Memory-Aware DVFS Governing Policy for Improved Energy-Saving in the Linux Kernel

Philkyue Shin, Dahun Kim, and Seongsoo Hong
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
{pkshin, dhkim, sshong}@redwood.snu.ac.kr

***Abstract*—Energy-aware computing is one of the most critical issues in modern computing systems. Linux has introduced the schedutil governor since its 4.7 kernel release, which dynamically scales the processor frequency level to reduce energy consumption. Although it has been widely used in most Linux-based systems, the governor often makes inaccurate decisions in frequency selection, thus leading to unnecessary energy consumption. In this paper, we propose an enhanced governor as an alternative. We first rigorously analyze the schedutil's policy and then find out that it does not take into account memory stalls when characterizing CPU performance via CPI (cycles per instruction). This yields inherent inaccuracy, particularly in modern SoCs where multiple CPU cores and accelerators incur a huge amount of memory traffic over the system bus. To rectify this problem, we reformulate the CPU performance estimation function of the schedutil governor via the memory stall cycle ratio so that it can dynamically reflect the effects of changes in the processor's frequency and the system's memory contention. We show that our CPU performance estimation function is easily integrated into schedutil. We also show that the memory stall cycle ratio, the key element of the function, can be efficiently calculated at runtime with performance monitoring units (PMU) commonly available in most modern SoCs. We have implemented our governor and conducted extensive experiments to validate its effectiveness. Experimental results show that our governor saves more energy than schedutil by up to 28.91% without noticeable performance degradation.**

Keywords—dynamic voltage and frequency scaling (DVFS), energy efficiency, memory stall, schedutil governor

I. INTRODUCTION

Energy-aware computing has been an issue of utmost importance in a wide variety of modern computing systems ranging from massively parallel distributed server clusters to small hand-held mobile devices. Numerous operating system-level power reduction techniques and algorithms have been investigated in both industry and academia. Two representative techniques are power state management (PSM) and dynamic voltage and frequency scaling (DVFS).

The PSM strategy puts the entire system or some of its idle components into a low-power or power-off state whenever possible. The `cpuidle` subsystem of the Linux kernel is a well-known example of the PSM strategy [1]. It utilizes CPU idle states and CPU hotplug that dynamically disables and enables CPU cores.

The DVFS strategy has been the subject of extensive research since it offers great potential for energy conservation. Optionally coupled with CPU allocation and scheduling algorithms, many DVFS policies have been proposed to reduce energy consumption by finding the optimal operating frequency and/or voltage of the underlying processor. Among those, the task-level DVFS policies work in two steps for each runnable task in the system: (1) They estimate a task's execution time and idle time for a pre-specified future time interval, and (2) they reclaim the estimated idle time for the slow but energy-efficient execution of the task. If the estimated idle time is accurate enough and the idle time reclamation is faithfully carried out, the system can save energy without degrading the externally observable performance of the task [2].

As the DVFS mechanism has been widely incorporated into modern popular SoCs, the DVFS policies have been implemented into various operating systems. Linux is representative of such operating systems [3]. Specifically, the Linux kernel provides several `CPUFreq` governors, each of which realizes a specific policy for controlling how the processor frequency level is scaled. Since its 4.7 kernel release, Linux has offered the `schedutil` governor, which is now successfully and widely exploited in Android smartphones. Moreover, the recently emerging energy-aware scheduler (EAS) mandates it as a collaborating DVFS governor [4]. Similarly, in the server domain, the transition from the long-standing `powersave` governor to the `schedutil` governor has been taking place since Linux 5.7 kernel release [5]. As such, the `schedutil` governor is applied in many Linux-installed systems.

Despite such accomplishment, much of the underlying theory and operation of the `schedutil` governor has not been well analyzed in the literature. In this paper, we analyze the governor and uncover its effectiveness and limitations. The key idea behind this governor is that it estimates the acceptable performance of each task in the system and finds a frequency for

future execution that will ensure the desired performance while leaving only indispensable idle time.

The `schedutil` governor measures the performance of a task in terms of an *instruction rate*. The instruction rate is defined as the number of instructions executed per unit time interval. The governor estimates the desired instruction rate of a task from its past behavior using the exponential moving average. It then translates a task's desired instruction rate into a processor bandwidth demand in terms of the number of cycles per unit time interval. We call it the *cycle rate*. The governor adds together the bandwidth demands of all the tasks that will be running on the processor. It then adds another 25% of the total processor bandwidth demand to reserve the idle time that will accommodate inevitable task blocking due to task synchronization and IO waiting. This way the governor can select a processor frequency that meets both the processor's bandwidth demand and idle time requirement.

The `schedutil` governor effectively reduces energy consumption through dynamic performance estimation and improved integration with the kernel scheduler. Suppose the governor predicted a smaller desired performance value for a task than the actual value due to an unusual transient condition. Then the task would be running at a lower frequency than necessary and thus consume CPU cycles in the reserved idle time to make up for the lack of cycles. This would increase the estimated desired performance value of the task in the future. Conversely, an overestimated desired performance value of a task would increase idle time, which would eventually reduce the estimated desired performance of the task.

Unfortunately, the `schedutil` governor shows an important drawback when it comes to modern SoCs that possess multiple CPU cores and accelerators such as graphics processing units (GPU) and neural processing units (NPU). Since their usual workloads include memory-intensive programs such as deep learning and graphics applications, they are prone to experiencing nontrivial memory contention and exhibit a huge variance in memory access time. Since a CPU core stalls and wastes cycles for nothing during memory access, memory stall cycles affect the number of cycles per instruction (CPI) of a processor. In our experiment that we report in Section III, such CPI value varies significantly with the operating frequency. This phenomenon will become more evident in the future as the performance gap between the processing units and the memory grows.

When estimating CPU performance via CPI, the `schedutil` governor took a rather simplistic approach in that it regarded the CPI value of a processor as an intrinsic value, regardless of memory stall cycles inside a processor. Such inaccurate modeling of a processor's CPI value becomes an obstacle to finding the optimal frequency quickly. Suppose the number of memory stall cycles per instruction decreases. The governor will still operate at the current frequency that is unnecessarily high for the decreased memory stall cycles. Even if the governor gradually lowers the frequency after calculating the desired performance of tasks, energy will be wasted meanwhile. Conversely, suppose the number of memory stall cycles per instruction increases. Then the governor will gradually increase the frequency, but it will degrade tasks' performance until the processor reaches the desired frequency.

As an alternative, we propose an enhanced governor that can reduce the average frequency of the system without degrading the performance of a given workload, compared to the original `schedutil` governor. We name our governor the *memory-aware* `schedutil` governor or `mSchedutil` for short. To address the abovementioned limitation of `schedutil`, we propose a dynamic CPU performance estimation function using the memory stall cycle ratio that can be efficiently calculated with performance monitoring units (PMU) commonly available in most modern SoCs. We replace the original CPU performance estimation function of `schedutil` with our function.

