3 PARETO GOVERNORS

3.1 Overview

In this chapter we develop new operating system governors (resource managers) that seek to operate the system at or close to Dynamic EO (power-performance Pareto frontier) so that Service-Level Agreements (SLAs) are satisfied. We call such governors *Pareto governors*.

We consider the following SLAs in this chapter. (See Chapter 1, Section 1.2, for more background on these SLAs.)

- **SLAee**: Maximize energy efficiency.
- **SLApower**: Maximize performance given a power cap/budget.
- **SLAperf**: Maximize power savings given a performance target.

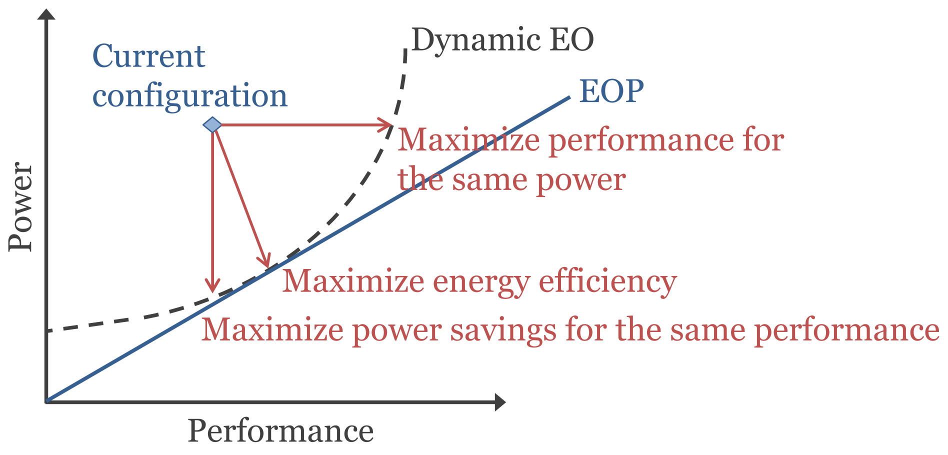

Current configuration
Dynamic EO
EOP
Maximize performance for the same power
Maximize energy efficiency
Maximize power savings for the same performance
Power
Performance

Figure 3.1: State transitions to Dynamic EO for meeting SLAs.

Figure 3.1 shows what transitions the Pareto governors must make to the current operating point to meet various SLAs.

Our governors manipulate two dynamic reconfiguration knobs—processor frequency (and voltage) scaling and cache prefetching. Processor caches improve performance by keeping recently or frequently used data on chip so that when the data is needed again, a long offchip access to main memory is avoided. However, caches are of finite size and previously used data may need to be evicted to make room for other data before being needed again. Cache misses happen when data is needed but is not in the cache either because it was never needed before, or was evicted from the cache. Cache prefetchers try to reduce cache misses by predicting data that is likely to be needed in the near future and proactively fetching it into the cache [201]. Unfortunately, the prediction may be inaccurate, leading to cache pollution, or not timely, leading to reduced or negative benefits [17, 201, 209].

Our governors use the BIPS (Billion Instructions Per Second) throughput metric to determine current application or to set performance targets. We assume the existence of user-supplied software routines that convert between BIPS and high-level performance metrics. Similar to existing governors in Linux, our new governors do not keep track of higher-level application constructs such transactions, queries, etc. The workloads that we consider for our experiments do not have any latency constraints. Other workloads that have latency constraints on high-level constructs should estimate their BIPS requirement and communicate that to the governors.

Our new governor for SLAee significantly improves BIPS-per-watt (energy efficiency), with a maximum improvement of 67% and an average (geometric mean) improvement of 30% (Section 3.8) compared to the performance-per-watt of the highest-frequency configuration (EP energy efficiency). Improving performance-per-watt is important as it translates into more work being done for the same energy cost or less energy being used for the same amount of work.

Our new governor for SLApower improves over traditional RAPL-based governors by controlling prefetch enable and governing for full system power (Section 3.9).

The SPECpower benchmark calculates a figure of merit called "overall ssj_ops/watt" ($= \frac{\text{avg. load}}{\text{avg. power}}$). This is the overall energy efficiency over all the load levels. With only static frequency selection (single frequency for the entire run), this figure of merit is maximized at 3 GHz achieving 16% higher figure of merit compared to that for the configuration with the maximum frequency. However, our new governor for SLAperf (Section 3.10) results in 31% higher figure of merit by carefully controlling frequency and prefetch settings dynamically. Note that the power meter that we use is not accepted by SPEC as valid for submitting reports and the figure of merit calculations mentioned above may be approximate.

The main contributions of this chapter are:

1. We develop a new OS governor that seeks Pareto-optimality for socket frequency (and associated voltage) scaling and hardware cache prefetching. To the best of our knowledge, this is the first work to develop governors that simultaneously control for these two knobs on a real system. OS governors currently do not seek to control for hardware prefetching.

2. We propose a two-level design for constructing SLA-aware governors. The first level predicts the Pareto frontier (Dynamic EO) and the second level chooses a state on the frontier that targets the desired SLA. This makes governors easily retargetable to different SLAs.

Section 3.3 describes existing governors in Linux. Section 3.6 develops a new governor for maximizing energy efficiency using frequency (and associated voltage) control. Section 3.7 extends this to include cache prefetch control. Section 3.8 discusses how to control for wall (full system) power in addition to socket and memory power. Sections 3.9 and 3.10 develop governors for maximizing performance under a power cap and minimizing power for a performance target.

3.2 Infrastructure

We use the Intel Haswell server, described in Section 2.2 of Chapter 2 and henceforth referred to as HS, for our experimental evaluations in this chapter. The OS acpi-cpufreq interface allows controlling frequency in 15 steps from 0.8–3.5 GHz (0.8–2.0 GHz and 2.1 GHz–3.5 GHz in steps of 200 MHz) and enabling/disabling the turbo boost region (3.5^{+} GHz). Although writing MSRs (Model Specific Registers) directly provides greater control, we use this interface for a fair comparison with the existing governors. (We make an exception to this rule in Section 3.10.3 where we consider all frequencies for our new reactive governor that seeks to minimize idle time.)

We measure socket power and DRAM power using an additional software thread that reads available RAPL (Runtime Average Power Limit) counters [52, 113] at 1 second intervals. This runs as a thread separate from application threads and any governor threads. The governors that we develop also read RAPL counters for power calculations. We measure wall power with a Watts Up? (.net) meter [107] at 1 second intervals. This also runs as a separate additional thread for experiments where we use wall power.

We consider 14 workloads from SPECOMP2012 [206], graph500 [88], hpcg [66, 188], and SPECpower [205]. Of these, graph500 and SPECpower are run to completion whereas the other workloads are run for the first 1200 seconds of their executions (a few runs of kdtree complete within this time at high frequencies).

3.3 Governors in Linux

The Linux acpi-cpufreq module includes the following governors [37] that control the operating frequency. The goal is to manage power-performance by either setting the frequency statically, or by varying it in response to processor utilization. The governors available in our system are shown below. The root user can dynamically change the governor.

- **PowerSave** (**S**): Sets all cores to the lowest frequency. The idea is to use the least amount of power to do the work, but performance may be less than what could be achieved on this machine.
- **OnDemand** (**O**): Periodically samples (default: 10 ms interval) cores to adjust frequencies based on core utilization. The idea is to reduce power by lowering frequency when the CPU is not fully utilized and increase frequency as utilization increases so that the performance impact is minimal. The **Conservative** (**C**) governor is a variant of the OnDemand governor with more conservative utilization thresholds for changing frequencies.
- **UserSpace** (**U**): The idea is to give the root user control of the frequency settings. On HS, the root user can set the socket frequency (all cores together) to any of the allowed frequencies. The interface does not allow per-core settings for HS. All cores transition to the highest frequency of any core in the socket. This mode is useful only if the workload is known well in advance so that it can be run with different frequencies to determine the best setting. This is not practical in most deployments but is useful in reasoning about improvement opportunities.
- **Performance** (**P**): Sets all cores to the highest frequency. The idea is to get the maximum performance. This governor also uses the maximum power.

To further distinguish between modes, we constrain **U** mode to exclude **S** or **P** mode frequencies, i.e., it operates in the range of 1.0–3.5 GHz.

While these governors attempt to control knobs (e.g., processor frequency) in the system, none of them seek to meet SLAs that deal with energy consumption, power limits, or performance targets.

3.4 Two-level governor design

There are two challenges in developing governors that seek to optimize for SLAs—dealing with multiple hardware reconfiguration knobs (DVFS, prefetching, and possibly more to be made available in future) and dealing with different SLAs. To simplify governor design and make them retargetable to different deployment scenarios, we propose the following two-level governor design:

1. **Pareto Predictor**: This predicts the power-performance Pareto frontier for the system and currently observed execution profile.

2. **Objective Selector**: This level selects the desired operating state from the Pareto frontier according to the SLA to be achieved.

The objective selector remains unchanged if the available knobs change and the Pareto predictor remains unchanged if new SLAs are targeted. We believe that this simplifies governor construction and portability.

The above separation is possible because of a few properties of the Pareto-optimal frontier:

- Configurations that optimize power-performance metrics lie on the Pareto frontier (Section 2.5, Property 3). This makes it sufficient to focus only on the frontier to meet SLAs.

- Power and performance of states on the Pareto frontier have the same monotonic ordering relation (Section 2.5, Property 2). This makes predicting effects of system configurations easier, e.g., reducing a power cap will reduce performance.

Section 3.6 describes the basic sampling and interpolation schemes used by the Pareto predictor. This is slightly extended in Section 3.7 to include an additional reconfiguration knob (cache prefetching). The objective selector for SLAee computes performance-per-watt for each predicted point on the frontier and selects the next system state to be the one that is expected to maximize performance-per-watt. Objective selector mechanisms for SLApower and SLAperf are described in Sections 3.9 and 3.10 respectively.

Modern systems [181] often include a centralized power-control unit (PCU) that collects telemetry information from functional blocks and performs control actions. Our proposed Pareto predictor can be colocated with or implemented by such a PCU to reduce runtime overheads.

Our new governors do not control frequency in the turbo boost region (3.5+ GHz) except for the SLAperf governor for SPECpower (Section 3.10). This is because we cannot control frequency exactly in that region, but a vendor implemented version of our work would not have this difficulty. It is possible to control performance indirectly by limiting power in this region at a fine granularity (e.g., 0.125W [141]) through the RAPL capability (also see Section 3.9). However, as we shall show, energy-efficient operations are usually at much lower frequencies. Hence we do not include the additional complexity of finely controlling the turbo region in our governors.

3.5 Deployment Scenarios

One deployment scenario makes the simplifying assumption that the system can be completely shutdown (zero power draw) when there is no work and (near-)instantly fully enabled when work arrives. As an example, Figure 3.2 shows the performance in Billion Instructions-Per-Second (BIPS) and power consumption in Watts (W) of graph500 [88] for different socket frequency settings on HS in this deployment scenario.

