Energy Characterization and Savings in Single and Multiprocessor Systems : understanding how much can be saved and how to achieve it in modern systems

Nicolas Triquenaux

Université de Versailles Saint-Quentin-en-Yvelines École Doctorale "STV"

Étude et sauvegarde de la consommation énergétique dans un environnement simple et multi processeurs : Comprendre combien peut être sauvegardé et comment y arriver sur des systèmes modernes

Energy Characterization and Savings in Single and Multiprocessor Systems :
Understanding how much can be saved and how to achieve it in modern systems

THÈSE

présentée et soutenue publiquement le 18 Septembre 2015

pour l'obtention du

Doctorat de l'université de Versailles Saint-Quentin-en-Yvelines
(spécialité informatique)

par

Nicolas Triquenaux

Composition du jury

Directeur de thèse :	William JALBY	- Professeur, Université de Versailles
Président :	Claude TIMSIT	- Professeur Emérite, Université de Versailles
Rapporteurs :	Christine EISENBEIS	- Directeur de Recherche, INRIA
	Philippe CLAUSS	- Professeur, Université de Strasbourg
Examinateurs :	Pierre VIGNERAS	- Ingénieur de Recherche, ATOS
	Jean-Christophe BEYLER	- Ingénieur de Recherche, Intel

Remerciements

Comme le dira chaque doctorant, il est facile de savoir quand une thèse commence mais beaucoup moins quand elle va finir et quel chemin elle va nous faire emprunter. Me voilà au bout du chemin. « Et heureux qui comme Ulysse, a fait un beau voyage », je tiens à remercier tous ceux qui ont jalonné ce périple de leurs présences, conseils et bonne humeur.

Je tiens tout d'abord à remercier mon directeur de thèse, William Jalby, pour la confiance et la liberté qu'il m'a accordé tout au long de mes travaux de recherche.

Je remercie ensuite M. Jean-Christophe Beyler de m'avoir proposé de faire une thèse, certes de manière quelque peu insistante dès le début de mon stage de fin d'étude. Je ne regrette en rien d'avoir décidé de le faire. Ceci m'a permis de voir jusqu'où je pouvais aller. Je le remercie aussi de m'avoir poussé dans mes moments de doutes.

Un très grand merci à l'équipe énergie qui est née et morte avec moi. Sans eux, je me serais trouvé bien seul face à cette mer capricieuse et changeante qu'est la thèse. Je remercie Benoit Pradelle pour avoir repris le flambeau quand Jean-Christophe est parti découvrir de nouveaux horizons. Je remercie Amina Germouche pour son expertise en programmation linéaire et pour son entreprise d'import export de pâtisserie algérienne soignant tout vague à l'âme. Jean Philipe Halimi pour avoir partagé parfois des problèmes communs quand il a fallu étudier les processeurs xeon-phi d'Intel. Alexandre Laurent pour m'avoir aidé à concevoir les premières briques d'UtoPeak. Et à tous, pour avoir partagé de bon moments gastronomiques.

Je remercie les rapporteurs pour avoir eu la patience de relire ma thèse juste avant leurs vacances. Je suis reconnaissant à Claude Timsit d'avoir su me donner goût à la recherche au cours de mes années d'étude et de m'avoir introduit au vaste monde du calcul scientifique. Enfin je remercie Pierre Vigneras de m'avoir laissé le temps de lui présenter mes travaux autour d'un café et d'avoir accepté de faire parti du jury.

Je remercie Jean Thomas Acquaviva qui, bien que ne faisant plus partie du laboratoire, m'a aidé dans la constitution du jury de thèse et m'a poussé un peu lorsque je laissais le travail fourni chez DataDirect Networks prendre le pas sur la rédaction de ce présent manuscrit.

Je remercie aussi Vincent Palomares, compagnon d'infortune, avec qui j'ai partagé une traversé du Styx mouvementé durant de longues nuits après avoir payé à Charon notre droit de passage en formulaire administratif.

Je remercie aussi mon épouse d'avoir fait preuve d'une grande patience sur la fin de la rédaction. La thèse est une amante qui ne souffre d'être délaissée et « l'enfer n'a pas de furie comme une femme dédaignée » – William Congreve.

Enfin je remercie ma belle-mère pour avoir fourni une expertise sans faille tout au long de la rédaction retirant en bien des endroits mes maladresses orthographiques et syntaxiques. Et je remercie aussi mes parents qui ont permis que je ne sois qu'a 15 minutes de marche de la faculté de Versailles.

À l'optimisme

Résumé:

Bien que la consommation énergétique des processeurs a considérablement diminué, la demande pour des techniques visant à la réduire n'a jamais été aussi forte. En effet, la consommation énergétique des machines haute performance a crû proportionnellement à leurs accroissements en taille. Elle a atteint un tel niveau qu'elle doit être minimisée par tous les moyens.

Les processeurs actuels peuvent changer au vol leurs fréquences d'exécution. Utiliser une fréquence plus faible peut mener à une réduction de leurs consommations énergétiques. Cette thèse recherche jusqu'à quel point cette fonctionnalité, appelé DVFS, peut favoriser cette réduction. Dans un premier temps, une analyse d'une machine simple est effectuée pour une meilleure compréhension des différents éléments consommateurs afin de focaliser les optimisations sur ces derniers.

La consommation d'un processeur dépend de l'application qui est exécutée. Une analyse des applications est donc effectuée pour mieux comprendre leurs impacts sur cette dernière. Basés sur cette étude, plusieurs outils visant à réduire cette consommation ont été créés. REST, adapte la fréquence d'exécution au regard du comportement de l'application. Le second, UtoPeak, calcule la réduction maximum que l'on peut attendre grâce au DVFS. Le dernier, FoREST, est créé pour corriger les défauts de REST et obtenir cette réduction maximum de la consommation énergétique.

Enfin, les applications scientifiques actuelles utilisent généralement plus d'un processeur pour leurs exécutions. Cette thèse présente aussi une première tentative de découverte de la borne inférieure sur la consommation énergétique dans ce nouvel environnement d'exécution.

Mots cléfs:
Caractérisation de puissance, Caractérisation d'énergie, Contrôle des ventilateurs, Contrôle dynamique de fréquences, Profilage dynamique, Applications parallèles, Consommation énergétique minimale

Abstract:

Over the past decade, processors have drastically reduced their power consumption. With each new processor generation, new features enhancing the processor energy efficiency are added. However, the demand for energy reduction techniques has never been so high. Indeed, with the increasing size of high performance machines, their power and energy consumptions have grown accordingly. They have reached a point where they have to be reduced by all possible means.

Current processors allow an interesting feature, they can change their operating frequency at run-time. As granted by transistor physics, lower frequency means lower power consumption and hopefully, lower energy consumption. This thesis investigates to which extent this processor feature, called DVFS, can be used to save energy.

First, a simple machine is analyzed to have a complete understanding of the different power consumers and where optimizations can be focused. It will be demonstrated that only fans and processors allow run-time energy optimizations. Between the two, the processor shows the highest consumption, therefore potentially exposing the higher potential for energy savings.

Second, the power consumption of a processor depends on the applications being executed. However, there are as many applications as problems to solve. The focus is then put on applications to understand their impacts on energy consumption. Based on the gathered insights, multiple tools targeting energy savings on a single processor are created. REST, the most naive, tries to adapt the processor state to the stress generated by the application, hoping for energy reduction. The second, UtoPeak, computes the maximum energy reduction one can expect for any tool using DVFS. It allows to evaluate the efficiency of such systems. The last one, FoREST, was created in order to correct all the flaws of REST and target maximum energy reduction.

