HAL
open science

DUF : Dynamic Uncore Frequency scaling to reduce power consumption

Etienne André, Remi Dulong, Amina Guermouche, François Trahay

To cite this version:

Etienne André, Remi Dulong, Amina Guermouche, François Trahay. DUF : Dynamic Uncore Frequency scaling to reduce power consumption. 2020. hal-02401796v2

HAL Id: hal-02401796

https://hal.science/hal-02401796v2

Preprint submitted on 28 Feb 2020 (v2), last revised 2 Aug 2021 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DUF : Dynamic Uncore Frequency scaling to reduce power consumption

Étienne André*, Rémi Dulong*†, Amina Guermouche*, François Trahay*
* Télécom SudParis
Institut Polytechnique de Paris
Evry, France
† University of Neuchâtel
Neuchâtel, Switzerland
Email : first.last@telecom-sudparis.eu

***Abstract*—Reducing the power consumption of applications has become one of the key challenges in high performance computing. Recent processor architectures differentiate processor core frequency from its uncore frequency. As a consequence, in addition to tuning processor core frequency with DVFS, power consumption can also be controlled through Uncore Frequency Scaling (UFS).**

This paper studies how the uncore frequency can be used as a leverage to improve power consumption. We propose DUF, a daemon process that dynamically adapts the uncore frequency to reduce an application power consumption with a user-defined limit on performance degradation.

The evaluation of DUF on two different architectures shows that with less than 3.5% performance degradation, DUF can reduce the socket power consumption by more than 12%. We also show that DUF can reduce the total energy consumption of by up to 13.18%.

***Index Terms*—Green computing, Power consumption, Uncore frequency, High Performance Computing**

I. Introduction

Reducing the power consumption of supercomputers has become one of the key challenges in high performance computing. As a matter of fact, Summit, the most powerful supercomputer consumes 10.90 MW [1] while the US Department of Energy sets a limit of 20 MW for future exascale machines [2].

Dynamically adapting the processor frequency according to the application workload is a common technique to control power consumption. It is widely used in recent architectures where limiting the power consumption and respecting the thermal design power (TDP), while using the processor to its maximum capacity (number of cores, vectorized instructions, ...) requires to lower the CPU frequency, which negatively impacts performance.

Recent processor architectures differentiate the processor core frequency (that affects the computation units and the L1/L2 caches) from its uncore frequency (which affects the last level cache and the memory controller) [6]. The Uncore Frequency Scaling (UFS) automatically selects the uncore frequency according to the CPU frequency, the energy and performance bias hints and cores stall cycles [5]. However, it does not fully benefit from the leverage provided by the uncore frequency: Figure 1 shows the effects of varying the uncore frequency in terms of slowdown (figure 1a) and power savings (figure 1b) for NAS Parallel Benchmarks EP and CG, and HPL. These figures report the relative slowdown and power saving over the default values (obtained with UFS) on the CHIFFLET platform. Both the applications and the platform are described in section III.

Figure 1a shows that uncore frequency does not impact EP performance, while figure 1b shows that setting the uncore frequency to 1.2 GHz reduces power consumption by more than 16 % compared to UFS. Hence, it is possible to achieve the same performance as UFS with a lower power consumption.

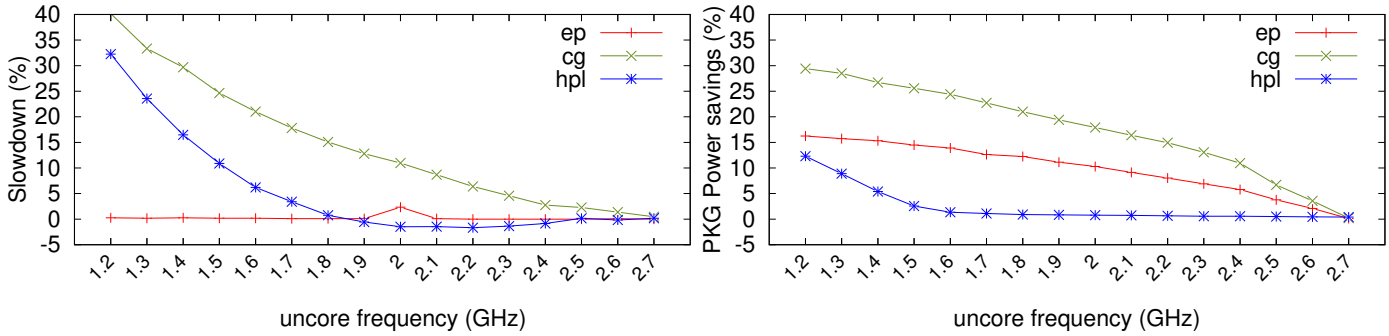
Interestingly, regarding HPL, reducing the uncore frequency actually slightly improves its performance by 1.47%. Since HPL power consumption reaches TDP, the core frequency is automatically lowered which degrades the performance. Manually lowering the uncore frequency decreases the power consumption. As a consequence, core frequency increases leading to better performance.