We have implemented `mSchedutil` on the NVIDIA Jetson AGX Xavier platform, which is one of the most popular SoCs that possess multiple CPU cores and accelerators. We have evaluated `mSchedutil` with various synthetic and real-world workloads. The results show that `mSchedutil` saves more energy than `schedutil` by up to 28.91% without noticeable performance degradation. We observe that `mSchedutil` becomes more effective when the system's memory contention intensifies and the system's load increases. Our experimental result also shows that `mSchedutil` incurs only negligible runtime overhead. We make the source code of `mSchedutil`, along with experimental workloads, publicly available so that anybody can evaluate or use `mSchedutil` freely.

The remainder of this paper is organized as follows. Section II surveys existing memory-aware DVFS policies. Section III presents an in-depth analysis of the `schedutil` governor. It explains the governor's operational behavior and the mathematical equation that determines the next frequency. Section IV describes the problem at hand. Section V describes our solution, `mSchedutil`. Section VI describes the implementation of `mSchedutil` along with the memory stall cycle ratio estimation method. Section VII reports on the experimental evaluation. Finally, Section VIII concludes this paper.

II. Related Work

DVFS-based policies have proven to be effective in reducing energy consumption in a wide variety of computing systems and thus have been extensively investigated in both academia and industry. Many of them target memory-intensive workloads such as deep learning and graphics applications. In such workloads, memory stalls may have a great impact on the energy efficiency of a processor. As such, researchers have examined the effect of memory stalls on energy efficiency and proposed memory-aware DVFS scaling policies [6-15].

Such policies can be classified into system-level and task-level depending on whether they measure the impact of memory stalls on the entire system or on each task in the system. The system-level DVFS policies develop analytical models that evaluate the amount of energy consumption changed due to memory stalls. Spiliopoulos et al. proposed stall-based and miss-based models which estimated memory stall time using PMU. They, in turn, developed an energy estimation model [6] utilizing the estimated memory stall time, frequency, voltage,

and IPC of a processor. Su et al. estimated the system's energy consumption using a linear regression model and memory stall time measured with the special PMU and the other nine PMU events in x86 [7]. Liang et al. proposed a function of the memory access rate and frequency of a processor to estimate the system's energy consumption [8]. Gupta et al. proposed a regression-based energy estimation model using the cache miss rate, voltage, and frequency of a processor [9].

One of the advantages of the system-level DVFS policies is that they can achieve intuitive energy-related goals such as power capping. However, it is often quite difficult to generalize them to be applicable to various SoC architectures. This is because each SoC has different energy-related characteristics such as power gating, which are difficult to model, and because they often rely on special or dedicated PMU events.

The task-level DVFS policies estimate a task's idle time varied due to memory stalls and reclaim the estimated idle time for the slow but energy-efficient execution of the task. Such policies are subdivided into prior knowledge-guided and runtime-driven depending on how they estimate the idle time.

The prior knowledge-guided DVFS policies estimate idle time based on the prior knowledge given by a compiler or an application. For example, some applications have QoS or deadline constraints. After estimating the execution time, one can easily calculate the slack time from the timing constraints. Hsieh et al. used an application's memory utilization and memory access rate to capture its memory access behavior [10]. Using such values and past behavior, they estimated the execution time of a mobile game. They then calculated CPU and GPU frequencies in such a way that they could reclaim the slack time while maintaining the desired frame rate of the mobile game. Bahn et al. designed a genetic algorithm that aimed at minimizing the power consumption in processors and memory while satisfying the deadline constraints of all jobs [11]. To do so, they included the memory footprint and the number of memory read/write operations in their job model and evaluated the worst-case execution time. Lu et al. attempted to locate code sections that had substantial slacks with the aid of a compiler [12]. They suggested decreasing the frequency while executing such code sections.

The prior knowledge-guided DVFS policies are capable of significantly reducing energy consumption in real-time and QoS applications. Obviously, they are not suitable for general-purpose computing systems that run applications whose timing characteristics are not known in advance.

The runtime-driven DVFS policies estimate the idle time of a task from its past behavior. Choi et al. constructed regression models that calculated the idle time for the expected workload using the ratio of the total off-chip access time to the total on-chip computation time [13]. Hsu and Feng presented a β algorithm that estimated idle time using the intensity level of off-chip accesses [14]. They measured the intensity level by the number of executed instructions per second. Cho et al. introduced a notion of operational intensity, which was defined as the number of memory operations per byte access [15]. They found the optimal frequency that matched the operational intensity using the roofline model.

The `schedutil` and `mSchedutil` governor are runtime-driven DVFS policies like those found in [13], [14], and [15]. The `schedutil` governor can quickly respond to changes in running tasks since it traces the amount of performed work on an individual task basis.

Despite its advantage, `schedutil` lacks support for memory-aware frequency scaling since it does not consider memory stalls when modeling the performance of a core. Thus, we develop and propose `mSchedutil` as an alternative.

III. Analysis of Schedutil Governor

In this section, we present our analysis of `schedutil`. We first explain how the governor works in five steps and then rigorously derive the governor's equation to calculate the next frequency. Finally, we show the limitation of the governor with experiments.

A. Operational Behavior

The `schedutil` governor is designed to recompute the frequency whenever one or more processors in the system experience workload changes. When a task is migrated to a processor with higher performance or the aggregated desired instruction rate of a processor gets smaller, `schedutil` ensures that the processor runs at a lower frequency; otherwise, the processor runs at a higher frequency. For each processor, the `schedutil` iterates the following steps in series.

[Step 1: Triggering] The governor is invoked by not only the periodic scheduling tick but also the task scheduler of the kernel.

[Step 2: Monitoring] The governor keeps track of the instruction rate for each task running on a processor.

[Step 3: Prediction] Upon its invocation, the governor estimates the desired instruction rate of each task for the future. It then adds together the estimated instruction rates of all the tasks to be running on the processor.

[Step 4: Cycle Conversion] The governor calculates the processor bandwidth demand needed for executing the total estimated instruction rate in terms of cycle rate.

[Step 5: Frequency Calculation] The governor computes the frequency for the future such that the processor can offer 80% of its total cycle rate for its tasks and leave the rest 20% as idle time.

The *triggering* step enables the governor to respond to the sporadic changes in the processor bandwidth demand as well as allows it to perform periodic decision-making. When a task is migrated into or out of a processor or when a task is forked or terminated on a processor, the kernel's task scheduler is called, which in turn invokes the governor. The tick scheduler also periodically triggers the governor.