Last, scientific applications generally need more than one processor to be executed in a decent time. The thesis also presents a first attempt to compute a lower bound in energy reduction when considering this new execution context.

Keywords:

Power characterization, Energy characterization, Fan control, Dynamic frequency scaling, Dynamic profiling, Parallel applications, Lowest energy consumption

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concerns on energy consumption and its ecological impacts did not rose overnight. Some people say that there always was a sub-part of the society to worry about the ecological impacts of the skyrocketing needs in energy worldwide. Others would tell that no one really care about such concerns as long as there is profit. And more dramatically,a few rave that unless an imminent huge disaster happens, every thing will remain unchanged because those who are wasting the most energy refuse to see its long term impacts. What is known, is that these concerns on energy consumption were publicly acknowledged during the first Earth Summit in 1972 along with many other ecological concerns. Following this Earth Summit, several treaties were issued to unit all the countries towards the same goal. During the second edition of the Earth summit, the United Nations Framework Convention on Climate Change (UNFCCC) was given birth. Signed by 165 countries, it is a framework for negotiating specific international treaties that may set binding limits on greenhouse gas. It helped designing the Kyoto protocol signed on December 11th 1997. From then, the interest in tackling the energy consumption issues as well as other ecological matters went snowballing.

In the world of High Performance Computing (HPC), the term "Performance" was until recently the preponderant criteria to evaluate how well an application is executed. However, with the ever growing demand for performance associated with a higher and higher complexity of the problems to be tackled, the size of HPC systems dramatically increased along with their power consumption. As an example, the first machine on november top500 list, Tianhe-2 consumes 17 Mw. To power such a machine is not the sole problem, the machine has to be cooled down to prevent overheating. For example, 0.7W of cooling is needed to dissipate every 1W of power consumed by one HPC system at Lawrence Livermore National Laboratory [131]. The data is a bit old, and the ratio surely has been enhanced, but still the amount of power to operate such gigantic machine is tremendous. And when looking at the overall picture, the growth will not stop. In 2013, U.S. data centers consumed an estimated 91 billion kilowatt-hours of electricity. This is the equivalent annual output of 34 large (500-megawatt) coal-fired power plants, enough electricity to power all the households in New York City twice over. Data center electricity consumption is projected to increase to roughly 140 billion kilowatt-hours annually by 2020, the equivalent annual output of 50 power plants, costing American businesses $13 billion per year in electricity bills and causing a yearly emission of nearly 150 million metric tons of carbon pollution [6]. Power and energy consumption can no longer be ignored and the energy consumption may replace the performance criteria to evaluate how well an application is executed.

Power and energy consumption have then to be optimized by any possible means. Dynamic Voltage Frequency Scaling (DVFS) was then selected as the flagship of that new crusade. As it will be seen later on, processors power scales quadratically to the voltage and linearly to the operating frequency. Moreover, coming from the transistor physics, the frequency and voltage are linked to eachother. Higher voltage

allows to operate faster, and slower operations allow lower voltage. Then by lowering the operating frequency, tremendous energy consumption can be saved, roughly cubic to the processor frequency. By using that interesting property, a wide range of tools and techniques were built and all of them reported energy reductions. However, Le Sueur *et. al.* [100] well expose the problematic brought with the tremendous decrease in transistor miniaturization. With smaller transistors, the core voltage was drastically reduced, from 5V with $0.8\mu m$ transistor feature size to 1.1-1.4V when considering $32nm$ transistor size. After that assessment, the authors pretend that the potential to save energy via DVFS is dramatically reduced. However, processors efficiency was enhanced with each new generation. Current processors expose a wide range of frequencies, fifteen for the SandyBridge and IvyBridge processors used during this thesis compared to ten for an older Westmere architecture. A wider range of frequencies, allows a better control over the application execution, thus better tuning for energy reduction. Deeper sleep state also are exposed with each new processor generation. As an example on the fourth generation of Intel processors [79], the deepest sleep state allows entire cores to shut down. Moreover, the new Haswell architecture now embarks the voltage regulator on the die, allowing a more efficient power management [64]. A legitimate question then arise when facing all these processor energy efficiency enhancements : is DVFS still a legitimate technique to reduce processors energy consumption while being used ?

This thesis will bring an answer to this question. It will be shown that DVFS techniques, even though they have a limited impact on sequential application, have a huge potential for energy saving for parallel applications.

The first part presents a vulgarization on energy and power consumption. In order to demytify what power and energy consumption mean, different power consumers of a simple machine are analyzed. For each of them, possible optimization mechanics are presented. For the processor power consumption, a more in-depth study is performed to understand how it consumes power under different kind of pressure and what are the means to reduce to consume less.

The second part provides an in depth analysis of applications energy consumptions. Even though the amount of energy is partially dictated by the hardware, the pressure the application puts on the processor and the time it takes to be executed also plays an important part. It will be demonstrated that an application can be seen as a sequence of different phases with different purposes and resource needs. By identifying each application phase, and adapting the processor operating frequency to each of them, the overall application energy consumption is reduced. One could see the application phases identification as the major challenge, however this thesis is not about finding the best phase identification algorithm but to match the processor speed to each application phase at best. Nevertheless, two different phase identification techniques are presented, either static or dynamic. It is demonstrated that the dynamic phase identification allows more flexibility and has less overhead than the static method for the purpose of the DVFS techniques presented thereafter. In total three different techniques are presented: REST, UtoPeak and FoREST. The first one, REST, is a first naive attempt to acknowledge if DVFS techniques are worth the effort. It shows that on various application execution set-ups more than 25% of their energy consumption can be saved. However, as for all the other DVFS based tool there is no clear view on which maximum level of improvement can be expected. To solve that, UtoPeak is created. It is a static tool accurately predicting for one application execution, the maximum energy reduction one can expect from

DVFS based tools. Though the impact of DVFS is limited on sequential applications, it shows that, for parallel applications, at most 45% of the application energy can be saved. Though UtoPeak is able to grant maximum energy reduction, it is static and needs multiple runs of the same application to gather enough information to produce a realistic prediction. FoREST, is created to dynamically achieve the maximum energy saving uncovered by UtoPeak in one application execution. Since it is a dynamic tool, and HPC people are not yet ready to sacrifice too much performances on the altar of energy reduction, FoREST also takes into account a performance degradation limit. FoREST performs the maximum energy reduction regarding that limit. Finally, FoREST adaptivity was further extended to take into account the overall machine energy consumption, and adapt the processor frequency to minimize the overall consumption. It demonstrates that DVFS is good to reduce processor energy consumption, but it cannot be used alone to achieve full machine energy savings.

The previous part demonstrates that DVFS is legitimate when considering parallel applications on a single processor. However does this transpose to multi-processor environments? The last part expose the first elements of an answer. It presents a static tool that predicts the maximum energy saving one can expect for one application regarding its execution set-up and confirms also that DVFS is still legitimate even when considering multiple processors environment.

Finally a conclusion and perspectives are presented.