For CG, if a small performance degradation is tolerated, reducing the uncore frequency significantly lowers the power consumption. For instance, if a 5% slowdown is tolerated for CG, the uncore frequency can be lowered to 2.3 GHz, which reduces the power consumption by 13 %.

Based on these observations, we propose DUF, a daemon process that dynamically adapts the uncore frequency to the application needs. DUF aims at reducing an application power consumption with a user-defined limit on performance degradation. DUF can be seen as providing different uncore frequency governors (performance at 0% slowdown, powersave at 100% slowdown), in a similar fashion to what is done for DVFS.

We compare DUF to the default behavior and to UPSCAVENGER on two architectures and 11 applications and benchmarks. The experiments show that (i) DUF reduces the power consumption by up to 16.34% for EP without altering its performance (ii) a 5.28% slowdown on MG allows for more than 16% power savings (iii) when running under power capping constraints, DUF outperforms UFS with a maximum performance improvement of 11.57% for EP.

[1] http://www.top500.org
[2] https://exascale.llnl.gov/



(a) Slowdown

(b) PKG power

Fig. 1: Uncore frequency impact on execution time and package power consumption on CHIFFLET.

The remainder of this paper is organized as follows: We describe DUF in Section II. Section III presents the measurement methodology we used in our experiments and the evaluation of DUF. Finally, we compare it to the related work in Section IV before concluding in Section V.

II. Dynamic Uncore Frequency (DUF)

In this section, we describe DUF [3] (that stands for Dynamic Uncore Frequency scaling), a daemon process that dynamically adapts the uncore frequency in order to trade a limited performance degradation for power savings. The aim of DUF is twofold: reducing the power consumption of an application, and limiting the performance degradation to a user-provided upper-bound.

A. Overview of DUF

The user of DUF specifies the sockets to monitor and a maximum performance degradation to tolerate. Then, DUF periodically invokes its *measurement module* that collects the CPUs performance counters. Using collected data, the *regulator module* decides whether the uncore frequency should be changed. The decision algorithm described in section II-C applies for each user-specified socket. It can be summarized as follows: DUF detects the application phases (memory intensive vs compute intensive) and, for each phase, measures the performance obtained with the maximum uncore frequency. It then decreases the uncore frequency until the performance degradation reaches the user-specified limit.

B. Measurement module

DUF *measurement module* collects various CPU hardware counters corresponding to the application FLOPS/s and the memory bandwidth in order to guide the *regulator module*. Then it computes the arithmetic intensity as the ratio between the FLOPS/s and the memory bandwidth. An arithmetic intensity greater than 1 indicates that the application is in a CPU-intensive phase. Otherwise, we assume that the application has entered a memory-intensive phase.

[3] available as open-source at:
https://gitlab.com/parallel-and-distributed-systems/DUF

C. Regulator module

In order to select the uncore frequency of a socket, DUF *regulator module* runs Algorithm 1 after every measurement period.

If a new application phase is detected, DUF sets the maximum uncore frequency and measures max_flops and max_bw (the achieved memory bandwidth) during the next measurement period. DUF assumes a phase change happened either because the application arithmetic intensity changed from CPU-intensive to memory intensive or the opposite (lines 4 to 9), or because the FLOPS/s and the memory or the L3 cache bandwith increased significantly (lines 13-16).

Otherwise, DUF checks how the previous decision impacted the FLOPS/s and the memory bandwidth. DUF considers that if the FLOPS/s dropped compared to the previous measurement, but the memory bandwidth remained stable, then the drop come from the behavior of the application itself rather than the impact of the uncore frequency. Based on this assumption, DUF decreases the uncore frequency (lines 10-12). Note that DUF considers the memory bandwidth as stable if it decreased by less than the tolerated slowdown. In other words, if the tolerated performance loss is 20% then the bandwidth is considered as stable if it dropped by less than 80%.

Finally, DUF decreases the uncore frequency as long as the performance remains within the user-specified threshold (lines 17-19) and increases it otherwise (lines 22-23). If a decrease is requested while the uncore frequency is at the minimum, DUF increases the measurement period as we reach a stable phase (line 21). The period is reset every time DUF changes the uncore frequency. DUF also increases the period if the requested uncore frequency is stable across iterations, indicating that a stable phase was reached. In all cases, we limit the measurement period to 10 times the initial period.

On the other hand, if, after an increase, the maximum frequency is reached, DUF considers that the application behavior changed. It updates the maximum FLOPS/s to the current ones and decreases the uncore frequency (lines 24-25).

