E-MOS: Efficient Energy Management Policies in Operating Systems

Vinay Gangadhar
vinay@cs.wisc.edu

Clint Lestourgeon
clint@cs.wisc.edu

Jay Yang
jkyang2@wisc.edu

Varun Dattatreya
channagirida@wisc.edu

Department of Computer Sciences
University of Wisconsin - Madison

Abstract

Current Linux Operating Systems' power governors do not provide a fine-grained control and management over the energy utilized by the applications running on a computing system. Linux power governors by default use ondemand CPU frequency governor or Intel's P-state driver for power management by considering the entire system load rather than individual application requirements. This system-wide and default kernel policy could have undesirable effects on OS power management and could lead to poor battery life and performance for modern CPUs. In this project, we first provide an overview of the difficulties in existing Linux Operating System power management and we explore the opportunities that application characterization presents for more efficient energy management solutions.

We present E-MOS (Efficient Energy Management Policies in Operating Systems), an energy management model which uses applications' characteristics to make frequency scaling policy decisions in order to achieve better energy efficiency while balancing the performance of the system. The decisions of this application aware policy can be applied through the ACPI (Advanced Configuration and Power Interface) user-space power governor. E-MOS is evaluated on variety of benchmarks and we see energy savings upto 2x for a 13% performance loss.

1. Introduction

The last decade of computing has seen mobile phones, wearables and other battery powered systems becoming more ubiquitous. This shift towards mobile computing has lead to system designers being faced with the challenge of prolonging battery life by limiting the energy consumed by system resources (CPU, Memory, Network) and improving system energy efficiency [13, 16] while running a multitude of user applications. In addition to energy consumption, managing system power dissipation is also very important in avoiding heat related problems that are prevalent in computing systems [9].

Most of the existing work on system energy conservation rely on techniques that dynamically characterize power consumption and the Quality of Service (QoS) requirements of applications. Based on these parameters, systems implement power management policies by using DVFS (Dynamic voltage frequency scaling) [6] to reduce the CPU operating frequency or switching to lower-CPU power states [11, 20] during periods of low activity. Reflecting this, the current power management governors in Linux primarily use one source of information while regulating power, CPU usage and system performance requirements. Based on the performance needs of a system, these governors vary the CPU frequency in an effort to manage energy consumption while maintaining QoS. There exists a wide variety of other sources of information, through platform independent interfaces like ACPI [1, 2], platform dependent interfaces like RAPL(Running Average Power Limit) and of-course the user as well. The rigid nature of current power management policies could harm the energy efficiency of a system, since each application could differ in its needs, and thus require different strategies to achieve efficient energy utilization. Some applications (such as those that interact with the user) run for shorter periods of time and require an immediate response. Other, more background tasks (like a disk scan) run for longer durations but

might not be latency bound. While the former class of applications could benefit from short periods of high performance, the latter could be executed more efficiently by operating at a lower performance level. Furthermore, similar issues occur with other system resources and the efficient management of these resources also affect the optimal energy management strategy. These issues could affect the systems that manage power only through CPU frequency scaling.

Given the issues identified, we aim to explore cases where the current power governors on Linux fail to yield a reasonable energy management solution. Furthermore, similar to the work of Liang et. al [10], we aim to show that by providing information that characterizes the needs of a particular application, better energy efficiency is achievable. We first profile applications and characterize them into buckets based on how compute intensive, cache sensitive and memory intensive they are. Based on this information, an analytical model called E-MOS is used to reason about the frequency scaling of cores and achieve better energy efficiency. The analytical model uses a decision table which uses the parameters of an application and power management objectives to be achieved as inputs. It then computes possible actions that need to be taken to optimize energy efficiency by providing different frequency scaling settings for the given system. The model achieves a trade off between energy and performance by reducing or increasing and/or maintaining the CPU frequency. We evaluate the model's results by implementing its suggestions and testing with the user-space power governor for a variety of applications. Our experiments scale the CPU frequency and we observe energy savings of up to 2x with 13% performance loss compared to the default Linux power governors.

This paper makes the following contributions:

- An analysis and case study of existing Linux power governors for different applications that can be categorized as compute intensive, cache sensitive and memory bound.
- An application aware analytical model called E-MOS which analyzes the trade off between energy efficiency and performance with application information.
- An improved energy-management policy, which tunes the CPU frequency to meet application demand while achieving better energy efficiency.
- A comparison of E-MOS with Linux's default power governors for a number of benchmarks.

Paper Organization We first discuss related work in the power management field in Section (§2). We present a brief overview of the existing Linux power management solutions in Section (§3). Section (§4) discusses a case-study of non-linear scaling with existing power governors and motivates our work. Section (§5) explains our application categorization and with a detailed analysis, reasons about the problems with existing Linux governors. Section (§6) explains our E-MOS analytical model and the decision table it uses to make application-aware energy management policy decisions. Section (§7) details our experimental methodology while Section (§8) gives a detailed evaluation and results of our model. We discuss the lessons we learned in Section (§9) and finally conclude in Section (§10).

2. Related Work

The Energy Management (EM) area in operating systems constitutes a diverse set of solutions to the problem of ensuring performance while decreasing power usage. The Linux kernel has some predefined governors for managing power but many papers have recognized that these general purpose solutions are certainly not optimal, and perhaps in some case not even adequate. There have been a number of attempts to remedy this and we enlist some of these solutions in this section. Dynamic Voltage Frequency Scaling [6] has been a popular hardware mechanism to save power and boost performance. Zeng et. al [20] focuses on directly managing power usage as one might manage other scarce resources. Choi et. al [7] dynamically profile applications to determine a good energy management policy at hardware level.