Part I

Power and Energy Popularization

Chapter 2

Introduction

As introduced previously, it is costly to operate data centers and any energy consumption reduction can be translated in significant money saving. It explains the existence of various energy reduction mechanisms seen in the next Parts. However before designing systems that grant energy reduction, the energy consumption has to be understood by itself.

The term Energy can be found in multiple domains, like mechanics, nuclear power, or electricity and is always used as a rate of performing work over time. The energy is generally computed as : $P \times T$ where P is the rate of activity, and T the time slice on which the work is done. In electricity the power P, is generated by an electric current passing through an electric potential. As an analogy consider a watermill being the electric potential and the flow of water being the electric current. The higher the flow of water on the water wheel the higher the grind force. The watermill energy quantifies the grind force over the period of use. When considering a computer, electrical power is what is needed to operate it and energy is used to quantify the consumed power until it is shutdown. Though, the watermill or the computer have just been considered as a unique entities, it is their composing parts that define the overall needed power to activate them. For the watermill a certain level of power is required to rotate the water wheel and additional power to rotate the grind wheel. It is the same for a computer with the motherboard, memory dims, disks, fans and the processors each need a fraction of the overall computer power. As an example, Table 2.1 shows the power needs of the different computer parts. Each stated value are coming from components data sheet that can be found in any computer hardware resellers.

The values showed in Table 2.1 are the worst case power consumption. Depending on the situation, each part can draw more or less power. Back to the watermill, the grind power will variate regarding meteorological factors. If huge waterfall happened upstream, the water's flow will grant more grind force. At the opposite, if a drought occur, the water's flow will decrease, decreasing the grind force. It is the same for a computer, depending on the usage scenario each computer part will draw more or less power as shown in Figure 2.1.

Figure 2.1 extracted from [44] shows different usage scenario measured on a Be-

Computer's Parts	Power Consumption	Ratio in Percentage
Processor	80W	57.7 %
Mother Board	35W	25.1 %
Ram modules	6W	4.3 %
HDD	6W	4.3 %
Fans	12W	8.6 %

Table 2.1: Computer Power Consumption Break-Down.

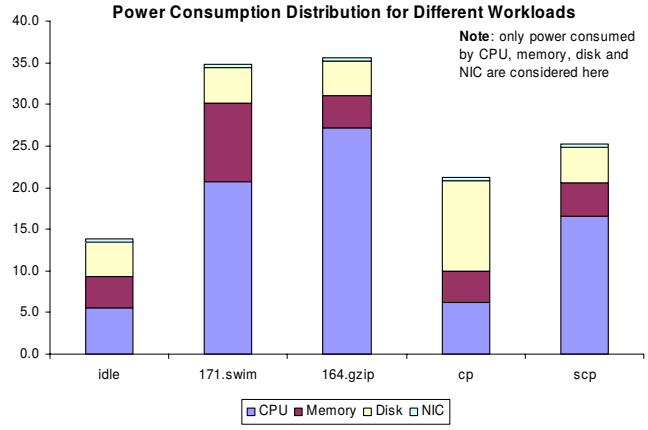


Figure 2.1: Power Profile for Several Workloads [44]

owulf node machine. It can be seen that each component consumes a part of the overall machine power. They can drastically change their needs regarding the computer usage scenario. It can be acknowledged that in most scenarios the processor is the part with the highest consumption. However, disks and memory are not to be neglected since they can consume more than 50% of the overall power in the *cp* scenario.

In the end, the energy consumption can vary regarding the duration and regarding different usage scenarios. Therefore it is essential to fully understand how each computer component consumes power. Relying on these insight, it is possible to determine the means to reduce these consumptions and design energy reduction techniques. However, before jumping into further details, it is as important to understand why the energy criteria is only used in this thesis and not for example energy-delay or other metrics derived from it. Once it is clear that the energy is considered as the baseline metric, the focus is put on each machine power consumer. Ram dimms and disk power consumption and optimization will be first studied. They do not represent a significant part of a single machine power consumption, however data centers do not use only a single disk and two ram dims, but many thousands, making them a non-negligible power consumer. The RAM dims and HDD study will be followed by the study of fans power consumptions and how these elements can prevent the processor from consuming more. It will also be seen in Chapter 5 that the fans are always operated at their maximum speed even when unnecessary , making them wasting power. Finally, in Chapter 6 the processor power consumption and the means to reduce it are presented. Unlike for RAM dims, HDD and fans, techniques to reduce the processor power consumption and energy are not presented in the chapter since they are the purpose of the thesis.

Chapter 3

Metrics

Metrics are essential to quantify, measure, and evaluate a system energy consumption. They form the basis of any optimization mechanism. Many metrics have been proposed and used to rate power or energy efficiency and can be classified into two types : metrics for single machine or processors, and those for parallel systems and clusters.

3.1 Single Machine

The most basic metric used in this thesis is purely the energy consumption, computed as $P \times T$ with P the power consumption of the studied systems and T a time period. However other metrics can be found. The most common is called energy-delay [23, 55, 69, 74, 118], computed as the $E \times T$ and sometimes called PT^2 [118]. It is intended to characterize the trade off between the energy and the delay. Pursued into that direction $ED2P = E \times T^2$ was created by Bose *et. al.* [23] to be used when considering Dynamic Voltage Frequency Scaling. It is supposed to cancel the influence of frequency scaling since E roughly scales with the square of the frequency and T with the inverse of the square of the frequency. Ge *et. al.* [51], based on the $ED2P$ metric, propose a weighted version of it, called *weighted-*$ED2P$. It is assumed to allow the user to influence the metric to decide which is the most important between the performance or energy.

Lastly, based on the assumption that the energy usually is not a linear function of the performance, Choi *et. al.* [28] proposed a relative performance slowdown δ. Based on that metric, power efficiency gain can be more accurately calculated in power management techniques [52, 73].

3.2 Parallel Systems And Clusters

Some other metrics were designed specifically for parallel systems. In [74], Hsu *et. al.* proposed the reciprocal variant of the energy delay product : $1/EDP$. In fact it rates the $FLOPS/W$ that can be delivered by the parallel system. It allows to rate if adding more hardware for an application execution is efficient regarding power consumption.

With ever growing parallel systems, two metrics were designed to quantify the cost of owning an HPC system, TCO, and a metric to rate the energy efficiency of the overall facilities needed to operate such machines, PUE. The Total Cost of Ownership consists of two parts: cost of acquisition and cost of operation. TCO is often quite difficult to calculate, and Feng *et. al.* [43, 74] proposed a way to approximate it with an already seen metric, the performance/power. The Power Usage Efficiency [18], PUE, metric is the ratio between the power drawn by the facility and the power that is actually used by the machine. If it is 1 then all the

power consumed by the facility is used to perform computations. If it is equal to two then half of the power drawn is lost to leak, heat, power converter....

In the end, a wide range of metrics exists to measure the energy and quantify its efficiency. However, PUE or TCO are too coarse a grain for the purpose of the current thesis. All the metrics as EDP, $ED2P$ or the weighted EDP put too much importance on the execution time. By artificially increasing the importance of the speed-up impact, it can obfuscate an actual increase on the energy consumption. As an example, suppose that an application is consuming $E_0 = P_0 \times T_0$. For some reason, the execution frequency has changed, the power factor is increased by a factor of 2 and the execution time is decreased by a factor of 1.5, the new application energy consumption is then: $E_1 = P_0 \times T_0 \times (2/1.5) = 1.33 \times E_0$. An increase in energy consumption happened. However, when considering EDP, it will state $EDP1 = P_0 \times (T_0)^2 \times (2/2.25) = 0.88 \times EDP0$, showing an improvement on the metric when there is actually an increased energy consumption.