Algorithm 1 Uncore Frequency Scaling algorithm

```
loop                                                    ▷ Every period
    flops ← measure_flops()
    oi ← measure_operational_intensity
    if oi > 1 and phase! = CPU then
        phase ← CPU
        FREQ=MAX_UFREQ
    else if oi < 1 and phase! = memory then
        phase ← memory
        FREQ=MAX_UFREQ
    if flops < old_flops then
        if bw/max_bw > 1 − perf_loss then
            DECREASE_FREQUENCY
    if flops > 2 ∗ old_flops then
        if bw > 2∗old_bw or l3_bw > 2∗old_l3_bw then
            FREQ=MAX_UFREQ
        else DECREASE_FREQUENCY
    if flops > perf_loss ∗ max_flops then
        if freq > min_freq then
            DECREASE_FREQUENCY
        else if period < 10 ∗ period then
            period = period ∗ 2
    else if freq < max_freq then
        INCREASE_FREQUENCY
    else max_flops ← flops
        DECREASE_FREQUENCY
```

III. Experiments

In this section, we evaluate if DUF meets its two objectives : saving power while limiting the performance degradation to a user-defined limit. We first provide the hardware settings of the experiment testbed. Then, we describe the different regulators that we study. We finally present the results of our experiments.

A. Experiments testsbed

This section describes the architectures and applications that we used.

1) Hardware settings: We used two servers from the Grid'5000 [2] platform. All platforms run under Intel Pstate with performance governor.

- Chifflet is equipped with two Intel Xeon E5-2680 v4 CPUs (Broadwell microarchitecture) with 14 cores per CPU, and 768 GiB of memory. The uncore frequency ranges from 1.2 GHz to 2.7 GHz.
- Yeti is equipped with four Intel Xeon Gold 6130 CPUs (Skylake microarchitecture) with 16 cores per CPU. Each NUMA node has 64 GiB of memory. The uncore frequency ranges from 1.2 GHz to 2.4 GHz.

2) Software testbed: We conducted the experiments using several applications.

- The NAS Parallel Benchmarks [1] provide a set of small applications.We use: BT, CG, EP, FT, LU, MG, SP, UA from NPB-3.3.1 OpenMP version. We choose the problem size so that each application execution time is in the [30s-400s] range. On Chifflet, EP, and FT run using the class D problem size. The remaining benchmarks run using class C. On Yeti, all benchmarks run using class D except SP for which we use class C. The OpenMP threads are bound to cores in a round-robin fashion.
- High Performance Linpack (HPL) [7] is a software package that solves dense linear algebra systems. We use HPL version 2.3 compiled with Math Kernel Library (MKL) version 2019.1.144. HPL uses a configuration file where we set NB to 224 on all platforms. N is set to 62720 on Chifflet and 91840 on Yeti. (PxQ) is set to (4x7) on Chifflet and (8x8) on Yeti.
- LAMMPS [8][4] performs molecular dynamics simulation. We use input file `in.lj` provided for the accelerate suite where we set the `run` value to 100000.
- Nwchem [14][5] is a computational chemistry application. We use the input data set `3carbo.nw` from the qdm provided files.

On all platforms, the applications were compiled with gcc 6.3.0 with -O3 flag. The machines were running Linux version 4.9.0-9. HPL, LAMMPS and nwchem were compiled against Open MPI 3.1.4. Finally, all platforms cores were used during all the experiments (28 on Chifflet and 64 on Yeti) while hyperthreading was disabled. Each experiment was run 10 times and we keep the average over the 8 runs between the minimum and maximum execution times. In all configurations, a maximum variation of 1.5% compared to the mean was observed.

3) Measurement framework: In section I and III-B1, we use LIKWID [13][6] to set the uncore frequency and measure the power consumption of the applications with `likwid-perfctr`. All the measurements are performed every second, with no overhead for all the applications. In section III-C, all measurements (DUF and UPSCavenger require collecting hardware counters in addition to power) are performed using the PAPI library [12][7].

B. Description and configurations of UFS, UPSCavenger and DUF

This section briefly describes the default UFS behavior. It also describes UPSCavenger algorithm and the configurations used for DUF.

1) Default behavior of Uncore Frequency Scaling: In order to understand the Uncore Frequency Scaling (UFS) default behavior on our experimental testbed, we measure the average uncore frequency when running applications with different profiles. For that purpose, we use two memory-intensive applications (CG and MG), and two CPU-intensive applications (EP and HPL).

Table I depicts the average uncore frequency range observed over the sockets. It also provides the average power consumed by the applications over all sockets.

[4]commit `aa2b88578`
[5]commit `67f5237ab`
[6]commit 267d
[7]git commit version `ceb64276`

	CHIFFLET		YETI	
application	Power (W)	ufreq (GHz)	Power (W)	ufreq (GHz)
HPL	120.48	2.4	124.61	[1.6-1.7]
NPB CG	79.2	2.7	123.37	[2.3-2.4]
NPB EP	100.83	2.7	116.08	2.4
NPB MG	82.69	2.7	123.32	[2.2-2.3]

TABLE I: Average observed uncore frequency on CHIFFLET and YETI.