Non-Optimal General Purpose EM Policies It has been recognized that the generic policies for energy management implemented in Linux by using the ACPI interface are not always the best performing, or most efficient. For example in the context of mobile

devices [10], the on-demand power governor used by the Android operating system does not guarantee low power consumption for all workloads. Recently (as of Linux 3.9), Intel introduced a P-state driver that is more platform specific to overcome some of the issues of the ACPI governors. On the opposite end of the spectrum, we see work done by Yanpei. et. al [11] that attempts to tackle this issue for data center workloads and emphasizes QoS guarantees rather than energy efficiency. Our work focuses on providing more dynamic energy management decisions based on the application information. Work on using user-space governor to drive the power policy and compare the implementation with the other Linux governors has been done in [10].

Existing Energy Efficiency Policies In the case of [20], energy is treated as another scarce resource and a currency based algorithm is used. Processes in this system are allocated a specific amount of power usage, and are given access to various power consuming operations in relation to the amount of power they are allocated. However, this is focused largely on the question of optimization of allocation, and not on the question of how to ensure that the energy is productively used. Other implementations [10, 7] attempt to form a correlation between the number of memory accesses and ideal operating frequency. They base this argument on the fact that some workloads see lower power when executed at higher frequencies. Their experiments show that this is a consequence of the number of memory accesses, whose frequency of operation remains constant. By getting a measure of the memory accesses in a workload, they use the memory access - CPU frequency correlation to set the operating frequency. Our approach is similar to the above work where we classify workloads depending on how CPU/memory bound and cache sensitive.

Application Profiling and decomposition Dynamic voltage and frequency scaling based on application profiling and decomposition has been done in Choi. et. al [7]. This idea on profiling is similar to our project's initial profiling step, but the workload is decomposed into two-parts here: On-chip and Off-chip, effectively indicating CPU sensitive and memory sensitive applications respectively. It exploits the idea that different workloads have different power management needs and these statistics could be exploited at run-time. They do not implement this in operating system and the energy management policy may be overridden by the kernel's own policies. Similarly, Choi et. al [8] work does DVFS for energy conservation classifying all the applications based on On-chip and Off-Chip computation ratio. Again this would result in a generic decision, based on ratios calculated offline. Power conscious fixed scheduling policy of applications with DVFS based on profiling information has been implemented in Shin et. al [17], which again does not integrate the decision making capability to operating systems. Snowdon et. al [18] also characterize applications but use this information to estimate an application's power consumption. They use this information to present the OS with a flexible power management policy. Their Koala policy needs a special mention as it is very closely related to our current work which aims at embedding application information into the energy management policy of the operating system, based on an application's needs.

3. Linux Power Governors

On today's Linux systems, depending on hardware availability and the version of the Linux kernel used in a distribution, power management is either handled by the ACPI governors or Intel's P-state drivers. The ACPI governors are platform independent solutions that base their decisions on system load and ACPI events and request CPU frequencies. On the other hand, Intel's P-state drivers were introduced in Linux 3.9 and support processors from Sandybridge onwards. The P-state drivers are more aware of the hardware capabilities of Intel's processors and request Performance states (P-states) rather than fixed frequencies.

ACPI governors These governors attempt to scale CPU frequency in order to save power. CPU frequencies can be scaled automatically depending on the system load, in response to ACPI events, or manually by user space programs. The infrastructure available in the Linux kernel to perform frequency scaling is called `CPUPOWER`. There are a number of ACPI gov-

ernors available in Linux but the ondemand governor is the most widely used. This governor sets the CPU frequency depending on the system load. In general, this governor tries to run the CPU load at high frequency. If the CPU load placed by the user abates, the ondemand governor will step back down through the kernel's frequency steppings until it settles at the lowest possible frequency, or the user executes another task to demand a ramp. Ondemand scales its frequency in a work queue context. In other words, once the task that triggered the frequency ramp is finished, ondemand will attempt to move the frequency back to minimum. If the user executes another task that triggers ondemand's ramp, the frequency will bounce from minimum to maximum.

P-state drivers Intel's P-state drivers work on the race-to-idle concept. Since most modern CPUs consume very little power when idle, these drivers attempt to execute workloads as quickly as possible and return the CPU to an idle state. Race-to-idle policies benefit from the power savings of having the CPU in an idle state for most of the time. Apart from exploiting the idle CPU power savings, the P-state drivers are also aware of the available Performance states (or P-states), which represent a voltage-frequency operating point for the CPU. Currently, two P-state driver algorithms are selectable by the user: powersave and performance. As their names imply, the powersave driver emphasizes power savings at the cost of performance while the Performance driver works in a manner that is similar to the ACPI ondemand governor.

4. Imperfect Scaling Study

This section focuses on a case study, which explores the frequency scaling capability of the system with two types of access patterns in applications. An assumption often made, with respect to the power utilization is that on a given CPU, changing the frequency will change the CPU performance by the corresponding amount (generally known as scaling). However, this is only true when considering the CPU execution. But, there are other architectural parameters whose performance do not scale along with CPU frequency. One particular example would be

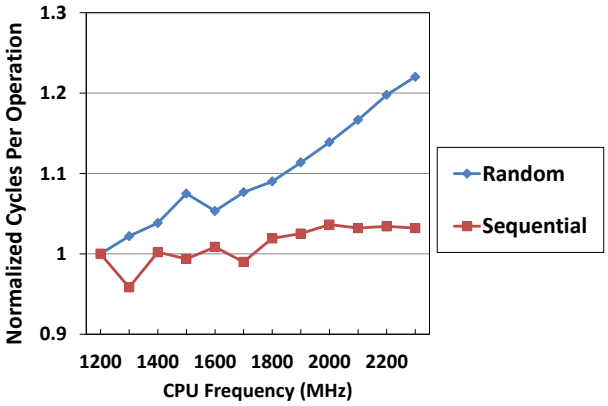

Figure 1: Frequency scaling effect for cache misses

memory. CPU caches hide much of this latency, but cache misses can be particularly damaging for the system energy efficiency, as the time frame is too short for the operating system to handle the cache misses intelligently, or even be aware of them. Yet, operating systems' take long enough time to make a decision that they measurably decrease the performance and/or efficiency.