The *monitoring* step allows the governor to have full knowledge of the past progress of each task in terms of the instruction rate. To delve into this step, we first introduce a *governor epoch* which is a time interval delimited by two successive governor invocations. Fig. 1 depicts the i^{th} governor epoch. The i^{th} governor epoch begins with the i^{th} governor

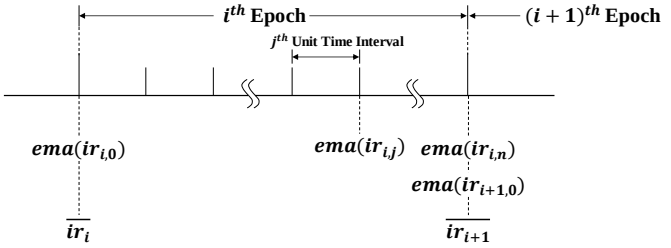

Fig. 1. A governor epoch divided into multiple unit time intervals.

invocation and ends with the $(i+1)^{st}$. Each governor epoch is subdivided into one or more unit time intervals. Fig. 1 shows that the i^{th} governor epoch consists of n unit time intervals.

Using several values measured at runtime, the governor calculates the instruction rate $ir_{i,j}(\tau)$ of a task τ in the j^{th} unit time interval of the i^{th} governor epoch for $1 \le j \le n$, as follows:

$$ir_{i,j}(\tau) = \frac{e_{i,j}(\tau)}{T} \cdot f_i\big(p_i(\tau)\big) \cdot IPC\big(p_i(\tau)\big) \tag{1}$$

where $e_{i,j}(\tau)$ is the time spent to execute τ in the j^{th} unit time interval of the i^{th} governor epoch and $p_i(\tau)$, $f_i\big(p_i(\tau)\big)$ and $IPC\big(p_i(\tau)\big)$ are respectively the processor hosting τ, its frequency, and its IPC value in the i^{th}governor epoch. $IPC(p)$ is treated as an intrinsic value of the processor p. T is simply the unit time interval size, which is 1ms in the current Linux implementation.

Let $\overline{ir_i}(\tau)$ be the estimate of the desired instruction rate of τ for the i^{th} governor epoch. On the $(i+1)^{st}$governor invocation, the *prediction* step computes $\overline{ir_{i+1}}(\tau)$ by repetitively calculating the exponential moving average of $ir_{i,j}(\tau)$ as follows:

$$ema\left(ir_{i,j}(\tau)\right) = \alpha \cdot ir_{i,j}(\tau) + (1-\alpha) \cdot ema\left(ir_{i,j-1}(\tau)\right) \tag{2}$$

where $ema\left(ir_{i,0}(\tau)\right) = \overline{ir_i}(\tau)$ and $(1-\alpha)^{32} = 0.5$. In the current Linux implementation, the smoothing factor α is determined such that the instruction rate measured 32ms ago is weighted half as much as the instruction rate measured 1ms ago. The 32ms is selected by the rule of thumb.

$\overline{ir_{i+1}}(\tau)$ equals to $ema\left(ir_{i,n}(\tau)\right)$ as shown in Fig. 1. We thus simply have:

$$\overline{ir_{i+1}}(\tau) = ema\left(ir_{i,n}(\tau)\right) \tag{3}$$

In turn, the governor adds up $\overline{ir_{i+1}}(\tau)$ of all tasks τ in the processor p_{i+1} 's runqueue and computes $\overline{ir_{i+1}}(p_{i+1})$. $\overline{ir_{i+1}}(p_{i+1})$ is the total estimated desired instruction rate of p_{i+1} for the $(i+1)^{st}$ governor epoch.

$$\overline{ir_{i+1}}(p_{i+1}) = \sum_{\tau \in p_{i+1}} \overline{ir_{i+1}}(\tau) \tag{4}$$

The *cycle conversion* step translates $\overline{ir_{i+1}}(p_{i+1})$ into the cycle rate $\overline{cr_{i+1}}(p_{i+1})$ by dividing it by the intrinsic IPC value of p_{i+1}. As a result, we get the following equation containing the CPI value, the reciprocal of the IPC value.

$$\begin{aligned}\overline{cr_{i+1}}(p_{i+1}) &= \overline{ir_{i+1}}(p_{i+1}) \cdot \frac{1}{IPC(p_{i+1})} \\ &= \overline{ir_{i+1}}(p_{i+1}) \cdot CPI(p_{i+1})\end{aligned} \tag{5}$$

From $\overline{cr_{i+1}}(p_{i+1})$, the *frequency calculation* step determines the frequency $f_{i+1}(p_{i+1})$ for the $(i+1)^{st}$ governor epoch that allows the processor p_{i+1} to spend 80% of its CPU cycles running the tasks in the runqueue and reserves the remaining 20% as idle time.

$$f_{i+1}(p_{i+1}) = 1.25 \cdot \overline{cr_{i+1}}(p_{i+1}) \tag{6}$$

We have explained how the `schedutil` calculates $f_{i+1}(p_{i+1})$ with six equations. We merge all these equations, as follows.

$$\begin{aligned}f_{i+1}(p_{i+1}) &= 1.25 \cdot \overline{cr_{i+1}}(p_{i+1}) \\ &= 1.25 \cdot \overline{ir_{i+1}}(p_{i+1}) \cdot CPI(p_{i+1}) \\ &= 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(ir_{i,n}(\tau)\right) \cdot CPI(p_{i+1}) \\ &= 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot f_i\big(p_i(\tau)\big) \cdot \frac{CPI(p_{i+1})}{CPI\big(p_i(\tau)\big)}\right)\end{aligned} \tag{7}$$

We refer to (7) as the *frequency equation* of `schedutil`.

B. Inherent Inaccuracy

The `schedutil` governor is indeed effective in practice. Also, it can be easily implemented and executed only with a small computational cost and memory space. Note that its major computational burden lies in computing $ema\left(ir_{i,j}(\tau)\right)$ for each task. Even this can be done cheaply by using the previously computed $ema\left(ir_{i,j-1}(\tau)\right)$ and calculating $ir_{i,j}(\tau)$.

Unfortunately, the `schedutil` governor has a noticeable downside that arises due to a simplified assumption on a processor's performance characteristics. Note that it models a processor's CPI value as a $CPI(p)$ function, which always returns a constant CPI value for a given processor p. However, we argue that a processor's CPI changes more drastically in the modern SoCs than the `schedutil` originally expected, due to increased memory contention. To show that our argument holds, we have performed an experiment.

In this experiment, we ran SPECrate 2017 on the NVIDIA Jetson AGX Xavier platform with a constant amount of memory contention being applied to the running benchmark [16]. We then measured CPI values while changing the frequency of the CPU. We repeated the measurement with each of the 23 benchmarks in SPECrate 2017. We found out that the CPI linearly increases with the frequency in all the measurements. To pictorially demonstrate the relationship between the frequency and the CPI, we selected the first three measurements and plot them in Fig. 2.