On CHIFFLET, CG, EP and MG run at the maximum uncore frequency (2.7 GHz). The uncore frequency for HPL is lower (2.4 GHz).We also observe that HPL reaches the thermal design power (TDP) of the machine (120 W). This behavior suggests that on CHIFFLET, the UFS policy first sets the uncore frequency to its maximum, and reduces it only when TDP is reached.

A similar behavior is observed on YETI: EP has a limited power consumption. Thus, the uncore frequency is set to the maximum (2.4 GHz). Meanwhile, since CG, MG, and HPL reach TDP, their uncore frequency is reduced.

2) UPSCAVENGER*:* We compare DUF with UPSCAVENGER, a tool that regulates the uncore frequency. Since UPSCAVENGER source code is not available, we implemented our own version of UPSCAVENGER based on the description in [4][8].

At every phase change, UPSCAVENGER updates the maximum DRAM power consumption to the one observed. Periodically, it : (i) decreases the uncore frequency if the DRAM power consumption is steady (ii) detects a phase change and resets the uncore frequency if the power consumption increases (iii) increases the uncore frequency if both the DRAM power consumption and the IPC deacrease (iv) otherwise it detects a phase change and resets the uncore frequency.

Unlike DUF, UPSCAVENGER does not consider a tolerated slowdown. It aims at reducing applications power consumption without degrading their performance. It considers a 5% measurement error. As a consequence, DUF can be seen a generalization of UPSCAVENGER, where UPSCAVENGER should approximately stand between DUF with a 0 % and a 5 % slowdown tolerance.

3) DUF *configuration:* In order to evaluate DUF, we use four different slowdown tolerances: DUF_0 (0 % tolerance), DUF_5 (5 % tolerance), DUF_{10} (10 % tolerance), and DUF_{20} (20 % tolerance).

DUF considers an error margin of 2% regarding accuracy of measurements. Finally, we set DUF uncore frequency step to 100 MHz and the measurement period to 200 ms. Lower measurement periods lead to an overhead on some applications. On the other hand, periods such as 500 ms are too large for short running applications such as LAMMPS or CG on CHIFFLET. From our observations, 200 ms offers a good trade off for all the applications. Note that we discuss how DUF could automatically change its measurement period in paragraph III-G.

[8]our implementation is available as open source at https://gitlab.com/parallel-and-distributed-systems/DUF/tree/master/PowerScavenger

4) Coping with UFS: The default UFS can only be disabled in the BIOS which we cannot access on Grid'5000. As a consequence, when running DUF, the default UFS runs as well. Therefore, a decision taken by DUF can be overwritten by the default UFS. Section III-B1 concluded that both CHIFFLET and YETI default UFS decreases the uncore frequency when TDP is reached. Thus, the frequency set can be lower than the one resquested by DUF. From our observations, this behavior occurs only for applications that reach TDP. In order to handle this situation, at every iteration, both DUF and UPSCAVENGER use the uncore frequency that was set by UFS to compute the next frequency.

C. Experiments results

We run the applications described in section III-A2 on CHIFFLET and YETI while running the regulators. Figures 2a and 2c report the measured slowdown, Figures 2b and 2d show the socket power savings, and Figures 2e and 2f depict the socket + DRAM energy savings. All the results are presented as a percentage over the default values (obtained with UFS) on each platform. In addition to DUF results, the figures also present UPSCAVENGER results and the best and worst values obtained by manually setting the uncore frequency.