To demonstrate this effect, we wrote a simple micro-benchmark, where a process takes large array (32MB), copies a random entry to another random entry in memory. We then operated the CPU core at variety of frequencies using `CPUPOWER` [3] utility and measured the execution time. This workload was then compared to another process that did the exact work of copying data from one point to another, but did so sequentially, avoiding most of the cache misses. As expected for both processes the time to execute decreases as CPU frequency increases. We can then plot the *efficiency*, computed here as the time to execute the process represented as the number of clock cycles per operation. For ease of comparison these were then normalized. Figure 1 shows the comparison of the random access workload with the sequential workload with increasing CPU frequency, normalized to the base 1200Mhz frequency.

Since both processes do the same amount of computation, the only factor that can affect the performance and efficiency is the memory access pattern. If the performance scaled perfectly with CPU frequency, we would expect a nearly flat line, with all cache hits as we see in the case of sequential access

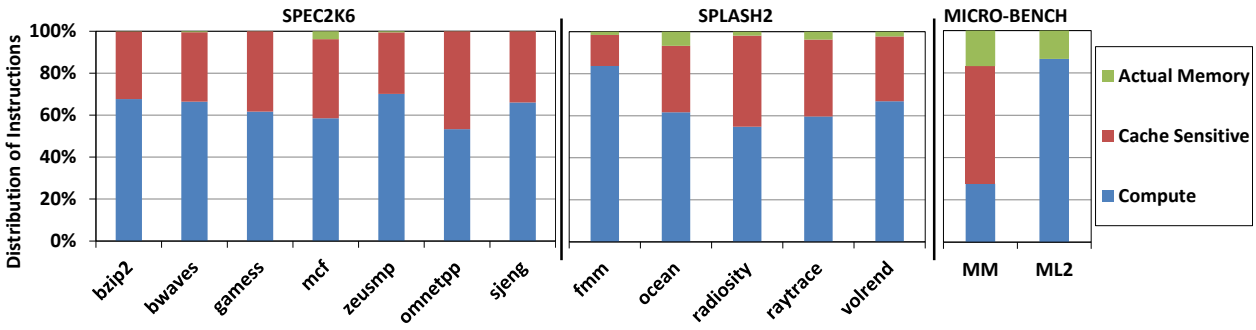

Figure 2: Distribution of Instructions in Applications

workload in the figure. But with random accesses, we see instead that the cycles per operation increase as the frequency increases, clearly demonstrating a decrease in the efficiency. This is because of time spent in handling cache misses, and CPU is not doing any useful work to hide this latency. It can be seen that at 2300Mhz, we are 20% less efficient than at 1200MHz. There are two mitigating factors with respect to this issue. First is programming practices – since cache misses are a well known performance issue, many applications are already written to avoid them as much as reasonable. Second is the hardware – in particular modern processors have both hyperthreading and out of order execution, both of these features allowing the processor to potentially work on other instructions while the cache misses are being served.

Operating system however, in this scenario does not take efficient decisions, to actually save energy by overriding the default policies. Instead, it still relies on its power governors to run at higher CPU frequency and thus waste energy. This could be a simple example, but nonetheless, as we will demonstrate in further sections with some real workloads, the inevitable constraint is memory bandwidth and/or latency, and in these cases a decrease in CPU frequency may very well lead to efficiency gains.

5. Power Governors Analysis

The case study in the previous section showed a contrived example and how cache misses are effected due to the imperfect frequency scaling in Linux power governors. In this section, we consider real applications and try to analyze how existing Linux power governors perform for different class of applications.

5.1. Application categorization

We chose applications from SPEC2006 [19] and SPLASH2 [5] to account for both single-threaded and multi-threaded applications' analysis. We also have written two micro-benchmarks which we call MICRO-BENCH suite from now, to exercise memory behavior. ML2 is a linked-list traversal workload and MM is a workload which has byte accesses larger than cache line size. To first analyze the distribution of instructions in these applications, we used the PIN [12] binary instrumentation tool for profiling. The PIN tool was modeled to account for total instructions in the application, instructions which get hit in the cache and instructions which miss in the cache and actually access the memory. The application profiling was mainly done to figure out which frequency must be scaled based on the time spent by these applications in CPU or DRAM or Caches. We could not get Disk (I/O) related instructions and I/O behavior of applications, as PIN cannot model file system accesses, and we have no control over the speed at which Disk accesses happen. So, our focus is mainly on the following three categories:

- *Compute Intensive*: The applications which spend most of their execution time in CPU core and have more computation instructions between load/stores to the memory. For these applications, CPU frequency is the important parameter.
- *Cache Sensitive*: The applications which have more memory access instructions, but due to the locality of the data accesses, most of them get the data in CPU caches. For these applications, again CPU frequency is important. For our analysis, we

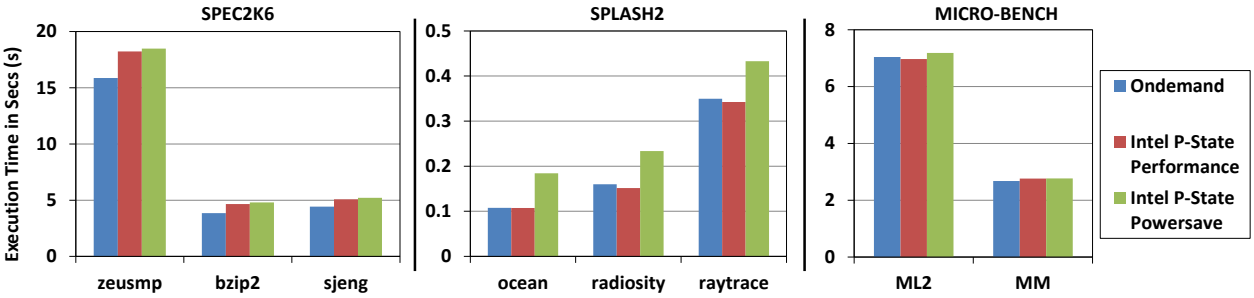

Figure 3: Execution time for Linux ondemand and Intel P-state power governors

assume that, CPU cycles will be wasted in getting the data from the caches, whenever it encounters load or store instructions. It is possible that just by profiling the memory related instructions and not analyzing the cache accesses, one could misinterpret the application as memory intensive and always scale the DRAM frequency leading to energy wastage.

