Phoronix Media. (Sept. 2011). *Phoronix Test Suite* [Online]. Available: http://www.phoronix-test-suite.com/

[11] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2006, pp. 347–358.

[12] X. Liu, X. Zhu, S. Singhal, and M. Arlitt, "Adaptive entitlement control of resource containers on shared servers," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manag.*, 2005, pp. 163–176.

[13] P. Padala, K. G. Shin, X. Z. Mustafa, U. Z. Wang, and S. S. Arif, "Adaptive control of virtualized resources in utility computing environments," in *Proc. Eur. Conf. Comput. Syst.*, 2007, pp. 289–302.

[14] T. Horvath, T. Abdelzaher, K. Skadron, S. Member, and X. Liu, "Dynamic voltage scaling in multitier web servers with end-to-end delay control," *IEEE Trans. Comput.*, vol. 56, no. 4, Apr. 2007, pp. 444–458.

[15] Y. Zhu and F. Mueller, "Feedback EDF scheduling exploiting dynamic voltage scaling," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2004, pp. 84–93.

[16] H. Jung and M. Pedram, "Continuous frequency adjustment technique based on dynamic workload prediction," in *Proc. Int. Conf. VLSI Design*, 2008, pp. 249–254.

[17] D. song Zhang, S. yao Jin, T. Wu, and H. wei Li, "Feedback-based energy-aware scheduling algorithm for hard real-time tasks," in *Proc. IEEE Int. Conf. Netw. Archit. Storage Netw. Archit. and Storage*, Jul. 2009, pp. 211–214.

[18] K. He, Y. Chen, and R. Luo, "A system level fine-grained dynamic voltage and frequency scaling for portable embedded systems with multiple frequency adjustable components," in *Proc. IEEE Int. Conf. Portable Inf. Devices*, May 2007, pp. 1–5.

[19] V. Hanumaiah and S. Vrudhula, "Temperature-aware DVFS for hard real-time applications on multi-core processors," *IEEE Trans. Comput.*, vol. 61, no. 10, pp. 1484–1494, 2011.

[20] P. Malani, P. Mukre, Q. Qiu, and Q. Wu, "Adaptive scheduling and voltage scaling for multiprocessor real-time applications with non-deterministic workload," *Design Autom. Test Eur. Conf. Exhib.*, 2008, pp. 652–657.

[21] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 5, pp. 658–671, May 2010.

[22] J. Li and J. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, Feb. 2006, pp. 77–87.

[23] X. Liu, P. Shenoy, and M. Corner, "Chameleon: Application-level power management," *IEEE Trans. Mobile Comput.*, vol. 7, no. 8, pp. 995–1010, Aug. 2008.

[24] K. Banerjee and E. Agu, "Powerspy: Fine-grained software energy profiling for mobile devices," in *Proc. Int. Conf. Wireless Netw. Commun. Mobile Comput.*, Jun. 2005, vol. 2, pp. 1136–1141.

[25] Z. Zong, A. Manzanares, X. Ruan, and X. Qin, "EAD and PEBD: Two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters," *IEEE Trans. Comput.*, vol. 60, no. 3, pp. 360–374, Mar. 2011.

[26] A. Kansal and F. Zhao, "Fine-grained energy profiling for power-aware application design," *SIGMETRICS Perform. Eval. Rev.*, vol. 36, pp. 26–31, Aug. 2008 [Online]. Available: http://doi.acm.org/10.1145/1453175.1453180.

[27] C. Xian, Y.-H. Lu, and Z. Li, "A programming environment with runtime energy characterization for energy-aware applications," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 2007, pp. 141–146.

[28] R. Kotla, S. Ghiasi, T. Keller, and F. Rawson, "Scheduling processor voltage and frequency in server and cluster systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, p. 234b, vol. 12, 2005.

[29] J. Barnett, "Dynamic task-level voltage scheduling optimizations," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 508–520, May 2005.

[30] C. Gniady, A. Butt, Y. C. Hu, and Y.-H. Lu, "Program counter-based prediction techniques for dynamic power management," *IEEE Trans. Comput.*, vol. 55, pp. 641–658, 2006.

[31] G. Contreras and M. Martonosi, "Power prediction for Intel XScale reg; processors using performance monitoring unit events," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 2005, pp. 221–226.

[32] M. Martonosi, S. Malik, and F. Xie, "Efficient behavior-driven runtime dynamic voltage scaling policies," in *Proc. 3rd IEEE/ACM/IFIP Int. Conf. Hardware/Software Codes. Syst. Synthesis*, Sep. 2005, pp. 105–110.

[33] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proc. 35th Int. Symp. Comput. Archit.*, Jun. 2008, pp. 363–374.

[34] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2006, pp. 359–370.

[35] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2003, pp. 93–104.

[36] C. Hu, D. Jimenez, and U. Kremer, "Toward an evaluation infrastructure for power and energy optimizations," in *Proc. 19th IEEE Int. Parallel Distrib. Process. Symp.*, Apr. 2005, pp. 143–151.

[37] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson, "Characterizing the impact of different memory-intensity levels," in *Proc. IEEE 7th Annu. Workshop Workload Char. (WWC-7)*, Oct. 2004, pp. 3–10.

[38] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, Nov./Dec. 2003.

[39] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Proc. 12th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 1/Oct. 2003, pp. 220–231.

[40] Intel Corp., *Intel-64 and IA-32 Architectures Software Developers Manual, vol. 3B: System Programming Guide, Part 2*, Nov. 2008.

[41] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, Jun. 2004, pp. 76–87.

[42] D. P. Bovet and M. Cesat, *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly Media Inc., 2006.

[43] Intel Corp., *Architectures Software Developers Manual, vol. 3A: System Programming Guide, Part 1*. Santa Clara, CA: Intel Corp., Nov. 2008.

[44] J. Ruggiero, *Measuring Cache and Memory Latency and CPU to Memory Bandwidth System*. Santa Clara, CA: Intel Corp., Nov. 2008.

[45] Fluke Corp. (Mar. 2006). *Fluke i30 AC/DC Current Clamp Technical Data* [Online]. Available: http://support.fluke.com/find-sales/Download/Asset/2747126_6001_ENG_B_W.PDF

[46] Agilent Technologies, Inc. (Dec. 2011). *Agilent 34410A/11A 6 1/2 Digit Multimeter User's Guide* [Online]. Available: http://cp.literature.agilent.com/litweb/pdf/34410-90001.pdf

Zhiming Zhang received the BS degree in computer engineering from University of Electronic Science and Technology, Sichuan, China, in 2009 He is currently working toward the PhD degree in computer engineering with the Department of Electrical and Computer Engineering, Iowa State University, Ames. His research interests include computer architecture and energy aware computing.

J. Morris Chang received the PhD degree in computer engineering from North Carolina State University, Raleigh. He is an associate professor with Iowa State University, Ames. His industry experience includes positions at Texas Instruments, Microelectronic Center of North Carolina and AT&T Bell Laboratories. His research interests include computer architecture, performance study of java virtual machines (JVM), and wireless networks. Currently, he is a handling editor of *Journal of Microprocessors and Microsystems* and the Middleware & Wireless Networks subject area editor of *IEEE IT Professional*.