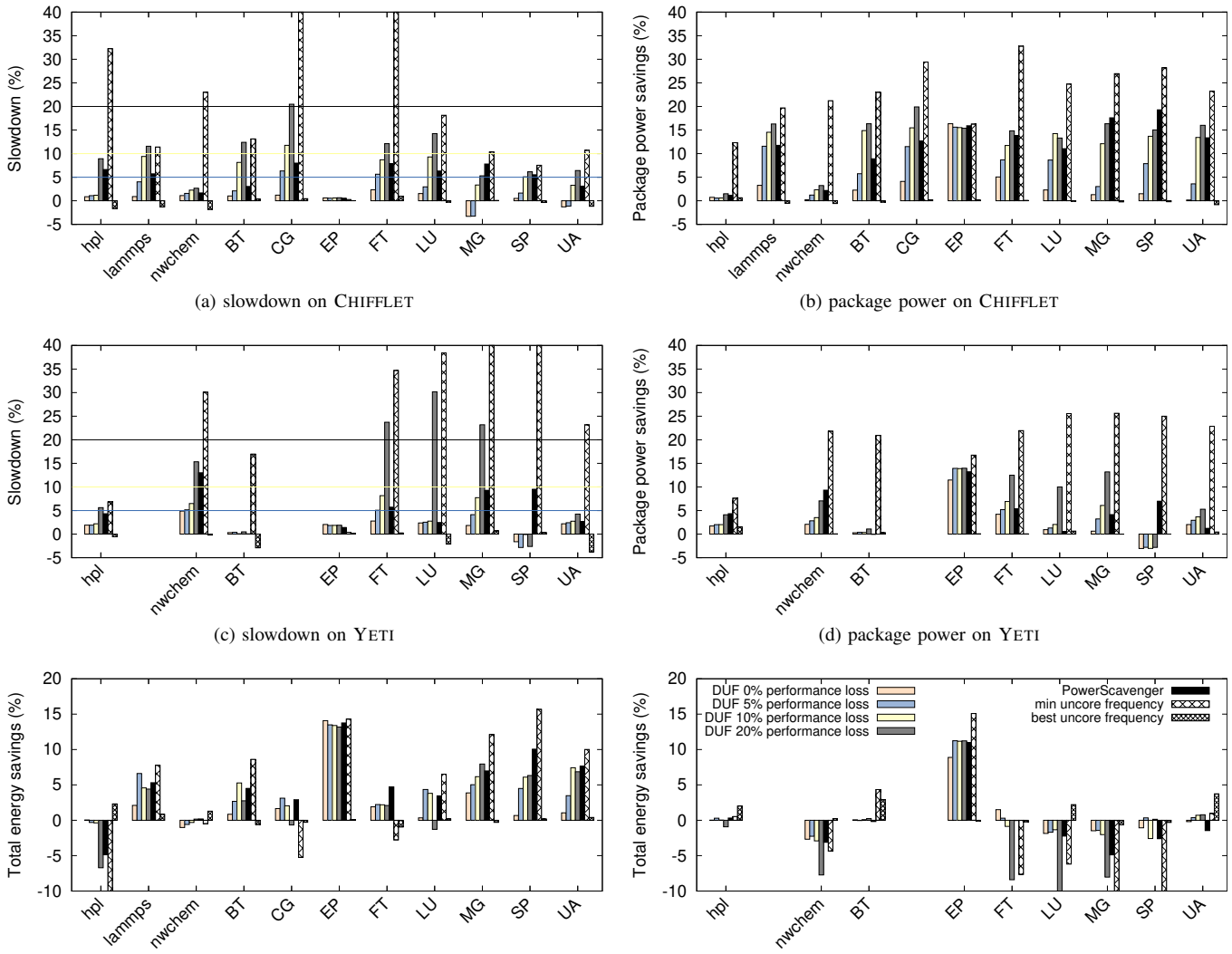
Note that, on YETI, measuring the needed information for DUF and UPSCAVENGER, without modifying the uncore frequency, impacts LAMMPS, CG and nwchem performance. The overhead is rather small for nwchem (2 %) but large for CG and LAMMPS. For this reason, we do not present CG and LAMMPS results on YETI.

D. Impact on execution time

This section evaluates how DUF and UPSCAVENGER affect the application execution time, and if the slowdown respects: a user-defined limit for DUF with a 2 % measurement error, and a 5 % measurement error for UPSCAVENGER.

Figures 2a, and 2c show that, on all platforms, DUF remains within the tolerated slowdown for the majority of applications. Only two applications exceed the limit on YETI: LU and FT with DUF_{20}. The overhead with LU is due to its memory bandwidth usage. As stated in section II-C, with DUF_{20}, the tolerated bandwidth drop is set to 80 %. However, for LU, a 80 % bandwidth decrease causes a large performance degradation. Regarding FT, its behavior is such that the flops suddenly double, but with an increase of L3 bandwidth by a factor of 1.5. However, the frequency is increased only if L3 bandwidth doubles as well (line 14 of Algorithm 1). As a consequence, the frequency is decreased which causes the overhead. These two problems are further discussed in section III-G. Overall, DUF respects the user-defined limit for 77 of the 80 tested settings.

The slowdown caused by UPSCAVENGER for half of the applications remains below the 5% measurement error. For the other half (hpl, LAMMPS, CG, FT, LU, MG, and SP on CHIFFLET, and nwchem, CG, FT, MG and SP on YETI), the slowdown is between 5 and 15%. This is because UPSCAVENGER assumes that as long as the memory power remains



(a) slowdown on CHIFFLET

(b) package power on CHIFFLET

(c) slowdown on YETI

(d) package power on YETI

(e) total energy on CHIFFLET

(f) total energy on YETI

Fig. 2: DUF impact on performance, power and energy consumption. The legend is provided in Figure 2f.

constant, the frequency can be decreased, regardless of the IPC and the L3 bandwidth. However, for some applications, (eg. HPL on CHIFFLET), both the FLOPS/s and the L3 bandwidth drop while the memory bandwidth is slightly impacted. As a consequence, UPSCAVENGER keeps decreasing uncore frequency while DUF increases it.