- *Memory Bound* These are the applications, which have lot of last level cache misses and the accesses reach memory (DRAM). Increasing the CPU frequency for these applications could lead to wastage of energy and for better performance DRAM frequency might have to be scaled rather than CPU frequency.

Figure 2 shows the application categorization and distribution of instructions across seven SPEC2006 benchmarks, five SPLASH2 benchmarks and two micro-benchmarks. We see that SPEC2006 has more compute intensive instructions with some of them having good cache locality. SPLASH2 has more cache sensitive instructions in average with some having good percentage of compute as well as memory related instructions. The micro-benchmark MM has lot of cache related instructions and many accessing memory. ML2 has zero cache access and all of the load/store instructions access the memory.

5.2. Power governors performance

We executed these applications on a Linux kernel with Ondemand [14] power governor as the default and also ran the applications on two power governors (performance and powersave) of Intel's recent P-state [4, 15] driver [1]. We considered Intel's P-state driver, as it has access to turbo boost frequency through direct driver interface. Figure 3 shows the execution time of selected benchmarks across all three applications suites (SPEC2K6, SPLASH2 and MICRO-BENCH) for three discussed power governors. It is seen that, all the three power governors have similar performance even though each has a distinct objective. Intel P-state performance governor is supposed to boost the performance with access to turbo boost frequency. Even though, P-state powersave governor has slightly worse performance compared to other two, as it tries to save power by reducing the frequency, the difference is not significant. The next subsection details on energy efficiency of each of these governors.

[1] Section 7 explains the methodology used for performance and energy estimation

5.3. Power governors energy efficiency

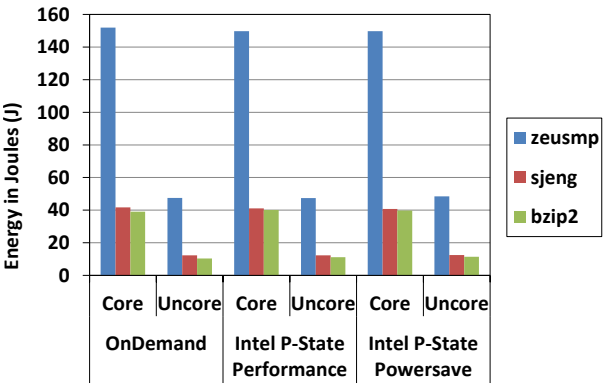

Figure 4: Energy Consumption for SPEC2006 workloads with ondemand and p-state governors

As seen with the execution time, we also wanted to analyze if the power governors perform similar with respect to energy efficiency. Figure 4 shows the energy consumption in Joules for three of the SPEC 2006 benchmarks. We see that the core and uncore

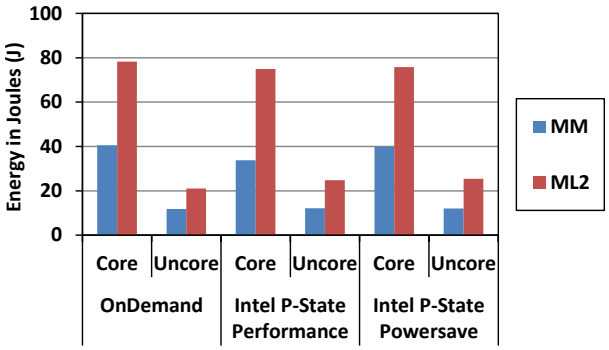

Figure 5: Energy Consumption for MICRO-BENCH workloads with ondemand and p-state governors

energy for all the three benchmarks across all three governors are similar. Figure 5 shows similar trend for memory intensive micro-benchmarks.

Figure 6 shows energy consumption for SPLASH2 benchmarks. In case of SPLASH2 benchmarks, it can be seen that for all three workloads, P-state powersave saves more energy around 2-3 joules compared to other two governors. This complements why powersave governor had worse execution time than other two. However, still the powersave governor does not save significant energy savings even though it reduces the CPU frequency to save lot of power.

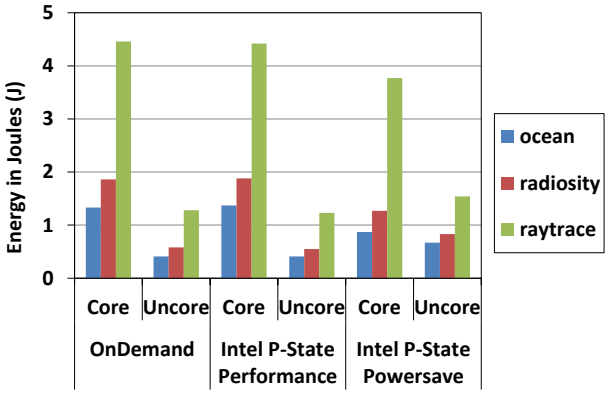

Figure 6: Energy Consumption for SPLASH2 workloads with ondemand and p-state governors

The takeaway from this analysis is that: 1) existing Linux power governors are optimized only for compute intensive benchmarks; 2) All the three power governors mainly use "race to halt" approach to execute workloads faster (scale CPU frequency) and thus try to save energy; 3) The existing power governors are not application-aware and do not rely on application characteristics to scale CPU or DRAM frequency; 4) ondemand governor scales frequency on overall system load rather than individual application requirements. Based on these takeaways, we believe that operating systems should give more freedom to user-space for energy management and making better policy decisions. User-space has more information about applications it is running than the underlying drivers or hardware. We now present our model E-MOS, which is an application-aware energy management model and discuss its implications on OS energy management policies.