Having the power and the execution time equally weighted allows a better understanding of what is at stake. Both power and execution time are orthogonal, and finding a sweet spot satisfying both criteria is complex. It is the whole underlying story of this thesis. This is why only the energy metric is considered, it allows all the presented systems to really acknowledge when there is an actual improvement.

Chapter 4

Hard Drive and Memory Energy Consumption

It has been seen previously in Table 2.1 that RAM dims and Hard-Drive Disks (HDD) do not account for a significant part of the overall power consumption of a computer. Yet, when considering a high performance computing machine, RAM dims and HDD are counted by thousands. As an example, consider the TITAN [99] which is ranked second in the top500 [161], it uses 584 Tera-bytes of RAM and 40 Peta-bytes of disk space. By doing a naive calculus, considering 16 Giga-byte RAM dims and 4 Tera-byte disks, it gives $36,500$ RAM dims and $10,000$ HDD. In addition, when considering $4W$ and $10W$ as power consumption for RAM dims and hard drives, 246 Kilo-Watt are consumed for just maintaining them powered. When considering the power cost of 6.65 Cents per Kilowatt-hour [7], the yearly cost for the operator to just provide power for the disks and the ram dims is $143,304$. It is less negligible than one could think when only considering a single computer. Reducing the power cost of RAM dims or hard drives can translate into significant money savings.

To perform global power reduction, the power consumption of each device has to be fully understood. Any electrical device consumes power in the same way. There is a part of the power consumption dedicated to perform work and another part dedicated to keep the device powered and ready to work. Later in this text, the first power section will be referred to as dynamic power, the second one as static power. It will be seen that reducing dynamic and static powers means modulating the state of the devices to best fit the actual workload. On the one hand, reducing static power generally translates into shutting down some piece of hardware because it reduces the amount of power needed to keep the devices on. On the other hand, reducing dynamic power, means decreasing the operating frequency of the device. Unfortunately, as it will be seen in Sections 4.1 and 4.2, the techniques that could be designed to reduce power consumption either only rely on simulations or are already implemented in current hardware. It was then decided not to push forward on the design of optimizations for RAM and HDD. However, knowing how they are consuming power is important to have a general overview.

4.1 RAM

RAM dims are essential to any computer system. It acts as a buffer, between CPU caches and hard disks, to reduce performances breakdown if a wanted data is not in the last level of CPU cache. Actual system heavily relies on RAM. As show previously with TITAN, 32GB of RAM are dedicated to a single CPU. Memory systems also draw a disproportionate share of power regarding their load [35, 117] because they are usually run at their highest speed to avoid any performance loss. However, it exists various range of workload that do need intensive RAM access.

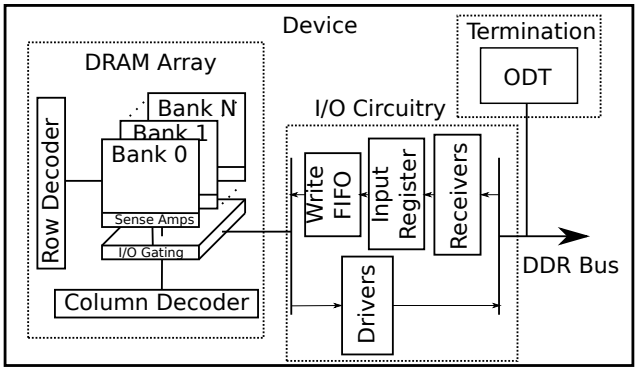


Figure 4.1: General overview of DRAM device structure, extracted from [35]

That gives RAM dims some opportunity to modulate their state and fit the workload needs, reducing their power consumption. Yet, to reduce dynamic or static power, all RAM consuming actors have to be identified.

4.1.1 Power Consumers

Figure 4.1 extracted from [35] shows a general overview of DRAM device structure. Each identified DRAM component consumes power. Based on [35] DRAM Array, power consumption scales accordingly to the memory bandwidth utilization. The larger the amount of accessed data, the higher the power consumption. The power consumption of I/O circuitry, is sensitive to both memory frequency and utilization. Indeed, as it is an interface between the DRAM array and the bus, it is also stressed when a large amount of data is requested. Finally, termination power, is adjusted to the bus electrical characteristics, and depends on its utilization. The sum of each power consumer defines the overall RAM dimm power consumption.

Basically, the overall RAM dimm power consumption scales with the bandwidth, since most of the components are bound to the bus utilization [35]. However, changing the memory operating frequency can reduce the overall power consumption. In electricity, based on Ohm's law, $P = U \times I$, where P is the power, U is the voltage supply and I the current intensity. Lowering U means reducing P. As it is explained later in Chapter 6, the voltage is modified by changing the operating frequency. It is then interesting to modify the operating frequency to reduce the overall power consumption of the main memory. In addition, some RAM elements scale in U and others in U^2 [93] when the frequency is changed. For the elements scaling in U^2, significant power saving can then be obtained when lowering the operating frequency. Though reducing the RAM operating frequency lowers the power consumption, it can impact the performances of an application that requests data into RAM. Consequently, lower performances, means increased application execution time, increasing the system energy consumption. The whole game is then to reduce RAM operating frequency regarding bus utilization to reduce RAM energy consumption [35, 38, 108].

4.1.2 Possible Optimizations

DRAM exposes different operating frequencies. One can change the RAM dims operating frequencies, however the frequency shift has to be performed inside the

BIOS [35]. A machine reboot is then needed. For people looking for power savings on their laptops, no significant impact is observed on their working process. However in an HPC environment, rebooting a set of machine to achieve power reduction is not affordable. That is why most of existing optimization mechanisms [101, 35, 38, 108] rely on power models simulations and are not usable in the practical world. Nevertheless, Malladi *et. al.* [117] states that many datacenter applications stress memory capacity and latency but not memory bandwidth, therefore by replacing the high speed DDR3 with mobile DDR3, they demonstrate a 3-5 reduction factor on memory power with negligible penalties. However, the cost in infrastructure to operate such optimization is not affordable. Still, it enforce the need of being able to reduce the frequency at run-time. David *et. al.* [35] details the different steps and their complexity to allow RAM dynamic frequency scaling. As all manufacturers and HPC machine operators seek a maximum energy efficiency, it may happen in the future.

As explained in the introduction, RAM is not the only device that can be interesting to optimize because of its usage quantity. Disks also consume an important amount of power. Similarly to RAM dims they are permanently operated at their highest performance state, to prevent any performance loss. However, when a purely CPU bound application are run, disks never are on the critical path. They can be put into an idle state or even shut down to reduce the application power footprint.

4.2 Disk

As for RAM, scientific applications more and more rely on disks. The applications tackle ever bigger problems increasing their time to solution. The probability for a hardware failure during the computation can no longer be ignored. Then to prevent the application from losing all the performed computations in case of an hardware failure, check-pointing is performed [12]. In addition, a data protection mechanism is needed to protect the already stored data from any disk failures. Additional disks are then required. As shown in the small example above, for the TITAN machine 10000 disks are used if the considered size is 4TB. It can be more dramatic for data centers which are purely storage oriented. Being able to optimize the energy needed to write or read data or even shutdown unused disk can save a significant portion of the overall system energy consumption.