The figures also show that the behavior of some applications does not allow DUF to slow them down. For instance, BT on YETI and nwchem on CHIFFLET keep switching phases, and HPL, LU, SP and UA on YETI naturally see their FLOPS/s drop by more than the tolerated slowdown.

E. Impact on power and energy consumption

This section evaluates how DUF and UPSCAVENGER reduce the power and energy consumption of the applications. Figures 2b and 2d show the package power saving when using DUF and UPSCAVENGER on CHIFFLET and YETI.

The figures show that the power savings for DUF and UPSCAVENGER are similar: for some applications (eg. HPL or nwchem) the power consumption is only slightly reduced, but for some other applications (eg. CG, EP, or SP on CHIFFLET), power savings exceed 15 %.

EP and hpl have the exact same behavior for all regulators. As reported in figure 1, EP is not impacted by uncore frequency, it reaches 13.73 % power saving on YETI regardless of the regulator. For hpl, DUF_{20} causes a 8.9% slowdown while only 1.50 % power savings are observed on CHIFFLET. This is because HPL power consumption and performance are roughly the same from 2.7GHz to 1.6 GHz as shown in Figure 1. However, as stated in section III-D, in HPL, the FLOPS/s decrease below the tolerated slowdown. Thus DUF does not manage to reach 1.6 GHz. UPSCAVENGER manages to reach lower frequencies but for very few iterations which is not enough to show an impact on power consumption.

For most applications, the power savings obtained with UPSCAVENGER are similar to those of DUF with equivalent slowdown. For instance, UA power savings with UPSCAV-

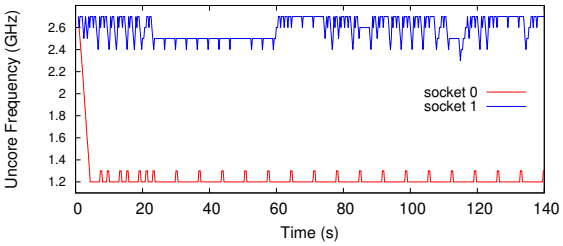


Fig. 3: Uncore frequency during SP execution with DUF_{20} on CHIFFLET

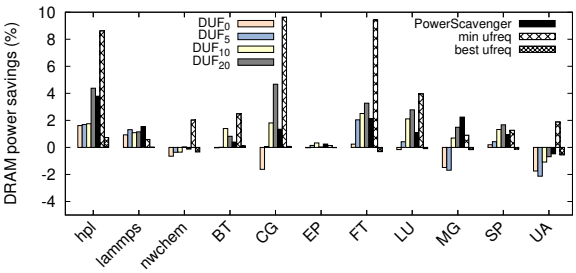


Fig. 4: DUF impact on DRAM power consumption on CHIFFLET

ENGER and DUF_{10} are equivalent and both tools have the same slowdown. This is also the case for CG where UPSCAVENGER performance and power savings are between DUF_{10} and DUF_{20}. This indicates that, for the same slowdown, DUF and UPSCAVENGER reach the same uncore frequency.

Other applications show better power savings with DUF compared to UPSCAVENGER. This is the case for LAMMPS on CHIFFLET where DUF_5 provides the same power savings as UPSCAVENGER with a smaller slowdown. Moreover, on YETI, most applications show better power savings with DUF compared UPSCAVENGER. This is the case for MG where UPSCAVENGER slowdown reaches 10% while its power savings are equivalent to those of DUF_5.

Finally, FT and SP on CHIFFLET and nwchem on YETI show better power savings with UPSCAVENGER compared to DUF despite a smaller overhead. For instance for FT, UPSCAVENGER performs better than DUF_{10} while providing better power savings. This is because for few iterations, FT performance is exactly at 10% which leads DUF to stop modifying the uncore frequency. On the other hand, the memory power consumption remains stable. Therefore UPSCAVENGER keeps reducing the uncore frequency.

Among DUF four configurations, DUF_{20} reaches the maximum power savings. For instance, CG provides the best savings at 19.90 % on CHIFFLET.

Finally, in some cases, with a small slowdown, large power savings can be reached. For instance, for MG on CHIFFLET, DUF_{10} degrades the performance by only 3.34 %, while saving 12.09 % of power.

1) Per-socket uncore frequency regulation: Since DUF handles each socket separately, the uncore frequency may not be the same on each socket. Figure 3 shows how the uncore frequency varies on each socket when running SP on CHIFFLET using DUF_{20}. It shows that, on socket 0, the frequency is most of the time at the minimum while it varies between 2.4 GHz and 2.7 GHz on socket 1. This is due to the fact that on socket 1, both the FLOPS/s and the memory bandwidth both keep increasing then decreasing. Thus, DUF keeps decreasing and increasing the uncore frequency. On the other hand, on socket 0, the memory bandwidth is stable and DUF manages to reduce the uncore frequency.