6. E-MOS Analytical Model

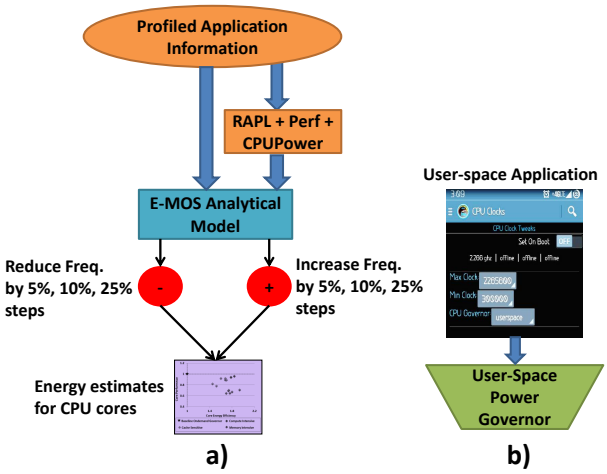

Figure 7: a) E-MOS model; b) Frequency setting in User-space governor

With the detailed analysis and reasoning out the anomalies of existing power governors in the previous section, we now present our E-MOS (Efficient Energy Management Policies in OS) analytical model which mainly relies on the principle of application-aware energy management. We believe that OS providing more freedom to user-space for energy management is beneficial, since user-space has more relevant information about applications and their behavior. Of-course, dynamic power management by changing the CPU frequency and DRAM frequency simultaneously based on application phases would be an ideal power governor behavior. But, our model as of now only statically decides

a frequency setting based on overall application characteristic. Although, E-MOS is an analytical model, one could modify the existing OS power governor to implement E-MOS. Our project aims to evaluate E-MOS model with variety of applications, and based on the results and analysis we aim to use the user-space governor to feed in the suggested frequency setting and get the energy benefits.

Figure 7 (a) depicts the overview of our E-MOS model. It takes the profiled application information – number of compute, cache sensitive, memory intensive instructions (Profiling output from PIN model) as input and uses Linux utilities to get the execution time (`perf`), energy (`perf` and `RAPL`) and frequency (`CPUPOWER`) estimates with the default power governors. These application level measurements are then used by a python framework model and it gets the new energy estimates by increasing or reducing the frequency by 10%, 15% and 25% scaling steps. The model, estimates the energy based on the runtime statistics collected for the application and equation below:

$$Energy(E) = Power(P) * ExecutionTime(T)$$

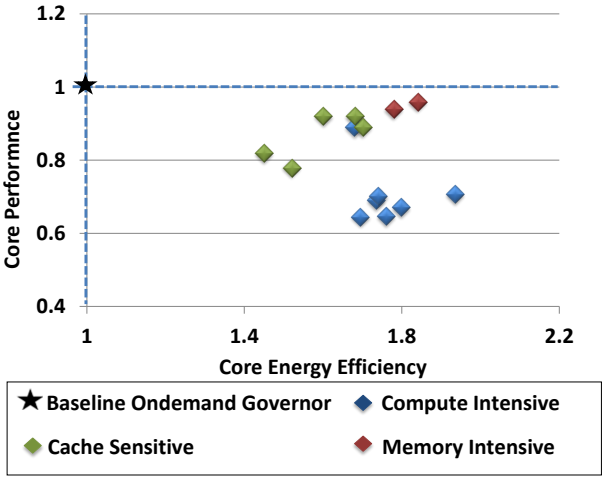

Figure 8: Energy estimates from E-MOS when frequency reduced by 30% (600Mhz)

Power is measured by `RAPL` utility again with the varied frequency and execution time is measured using `perf`. These new energy estimates are then plotted with ondemand power governor as baseline. Figure 8 and Figure 9 shows one such example of E-MOS reducing and increasing the CPU core frequency by 25% (600 Mhz). Ideally, you want to vary both CPU core and DRAM frequency for different application types and then evaluate over baseline model. But, current Linux APIs are not sophisticated enough to measure the runtime frequency and energy estimates for DRAM. Also, there is very limited interface to modify the runnable frequency of DRAM.

6.1. Core frequency reduction

Figure 8, plots all the applications with their performance on y-axis and core energy efficiency on x-axis normalized to ondemand governor. We can see that the energy efficiency of all the applications increases significantly upto 1.9x with the core frequency reduction by 600Mhz. But, the performance of compute intensive applications takes a hit, as they mainly depend on core frequency. Interestingly, memory intensive applications' performance is very close to ondemand governor as they do not much depend on core frequency. Even, cache sensitive applications lose only around 15% performance with such a large core frequency reduction. This, indicates that there is potential in energy efficiency gain while executing cache sensitive and memory intensive applications by reducing the core frequency.