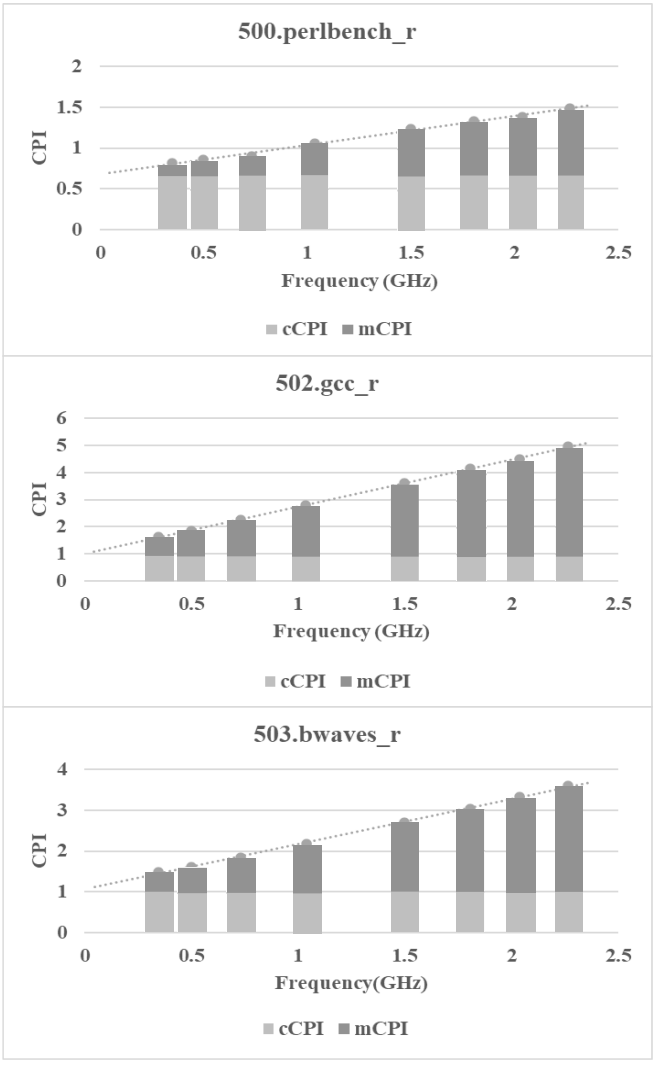

Fig. 2. Linear relationship between frequency and CPI.

The linear relationship between the frequency and the CPI of a processor comes from memory stalls occurring inside the processor during memory access. Note that for given memory access time, the memory stall cycle count of a processor is proportional to the operating frequency. We refer to such varying memory stall cycle count per instruction as *memory stall CPI* or *mCPI* for short.

In addition to such memory stall cycles, a processor's CPI includes the cycles for the processor to spend executing the instruction. For a given CPU instruction, such execution cycles are intrinsic to the CPU design and the cycle count remains constant regardless of the operating frequency. We call it *computation CPI* or *cCPI* for short.

Obviously, CPI is the summation of cCPI and mCPI. The cCPI is simply the y-intercept of the linear CPI function of the frequency since the CPI of a processor converges to its intrinsic cCPI value as the operating frequency converges to zero. Similarly, the mCPI is a multiplier of the frequency and the slope of the CPI function. In Fig. 2, we color cCPI in light gray, and mCPI in dark gray.

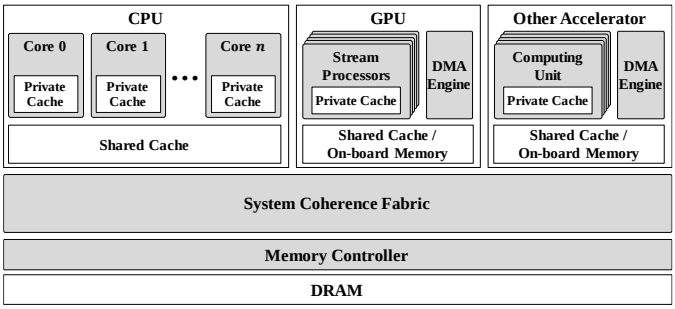

Fig. 3. Target architecture model.

In order to eliminate the inherent inaccuracy of the `schedutil`, one needs to be able to compute the CPI value at the beginning of every governor epoch when the `schedutil` governor is invoked. Unfortunately, it is quite difficult to compute it since the slope of the CPI function dynamically changes with the workload running on the system.

IV. Problem Description

In this paper, we aim at enhancing the `schedutil` governor of the Linux kernel by addressing its inherent inaccuracy which was analyzed in the previous section. Specifically, we attempt to reduce the energy consumption of the CPU in the system while maintaining the rate of instructions that are executed by the running tasks, compared to the original governor.

To achieve this, we model the CPI as a linear function of the frequency for a given workload so that we can calculate a more accurate CPI value than the original governor when the frequency equation (7) is called. This surely lowers the average operating frequency of the CPU. Thus, the problem at hand is to formulate a new frequency equation for `mSchedutil` with the new CPI function and evaluate the equation dynamically when needed.

In this section, we first briefly describe the target system model and then formally present our problem.

A. Target System Architecture

Our target system is an SoC possessing multiple CPU cores and one or more accelerators. The system has a multi-level cache and onboard memory. The main memory is shared by all the processing units in the SoC. Since the memory controller operates with its own frequency and voltage level, memory access latency is not affected by the CPU frequency. Fig. 3 pictorially shows the target architecture that is typical of modern SoCs including the NVIDIA Jetson AGX Xavier platform.

B. Problem Definition

To formally present our problem, we start with some necessary definitions. We define a more accurate CPI value function $CPI(p, f)$ than $CPI(p)$. We elaborate on this function using mCPI and cCPI. We then reformulate the frequency equation of `schedutil` by substituting $CPI(p)$ with $CPI(p, f)$. As a result, we get a new frequency equation for `mSchedutil`.

Definition 1. For a given workload running on the system, we let $CPI(p, f)$ be the CPI value of the processor p running at the frequency f.

We in turn define mCPI and cCPI for $CPI(p, f)$.

Definition 2. For $CPI(p, f)$, we denote the number of memory stall cycles per instruction as $mCPI(f)$ and the number of computation cycles per instruction as $cCPI(p)$.

The following holds trivially from **Definition 2**.

$$CPI(p, f) = mCPI(f) + cCPI(p) \tag{8}$$

We fit $mCPI(f)$ into a linear function of f using a proportional constant m. As a result, we have the following equation.

$$mCPI(f) = m \cdot f \tag{9}$$

Note that we model m as constant for a given workload in the system for the sake of simplicity; it dynamically changes with the running workload.

We then model $cCPI(p)$ as a linear function of $CPI(p)$ using another proportional constant c. The $CPI(p)$ is originally defined in `schedutil` and denotes the CPU performance that is constant regardless of frequency.

$$cCPI(p) = c \cdot CPI(p) \tag{10}$$

By modeling $cCPI(p)$ this way, $CPI(p, f)$ subsumes $CPI(p)$. Depending on the proportional constants, $CPI(p, f)$ obtains the same or a more accurate CPI value than $CPI(p)$. We determine the proper proportional constants that obtain an accurate CPI value at runtime.