2) Impact on DRAM power consumption: Figure 4 depicts the DRAM power savings on CHIFFLET. On YETI, the same trend is observed where DUF_{20} provides the best power savings for most applications.

Figure 4 shows that for most applications, the DRAM power consumption corresponds to the default DRAM power consumption $\pm$ 2 %. HPL, CG, FT and LU show a larger difference reaching a maximum power savings of 4.67 % with CG under DUF_{20}. Regarding UPSCAVENGER, the behavior is similar to package power savings where DRAM power savings are equivalent to DUF for the same slowdown. This is the case for HPL, BT, CG, LU, MG and SP.

3) DUF impact on energy consumption: Figures 2e and 2f show how DUF and UPSCAVENGER impact applications energy consumption. We consider both socket and DRAM power consumption when measuring the energy consumption.

On CHIFFLET, both UPSCAVENGER and DUF allow for energy savings for all applications except for HPL. The maximum savings reach 13.18% for EP with all configurations. MG and SP show the best energy savings with DUF_{20}. For the other applications, limiting the overhead provides better energy savings. For instance, with a slowdown of 4.02 %, energy savings reach 6.61 % for LAMMPS.

On YETI however, UPSCAVENGER and DUF only reduce the energy consumption of EP by 11.33%. Due to its significant slowdown, DUF_{20} leads to the worst energy consumption, while the other regulators show similar consumption.

F. Improving performance with uncore frequency

As stated in section I, in addition to improving power consumption, uncore frequency can be used as a leverage to improve the performance of applications that reach TDP. However, the observed performance improvements were rather small. In order to better observe this behavior, we use powercapping, to put a stronger constraint on the default UFS.

Figure 5 shows the performance increase when using DUF_0 and UPSCAVENGER on YETI. We set the powercap to 100W for all applications except EP (98 W), nwchem (90 W) and SP (80 W). The results are compared to those with default UFS under the same constraints. Note that we only run these experiments on YETI because powercapping is not enabled on CHIFFLET. We do not provide the results for SP with UPSCAVENGER as they show too much variation.

The results show that for most applications, using DUF improves performance, with a maximum of 11.57% for EP. UPSCAVENGER shows a similar trend, but with lower performance improvements. The reason behind the performance

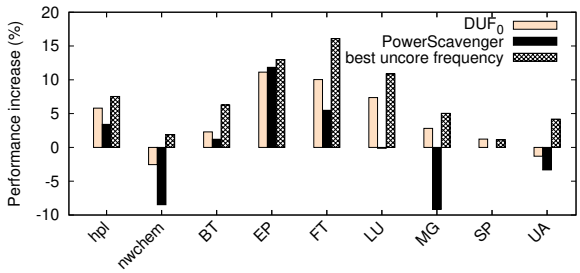


Fig. 5: Performance increase when using DUF under power-capping on YETI

increase lies in the core frequency. For instance for FT, when DUF is not used, the core frequency varies between 1.6 and 1.8 GHz for the four sockets, and UFS sets the uncore frequency to 2.21 GHz or higher. When using DUF, the average core frequency is 2.1 GHz and the uncore frequency goes as low as 1.73 GHz. Thus, by limiting the power consumption with the uncore frequency, DUF allows to increase the core frequency, which improves the application performance. This shows that even if YETI uncore frequency scaling algorithm is more reactive to the behavior of the applications, DUF is actually able to better match the needs of the application being executed.

Because of their behavior, nwchem and UA show performance loss. UA behavior for instance leads to frequently resetting the uncore frequency. As a consequence, it is higher than when using UFS which leads to a lower CPU frequency.

G. Limitations and possible improvements

DUF evaluation shows how it can improve power consumption while respecting the tolerated slowdown. However, we identified some limitations which are discussed in this section.

As stated in section II-C, DUF assumes that the bandwidth drop is correlated to the performance drop. This assumption does not reflect the real impact of memory bandwidth and impacted LU performance on YETI as stated in section III-D. Moreover, DUF assumes that an increase in the FLOPS/s can come with an increase in the L3 or memory bandwidth with same factor. FT on YETI showed that this is not the case. Modeling the impact of uncore frequency on L3 cache and memory bandwidth is definitely needed to handle this issue.

Finally, depending on the application, DUF period should adapt if the application behavior varies too frequently by studying how often the phase changes.

H. Conclusions

The evaluation of DUF exhibits how uncore frequency can improve power consumption. It also showed the potential of uncore frequency as a leverage to improve performance. The overall conclusions of DUF are:

- DUF can adapt to different architectures and different applications;
- DUF manages to stay within the tolerated slowdown;
- DUF manages to reduce socket and memory power consumption. For some applications (such as EP), DUF reaches significant power savings (16.34%) without degrading the performance;
- By slightly degrading the performance of applications, DUF significantly reduces their power consumption;
- DUF manages to improve applications performance under power capping constraints by allowing the cores frequency to be increased;
- Compared to UPSCAVENGER, DUF manages to better respect the applications slowdown and shows better power savings for many applications.