6.2. Core frequency increase

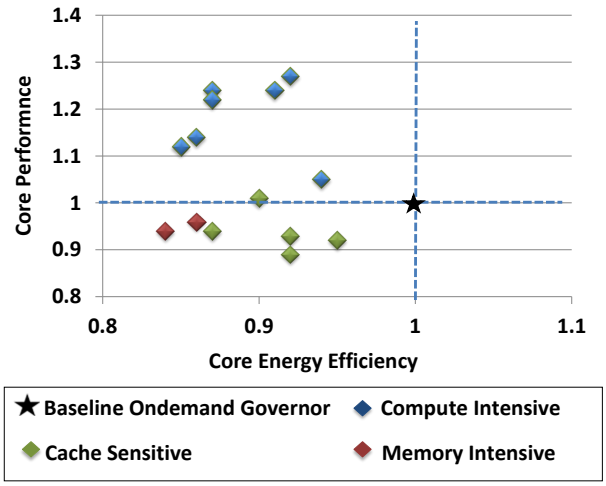

Figure 9: Energy estimates from E-MOS when frequency increased by 30% (600Mhz)

Figure 9, shows a scenario where the core frequency is increased by 600Mhz. Here, compute intensive applications have speedup upto 1.25x which is expected. But, there is not much speedup improvement for cache and memory applications with the increase in core frequency. This, indicates that, increasing core frequency has not much effect on performance of memory and cache applications and you might not want to decrease core frequency when executing memory and cache applications to gain better energy efficiency.

6.3. E-MOS energy management policies

Now, based on the E-MOS model energy results for various applications, and different frequency settings, we formulated a policy decision table to be followed by the power governors based on the application type and the objective. These policies are formulated based on the energy estimates from the E-MOS model. Table 1 shows the policy decisions to be taken for various applications based on the energy or performance objective. Even though, E-MOS model does not control or evaluate DRAM frequency, based on the energy estimated for two scenarios mentioned above, we formulated the policies for DRAM frequency setting too. All our further evaluations and results only use core frequency setting policies and all the evaluation is optimized for energy as the first class constraint. Figure 7 (b) shows how based on the suggested frequency setting of E-MOS model, user-application can scale the CPU core frequency setting using user-space power governor. With user-space power governor one can manually set the core frequency and pin the application to run with that. Thus, E-MOS model along with the user-space power governor can act as an application-aware energy management tool, for better energy efficiency.

Application	Objective	CPU Freq.	DRAM Freq.
Compute	Energy	Maintain\Increase	Decrease
Compute	Performance	Increase	Maintain
Cache	Energy	Decrease	Decrease
Cache	Performance	Maintain	Increase
Memory	Energy	Decrease	Maintain\Increase
Memory	Performance	Maintain	Increase

Table 1: E-MOS Policies Decision Table

6.4. Limitations

We believe our first order E-MOS model, is preliminary model and lot of improvements are needed for dynamic power management. Once such scenario is where a single application has multiple execution phases with compute and memory intensive and you need support to dynamically switch to the corresponding phase by increasing/decreasing the core and DRAM frequency. Also, since this is an analytical model, more realistic estimates can be obtained by integrating this model inside an actual kernel power governor. We aim to implement this as future work.

7. Methodology and Evaluation

All the experiments are executed on an Intel core i7 3630, running 32-bit Ubuntu 15.04 with Linux kernel 3.19. The available scalable CPU frequencies are: 0.8Ghz to 3.4Ghz at steps of 300Mhz. With turbo, a single core can execute with a frequency of 3.4GHz. All power/energy measurements were made in terminal mode to reduce the added noise of the desktop environment. A number of system power measurement methods were examined while working on this project, including: a wattsup meter, `PowerTOP`, `PowerStat` and `perf`. After testing and comparing each of these methods, we decided to use `perf`, integrated with the `RAPL` driver, since it provided us with the best (and most repeatable) estimate of the energy consumed by an application.

`perf` uses Intel's RAPL (Running Average Power Limit) to estimate the energy consumed by an application. RAPL was designed by Intel to manage the thermals of a processor, i.e. ensure that their chips run within the thermal limits. It does so by reading a few Machine Specific Registers (MSRs) and using a software power model to estimate power and energy. The RAPL interface was released with Linux 3.13. Using RAPL, estimates of energy consumed by the cores as well as the package energy (package energy includes the core energy consumption as well as the cache numbers and uncore numbers) can be obtained. Intel have conducted tests [15] that validate the accuracy of the RAPL estimates. There is an inherent issue with RAPL since it uses 32-bit counters that do

not generate an interrupt on overflow. `Perf` handles this by gathering multiple periodic samples and using 64-bit counters that are updated in sync with the RAPL counters. While `perf` handles the counter overflow issues, its counters are incremented in steps of 0.23 nJ and this requires the final values to be scaled. The `perf stat` tool handles this scaling at the cost of a few extra clock cycles due to the additional floating point arithmetic. `perf` allowed us to measure the energy consumed by the cores and the cache. We measured the energy consumed by a benchmark by using `perf stat` and the baseline energy consumption was measured by calling the RAPL interface and executing the Linux `usleep` function for durations equal to the execution time of the benchmark. All our energy numbers were obtained by executing a benchmark and measuring the corresponding system energy for 10 iterations. The lowest measured values were then used in our analysis.

Limitations The `perf` energy consumption numbers that we collected for the cores are measured for the entire power plane. Therefore, even though benchmarks could run on a single core, our numbers are for all cores. We have attempted to reduce the impact of this by measuring the baseline energy consumption.

8. Results

In this section, we present our E-MOS model evaluation and the results of the model i.e chosen frequency settings, for all the initial applications we profiled. One important thing to remember is, we have optimized the model to work on energy as the first class objective and so the frequency setting chosen by E-MOS is more aligned towards energy efficiency with reasonable performance loss. We present our results based on each application type and the chosen frequency setting. We try to reason out why E-MOS would have chosen that setting and show the energy efficiency gains for each category. For all the graphs, both the energy efficiency and performance are normalized and are relative to the ondemand governor findings.