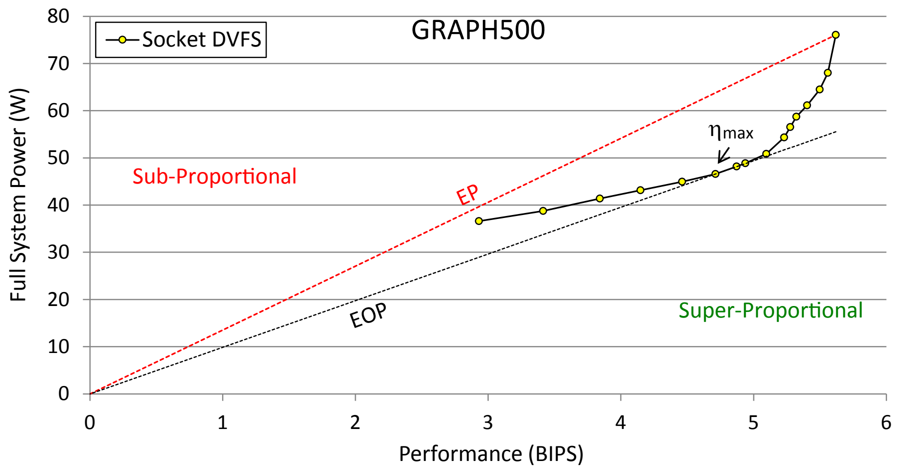


Figure 3.2: Example power-performance profile with Wall Power.

The SPECpower workload also measures the wall power of the system as the power cost to complete work. Additionally, SPECpower includes power measurements when the system is under zero load, that is, no work done.

In practice, this deployment scenario may not be realizable as there is usually a non-trivial latency cost while booting the system. An alternate deployment scenario assumes that the system transitions to active idle (non-zero idle power) when there is no work and is (near-)instantly fully enabled when work arrives. This minimize delays due to system wakeup when a workload is dispatched. In this scenario we are interested

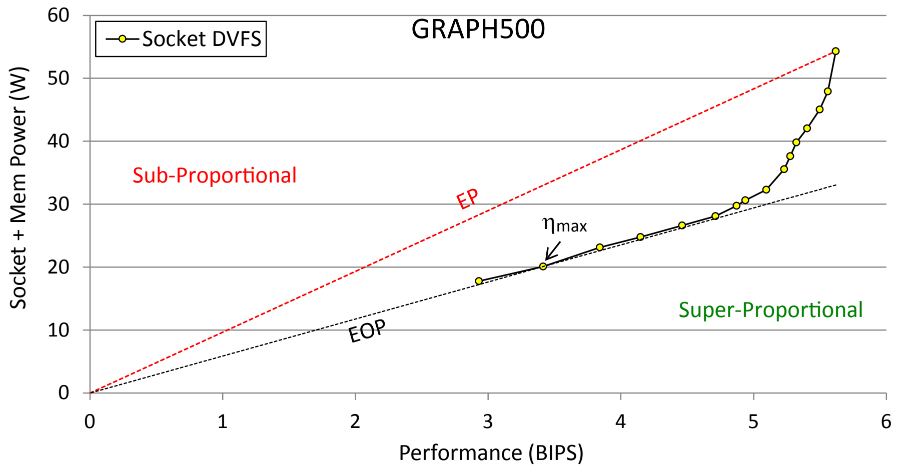


Figure 3.3: Example power-performance profile with Socket + Mem Power.

in the extra cost, measured by the socket and memory RAPL (Running Average Power Limit [113]) energy counters, to execute the workload. We will start with this deployment scenario for developing our governors. Figure 3.3 shows the power-performance profile for graph500 in this scenario.

For both scenarios, the EP line connects the origin (no extra power used for no work done) to the highest performing point with the default Peak Performance Configuration (all cores at their highest frequencies, prefetching enabled). While performance remains the same in both scenarios, the power consumption accounted for varies. Section 3.8 shows how to convert between the two power models. The states having the maximum efficiency (η_{max}) are higher-performing states in the first scenario, than in the second scenario, to compensate for idle power.

EP delivers performance in proportion to the extra power used. The EP line divides the power-performance landscape into two regions—“Sub-Proportional”, where the performance is less than proportional to the extra power (equivalently, lower energy efficiency than that of EP), and “Super-Proportional”, where the performance is more than proportional to the extra power (equivalently, higher energy efficiency than that of EP). Our governors aim to operate the system at Dynamic EO (power-performance Pareto frontier) that automatically accounts for super-proportional behavior, if it manifests.

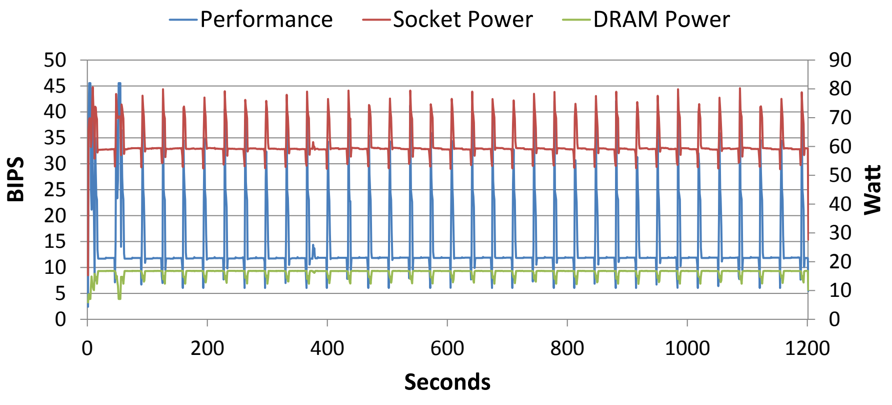
3.6 SLAee: Maximize energy efficiency

Our first goal is to develop a new governor that *seeks Pareto-optimal operation* and by doing so improves energy efficiency, measured as performance-per-watt (BIPS/Watt). For this section, we consider system power as the sum of socket power and DRAM power, both of which are estimated using RAPL counters. This corresponds to the second scenario (Figure 3.3) discussed in Section 3.5.

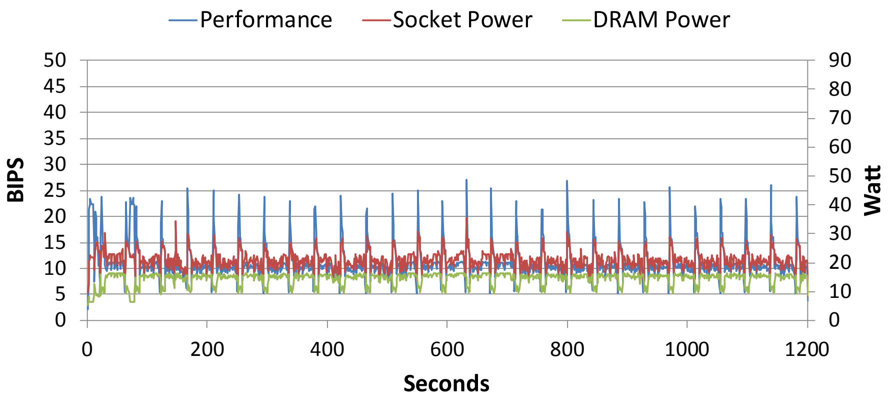
Figures 3.4a and 3.5a show example power-performance traces for applu (SPECOMP2012) and graph500 in **P** mode. Figures 3.4b and 3.5b show energy-efficient (but lower-performing) executions using our new governor that we will describe shortly. Lower-performing executions that save energy may be desirable in situations with relaxed performance constraints, e.g., in batch executions, and when energy costs are important to the user.

Workload applu performs several iterations, each with a memory-intensive portion followed by a compute-intensive portion; the performance and power spikes indicate iteration boundaries. The DRAM power drops during the compute-intensive part of each iteration due to less memory accesses. Workload graph500 runs 64 iterations of breadth-first search after initialization. Both applu and graph500 exhibit long-term periodic behavior in both performance and power readings, with periods of tens of seconds corresponding to iteration lengths. Long-term stability in average power-performance profiles reduces differences between fixed-time and fixed-work experiments.

HS exhibits significant opportunities in improving BIPS /Watt (equivalently, Instruc-



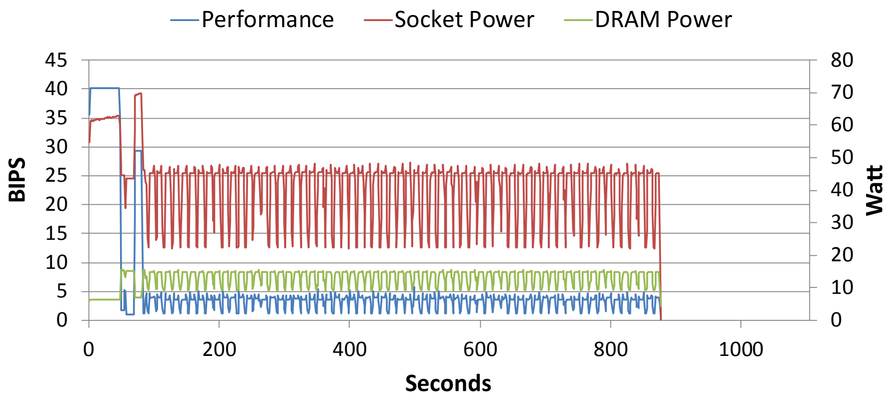
(a) applu. Instructions/nJ = 0.18.



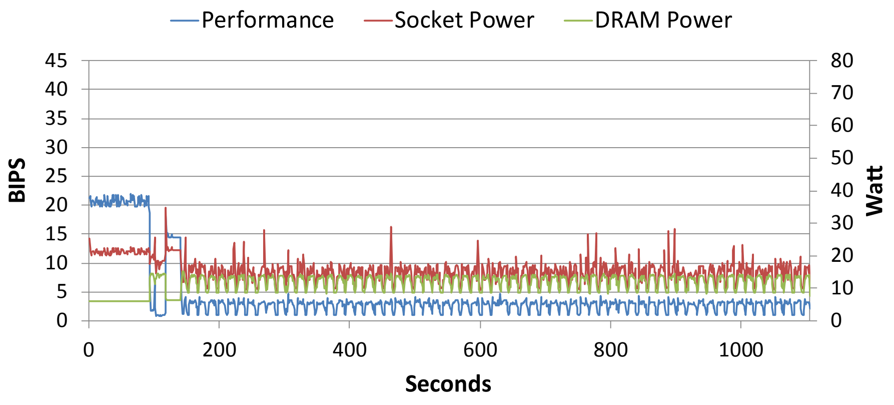
(b) applu. Instructions/nJ = 0.32.

Figure 3.4: Power-Performance traces for applu in **P**-mode and **R(10)**-mode on HS. Higher Instructions/nanoJoule implies more energy efficiency.

tions/nanoJoule) by changing frequency settings alone. BIPS changes between 1.18x (swim) to 4.86x (bwaves) in going from **S** to **P** modes whereas power changes between 2.52x (swim) to 5.67x (botsalgn), leading to a BIPS/Watt range of 1.29x (imagick) to 2.14x (swim) between best and worst values for that workload over all frequencies. For all these workloads, the minimum BIPS/Watt happens for **P** mode. applu and graph500 show a BIPS/Watt range of 1.84x and 1.67x respectively (also see Figure 3.6).