4.2.1 Power Consumers

Basically a non solid-state hard-drive is composed of multiple actors: spinning magnetic platters, an arm moving the read/write head across the platters tracks and finally some electronic that hosts some buffers and the disk scheduler. Basically the power consumption can be divided into two parts. The power consumed in the mechanical parts and the power consumed in the electronic of control. When performing write or read operations the mechanical parts are the preponderant consumers. Then, the main idea to save energy is to prevent the mechanical parts to be permanently powered. In [77], the author shows that even at rest, *i.e.* not servicing requests, the mechanical parts still are the major consumer. Indeed, the magnetic platters always are rotating in order to grant the best response time then they are always consuming power. In [77], it is also shown that even when the magnetic platters are down the disk still consumes power. Indeed the electronic part is still

powered to acknowledge incoming requests in order to spin-up the magnetic platters back to their nominal speed. Multiple optimization scenarios can be derived from the different disk states and generally target the unnecessary power consumption generated by the mechanical part.

4.2.2 Possible Optimizations

Multiple optimization strategies [66, 134, 167] rely on the fact that in standby mode, all the mechanical components are shutdown. Once a disk is recognized as unused, it is put in standby mode. Though it cuts off almost the entire disk consumption, the cost to spin up the magnetic platters at its full speed is not negligible. It can be up to 4 times the average disk power consumption [77]. Moreover, as the rotation speed is a controlled system it takes some time to reach the nominal rotation speed. It has to be ensured that the disk will be powered long enough to counter that 4 times disk power pick. If not, the optimized disks will artificially consume more power. Others designed disks with dynamic speed control [61]. Instead of purely stopping the platters from spinning, different speed settings are used. The spin speed is then adapted to the request rate [108]. Some others would also consider data placement algorithms. As an example, putting the frequent data at a low Logical Block Number, i.e. the beginning of the disk, where more data can be fetched in one platter rotation [77]. Even though, it exists various ways to optimize Hard Drive energy consumption, all those optimizations are now embedded in current Hard Drives [40]. As for the previous section, it was also decided not to put additional efforts in designing power and energy optimization for Hard Drive Disks.

In the end, in the actual technology state the described optimization scenarios either are unusable in the practical world or are already embedded in current hardware. However, the increasing demand for checkpointing will increase the application dependency to disks. One way to leverage that is burst buffers [126, 163]. Data written by an application to a burst buffer is stored in a significant ram pool until they are stored on disks generally with a redundancy mechanism. The rising interests in burst buffers will further increase the demand for RAM dims or disks, certainly forcing manufacturers to provide more practical ways to optimize energy consumption.

However, for fans one practical way to modify their power consumption is by modulating their rotation speed. Though they are widely used to cool HPC computers or data centers, they are always used at their maximum speed. Therefore, the hardware is always kept at a cool temperature preventing failures due to overheating. However, there is no need to use fans at their full speed since the hardware will operate the same way if the sustained temperature is 20 °C or 50 °C. Moreover, higher speeds mean higher power consumption. The next chapter presents a fan speed optimization technique to lower their energy consumption while preventing the processor to overheat.

Chapter 5

Fans

5.1 State Of The Art

Cooling systems are as important as the machine itself since they keep the hardware in a safe range of temperature and prevent failure due to overheating. Multiple ways to look at cooling systems exit. Two distinct approaches are generally considered to reduce cooling-related energy consumption in a data center or a supercomputer. The first is the general approach where the solutions make large-scale decisions. For instance, energy-centric job allocation [8, 13, 16] or task migration [53, 146, 147, 148] are typical systems helping to reduce energy dissipation. The second approach directly targets the air cooling and tries to reduce its consumption through precise tuning [76, 128]. The tool presented in this chapter, DFaCE considers a narrow scale: instead of considering the supercomputer as a whole, it considers nodes, enabling finer grain tuning.

At the server level, there are two different approaches of the cooling. Either by building theoretical models or by building dynamic systems to react according to what is observed on the system. The first category is about designing theoretical models to estimate the temperature induced by a given load or the best air cooling setting for a given temperature [67, 112, 138]. Rao *et. al.* [138] use such a theoretical model to determine the best CPU frequency according to the chip temperature. However, the model does not take into account the fan power consumption as DFaCE does. Moreover, even though DFaCE does not take into account processor frequency scaling to optimize the overall processor power consumption and not just its leakage, it can be transparently run concurrently to any Dynamic Voltage Frequency Scaling (DVFS) techniques to achieve the same purpose. Heo *et. al.* [67] and Liu *et. al.* [112] both propose models to quantify processor power leakage, however they are not used to control the fan speed.

At the opposite, other researchers propose using theoretical models with the goal of optimizing fan settings [155, 166]. Shin *et. al.* [155] theoretically model the effects of the temperature to simultaneously set the optimal fan speed and processor frequency. Their system could lead to performance degradation since DVFS is considered as an option for cooling the CPU down. DFaCE only considers fans and therefore cannot degrade the programs performance. Similarly, Zhikui *et. al.* [166] propose a Fan Controller (FC) based on a theoretical model that sets the best fan speed for several fans as soon as the CPU load changes. FC, unlike DFaCE, does not consider power leakage: FC minimizes the fan power consumption while maintaining the system temperature at an arbitrary temperature threshold. The presented systems consider theoretical analysis and models to determine an efficient fan setting. Therefore, the solutions suffer from bias induced by the approximations needed to model the complex temperature-related physics.

Theoretical models are built for a specific system or context and external events, such as fan failures or local hot-spots, cause them to be temporarily inaccurate as

the models parameters may change without being reevaluated. DFaCE is not based on theoretical representations of the problem and does not have to approximate the problem because the effects of fan settings are directly evaluated on the computer itself. Dynamic systems react according to what they observe on the system they run on; they do not suffer from the flaws of theoretical models. One such dynamic system, Thermal-Aware Power Optimization for servers (TAPO-server), was proposed by Wei *et. al.* [76]. TAPO-server regularly switches the fan speed to determine if the fan speed has to be increased or decreased in order to reach the minimal power consumption. The authors present convincing results, but TAPO-server does not take into consideration the quick variations of the heat generated by the device. It also restarts the learning process at every major system load change. Moreover, it is unable to handle more than one single fan; whereas, DFaCE is dedicated to multiple fan control, allowing it to efficiently optimize the cooling power consumption. DFaCE also works in two distinct sequences: once the best setting is learned for a given load, it is immediately applied as soon as the load is observed again. Such knowledge capitalization and the ability to reuse the optimal learned fan settings is a key advantage over existing dynamic systems, which are currently unable to react as quickly as DFaCE.

Additionally, several mechanisms were described in the patent literature although they often are similar to the system presented before. For instance, many patents [41, 54, 91, 98] perform simple fan control close to what is achieved by thermal-directed fan control. The work described in [130, 58] is close to TACO-server and suffers from the same flaws.

5.2 Motivations

Although the CPU and the memory are often identified as the major consumers, the cooling system accounts for a non-negligible part of the overall energy consumption [57]. Except for some uncommon configurations [37, 29, 94], fans are still often in charge of cooling a computer. It is common for a PC to be cooled by several fans, each potentially consuming as much as $10W$ at full speed. As the fan power consumption can account for a large part of the total energy consumption, depending on the number used, fans are a good target for energy optimization.