8.1. Compute intensive workloads

Figure 10 shows the relative core energy efficiency and speedup for SPEC2006 benchmarks which are mainly compute intensive workloads. E-MOS chose a frequency setting of 1.8Ghz to 2.4Ghz for the best energy efficiency among the available scalable frequency set. The geometric mean for 2.4Ghz indicates upto 1.4x of energy efficiency with just 3% performance loss. Provided E-MOS was optimized for energy savings, 3% performance loss with such significant energy gains is a good indication. You can still boost the application until turbo boost capability but you will lose lot of energy gains. So did E-MOS not choose setting above 2.4Ghz as the power consumption factor affects more to energy utilization than the lower execution time. With lower frequency of 1.8Ghz, you gain upto 1.6x energy efficiency with performance loss of 15%. You might want to go to lower frequency settings when energy is utmost important with situations like low battery in mobile phones.

8.2. Cache sensitive workloads

Figure 11 shows the relative core energy efficiency and speedup for SPLASH2 benchmarks which have both compute and cache sensitive workloads. E-MOS chose a frequency setting of 1.8Ghz to 2.1Ghz for best energy efficiency among the available scalable frequency set. This rationale makes sense as you want to slightly reduce the core frequency when most of the instructions are accessing cache and especially with the random workload we saw in our case study, you don't want to spend too many CPU cycles at higher frequency wasting energy. The geometric mean for 2.1Ghz indicates upto 1.5x of energy efficiency with just 4% performance loss. With one setting lower, for 1.8Ghz you can gain energy efficiency upto 1.62x and a 9% performance loss. We believe for cache workloads, 2.1Ghz is good, as modern processors are equipped with prefetching and out of order execution mechanisms which hide the cache latency effectively and thus want to be executing at slightly higher frequency than 1.8Ghz. This result was interesting as we expected E-MOS to still choose a lower setting, but we got around 1.5x

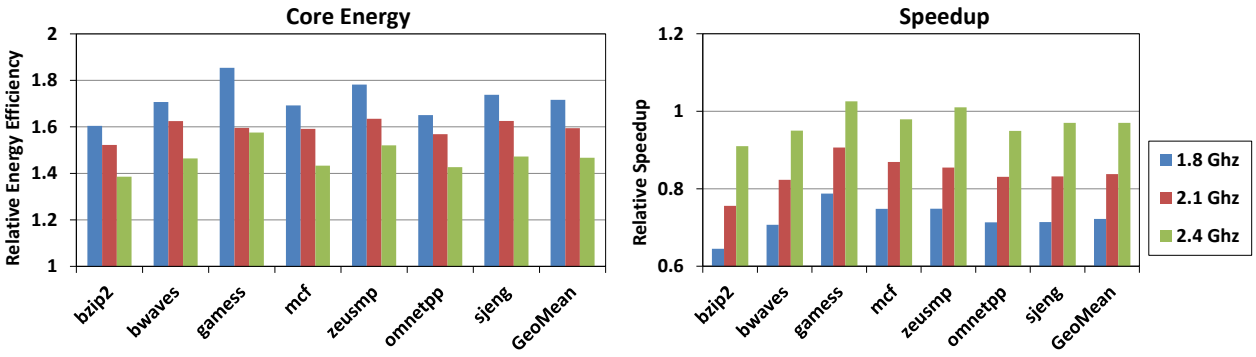

Figure 10: SPEC2006 run with user-space governor with frequency setting 1.8Ghz to 2.4Ghz

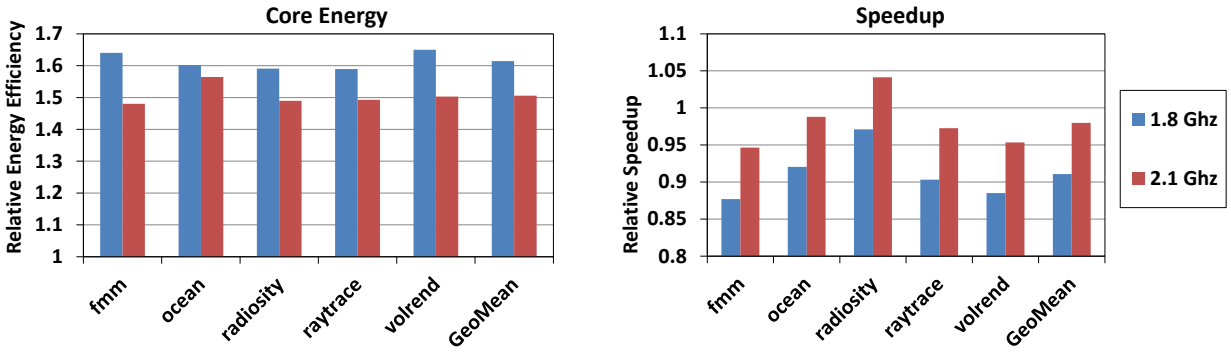

Figure 11: SPLASH2 run with user-space governor with frequency setting 1.8Ghz to 2.1Ghz

energy efficiency with a higher setting.

8.3. Memory intensive workloads

Figure 12 shows the relative energy efficiency and speedup for micro-benchmark suite MICRO-BENCH which are memory intensive workloads. E-MOS chose a frequency setting of 1.2Ghz to 1.8Ghz for best energy efficiency among the available scalable frequency set. The geometric mean for 1.2Ghz indicates upto 2x of energy efficiency with 13% performance loss. it is interesting to see that even with the frequency increasing to 1.8Ghz, there is no significant increase in performance for these workloads. So, it is better to run these applications at lower frequency with half the energy (2x energy efficient) than what you need for ondemand power governor. You would see performance increasing if you have capability to increase the DRAM frequency. However, in that case, you have to get the energy estimates of the entire system, which includes core, uncore and DRAM.