(a) graph500. Instructions/nJ = 0.10.



(b) graph500. Instructions/nJ = 0.16.

Figure 3.5: Power-Performance traces for graph500 in **P**-mode and **R(10)**-mode on HS. Higher Instructions/nanoJoule implies more energy efficiency.

We implement a simple reactive, **R(t)**, mode of operation to exploit the improvement potential. Our approach is to sample power and performance at a few different frequencies, then use that information to interpolate the frontier. Referring to Figure 3.2 as an example, we see that at least three samples are needed to target super-proportionality. In contrast, aiming for proportionality would require only two points, but the non-linearity in system behavior between the points could not be predicted or controlled.

We implement two power-performance predictors (in software)—one for the socket subsystem and the other for the memory subsystem. The socket predictor sets the frequency to 0.8 GHz (lowest frequency), 2.1 GHz (midpoint frequency) and 3.5 GHz (nominal frequency) in three consecutive intervals of t ms each and observes the power, performance, memory read and write bandwidths for each setting. It then interpolates (quadratic or piecewise linear) the effects for the other frequencies.

A software coordination module, running on one of the cores, reads the socket predictions and DRAM predictions every $51\mathrm{t}$ ms (immediately after the $3\mathrm{t}$ socket sampling), composes the predictions, estimates the frontier and selects the best frequency. The length of the interval that the system runs in this state is $48\mathrm{t}$. It is during this time that the DRAM predictor is periodically invoked (every $12\mathrm{t}$ ms) to adjust a computed linear regression between DRAM power and read and write bandwidths (two variables) based on current readings. The regression is reset every 17 observations ($204\mathrm{t}$ ms) to react faster to phase changes. We choose this value since 17 is not divisible by 4 ($48\mathrm{t}/12\mathrm{t} = 4$), so the regression will not be reset at the time when the readings are needed for the power estimations with the interpolated values.

Since the optimal frequency will be in of the high/mid/low ranges, the sampling overhead is approximately $\frac{2\mathrm{t}}{51\mathrm{t}}$~4%. The workload continues to execute, although sub-optimally, in those two sampling intervals, so the temporal overhead is usually $< 4\%$. Figures 3.4b and 3.5b show power-performance traces for applu (SPECOMP2012) and graph500, in **R(10)** and the improvement in BIPS/Watt.

For all of the timing intervals mentioned above, we do not account for additional governor overheads due to system calls, interpolation, estimations, etc. So, the actual intervals will be slightly longer. The governor in Section 3.10.3 accounts for these overheads.

There are three main issues in implementing the interpolant for the socket predictor:

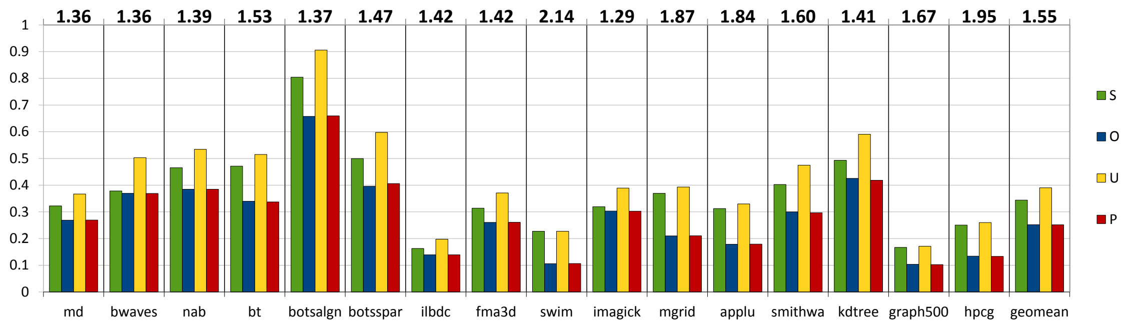
1. Getting successive sample points that show non-decreasing performance and power with increasing frequency.
2. Getting sample points with acceptable measurement noise/jitter.
3. Dealing with non-convexity of the frontier.

The first issue arises when the workload exhibits local phase behavior. The second issue arises with rapid sampling that makes the jitter in the energy measurements seem to be higher than that in the timing measurements leading to occasionally unrealistic power calculations. The third issue arises when the number of samples is not enough to correctly estimate the shape of the frontier.

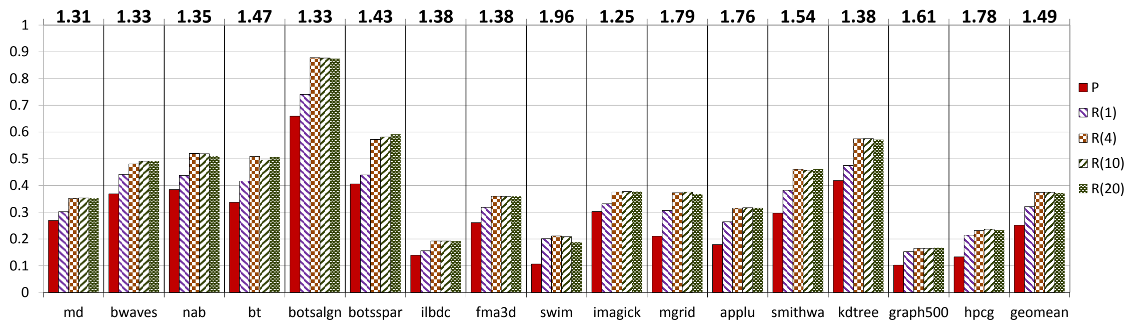
To deal with the first two issues, we disregard samples if either decreasing values are found or if power readings differ in more than 10x between the three samples and the coordinator transitions to 3.5 GHz. While other default actions are possible, we choose to penalize ourselves when we are not confident about the interpolation. On average (geometric mean) less than 2% of samples are discarded, but the frequency can occasionally be high, e.g., ~10% for bwaves in **R(1)**. We do not correct for the third issue and our results will be suboptimal for non-convex frontiers.

Figure 3.6 compares the energy efficiency (performance-per-watt) with different modes of operation. For all these workloads, the **P** mode has the lowest efficiency. The numbers at the top show maximum gains (e.g., 1.36 implies 36% gains) in energy efficiency over **P**-mode by selecting the optimal frequency in **U** or **S** modes (Figure 3.6a) and **R(10)** mode (Figure 3.6b). We observe that:

- *The potential rewards for selecting optimal configurations are significant*: The efficiency improvements were 28.6% (imagick) to 113.7% (swim) over **P** (geometric mean: 55%



(a) The Top line shows Performance-per-Watt of the best **U** mode frequency relative to **P** mode



(b) The Top line shows Performance-per-Watt of **R(10)** relative to **P** mode

Figure 3.6: BIPS-per-Watt on HS with different policies.

over **P**, 13.4% over **S**). The **O** and **P** modes are suboptimal for this metric for every workload.

However, the improvements come at a performance cost. Compared to **P** mode, the most energy-efficient frequency setting for each workload resulted in a reduction of BIPS of around 12.8% (mgrid) to 45.3% (botsspar), with a geometric mean of 35.4%. So, there is a tradeoff between energy savings and performance loss. For batch executions the performance loss may be tolerable. For other executions, Section 3.10 discusses a governor that try to reduce energy while meeting a given performance constraint.

- *There is no single best static frequency setting*: The best static frequency settings for the different workloads were 0.8 GHz (swim), 1.0 GHz (applu, graph500, hpcg), 1.4 GHz (mgrid), 1.8 GHz (bt, botsspar), 2.0 GHz (ilbdc, smithwa, kdtree), 2.1 GHz (md, nab, botsalgn, fma3d), 2.3 GHz (bwaves, imagick).

- *Rapid profiling and reconfigurations are not necessary for long running workloads*: We did a sensitivity analysis with **t**=1 msec, 4 msec, 10 msec, and 20 msec. The resulting performance-per-watt numbers indicate that **R(20)** (geometric mean: ~48% over **P**, 8.3% over **S**), **R(10)** (geometric mean: ~49% over **P**, ~9% over **S**), and **R(4)** (geometric mean: 48.7% over **P**, 8.8% over **S**) improved over **R(1)** (geometric mean: 27.5% over **P**, -6.7% over **S**). For the rest of our discussion on this governor we will focus only on t=10 msec, that is, **R(10)**.

We observe that many workloads exhibit long-term variation and periodic behavior. For example, applu shows ~34.3 sec periodicity in **P** mode (Figure 3.4a) and ~42.3 sec with **R(10)** (Figure 3.4b). graph500 exhibits ~12.3 sec periodicity in **P**-mode (Figure 3.5a) and ~15 sec with **R(10)** (Figure 3.5b). We expect long training intervals that track considerable execution history to work well with such workloads.

The socket predictor could use a variety of interpolants, e.g., piecewise linear or quadratic, to predict power and performance for different frequencies from the profiled data. The choice of the interpolant trades off accuracy with computation cost.

We use quadratic interpolation for the socket predictor. A piecewise linear interpolation would be faster, but for the performance-per-watt metric, only one of the sample frequencies (0.8/2.1/3.5 GHz) would get chosen as the optimal frequency. This is because $\mathrm{Perf}(f) = af + b,\ \mathrm{Pwr}(f) = cf + d \implies \mathrm{Perf}(f)/\mathrm{Pwr}(f)$ is monotonic in f. So, the maxima will always occur among the end points of the interval. This is a generic result and is not limited to using frequency as the independent variable.

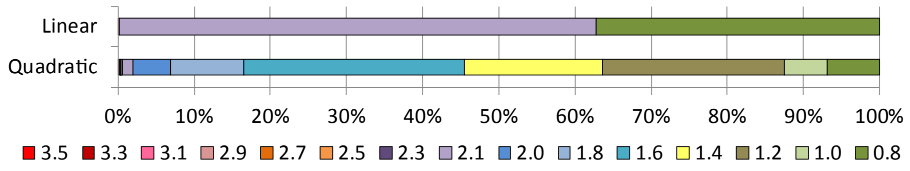


Figure 3.7: R(10) freq. distribution for applu (0.8–3.5 GHz).

To overcome this, we also evaluated quadratic interpolation for an alternative socket impact predictor without needing to change the coordinator. Figure 3.7 shows the frequency distribution for both schemes for applu. While Linear fluctuates mostly between 0.8 and 2.1 GHz, resulting in ~66% improvement over **P**-mode, Quadratic selects more frequencies in between resulting in 76.4% improvement. mgrid showed similar improvements.