In general, the temperature of the main computer components impact the speed of the fans. A common for controlling system is *thermal-directed*[41, 54, 91, 98]: fans accelerate when the temperature increases in order to maintain the system temperature below an arbitrary threshold, which is often set to a conservative value. Thermal-directed fan control focuses only on temperature management, trying to avoid hardware failures due to overheating, and ignoring energy consumption. Typically, it results in fans unnecessarily rotating at high speeds and consuming too much energy.

Moreover, slowing fans down increases the temperature and, apart from the increasing risk of hardware failures, it increases the power leakage of several components including the CPU. Leakage power is consumed due to transistor imperfection. Power leakage can represent up to 40% of current processor power consumption [112, 123]. Thus, efficiently managing fan speed consists in determining the optimal fan setting, which simultaneously minimizes the processor power leakage and fan power consumption, oversimplifying it would be "cooling enough but not too much". Figure 5.1 illustrates the impact of fan speed where the optimal fan setting

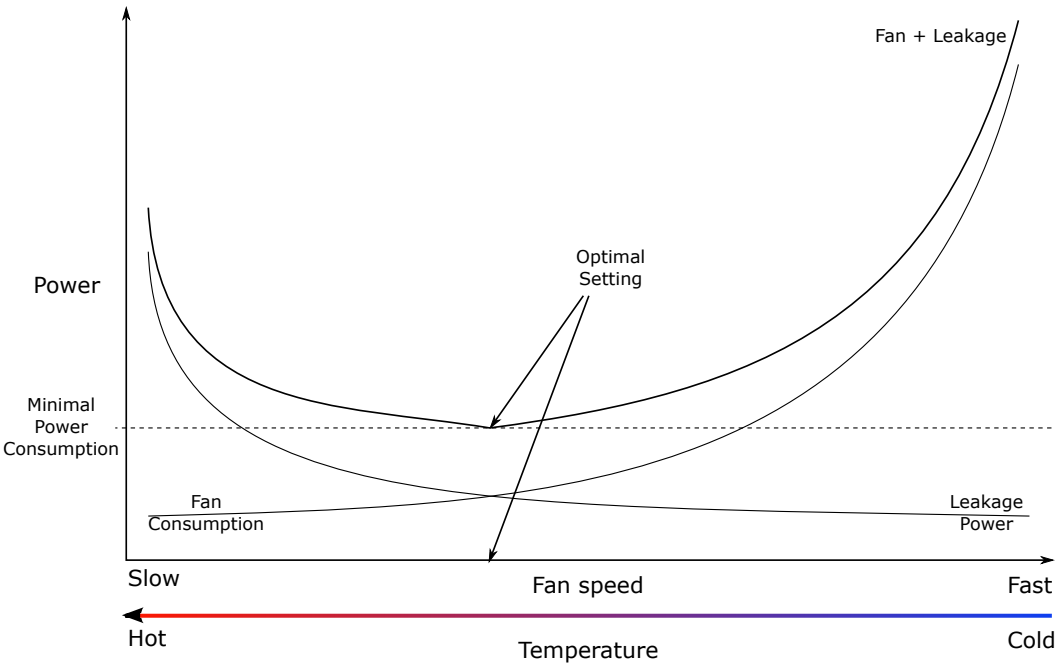


Figure 5.1: Illustrating the balance between fan power consumption and power leakage.

is the one leading to minimal power consumption from both CPU power leakage and fan power consumption. The optimal fan controller has to be able to perform a subtle fan control: it must optimize the power consumption of a computer by using fan speeds that simultaneously minimize fan consumption and power leakage. The fan controller, in this chapter only takes into account fan speed. It does not aim to optimize the airflow either, since it will add a non negligent overhead to an already long converging technique.

Finally, to be able to determine the optimum fan setting as shown in Figure 5.2, it requires a precise knowledge of the fans consumption and of the controlled processor power leakage.

5.3 Power Characterization

5.3.1 Fan Power

The fan power consumption is exponential to its speed. It is common behavior for fans [76, 166] and is shown in Figure 5.2. Different techniques can be used to measure the fan power consumption at different speeds. A straight forward approach consists in plugging a power meter directly onto the fan while controlling its power supply to vary its speed. Such an approach avoids any potential noise as only the fan power is measured, but it requires the fan to be extracted from the computer and to be independently controlled, which often is troublesome.

The actual method employed is based on a power meter, plugged to the computer, which measures the overall system power consumption. As the power meter measures the consumption on the wall as opposed to using probes, the method is non-invasive and more easily achieved. While maintaining the node in an idle state, a dedicated software controls the fan speed while the power meter measures the

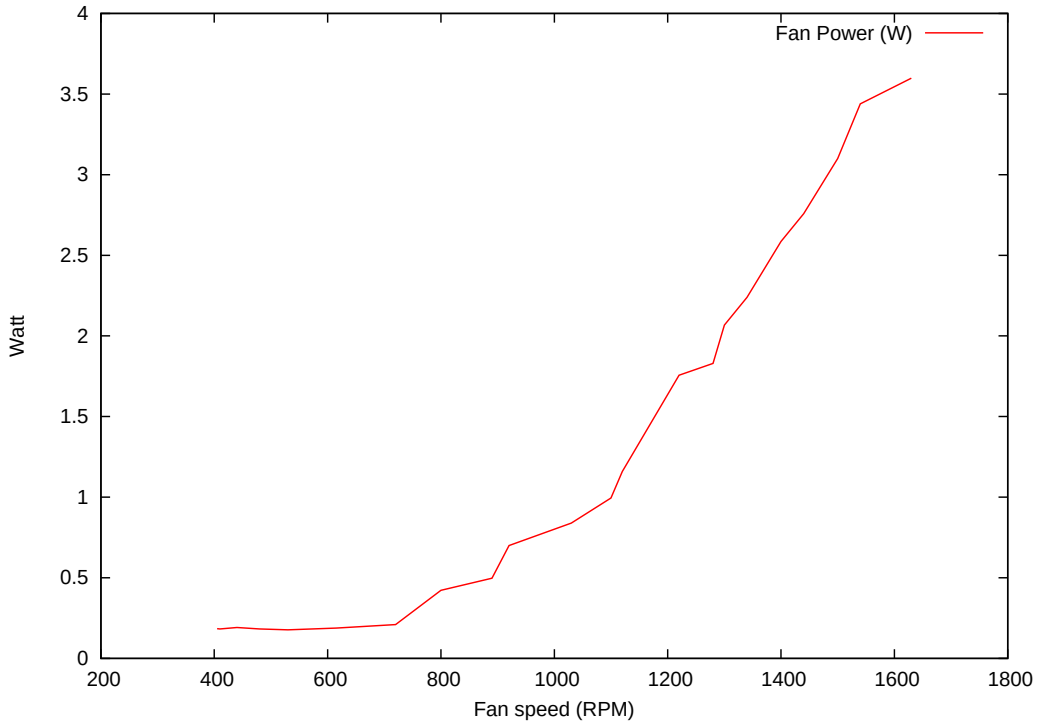


Figure 5.2: Power consumed by a standard 120mm CPU fan at different speeds.

node consumption for different fan speeds. The power consumption is not exact if the fan is not completely shut down at the minimal speed setting. However, the relative power consumptions at different speeds are correct enough to determine the setting for minimal power consumption. This gives the system a lightweight technique to compare power consumptions. Accepting the minimal setting as being a non-optimal consumption, the fan consumption can be considered null at its lower setting. The fan's consumption for every speed is computed by a simple difference between the present and the minimal speed. Let $pw_{fan}(fs)$ be the whole system power consumption for a fan speed fs. The fan power consumption when the fan runs at speed fs is then computed as $pw_{fan}(fs) - pw_{fan}(0)$. The power consumption measured with the current method is not exact if the fan under evaluation is not completely shut down at the minimal speed setting. However, the relative power consumptions at different speeds are correct, which is sufficient to build a power profile of each fan speed. The higher the speed, the higher the power consumption. When considering Figure 5.2, high fan speeds must indeed be avoided. However, small speed variations at the highest fan speed provides significant power savings. Power consumption profiles depend on the fan model, so different fans may lead to different potential gains. The profile presented in Figure 5.2 is representative of the general case.