From (8), (9), and (10), we derive $CPI(p, f)$ as follows.

$$CPI(p, f) = m \cdot f + c \cdot CPI(p) \tag{11}$$

In order to derive a frequency equation for our proposed governor, we modify the frequency equation of `schedutil` by substituting $CPI(p)$ with $CPI(p, f)$. The resultant equation follows:

$$f_{i+1}(p_{i+1}) = 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot f_i(p_i(\tau)) \cdot \frac{CPI(p_{i+1}, f_{i+1}(p_{i+1}))}{CPI\left(p_i(\tau), f_i(p_i(\tau))\right)}\right) \tag{12}$$

From (11), we rewrite (12) as follows:

$$f_{i+1}(p_{i+1}) = 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot f_i(p_i(\tau)) \cdot \frac{m_{i+1} \cdot f_{i+1}(p_{i+1}) + c_{i+1} \cdot CPI(p_{i+1})}{m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))}\right) \tag{13}$$

Consequently, our problem at hand is to efficiently evaluate (13) at runtime to obtain the frequency $f_{i+1}(p_{i+1})$ of the processor p_{i+1} at the beginning of the $(i+1)^{st}$ governor epoch.

V. Solution Approach: Approximation of the Frequency Equation

The `mSchedutil` governor must evaluate the frequency equation (13) at the beginning of each governor epoch in order to determine the frequency of a processor. Unfortunately, it is not practically feasible to compute (13) in its current form since the coefficients m_{i+1}, c_{i+1}, m_i, and c_i vary dynamically with the workload running on the system in each governor epoch. We thus take an approximation approach to get rid of these coefficients.

In this section, we derive an approximate frequency equation from (13) using the memory stall cycle ratio and linear regression. In designing our approximation, we intend to derive a frequency equation that consists of such values that can be easily measured on the fly via PMUs already built-in most COTS SoCs.

A. Reformulating Frequency Equation with Memory Stall Cycle Ratio

To derive a new frequency equation from (13) for our `mSchedutil` governor, we first eliminate the two coefficients m_{i+1} and c_{i+1}, then introduce the memory stall cycle ratio, and finally eliminate the remaining two coefficients m_i and c_i.

Firstly, we substitute m_{i+1} and c_{i+1} with m_i and c_i in (13). We have observed that the characteristics of the system's workload change little during the short 4ms governor epoch. We thus consider that the coefficient measured in a governor epoch remains unchanged in the subsequent governor epoch. As a result, we have the following equation:

$$f_{i+1}(p_{i+1}) = 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot f_i(p_i(\tau)) \cdot \frac{m_i \cdot f_{i+1}(p_{i+1}) + c_i \cdot CPI(p_{i+1})}{m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))}\right) \tag{14}$$

Secondly, we proceed to eliminate m_i and c_i from (14). As $f_{i+1}(p_{i+1})$ appears on both sides of the equal sign in (14), we need to solve (14) for $f_{i+1}(p_{i+1})$. The hard part in doing so is to compute $\left(m_i \cdot f_{i+1}(p_{i+1}) + c_i \cdot CPI(p_{i+1})\right) / \left(m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))\right)$. To ease this step, we reformulate this formula as below:

$$\frac{m_i \cdot f_{i+1}(p_{i+1}) + c_i \cdot CPI(p_{i+1})}{m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))} = \frac{m_i \cdot f_i(p_i(\tau))}{m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))} \cdot \frac{f_{i+1}(p_{i+1})}{f_i(p_i(\tau))} + \left(1 - \frac{m_i \cdot f_i(p_i(\tau))}{m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))}\right) \cdot \frac{CPI(p_{i+1})}{CPI(p_i(\tau))} \tag{15}$$

We note that the term $m \cdot f / (m \cdot f + c \cdot CPI(p))$ in (15) is the ratio of the memory stall cycle count to the instruction cycle count. We formally define the ratio in what follows.

Definition 3. The memory stall cycle ratio $mr(p, f)$ is defined as below:

$$mr(p,f) = \frac{m \cdot f}{m \cdot f + c \cdot CPI(p)} = \frac{mCPI(f)}{CPI(p,f)}$$

We now rewrite (15) using **Definition 3** and have the following equation:

$$\frac{m_i \cdot f_{i+1}(p_{i+1}) + c_i \cdot CPI(p_{i+1})}{m_i \cdot f_i(p_i(\tau)) + c_i \cdot CPI(p_i(\tau))} = mr\left(p_i(\tau), f_i(p_i(\tau))\right) \cdot \frac{f_{i+1}(p_{i+1})}{f_i(p_i(\tau))} + \left(1 - mr\left(p_i(\tau), f_i(p_i(\tau))\right)\right) \cdot \frac{CPI(p_{i+1})}{CPI(p_i(\tau))} \quad (16)$$

We plug (16) into (14) and obtain the following equation:

$$f_{i+1}(p_{i+1}) = 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot f_i(p_i(\tau)) \cdot \left(mr\left(p_i(\tau), f_i(p_i(\tau))\right) \cdot \frac{f_{i+1}(p_{i+1})}{f_i(p_i(\tau))} + \left(1 - mr\left(p_i(\tau), f_i(p_i(\tau))\right)\right) \cdot \frac{CPI(p_{i+1})}{CPI(p_i(\tau))}\right)\right) \quad (17)$$

Solving (17) for $f_{i+1}(p_{i+1})$, we end up with the following equation:

$$f_{i+1}(p_{i+1}) = 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot f_i(p_i(\tau)) \cdot IPC(p_i(\tau)) \cdot \left(1 - mr\left(p_i(\tau), f_i(p_i(\tau))\right)\right)\right) \cdot CPI(p_{i+1}) \div \left(1 - 1.25 \cdot \sum_{\tau \in p_{i+1}} ema\left(\frac{e_{i,n}(\tau)}{T} \cdot mr\left(p_i(\tau), f_i(p_i(\tau))\right)\right)\right) \quad (18)$$

(18) is the frequency equation of `mSchedutil`.

B. Estimating Memory Stall Cycle Ratio

To evaluate (18), the `mSchedutil` governor needs to additionally acquire the memory stall cycle ratio since the other values are already known through the `schedutil` governor. Such acquisition must be carried out at a low runtime cost since it is repeated every governor epoch.

In the literature, there exist several estimation methods for the memory stall cycle ratio, some of which suggest implementing a dedicated hardware performance counter [17], [18], [19] and others estimate it using special PMUs [7], [20]. Unlike these approaches, we intend to come up with a method that is as hardware-agnostic as possible. Our method is based on linear regression and relies only on commonly available PMU events that can be found in most COTS SoCs.