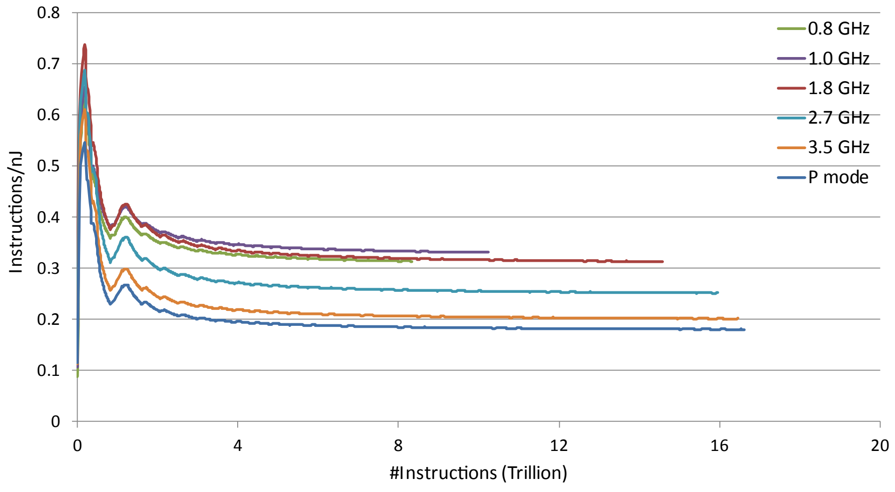


Figure 3.8: Average energy efficiency of applu as a function of the number of instructions executed and processor frequency.

For our evaluations, we run SPECOMP workloads for the first 1200 seconds. Only a few runs (when run at high frequencies) of kdtree finish within this time while other runs and all runs of other workloads do not finish. Doing fixed-time experiments, as opposed to fixed-work experiments, runs the risk of comparing efficiency numbers from incomparable runs. However, for these workloads, the differences are small. For example, Figure 3.8 shows that, for any operating frequency, the average energy efficiency (over the total instructions executed so far) of applu stabilizes after a sufficiently large number of instructions have executed. So comparing across runs having different, but large, number of instructions is possible.

We re-evaluated results using the same number of instructions (minimum across all policies from the fixed-time runs) for each workload. In terms of **U**-mode gains over **P**-mode, applu changed from 84% to 80% whereas botsspar changed from 47% to 52%. The geometric mean changed < 2% for all policies. All trends remained the same.

3.7 SLAee: Adding L2 Prefetch Control

Hardware prefetching on Intel x86 machines can be enabled or disabled by writing specific values to Model-Specific Registers (MSRs) [112]. All prefetchers are enabled by default. In this study we keep the DCU (L1 Data Cache) prefetchers enabled, but dynamically enable or disable the L2 prefetchers. We set the prefetching mode for all cores identically.

Figure 3.9 shows one example workload each for beneficial prefetch (mgrid) and harmful prefetch (md) with all possible socket DVFS settings. When prefetching is disabled, md shows 14% improvement in peak performance and 13.6% in maximum energy efficiency whereas mgrid shows 12.3% loss in peak performance and 14.9% loss in maximum energy efficiency. Since prefetching benefits are workload dependent, a static prefetch setting will always be suboptimal for some workloads.

We extend the frequency governor in the following simple way: Instead of taking one sample at 2.1 GHz, we take two samples—once with prefetching enabled and once

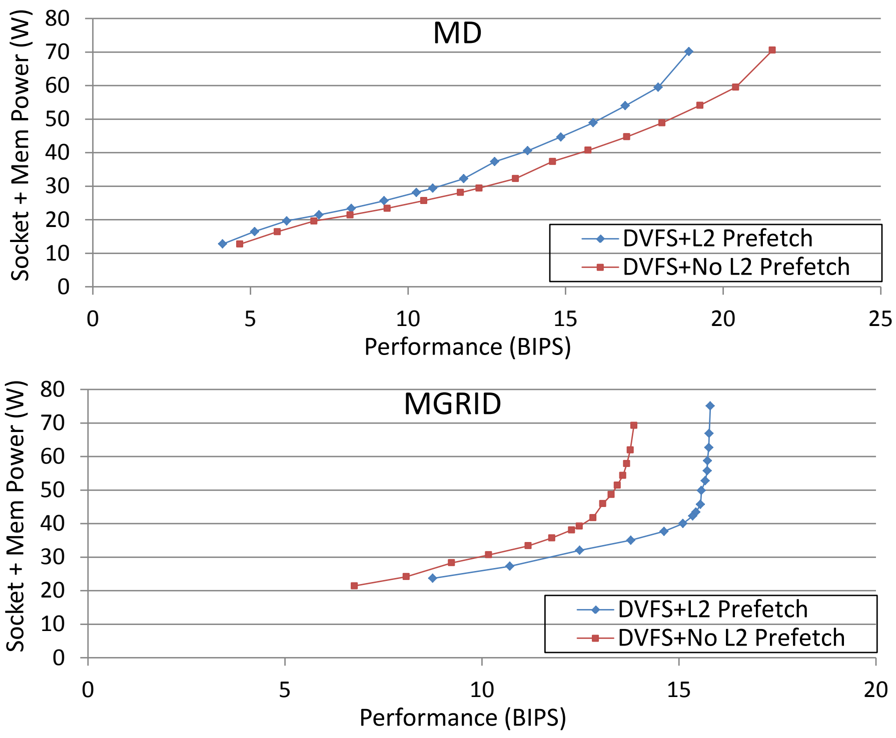


Figure 3.9: Example profiles. States at the Pareto frontier have L2 prefetching disabled for md, but enabled for mgrid.

disabled. We choose the prefetching mode that gives better performance and continue with that for the remaining two samples and estimating the frontier for that interval. Our choice of 2.1 GHz for taking the initial two samples is motivated by the need to keep the overhead of taking an extra sample small. A mid-range frequency, such as 2.1 GHz, is likely to incur a lower additional overhead for this than a high frequency, such as 3.5 GHz since the energy-efficient operations for most workloads are not at high frequencies.

Similar to **R(t)**, we name the new governor **RF(t)** (Reactive with prefetch control), parametrized by **t**, the length of the profiling interval in milliseconds. Figure 3.10 shows the distribution of prefetch modes (enabled/disabled) selected by **RF(10)** for our workloads. As expected, md ran with prefetching mostly disabled whereas mgrid ran

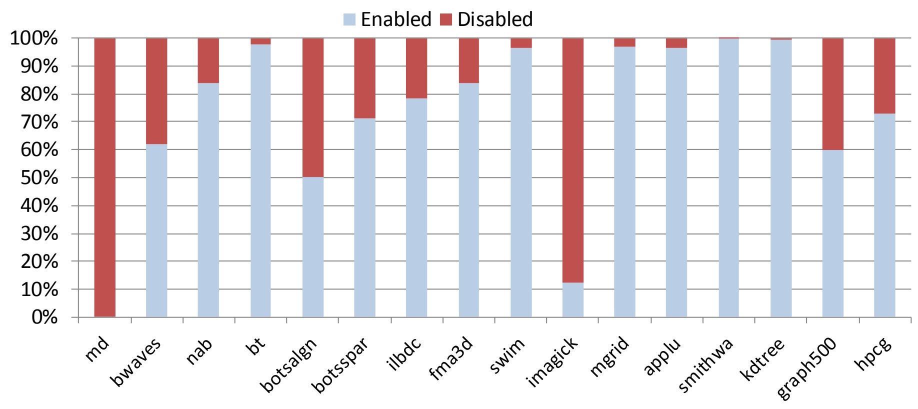


Figure 3.10: L2 Prefetch mode distribution by **RF**(10).

with prefetching mostly enabled. Apart from mgrid, workloads bt, swim, applu, smithwa and kdtree also predominantly chose to keep prefetching enabled. Other workloads chose between both enabled and disabled modes.

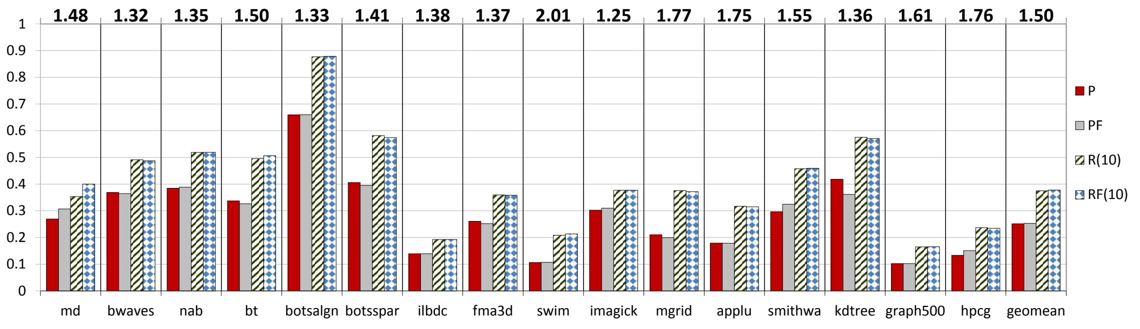


Figure 3.11: BIPS-per-watt of governors with (**RF**(10)) and without (**P**, **PF**, **R**(10)) dynamic control for L2 Prefetching.

Figure 3.11 shows performance-per-watt for four governors—**P**, **R(10)** (prefetching always enabled), **PF** (prefetching always disabled) and **RF(10)** (prefetching dynamically controlled). The secondary horizontal axis (top) shows the improvement (e.g., 1.48 implies 48% improvement) in performance-per-watt of **RF(10)** compared to **P**. **RF(10)** improved performance-per-watt beyond **R(10)** for md (48% instead of 31%) but did not create significant differences for other workloads.

To summarize our results so far, we find that the **P**, **PF** and **O** modes can always be improved by the other policies. **S** works best for swim, well for hpcg, but can be improved by **U**, **R** and **RF** for the other workloads. **U** is a good policy to use provided that workload is known in advance and profiling experiments can be carried out. **R** reaches close to **U** but is unable to outperform it. This is likely because these workloads have long-term stable behavior (see for example, Figure 3.8) making the best static frequency not a bad choice. On the other hand, **R** suffers from runtime profiling overheads and prediction errors due to sampling inconsistency and non-convexity of the frontier. The situation changes for SPECpower (see Sections 3.10.2 and 3.10.3), where reactive governors do significantly better than static frequency settings. **RF** further improves upon **R** if disabling prefetch is useful but does not hurt energy efficiency if not, so it represents the best of both prefetch modes.

3.8 SLAee: Adding Control for Wall Power

For the experiments in Sections 3.6 and 3.7, we consider system power as the sum of processor and memory power as estimated by the RAPL counters. We do not consider the power consumptions for other components (e.g., power supply, network interfaces, hard disks, etc.) that account for 20-30W power, with system idle power of HS at ~26W. HS has a 350W, 80 Plus Gold PSU (Power Supply Unit) [218, 230]. We will now correct for the extra system power considering a deployment scenario where the cost to execute a workload includes the wall (full-system) power.

Measuring wall power requires external power meters (e.g., Watts Up? meters) and are usually available only at long measurement granularities (e.g., minimum 1 sec intervals) as opposed to the millisecond granularity of RAPL counter measurements used by our governors (**R(10)** requires a measurement interval granularity of 10 msec).