The fan used for Figure 5.2 is a large fan, similar to the ones usually found in desktop computers. However, fans used to cool down cluster nodes are, in general, smaller fans, operating at higher speeds, consuming more than $10W$ at full speed. In some cases, they even account for up to 20% of the total node power consumption [166]. As a result, greater power savings may then be expected when optimizing a cluster node compared to the desktop computer.

By using the presented methodology, a power consumption profile is built for every connected fan. Such characterization is performed only once, to limit the online overhead. By using the power profiles, the optimization mechanism presented later will choose among the fan speeds the best one regarding the processor leakage and fan power consumptions. However, to grant the system the possibility to also acknowledge the impact of different fan speeds on processor power leakage, an accurate leakage profile has to be built.

5.3.2 Power Leakage

As a reminder to find the optimal fan setting, the processor leakage is also needed. The power leakage is specific to CPU model since it mainly comes from imperfections within the fabric. The CPU leakage is consumed in three areas [129]. The leakage current is the current either going through the substrate or through a not fully closed transistor. The recharge current is due to parasitic capacitance of wires and inputs. Finally, the shoot-through current happens during the CMOS transistor commutation. Equation 5.1 summarizes the three leakage composing the leakage power.

$$\begin{aligned} P_{leak} &= P_{current} + P_{capacitance} + P_{commutation} \\ P_{current} &= I_L \times U = \frac{U^2}{R_L} \\ P_{capacitance} &= U^2 \times C_P \times f \\ P_{commutation} &= \frac{U^2 \times f}{R_S} \end{aligned} \tag{5.1}$$

With P_{leak} as the total leakage power, $P_{current}$ the loss due to leakage current, $P_{capacitance}$ the loss in parasitic wires capacitance, and finally $P_{commutation}$ lost transistors commutation. U is the CPU supply voltage, I_L and R_L characterize the inductance and resistance of the substrate. C_P is the wire capacitance, the longer the wire, the higher the capacitance is. R_S represents the resistance of all the components on the path from the voltage supply and the ground. Finally, f is the CPU's working frequency. Each one of them is squared proportionals to the supply voltage and/or linear proportional to CPU frequency. Moreover, the leakage power P_{leak} is also linear proportional to the die temperature [112]. As fan settings impact the processor temperature, by substitution it also impacts the CPU power leakage, then to correctly measure the power leakage the fan must be stopped. Moreover, as shown in Chapter 6, the processor have the ability to change on the fly its operating frequency and thus its voltage supply level. If the voltage varies, the power leakage will also vary as shown in Equation 5.1. Then by forcing the hardware to use only one frequency, the supply voltage remains constant, allowing to measure the leakage power evolution regarding the die temperature.

Algorithm 1 CPU intensive kernel used to generate CPU heat.

```
num = srandom(42)
while true do
  res += sqrt(num)
end while
```

To measure the leakage, the fans are stopped and a single frequency level is set. An artificial compute intensive task, presented in Algorithm 1, is then launched. The chosen load forces the CPU to increase its temperature. To ensure that a wide range of temperatures are reached, the multiple instances of the same benchmark are launched in parallel on the different processor cores.

A wide range of temperatures is obtained, from the ambient temperature when the processor is idle, to the critical temperature when the processor is heavily loaded as shown in Figure 5.3. To achieve such range of temperatures, the fans are also shut down to allow the CPU to heat up. Algorithm 2 shows the used methodology to achieve different CPU temperatures.

Algorithm 2 CPU heat generation and measurements

```
CPU_Freq = max
for Core=0 to maxNbCore do
  kill all kernel instance
  repeat
    fans_speed = max
  until system is cooled down
  /* stop all the fans */
  fans_speed = 0
  /* launch one instance per Core */
  CPU-kernel(Core)
  for sample=0 to 400 do
    measure power and temperature
    sleep 1 second
  end for
end for
```

Firstly, the CPU is cooled down as much as possible to allow all the CPU cores to start at the same temperature. All the fans are then shut down to allow the processor to get beyond 60 degrees Celsius. After that, the stress is started while periodically measuring the power consumption and the temperature. When the current CPU load is fully sampled the next load level is started. At the end, a temperature and power consumption profile is available for each stress level. Figure 5.3 displays the available data. The y-axis displays each sample power consumption regarding the load level and the x-axis displays the range of reached temperatures. Each floor is obtained by increasing the number of concurrent benchmark execution. Due to the leakage power a slight increase on the power consumption can be observed for each load level while the temperature increases.

Due to the increased CPU activity a huge gap between each load level can be seen in figure 5.3. Generally, the consumed power is approximated as follows [27, 42, 50, 165, 162]:

$$\begin{aligned} P &= P_{dynamic} + P_{static} \\ P_{static} &\simeq cst \\ P_{dynamic} &\simeq A \times C \times V^2 \times f \end{aligned} \tag{5.2}$$

As shown in Equation 5.2 the full CPU power consumption P is function of the power consumed while performing operations $P_{dynamic}$ and of P_{static}. The leakage

power presented in Equation 5.1 is considered constant for a fixed frequency and temperature. The gaps between the different processor loads are due to the increased $P_{dynamic}$. The dynamic power is a function of A the percentage of active gates, C the total capacitance load, V the supply voltage, and f the processor frequency. As the experiment is run on the same processor with a fixed frequency and voltage, C, V and f remains unchanged between two different load executions. However, as more cores are used, the percentage of active gates raises, increasing A. The activity factor is then solely responsible for the gaps between consecutive load executions.

It can also be noticed on Figure 5.3 that for some temperature ranges, such as [48-50], [57-59], and [63-73], two power consumptions are available. As explained above, the difference between them comes from an increased number of used gates and can be expressed as follows. $P2$, and $P1$ are the two power consumptions obtained for the same temperature and $P2 > P1$.

$$\begin{aligned} P2 - P1 &= (P_{NBcore+1} + P_{leak})(P_{NBcore} + P_{leak}) \\ P2 - P1 &= P_{NBcore+1} - P_{NBcore} \\ P2 - P1 &= C \times \times V^2 \times f \times (A_2 - A_1) \\ P2 - P1 &\propto A_2 - A_1 \end{aligned} \tag{5.3}$$

As told above the power leak is considered as constant for a fixed temperature and frequency, P_{leak} can dropped from the equation. Only the dynamic powers $P_{NBcore+1}$ and P_{NBcore} remain. As the same processor, frequency and voltage are used for the two executions, the difference $P2 - P1$ is proportional to the difference of activity factors.