Overall, E-MOS analytical model with user-space power governor provided a good scalable frequency setting for all the three type of applications with better energy efficiency compared to ondemand power governor. The results backed our intuition that application-aware energy management is better and it makes sense for OS to give onus to user-space to make more better policy decisions. However, with more community support and integrating E-MOS type application-aware models into existing power governors could lead to better energy management solutions in OS. It is an interesting analysis from this project towards achieving the goal of energy efficient computing and with tools getting better we can expect to see more energy efficient solutions in coming days.

9. Lessons learned

Power management is still a challenge but, based on the evolution of tools and interfaces that we have explored over the course of working on this project, the tools needed to create a more efficient power management system are improving all the time. With RAPL (on Intel's processors at least) and libraries like PAPI, accurate power estimation is now much

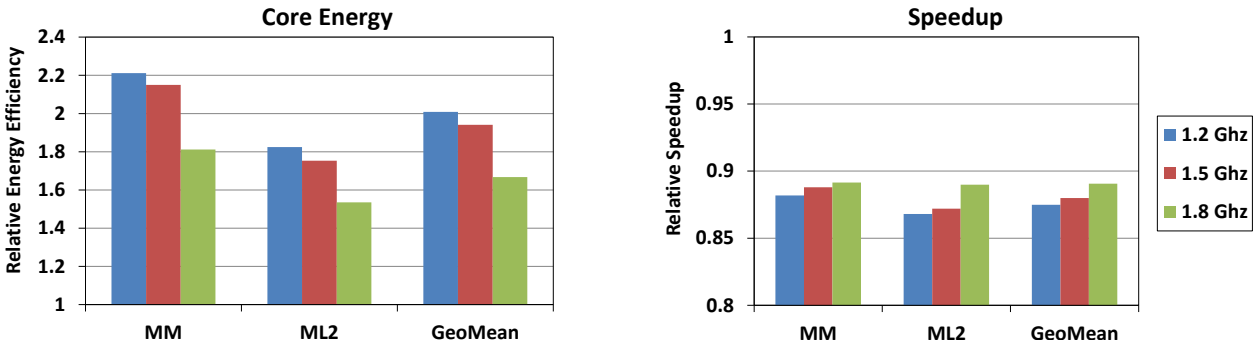

Figure 12: MICRO-BENCH run with user-space governor with frequency setting 1.2Ghz to 1.8Ghz

simpler. Our implementation can be extended to work with real time, interactive applications by using the CPU's performance counters with RAPL's power estimates.

Speaking of interactive applications, it became apparent to us that performance benchmarks might not be the best method of evaluating a power management policy, based on our tests. The performance centric nature of benchmarks like SPEC are good workloads for power management policies and, as can be seen from our comparisons, might not be the best way of differentiating any potential improvements,

Looking closer at the results of our Linux comparisons, and taking into consideration the possible issues with using performance benchmarks, it would seem like the P-state drivers do not make power management a solved issue. Based on our analysis, there are gains to be had with application specific policies.

10. Conclusion

Through this project, we aimed to prove that energy efficient policy decisions can be made by taking application characteristics into account. While our implementation is limited to a static analysis of workloads, our results show that this idea does have merit. This concept can be extended to a dynamic implementation by using performance counters and libraries like PAPI and interfaces like RAPL continue to make this more accessible.

We implemented an analytical model, E-MOS, that can take application characteristics and system power requirements and would suggest an optimum CPU frequency setting. E-MOS was designed to balance performance and energy consumption and our results show that we can achieve up to 2x energy efficiency with a performance loss of 13%. Considering that our data does not include Intel's turbo frequency boost, there is some room for performance improvements as well. Overall, a user-space power governor implementation that takes these application features into account has the potential to be more energy efficient than current defaults like the ondemand governor, even if it means a small loss in performance.

11. Acknowledgments

We thank Prof. Mike Swift for all the valuable inputs he gave during the course of the project. We also thank our peer reviewers for their valuable feedback for making this paper better and easy to read.

References

[1] [Online]. Available: https://www.kernel.org/doc/Documentation/power/apm-acpi.txt

[2] [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[3] [Online]. Available: http://linux.die.net/man/1/cpupower-monitor

[4] [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt

[5] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1992, pp. 316–322.

[6] R. Z. Ayoub, U. Ogras, E. Gorbatov, Y. Jin, T. Kam, P. Diefenbaugh, and T. Rosing, "Os-level power minimization under tight performance constraints in general purpose systems," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 2011, pp. 321–326.

[7] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proceedings of the 2004 international symposium on*

Low power electronics and design. ACM, 2004, pp. 174–179.

[8] ——, "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," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 18–28, 2005.

[9] A. Kumar, L. Shang, L.-S. Peh, and N. K. Jha, "Hybdtm: a coordinated hardware-software approach for dynamic thermal management," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 548–553.

[10] Y. Liang, P. Lai, and C. Chiou, "An energy conservation dvfs algorithm for the android operating system," *Journal of Convergence*, vol. 1, no. 1, 2010.

[11] Y. Liu, S. C. Draper, and N. S. Kim, "Sleepscale: runtime joint speed scaling and sleep states management for power efficient data centers," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 313–324.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[13] T. L. Martin and D. P. Siewiorek, "Balancing batteries, power, and performance: system issues in cpu speed-setting for mobile computing," Ph.D. dissertation, PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.

[14] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, vol. 2. sn, 2006, pp. 215–230.

[15] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, no. 2, pp. 20–27, 2012.

[16] J. Scaramella, "Worldwide server power and cooling expense 2006-2010 forecast," *International Data Corporation (IDC)*, 2006.

[17] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Design Automation Conference, 1999. Proceedings. 36th*. IEEE, 1999, pp. 134–139.

[18] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: A platform for os-level power management," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 289–302.

[19] C. D. Spradling, "Spec cpu2006 benchmark tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[20] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "Ecosystem: Managing energy as a first class operating system resource," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 123–132, 2002.