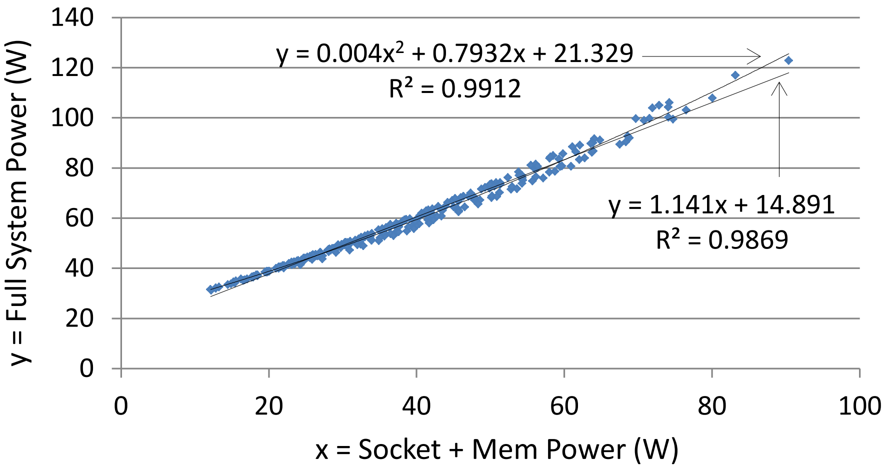


Figure 3.12: RAPL and Wall Power correlation.

To correct for the extra power we need to model the relation between the socket+mem power (measured by RAPL counters) and wall power (measured by a Watts Up? meter). Figure 3.12 shows the two values for all frequency settings for our workloads. We then fit a linear model and a quadratic model to the data. The slope of the straight line is consistent with the power efficiency of 87% (1.141^{-1}~0.876) of this PSU at <50% load [218, 230]. However, the y-intercept suggests a system idle power of ~20W (@x=~4.6W, socket+mem idle power) instead of the ~26W actually observed. Conversely, the quadratic model gives a better fit for idle power as well as a slightly better fit overall. One reason for this could be that load-dependent variations in the power efficiency of the PSU give rise to some non-linearity. In that case the SPUE (Server PUE) metric [104] that tracks power supply overheads may be more accurately characterized by a formulation that includes both load-dependent and load-independent factors than by a single number, e.g., 1/(rated power efficiency of the PSU). We use the quadratic model to estimate wall power from socket+mem power.

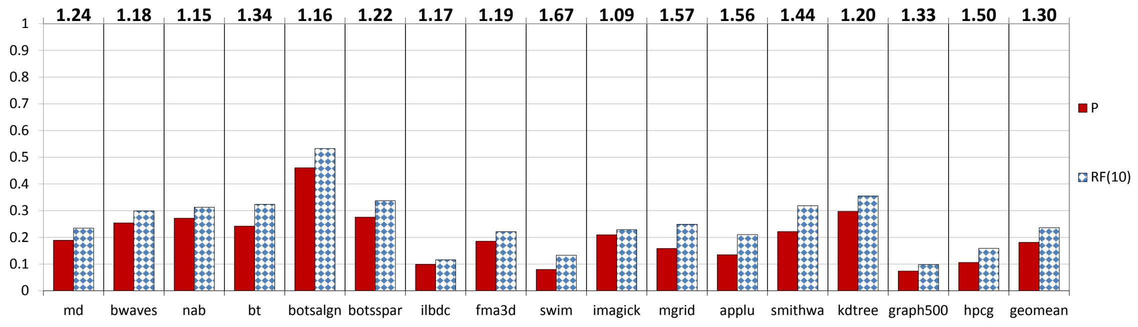


Figure 3.13: BIPS-per-watt of governors **P** and **RF**(10) with wall (full-system) power.

Figure 3.13 shows the performance-per-watt for the baseline **P** governor and our new **RF** governor, with wall power for the run measured using a Watts Up? meter. During the run, **RF** uses RAPL counters for profiling, then applies the wall power model mentioned above to the interpolated power numbers.

The maximum gains in performance-per-watt achieved by **RF** over **P** is 67% (swim) while the geometric mean over all workloads is 30.2%. Although the improvements are somewhat smaller than those in Figure 3.11 due to consideration of the extra system power for both the governors, this is significant improvement in energy efficiency realized on a real server machine. Since CPUE is directly proportional to energy consumption and inversely proportional to energy efficiency, we have $\frac{\mathrm{CPUE}(\mathbf{P})}{\mathrm{CPUE}(\mathbf{RF})} = \frac{\mathrm{E}(\mathbf{P})}{\mathrm{E}(\mathbf{RF})} = \frac{\eta(\mathbf{RF})}{\eta(\mathbf{P})} = 1.302$. So the average energy savings of **RF** over **P** is $\frac{\mathrm{E}(\mathbf{P})-\mathrm{E}(\mathbf{RF})}{\mathrm{E}(\mathbf{P})} = 1 - \frac{\mathrm{E}(\mathbf{RF})}{\mathrm{E}(\mathbf{P})} = 1 - 1.302^{-1} = 23.2\%$.

Energy savings reduce operational expenses that in turn reduces TCO for datacenters. **RF** thus opens up opportunities either for cost savings by using ~23% less energy on average to do the same work or for revenue generation by doing ~30% more work on average for the same energy cost.

While it improves energy-efficiency, **RF** loses performance, with respect to **P** mode, ranging from 0.2% (md) to 30% (bt) with a geometric mean of 19.5%.

3.9 SLApower: Maximize performance within a power cap/budget

None of the standard Linux governors **S**, **C**, **O**, **U**, **P** deal with power caps/limits. There is no way for the user to specify power caps to these governors.

The RAPL [113] capabilities include mechanisms to enforce a limit on the power consumption. One advantage of RAPL limits over frequency settings is that they can be fine-grained (e.g., units of 1/8 W) leading to greater control of the state space. Another advantage is that since RAPL limits are enforced by the hardware, the management overhead is lower than that of a software-controlled governor. Prior works [141, 213] have used power limiting as a mechanism to improve energy efficiency.

There are two main disadvantages of the RAPL power-capping mechanisms:

1. Capping of wall power cannot be directly specified. One needs to use a model, similar to the one that we developed in Section 3.8, to convert wall power limits to RAPL domain power limits.

2. Management of non-frequency resources does not automatically happen through the RAPL mechanisms. For example, setting or clearing RAPL limits does not affect prefetching status (enabled/disabled). So, workloads such as md that benefit significantly from prefetch control would not see those advantages with the RAPL approach. From this perspective, RAPL guarantees a power cap, *not best performance within that power cap*. Pareto optimality provides the stronger guarantee.

The first limitation can be easily overcome, but the second limitation is more profound and needs more effort to address. This limitation is likely to be accentuated with the addition of more reconfigurable knobs in future systems. Our new governor for SLApower improves upon the RAPL governor in this aspect. We demonstrate it by controlling prefetch settings as well as socket frequency to get the maximum performance within a specified power budget.

For the RAPL experiments in this section, we limit average socket power over 1 second intervals from 10W to 80W in steps of 5W. For our system, we could enforce a power cap only for the entire socket, not for the memory or for other components.

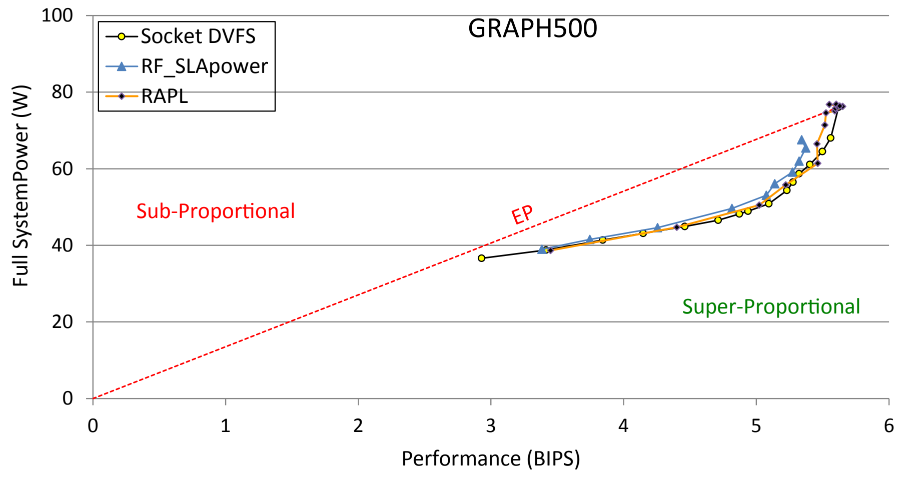
We develop a new governor (by modifying the objective selector) to select the next state that is predicted to use the highest power among all states with less power consumption than the SLA target. We name this governor **RF_SLApower(t)**. There are three differences between this governor and the RAPL governor:

1. We specify limits on full system power or socket power as needed. The Pareto predictor uses the model shown in Figure 3.12 to estimate wall power from socket and memory RAPL energy counter measurements. We specify power limits in steps of 5W from 35W to 80W for graph500 and 105W for md. In contrast, for the RAPL governor we capped socket power and measured the resulting system power. Since the RAPL interface does not allow specification of full-system power caps, the two power limits do not have an exact correspondence.

2. We impose a power limit on every reconfiguration interval, which is 510 msec long for **RF_SLApower(10)**. So it is somewhat more strict than the sliding window of 1 second that we used for the RAPL experiments.

3. We do not set turbo mode frequencies. This is because our Pareto predictor has limited control over frequency selection and hence limited insight on power consumption in turbo mode. So, the maximum frequency that we set is the nominal

frequency (3.5 GHz). However, this limits the maximum performance that can be attained.



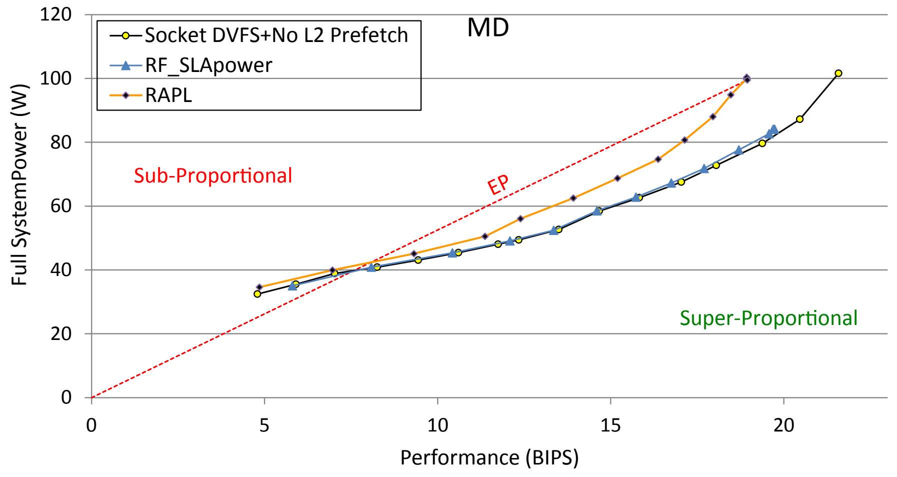


Figure 3.14: Power-performance profiles for graph500 and md for SLApower.