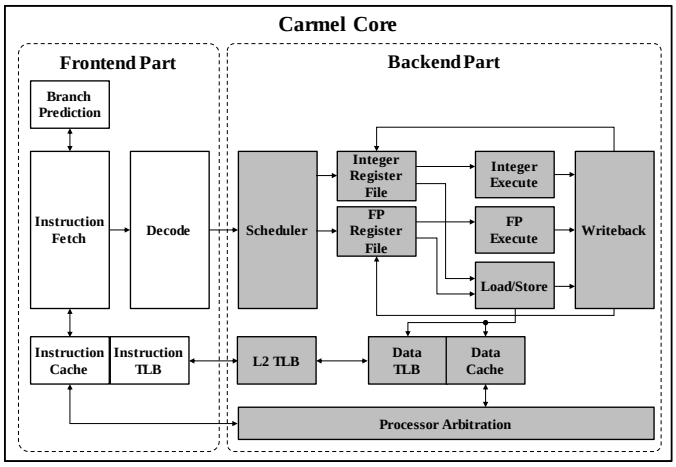

Fig. 4. Frontend and backend of the pipeline in the Carmel core.

To design our estimation method, we comprehensively analyze the stall cycles of instruction and classify them into three categories: bad speculation, frontend, and backend. The modern superscalar, out-of-order microarchitecture such as the Arm v8.2 Carmel CPU core has a pipeline that consists of the frontend part and the backend part as shown in Fig. 4. The branch predictor in the frontend part may incur bad speculation stalls and the rest of the frontend part may experience frontend stalls. The backend part may incur backend stalls [21].

Let $sr(p,f)$ be the ratio of the bad speculation stall cycle count to the total cycle count. Similarly, let $fr(p,f)$ and $br(p,f)$ be the ratios of the frontend and backend stall cycle count to the total cycle count, respectively. We model the memory stall cycle ratio $mr(p,f)$ as a linear combination of $sr(p,f)$, $fr(p,f)$, and $br(p,f)$, as follows.

$$mr(p,f) = \alpha_1(p) \cdot sr(p,f) + \alpha_2(p) \cdot fr(p,f) + \alpha_3(p) \cdot br(p,f) - \alpha_4(p) \quad (19)$$

Our model (19) is based on our previous work [22]. It shows that the backend stall cycle ratio is a sum of the memory-related term that is proportional to the frequency and a constant term related to instruction execution. So do the other two memory stall cycle ratios. The regression coefficient $\alpha_4(p)$ denotes the constant stall cycle ratio.

We simply obtain the regression coefficients $\alpha_1(p)$ through $\alpha_4(p)$ with a multiple-linear regression tool [23]. At runtime, our governor can easily evaluate $mr(p,f)$ by computing dependent variables $sr(p,f)$, $fr(p,f)$, and $br(p,f)$ using values from related PMU events.

VI. Putting It All Together

In this section, we present the implementation details of `mSchedutil`. Particularly, we describe the internal structure and operation of `mSchedutil` and explain the PMU-based

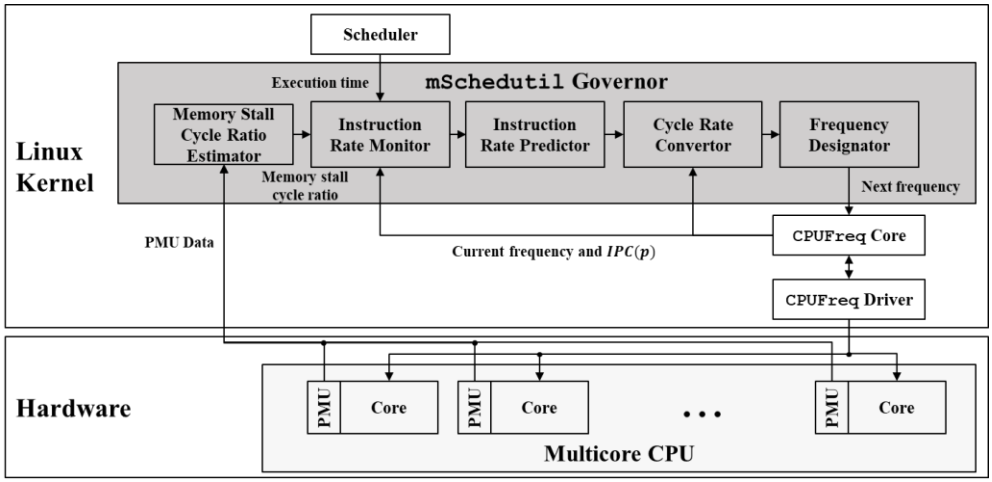

Fig. 5. Overall structure of the `mSchedutil` governor.

implementation of the regression model for estimating the memory stall cycle ratio.

A. Structure and Operation of the Proposed Governor

Fig. 5 pictorially depicts the overall structure of `mSchedutil`. It consists of five components: (1) instruction rate monitor, (2) instruction rate predictor, (3) cycle rate convertor, (4) frequency designator, and (5) memory stall cycle ratio estimator. We integrate these components into the `CPUFreq` subsystem of the Linux kernel.

The instruction rate monitor computes the past instruction rate of each task running on a given processor. It reads in a task's execution time from the kernel scheduler and the frequency and $IPC(p)$ values of the processor from the `CPUFreq` core module. In order to reflect memory stall in computing an instruction rate, it also reads in a memory stall cycle ratio from the memory stall cycle ratio estimator.

The instruction rate predictor computes the exponential moving average among instruction rate values provided by the previous component. It then produces a desired instruction rate for the next governor epoch. To do so, it adds together the desired instruction rates of all the individual tasks to be running on the processor.

The cycle rate convertor calculates the desired cycle rate for the next governor epoch. Finally, the frequency designator determines the next frequency according to (18).

The memory stall cycle ratio estimator calculates the $mr(p,f)$ value. In our implementation, we intentionally decouple it from the rest since it is highly dependent on the target SoC. We elaborate on this issue in what follows.

TABLE I. PMU EVENTS USED FOR ESTIMATING MEMORY STALL CYCLE RATIO.

Event Number	Event Mnemonic	Description
0x08	`INST_RETIRED`	Instruction architecturally executed
0x11	`CPU_CYCLES`	Cycles elapsed since PMU enabling
0x22	`BR_MIS_PRED_RETIRED`	Instruction architecturally executed, mispredicted branch
0x23	`STALL_FRONTEND`	No operation issued due to the frontend
0x24	`STALL_BACKEND`	No operation issued due to backend

B. Memory Stall Cycle Ratio Estimator

To construct the memory stall cycle ratio estimator, we need to determine the regression coefficients $\alpha_1(p)$ through $\alpha_4(p)$ offline. To do so, we have collected 1,458 tuples of $[mr(p,f)$, $sr(p,f)$, $fr(p,f)$, $br(p,f)]$ by varying benchmarks, CPU frequencies, and memory frequencies. Specifically, we ran 27 benchmarks from SPEC 2017 on the NVIDIA Jetson AGX Xavier platform at the CPU frequency changed in nine steps and at the memory frequency changed in six steps.