IV. RELATED WORK

Adapting uncore frequency is a recent research topic. In [3] the authors provide a machine learning technique to predict the optimal uncore frequency to be used and showed that the nature of the application impacts the energy saving that can be reached. The authors also study the impact of different performance loss policies. However, the proposed tool is static and needs a training phase on all possible frequencies before deciding the best frequency to run the applications.

Won et al. use a similar approach: they design an artificial neural network to characterize applications and to apply the best uncore power management policy to a network of chips [15]. In this study, the authors emulate a new hardware mechanism that would implement their approach.

In [9], [11] the authors present a study of the potential energy savings using DVFS and UFS for the application GAMESS. They proposed a performance and a power model, and a runtime to adjust both core and uncore frequencies. The runtime also takes a maximum performance degradation limit. The results show great energy savings with, in some cases, very low overhead. However, this work targets only GAMESS. The models were later used to design a tool that distributes a power budget over socket and memory [10]. However, the tool computes the performance obtained with all core and uncore frequencies, rather than adapting to the behavior of the applications like DUF.

The work presented in [4] is the closest to DUF. In section III-C, we compared DUF to our own implementation of UPSCAVENGER. We showed that although UPSCAVENGER is able to provide great power savings, DUF manages to better respect the performance slowdown while still providing power savings.

V. CONCLUSION AND FUTURE WORK

This paper presents DUF, a daemon process that dynamically adapts the uncore frequency in order to reduce the power consumption of applications while limiting the performance degradation to a user-defined limit. The evaluation shows that DUF significantly reduces the power and energy consumption while respecting the slowdown limit. It also shows how uncore frequency can be used as a leverage to improve performance.

As a future work, we plan to further study how to dynamically set the power cap to the application need and combine it to uncore frequency management. We also plan to target DVFS in DUF. Finally, since DUF handles each socket separately,

we will study how applications with different behaviors can be run on each socket in order to maximize power savings.

REFERENCES

[1] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The nas parallel benchmarks. IJHPCA **5**(3), 63–73 (Sep 1991)

[2] Balouek, D., et al.: Adding virtualization capabilities to the Grid'5000 testbed. In: CCIS, vol. 367, pp. 3–20. Springer (2013)

[3] Bekele, S.A., Balakrishnan, M., Kumar, A.: Ml guided energy-performance trade-off estimation for uncore frequency scaling. In: SpringSim. pp. 1–12 (April 2019)

[4] Gholkar, N., Mueller, F., Rountree, B.: Uncore power scavenger: A runtime for uncore power conservation on hpc systems. In: SC'19 (2019)

[5] Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., Geyer, R.: An Energy Efficiency Feature Survey of the Intel Haswell Processor. In: IEEE Int. Parallel and Distributed Processing Symposium. pp. 896–904 (2015)

[6] Hill, D.L., Bachand, D., Bilgin, S., Greiner, R., Hammarlund, P., Huff, T., Kulick, S., Safranek, R.: The uncore: A modular approach to feeding the high-performance cores. Intel Technology Journal **14**(3) (2010)

[7] Petitet, A., C. Whaley, R., Dongarra, J., Cleary, A.: Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers (September 2000), http://www.netlib.org/benchmark/hpl

[8] Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of computational physics **117**(1), 1–19 (1995), http://lammps.sandia.gov

[9] Sundriyal, V., Sosonkina, M., Westheimer, B., Gordon, M.: Core and uncore joint frequency scaling strategy. Journal of Computer and Communications **06**, 184–201 (01 2018)

[10] Sundriyal, V., Sosonkina, M., Westheimer, B., Gordon, M.: Maximizing performance under a power constraint on modern multicore systems. Journal of Computer and Communications **07**, 252–266 (01 2019)

[11] Sundriyal, V., Sosonkina, M., Westheimer, B.M., Gordon, M.: Comparisons of core and uncore frequency scaling modes in quantum chemistry application gamess. pp. 13:1–13:11. HPC '18, Society for Computer Simulation International (2018)

[12] Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with papi-c. In: Tools for High Performance Computing 2009, pp. 157–173 (2010)

[13] Treibig, J., Hager, G., Wellein, G.: LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In: ICPPW. pp. 207–216 (2010)

[14] Valiev, M., Bylaska, E.J., Govind, N., Kowalski, K., Straatsma, T.P., Van Dam, H.J., Wang, D., Nieplocha, J., Apra, E., Windus, T.L., et al.: Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. Computer Physics Communications **181**(9), 1477–1489 (2010)

[15] Won, J., Chen, X., Gratz, P., Hu, J., Soteriou, V.: Up by their bootstraps: Online learning in artificial neural networks for cmp uncore power management. In: HPCA. pp. 308–319 (Feb 2014)