We investigate enforcement of SLApower using **RF_SLApower(10)** and the RAPL governor for two of our workloads: graph500 and md. Figure 3.14 shows the power-performance profiles for both approaches. Both workloads exhibited behavior close to Pareto optimal with **RF_SLApower(10)**. The RAPL governor works well for graph500, but causes Pareto-dominated (hence suboptimal) operations for md as it does not control prefetch settings.

We also observe that **RF_SLApower(10)** falls short of achieving the maximum performance at the upper end of operating range. This is because the maximum allowable frequency in this governor is 3.5 GHz (nominal frequency) since we cannot effectively control the turbo range. One way to get around this limitation could be to use RAPL to control DVFS settings and have **RF_SLApower** control other settings, e.g., prefetch settings. Of course, one needs to to calculate the appropriate RAPL power caps from system power caps using the inverse of the power mapping function shown in Figure 3.12.

3.10 SLAperf: Maximize power savings given a performance target

None of the standard governors **S**, **C**, **O**, **U**, **P** deal with performance targets. There is no way for the user to specify performance targets to these governors. The RAPL capabilities (see Section 3.9) allow power caps to be specified, but not performance targets to reach.

We name our new governor **RF_SLAperf(t)**. It allows the user to specify performance targets in absolute or relative (with respect to peak) BIPS. The governor is agnostic of higher-level performance goals, e.g., transactions per second or response latency distributions. The user needs to have a mapping between such performance goals to one of the performance targets mentioned above. We will now describe the governor designs for these two types of performance targets.

3.10.1 Governing for absolute performance targets

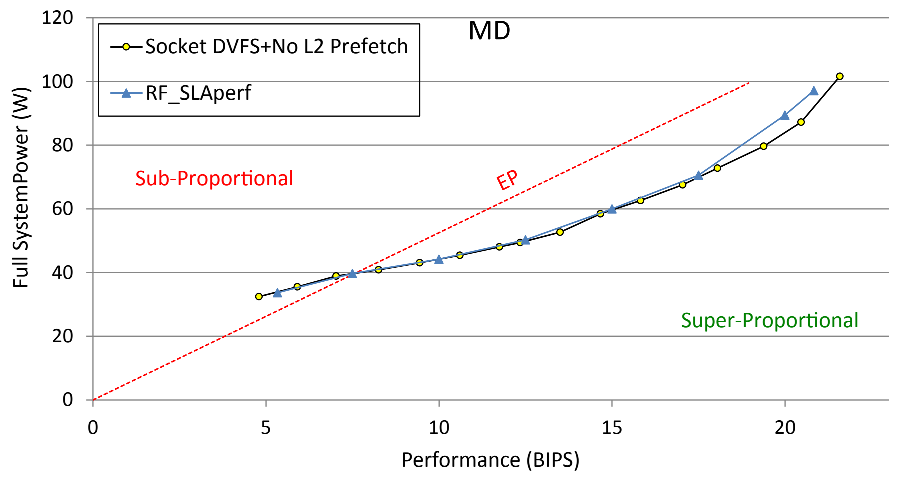
To govern for this SLA, we keep the Pareto predictor intact, but modify the objective selector to additionally keep track of the performance so far (time elapsed and instructions executed). This allows it to set the desired performance target for the next interval so that if the target for the interval is met, then the average performance so far would be that required by the SLA. The objective selector makes one of three possible choices:

1. If the average performance so far is greater than the SLA, the lowest performing point is chosen.
2. Otherwise, if the next interval target is greater than the best performance predicted for 3.5 GHz, turbo mode is chosen.
3. Otherwise, the point on the frontier that meets or just exceeds the next interval target is chosen.

The above is a simple policy for the objective selector. Other policies are possible and they will be useful particularly if they include workload semantics and predictions of future workload behavior since the Pareto predictor lacks both these dimensions.

We investigate enforcement of SLAperf using **RF_SLAperf(10)** for two of our workloads: md and graph500. md has mostly homogeneous behavior during its execution and we will show that it can be governed well to meet the SLA. On the other hand, graph500 has significant heterogeneity (different execution phases) and, as we shall show, cannot be governed well without prior knowledge of the phase behavior.

md has an average performance range of 4.8–21.6 BIPS at the frontier depending on the frequency setting and L2 prefetching disabled. We select SLA targets of 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0 and 22.5 BIPS (unreachable). graph500 has an average performance range of 2.9–5.6 BIPS at the frontier depending on the frequency setting and L2 prefetching enabled. We select SLA targets of 3.0, 3.5, 4.0, 4.5, 5.0 and 5.5 BIPS.



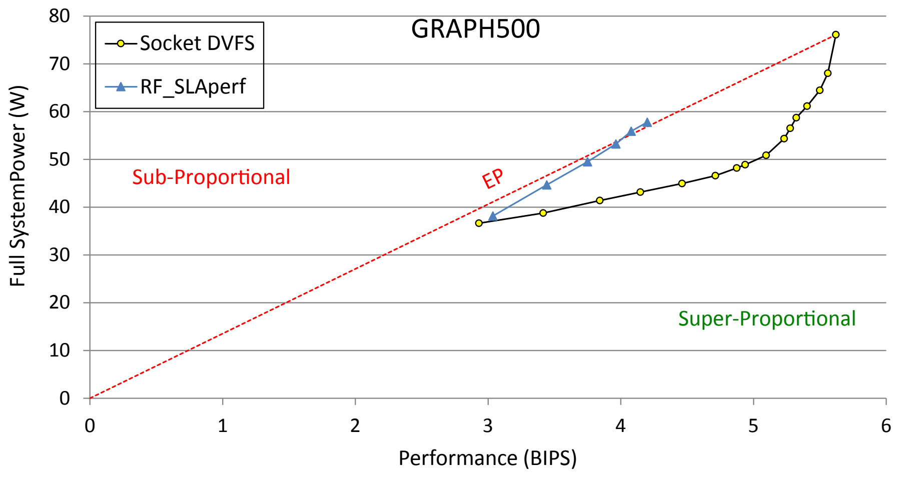


Figure 3.15: Power-performance profiles for md and graph500 for SLAperf.

Figure 3.15 shows the power-performance profiles and expected behavior for both workloads. The profile for md was at the frontier except at high performance targets. Its highest performance is around 3.5% less than the maximum possible. This is due to sampling/profiling and prediction overheads in the governor. Also, when it is given a target close to its highest performance, it tries to compensate for the loss by transitioning more into turbo mode resulting in more power consumption and consequently, Pareto-dominated states.

The governor failed to meet the SLA for most points of graph500 and the profile was quite suboptimal, being closer to proportional than Pareto optimal. However, as we explain below, this is primarily due to the non-homogeneous nature of the workload rather than incorrect state transitions chosen by the governor.

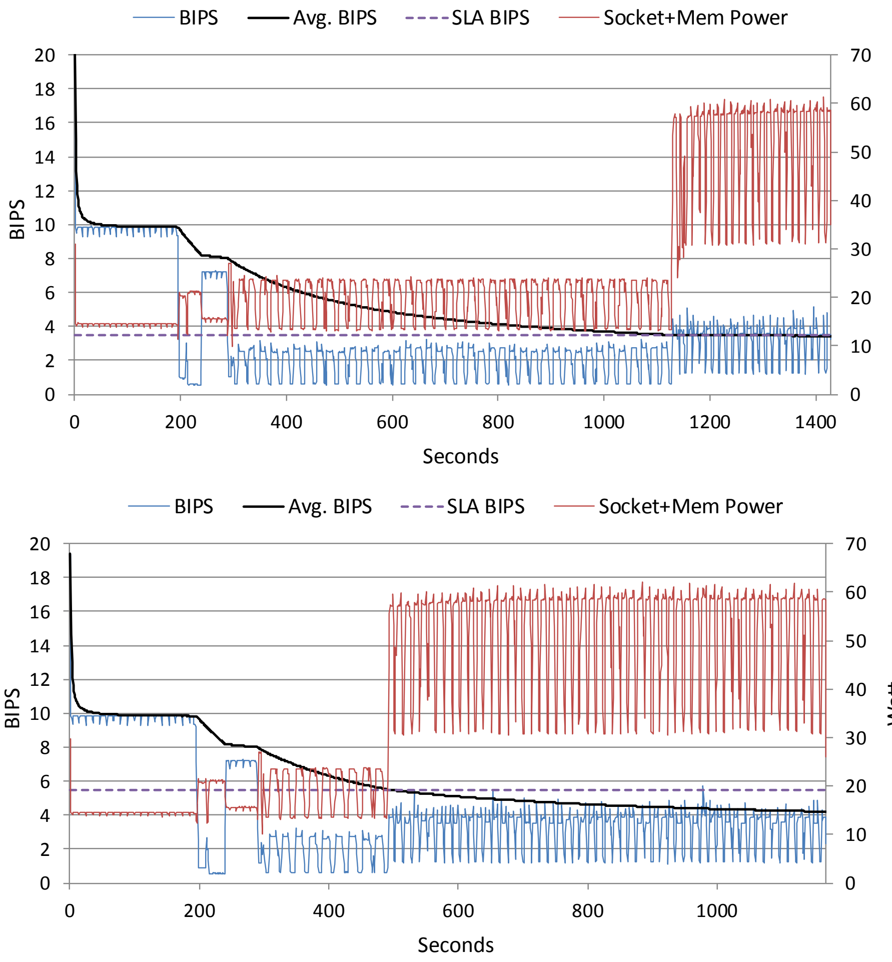


Figure 3.16: Execution profiles for graph500 with SLA = 3.5 BIPS and 5.5 BIPS.

Figure 3.16 shows detailed execution profiles for graph500 for two SLAs: 3.5 and 5.5 BIPS. The primary vertical axis (left axis) and the blue line show the number of instructions executed per second in BIPS. The secondary vertical axis (right axis) and the red line show the power (socket+memory) in Watts. The Avg. BIPS line shows the average BIPS attained by the workload execution so far. The dashed line shows the average BIPS expected to be reached over the entire execution. The governor seeks state transitions that will maintain the average BIPS equal to the SLA BIPS.

graph500 has non-homogeneous behavior—the initial ~290 seconds is mostly a high performance phase where high BIPS is possible whereas the remainder of the execution (successive iterations of breadth-first search) consists of a low performance phase. Initially, the average BIPS is higher than the SLA BIPS, so the governor reduces frequency to the lowest possible to save power. Eventually, execution enters the low performance phase and the average BIPS starts dropping more rapidly. However, the governor continues with a low frequency execution as the SLA BIPS is still lower than the average BIPS. After a while (1152 seconds for SLA=3.5, 491 seconds for SLA=5.5), the average always remains below the desired SLA although the governor transitions to higher frequencies. By now it is too late to take corrective action because of the nature of the low performance phase. The governor ends up transitioning to turbo mode (observe the sharp increase in power consumption), but average BIPS continues to drop. This is more pronounced for SLA=5.5, where the SLA is breached earlier in the execution, than for SLA=3.5.