To obtain a tuple at each measurement run, we used the PMU events listed in Table I. These are basic Arm PMU events and similar ones are found in many other microarchitectures like x86. Specifically, we obtain $sr(p,f)$ by dividing the `BR_MIS_PRED_RETIRED` value by the `INST_RETIRED` value since $sr(p,f)$ is approximately equal to the ratio of the number of mispredicted branch instructions to the number of retired instructions. Similarly, $fr(p,f)$ and $br(p,f)$ are `STALL_FRONTEND` and `STALL_BACKEND` divided by `CPU_CYCLES`, respectively.

Calculating $mr(p,f)$ is a bit complicated. According to **Definition 3**, we need $CPI(p,f)$ and $mCPI(f)$. We can get $CPI(p,f)$ easily by dividing `CPU_CYCLES` by `INST_RETIRED`. To get $mCPI(f)$, we need to determine $cCPI(p)$ according to (8). To do so, we rely on simple linear regression among tuples $[f, CPI(p,f)]$ since $cCPI(p)$ is the y-intercept of the linear CPI function of f as stated in (11). It is trivial to obtain such tuples.

VII. EXPERIMENTAL EVALUATION

To assess the effectiveness of `mSchedutil` over `schedutil`, we have performed four experiments. In the first and second experiments, we analyze how much energy `mSchedutil` saves compared to `schedutil` under various intensities of the system's memory contention and various amount of the system's load, respectively. In the third experiment, we evaluate the energy efficiency of the governors on real-world applications. Finally, in the fourth experiment, we assess the runtime overhead of `mSchedutil`.

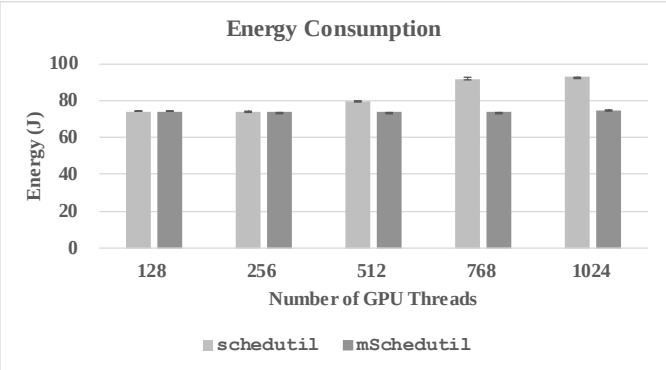

Fig. 6. Energy consumption under `schedutil` and `mSchedutil` with the system's memory contention varied.

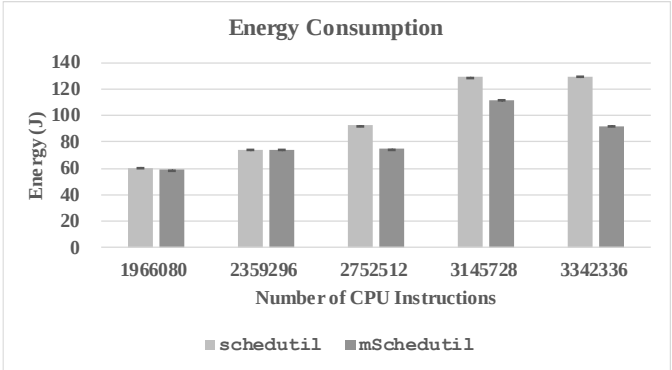

Fig. 7. Energy consumption under `schedutil` and `mSchedutil` with the system's load varied.

In this section, we give a detailed account of the experimental setup and present the experimental results.

A. Experimental Setup

Our target platform was the NVIDIA Jetson AGX Xavier platform hosting Ubuntu for Arm 18.04.1 based on Linux kernel 4.9.108 and JetPack 4.1.1. This platform supported seven power modes and we chose mode 5 for our experiments. We did so to make the effect of a governor more visible. A CPU core in mode 5 can consume up to 2.2 times as much energy as in the default mode. We measured consumed CPU energy using the integrated INA3221 power monitor in the Xavier platform.

For the first and second experiments, we created synthetic workloads for CPU and GPU, respectively. The CPU workload is a simple thread that periodically executed a fixed mix of arithmetic and memory instructions. The GPU workload consists of multiple threads executing only memory instructions.

For the third experiment, we collected applications for Arm Linux and designed five test scenarios. These scenarios include deep learning, game, office, video, and web browsing. The deep learning scenario runs a deep learning object detection application we took from the NVIDIA GitHub [24]. The game scenario runs an OpenArena game demo [25]. The office scenario renders LibreOffice files periodically [26]. The video scenario plays HEVC video files with the totem player [27]. The web browsing scenario renders offline chromium web pages periodically [28].

B. Experimental Result

In the first experiment, we ran the synthetic workloads for two minutes under the two governors and measured the total amounts of CPU energy consumption in Joule. We repeated this measurement, changing the number of GPU threads to see the effect of memory contention. We kept the throughputs of the CPU workload identical under both governors by precisely executing 2,752,512 instructions every 10ms in the CPU workload. We repeated each measurement 100 times and calculated the mean value and standard error.

Fig. 6 shows the result. The x-axis is the number of GPU threads and the y-axis is the CPU energy consumption under the two governors. The result shows that `mSchedutil` saves more energy than `schedutil` by up to 0.01% to 19.25%. It also shows that `mSchedutil` works more effectively as the memory contention gets increased.

In the second experiment, to see the effect of the system's load, we performed the above measurements with the CPU workload varied and the memory contention fixed to 1024 GPU threads. Fig. 7 shows that `mSchedutil` saves energy by 2.16% to 28.91% compared to `schedutil`. As the system's load gets increased, `mSchedutil` tends to achieve more energy savings because the larger the system's load, the larger the margin for improvement.

In the third experiment, we measured the CPU energy consumption under real-world workloads. We also measured the instruction rates of the workloads under `schedutil` and `mSchedutil`. Since we could not keep the throughput of the real-world applications identical under the two governors, we needed to compare the throughput of applications using the measured instruction rates.