This case study highlights a challenge with targeting this SLA for non-homogeneous workloads. There was no way that the governor could have known about future execution characteristics without such information being provided to the objective selector. The executions were doomed to miss SLA quite some time before the violations started to appear. Timely prediction of maximum attainable performance in future execution intervals is needed to resolve this issue.

Another challenge for targeting this SLA is that it is highly sensitive to the accuracy of online performance predictors. For our governors, we sample execution at different operating points and use that data to predict performance for the next interval using interpolation for intermediate points assuming convex behavior. This also assumes that the behavior in the next interval will closely match the characteristics of the sampled execution. Some inaccuracy in performance prediction is inevitable when one or more of these assumptions are violated. These issues are not specific to our governor design and must be addressed by any control policy seeking to target this SLA.

3.10.2 Governing for relative performance targets

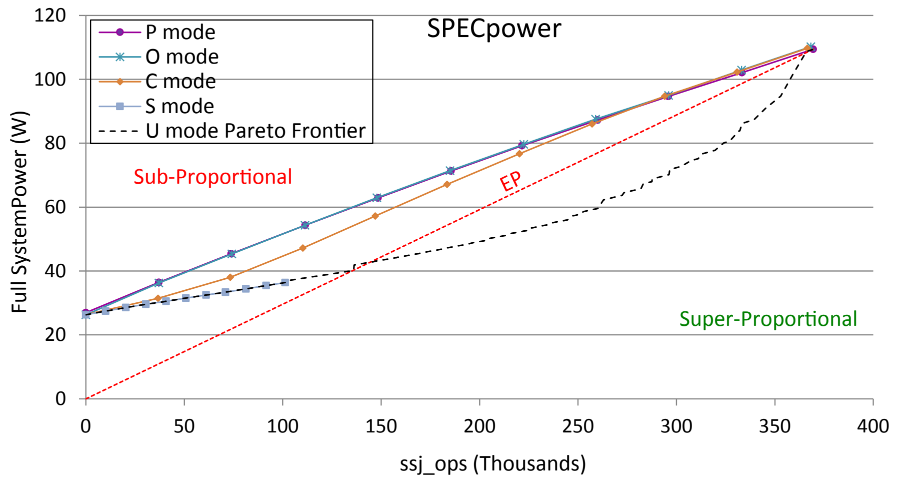


Figure 3.17: Power-performance profiles for SPECpower with Linux governors.

Figure 3.17 shows power-performance profiles for SPECpower with the default governors. **P**-mode and **O**-mode perform similarly and use the highest power for the achieved load. **S**-mode uses the lowest power for the achieved load, but the maximum load achievable is low (see below). **C**-mode works better than **P**-mode or **O**-mode at low loads but in general consumes significantly more power than **S**-mode or the **U**-mode Pareto frontier for the same load. For these experiments we "niced" the power measurement daemon and set the ignore_nice_load parameter of the **O** and **C** governors to discount activity by the daemon while calculating processor utilization.

Even though the **S**-mode profile lies on the Pareto frontier, there are two limitations with this policy:

1. It can only serve up to around 27% of peak load. So it will fail the performance requirement (load served must be within a certain limit of the offered load; see Chapter 2, Section 2.2, for details) at higher loads. The other governors in Figure 3.17 can serve high as well as low loads.

2. Although RUE = 1 for this governor, LUE $\geqslant$ 1.58. This means at least 58% excess energy used compared to operating with the best load even though it is operating at the Pareto frontier. Actually, its profile does not even enter the Super-Proportional region.

So, restricting servers to this policy is not a good idea. The other governor profiles are distant from Dynamic EO, thus having large RUE values. So, using these policies will lead to significant energy waste.

To govern for this SLA, we modify the Pareto predictor to also sample turbo mode so that peak BIPS can be estimated. Instead of limiting the maximum profiling frequency to 3.5 GHz, we now have the maximum profiling frequency as 3.7 GHz (midpoint of the turbo range 3.5–3.9 GHz) since we do not know the actual temperature-dependent operating frequency.

For any load, the target relative BIPS is set to be equal to that load level (load relative to maximum load). For example, attempting to service 70% load level sets the target relative BIPS to 0.7. The objective selector selects the lowest-power configuration that is predicted to have relative BIPS greater than or equal to the target relative BIPS.

We call this new governor **RF_SLAperf(t)**. The parameter t, in milliseconds, determines the length of the intervals as follows. For 100% target performance,

1. Turn prefetching off and profile for t msec.
2. Turn prefetching on and profile for t msec. Choose and set prefetching mode that performed better.
3. Run with that setting for the next 48t msec.

For this target performance, the frequency is kept at 3.7 GHz and not changed.

For a lower target performance, the plan is

1. Set frequency to 2.1 GHz (midpoint frequency). Turn prefetching off and profile for t msec.
2. Turn prefetching on and profile for t msec. Choose and set prefetching mode that performed better.
3. Set frequency to 3.7 GHz (turbo frequency). Profile for t msec.
4. Set frequency to 0.8 GHz (lowest frequency). Profile for t msec.
5. Estimate the Pareto frontier, select the best frequency, run with that setting for the next 48t msec.

DRAM bandwidth and energy consumption are profiled as before.

Figure 3.18 shows power-performance profiles with governors **R_SLAperf(10)** and **RF_SLAperf(10)**. **R_SLAperf(t)** has prefetching always enabled whereas **RF_SLAperf(t)**

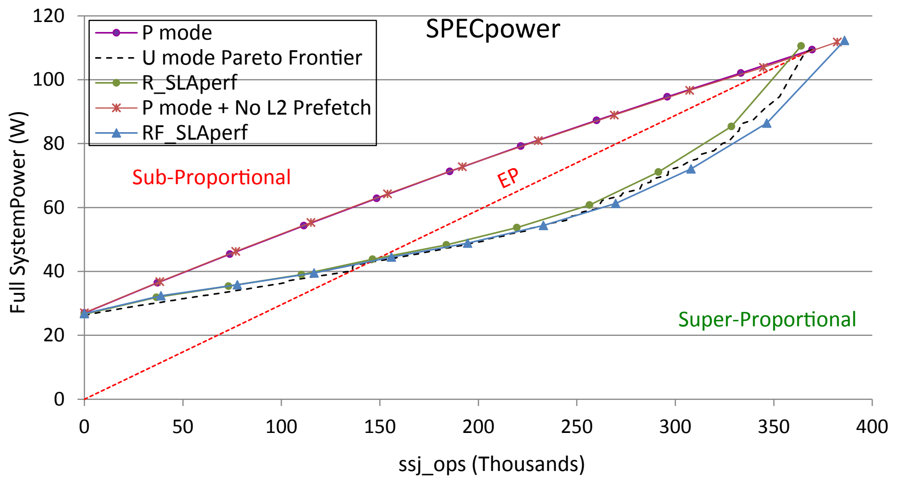


Figure 3.18: Power-performance profiles for SPECpower with **RF_SLAperf(10)** and **R_SLAperf(10)**.

dynamically controls it. Controlling prefetch settings increases the maximum load achievable compared to **P**-mode that always has prefetching enabled. **RF_SLAperf(10)** outperforms **P**-mode by around 4%. This outcome is qualitatively similar to md (See Figure 3.14). The increase in maximum performance can also be observed by running in **P**-mode with prefetching disabled (**P**-mode + No L2 Prefetch profile).

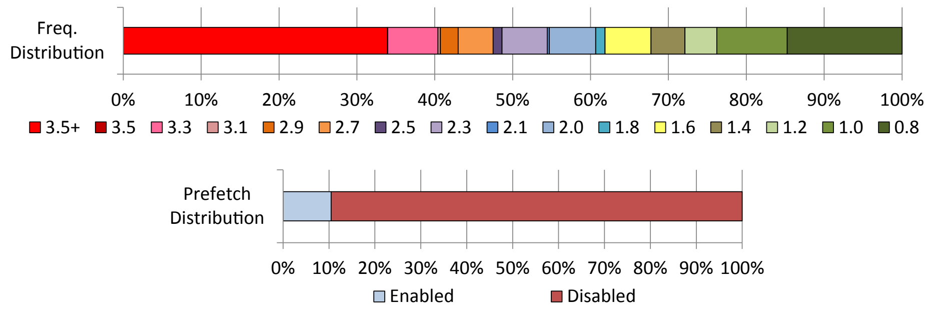


Figure 3.19: Distributions of frequency and prefetch settings for SPECpower with **RF_SLAperf(10)**.

Figure 3.19 shows distributions of frequency and prefetch settings selected by governor

RF_SLAperf(10) over the entire run that includes both calibration and measurement intervals of SPECpower (see [205], or, a brief description of SPECpower intervals in Section 2.2). As expected, it disabled prefetching for most of the time.

RF_SLAperf(10) also chose a number of different frequencies for each run depending on the load serviced. By default, SPECpower does 3 calibration intervals followed by 11 measurement intervals. During the calibration intervals and the first measurement interval, the server is offered very high load. So we expect that during these times the maximum frequency would be chosen by the governor. Thus for at least (3+1)/(3+11) = 28.6% of the time, the maximum frequency should be chosen. The last measurement interval is "Active Idle", that is zero load is offered, so we expect to select the lowest frequency during this interval which is around 1/(3+11) = 7.1% of total time. We observe from Figure 3.19 that the 3.5+ GHz setting (that is, maximum frequency in turbo mode) was chosen for around 34% of the time and 0.8 GHz (the lowest frequency) was chosen for around 15% of the time.

3.10.3 Governing to minimize idle time

So far, a performance target (absolute or relative) needed to be specified to the governors so that they could ensure that SLAperf is satisfied. We will now discuss a new governor that aims to service the offered load without keeping any processing contexts idle. We will demonstrate its action in the context of the SPECpower workload execution.

The key idea is to predict the highest frequency such that there are no idle cycles. Let α denote the number of active (that is, not idle) cycles per second in the last interval. The governor estimates the optimal value of target frequency for the next interval to be $\frac{\alpha}{8}$. The division by 8 is done since there are 8 logical threads on HS. This assumes that in the next interval,

1. the load will remain the same (or at least, not increase) and
2. all threads will be equally active

In case these assumptions are not true, the system may not be able to serve the offered load. To protect against this situation, the governor increases the estimated target frequency by a step whenever it equals the current frequency and doubles the value of the step. This facilitates an exponential ramp-up of frequency over successive intervals so that the offered load is served. On the other hand, if the estimated target frequency is less than the current frequency, the frequency is set to the target frequency and the step is re-initialized. Additionally, our governor selects the best prefetch setting for the next interval.

We call our new governor **RF_Active(t,p)**. Following are the three main steps undertaken by our governor in each interval.

1. Turn prefetching off and profile for p/2 msec
2. Turn prefetching on and profile for p/2 msec. Choose prefetching mode that performed better.
3. Estimate and set target frequency for the remaining interval, so that the total interval is t msec.

Figure 3.20 shows the power-performance profile of our new governor with different parameter values: **RF_Active(10,2)**, **RF_Active(100,20)**, and **RF_Active(500,20)**. We statically determine the length of the interval in Step 3 by taking into account governor overheads for system calls, profiling, and estimations. Note that this governor does not predict either power or performance for any configuration, but manages to operate the system very close to Dynamic EO.

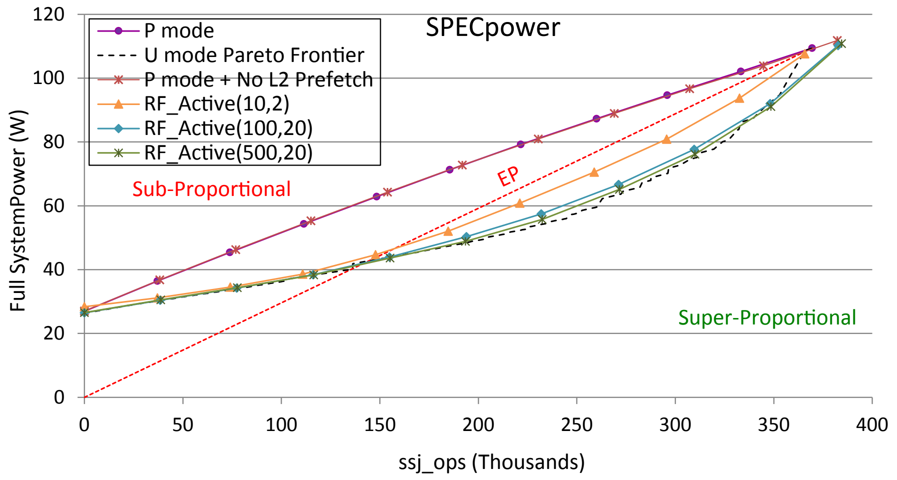


Figure 3.20: Power-performance profiles for SPECpower with **RF_Active(10,2)**, **RF_Active(100,20)**, and **RF_Active(500,20)**.

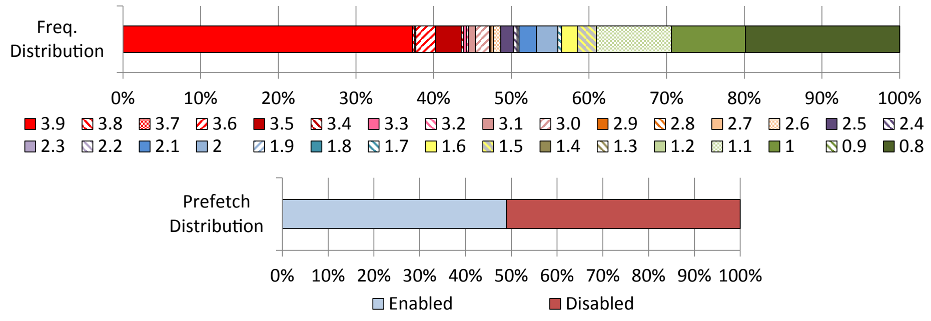


Figure 3.21: Distributions of frequency and prefetch settings for SPECpower with **RF_Active(10,2)**.

Figures 3.21–3.23 show the distributions of frequency and prefetch selections made by **RF_Active(10,2)**, **RF_Active(100,20)**, and **RF_Active(500,20)** respectively. For these experiments, we consider all frequencies, not just the ones available through the OS acpi-cpufreq interface. **RF_Active(10,2)** suffers from more overheads and less accurate selection of resource settings with shorter intervals—it keeps prefetching enabled for a larger fraction of time and also selects the maximum frequency more often than

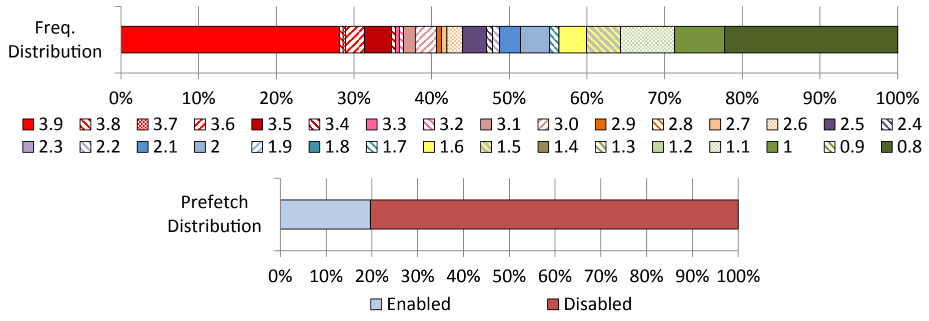


Figure 3.22: Distributions of frequency and prefetch settings for SPECpower with **RF_Active(100,20)**.

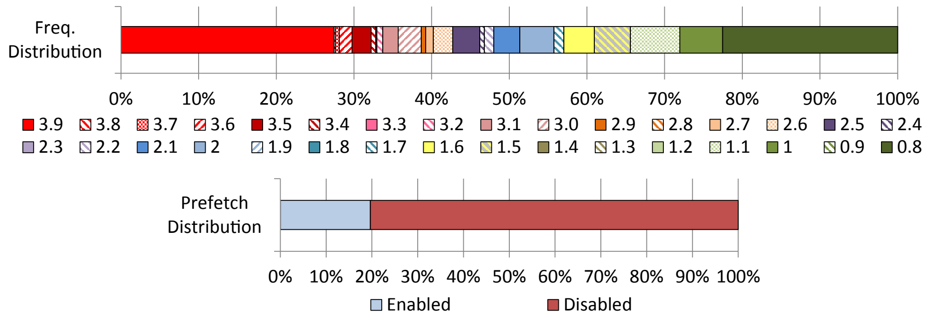


Figure 3.23: Distributions of frequency and prefetch settings for SPECpower with **RF_Active(500,20)**.

RF_Active(100,20) or **RF_Active(500,20)**. Compared to **RF_SLAperf(10)** (Figure 3.19), **RF_Active(100,20)** and **RF_Active(500,20)** select the lowest frequency more often and the maximum frequency less often but keep prefetching enabled for longer.

3.11 Limitations

We will now discuss a few limitations of our governor designs—socket-wide control, intrusive profiling, sampling inconsistency and non-representativeness, and non-zero reaction times.

3.11.1 Socket-Wide Control

Our governors select the same resource setting for the entire socket. This works well for HS, but newer machines offer capabilities for more fine-grained control, e.g., per-core DVFS settings. Having the same frequency setting for all cores may be suboptimal for non-homogeneous workloads if the hardware supports different per-core settings.

Even for machines supporting per-core control, we expect that we can continue with the current profiling strategy (low frequency, middle frequency, high frequency) for all cores. Our idea is to predict power-performance Pareto frontiers for each core and use those to select optimal per-core settings. However, to do that we need to know the performance and energy consumption of each core for each frequency setting profiled.

Current processors already report per-core performance. On HS we measure the energy consumption of the entire socket (HS does not report per-core energy consumption) but this will not be sufficient to determine per-core frontiers. If future processors report per-core energy consumption, then those values can just be used. Otherwise, per-core energy consumption needs to be predicted. One way to do this is to use per-core performance counter values, e.g., BIPS and generated memory bandwidth. Offline regression models that correlate performance counts with energy consumption could help to identify which counters are significant for the given machine.

Determining per-core prefetch settings is more difficult. Whether or not prefetching is useful depends on many factors such as the cache access patterns, the prefetching algorithm, and the timeliness of initiating prefetch. At this point we cannot suggest any efficient user-level or OS-level strategy to accurately determine optimal settings without additional hardware support from the vendor.

Constructing Pareto frontiers, using interpolation, individually for many cores can be costly in software. Hardware support for doing this in the PCU would be very useful. Heuristics that trade off accuracy vs cost can be used if a software solution is required. For example, since piecewise linear interpolation only selects end points for optimal values (Section 3.6), interpolation is not required and only the end points can be considered. This would be faster than quadratic interpolation, but may be suboptimal for some workloads.

3.11.2 Intrusive Profiling

Our governors try out a few resource configurations (e.g., socket frequencies) to determine their effectiveness before selecting the predicted optimal configuration. This intrusive profiling may be costly both in terms of reconfiguration latency and energy for some resources, e.g., cache configurations. In Chapter 4 we describe a novel method for predicting cache performance using reuse distance distributions of cache accesses. That method does non-intrusive profiling, but requires new hardware support.

3.11.3 Sampling Inconsistency and Non-representativeness

Our governors infer the impact of DVFS on the Pareto frontier by sampling execution with three frequencies—lowest, high, and intermediate—over three consecutive intervals and then fitting a quadratic polynomial to the measured power-performance values. This strategy works if the samples have consistent properties and are not drawn from different execution phases. To protect against making invalid interpolations we implement simple checks on the sampled power-performance values, e.g., performance at the lowest frequency should not exceed that at the high frequency, etc. The sample inconsistency problem can be avoided by an alternate approach [202, 207] that samples performance counters once, then predicts power and performance for other configurations using a precharacterized model. However, this approach is limited by the number of performance counters that can be concurrently read on real systems (can affect prediction accuracy) and also by the availability of the particular counters on different platforms (can affect portability).

3.11.4 Non-Zero Reaction Times

Our **R(t)** governors logically partition execution time into epochs with each epoch consisting of a profiling phase followed by a prediction phase (execution with the predicted optimal settings). The plan is t-t-t-48t without prefetch control and t-t-t-t-48t with prefetch control. The profiling phase lasts for t-t-t (total: 3t) or t-t-t-t (total: 4t) time, resulting in the epoch time being 51t or 52t. A small amount of extra delays exist due to overheads in system calls and calls to library functions to read system state and to execute the governor code.

Disregarding schedule variance (exact schedule timing may not be possible) and resource reconfiguration latency, the epoch time represents the worst-case time that the system needs to react to changes in execution characteristics. Shortening the prediction phase will allow faster reaction times at the cost of increasing profiling overhead.

The unit of time, t, cannot be made very small in part due to limited update frequency of the RAPL counters, usually about once every millisecond [96, 113]. In contrast, the DVFS transition time is typically a few tens to few hundreds of microseconds [95, 168] which is about one to three orders of magnitude smaller than t. However, these limits are due to hardware constraints and will affect other governors as well.

3.12 Conclusion

This chapter focused on online mechanisms to reduce RUE by constraining the system to operate close to the Pareto frontier. We developed new OS governors that seek

Pareto optimality in the presence of frequency scaling and cache prefetching and thereby improve the energy efficiency of a modern Intel Haswell server machine. We demonstrated improvements in performance-per-watt by up to 67% (maximum gains) and 30% on average (geometric mean over all workloads). This opens up significant opportunities for revenue generation or cost savings.

We proposed a two-level design to construct governors that are aware of Service-Level Agreements (SLAs) and can be easily retargeted to optimize for different SLAs. We also presented case studies and discussed challenges in governing for maximizing performance within a power cap and minimizing power for a performance target.