Fig. 8 shows the result. When comparing the throughputs of the applications, the two governors do not make a significant

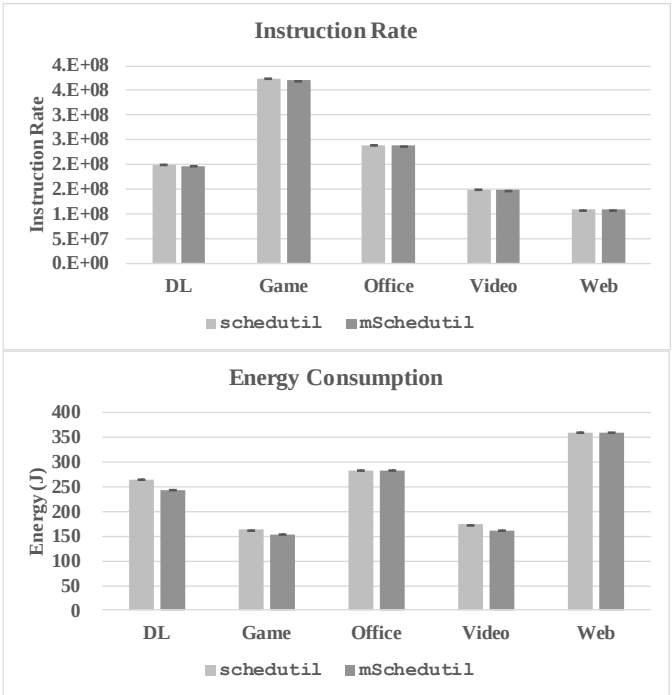

Fig. 8. CPU energy consumption under `schedutil` and `mSchedutil` (real-world applications case).

difference. In the experiment, `mSchedutil` saves more energy than `schedutil` by 7.97%, 5.23%, and 7.03% in the deep learning, game, and video scenarios, respectively.

In the fourth experiment, we separately measured the execution times of `schedutil` and `mSchedutil` code and compared them. Our governor took only 7.73% longer time than `schedutil`. Considering the short execution time of `schedutil`, such overhead is negligible.

VIII. Conclusion

We presented a memory-aware DVFS governor, `mSchedutil` that improves the energy-saving efficiency of `schedutil` in Linux. The `schedutil` governor estimates CPU performance without considering memory stall cycles although CPU performance expressed by CPI is dynamically affected by the memory stall. After rigorously analyzing the frequency equation of `schedutil`, we reformulated its inaccurate CPU performance estimation function using the memory stall cycle ratio so that the resultant `mSchedutil` governor could adaptively reflect the effect of the system's memory contention. We also showed that the new CPU performance estimation function is easily integrated into the original governor.

We implemented `mSchedutil` as an extension to the `schedutil` governor inside the `CPUFreq` subsystem of the Linux kernel. As part of the implementation, we designed a regression model that enabled our governor to efficiently calculate the memory stall cycle ratio at runtime based on our analysis of the pipeline structure of the modern superscalar multicore CPU. We demonstrated that our implementation worked at low runtime cost using only PMU events commonly available in most modern SoCs.

We extensively evaluated `mSchedutil` via diverse experiments. The results showed that `mSchedutil` saved energy by up to 28.91% compared to the `schedutil` governor.

Acknowledgment

This work is supported by Samsung Electronics' university R&D program.

References

[1] Rafael J. Wysocki, "CPU Idle Time Management," [online]. Available: https://www.kernel.org/doc/html/latest/driver-api/pm/cpuidle.html

[2] Stefanos Kaxiras and Margaret Martonosi, "Computer architecture techniques for power-efficiency," Synthesis Lectures on Computer Architecture 3.1 (2008): 23-29.

[3] Rafael J. Wysocki, "CPU Performance Scaling," [online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html

[4] Arm, "EAS Overview and Integration Guide," [online]. Available: https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/energy-aware-scheduling

[5] Rafael J. Wysocki, "[GIT PULL] More power management updates for v5.7-rc1," [online]. Available: https://lore.kernel.org/lkml/CAJZ5v0ji9p4_whgcJbh6mm8cdYpruHEzOsTqje7JedD45wH5Dg@mail.gmail.com

[6] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas, "Green governors: A framework for continuously adaptive dvfs," IEEE International Green Computing Conference and Workshops, 2011.

[7] Bo Su et al., "PPEP: Online performance, power, and energy prediction framework and DVFS space exploration," 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.

[8] Wen-Yew Liang, Ming-Feng Chang, and Yen-Lin Chen, "Optimal Energy Saving DVFS Approach of Embedded Processors," Journal of Information Science and Engineering 33.5, 2017, 1121-1140.

[9] Manjari Gupta, Lava Bhargava, and S. Indu, "Dynamic workload-aware DVFS for multicore systems using machine learning," Computing 103, 2021, 1747-1769.

[10] Chen-Ying Hsieh et al., "MEMCOP: memory-aware co-operative power management governor for mobile games," Design Automation for Embedded Systems 22, 2018, 95-116.

[11] Hyokyung Bahn, and Kyungwoon Cho, "Evolution-based real-time job scheduling for co-optimizing processor and memory power savings," IEEE Access 8, 2020.

[12] Teng Lu, Partha Pratim Pande, and Behrooz Shirazi, "A dynamic, compiler guided DVFS mechanism to achieve energy-efficiency in multi-core processors," Sustainable Computing: Informatics and Systems 12, 2016, 1-9.

[13] Kihwan Choi, Ramakrishna Soma, and Massoud 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 transactions on computer-aided design of integrated circuits and systems 24.1 (2004): 18-28.

[14] Chung-hsing Hsu and Wu-chun Feng, "A power-aware run-time system for high-performance computing," SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, 2005.

[15] Seong Jin Cho, Seung Hyun Yun, and Jae Wook Jeon, "A powersaving DVFS algorithm based on operational intensity for embedded systems," IEICE Electronics Express 12.3 (2015): 20141128-20141128.

[16] Standard Performance Evaluation Corporation, "SPEC CPU®2017 Utilities," [online]. Available: https://www.spec.org/cpu2017

[17] Stijn Eyerman and Lieven Eeckhout, "A counter architecture for online DVFS profitability estimation," IEEE Transactions on Computers 59.11 (2010): 1576-1583.

[18] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," Proceedings of the 7th ACM international conference on Computing frontiers, 2010.

[19] Jawad Haj-Yahya et al., "SysScale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors," ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020.

[20] Bo Su et al., "Implementing a leading loads performance predictor on commodity processors," USENIX Annual Technical Conference (USENIX ATC 14), 2014.

[21] Ahmad Yasin, "A top-down method for performance analysis and counters architecture," IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.

[22] Jungho Kim, et al., "Memory-aware fair-share scheduling for improved performance isolation in the Linux kernel," IEEE Access 8 (2020): 98874-98886.

[23] Leona S. Aiken, Stephen G. West, and Steven C. Pitts, "Multiple linear regression," Handbook of psychology (2003): 481-507.

[24] dusty-nv, "jetson-inference," [online]. Available: https://github.com/dusty-nv/jetson-inference

[25] OA Team, "OpenArena," [online]. Available: https://openarena.ws

[26] LibreOffice developers, "LibreOffice," [online]. Available: https://www.libreoffice.org

[27] The GNOME Project, "Totem player," [online]. Available: https://wiki.gnome.org/Apps/Videos

[28] Chromium developers, "Chromium," [online]. Available: https://www.chromium.org