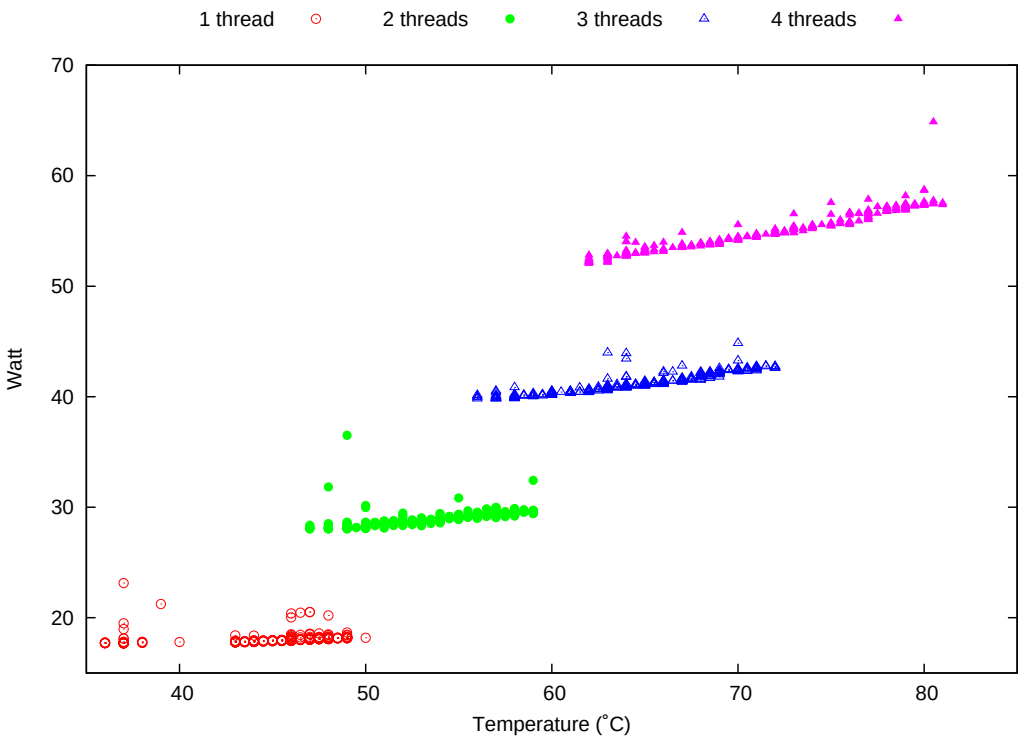


Figure 5.3: Power consumption of a Intel Core i5 2380P processor at different temperatures.

For each point belonging to the same overlapping temperature range, the difference between the two activity factors remain the same since the hardware has not changed. Then by subtracting the increased activity factor to all the points belonging to the same overlapped ranged, its impact on the measured power consumption is nullified. The leakage power then is the only factor responsible for power increase regarding temperature as shown in Figure 5.4.

Number of load's thread	Power difference (W)	Standard deviation
One vs Two	9.98	4.82%
Two vs Three	10.52	4.62%
Three vs Four	11.92	4.26%

Table 5.1: Power cost of using an extra processor core

The difference between each consecutive thread load seems to be 10W when considering the overlapping temperature ranges on Figure 5.3. Table 5.1 shows the exact extra cost for each number of used cores. The column power difference shows the constant value subtracted to each point of consecutive higher load. For example, 9.98 W were subtracted to each power setting measured while using a two thread load. The column *standard deviation* shows the variation noticed for each overlapped point. As all standard deviations are lower than 5% the cost of using an extra core is considered as constant. The result of the normalization is displayed in Figure 5.4.

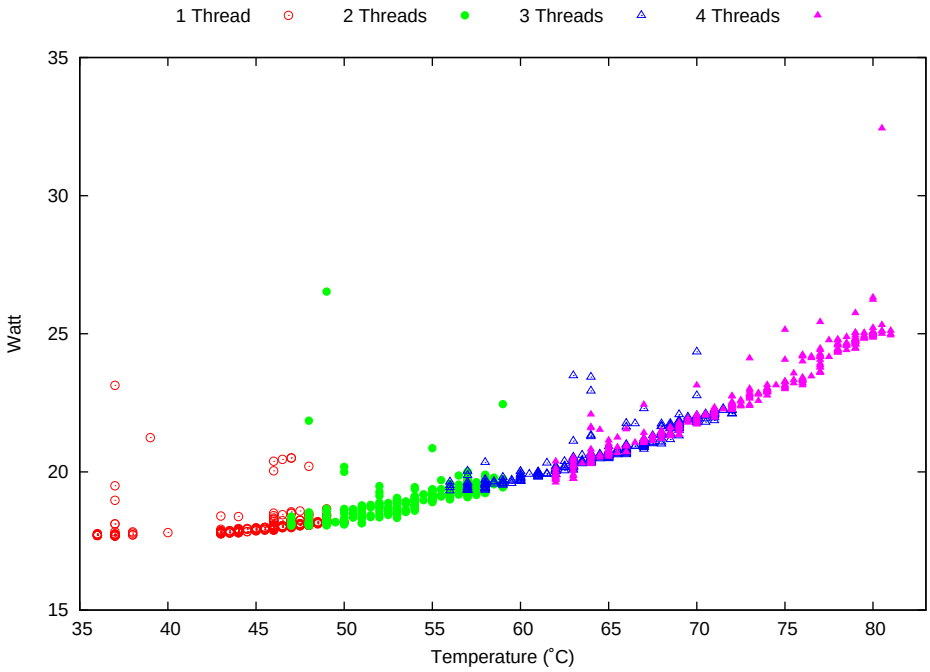


Figure 5.4: Normalized Power consumption of a Intel Core i5 2380P processor at different temperatures.

In Figure 5.4 it can now be clearly see the power increase due to the temperature. Similar to the method employed to deduce fan power consumption, power leakage is calculated by a simple difference based on the data displayed in Figure 5.4. Let

$pw_{HEAT}(t)$ be the CPU power consumption when operating at the temperature t. The power leakage induced by the temperature t, $pw_{leak}(t)$, is then $pw_{HEAT}(t) - pw_{HEAT(t_{idle})}$, with t_{idle} being the temperature reached when the processor is idle. The measurement procedure does not take into account the power leakage of the idle CPU. However, it correctly evaluates the relative power leakage for two different temperatures. The relative power leakage is sufficient for determining the fan setting leading to the minimal power consumption. The power leakage evolution regarding CPU temperature is displayed in Figure 5.5. It can be noticed that the power leakage is linear proportional to the temperature as already proven in previous work [68, 112, 139].

The way of measuring the leakage can lead to missing values for some temperatures. For example, when the processor temperature evolves quickly, the time spent at a particular temperature state is insufficient to measure relevant power consumptions. This is clearly noticeable for the one thread load in Figure 5.4. Such missing values are linearly interpolated from the closest values. As a result, the method provides a full characterization of the power leakage at different temperatures.

Similarly to fan power consumption, leakage information only has to be measured once for each CPU model under its control. It was decided to focus solely on CPUs, as processors are the devices most impacted by temperature variations and power leakage [68, 112, 139].

Finally, the result of the characterization consists in two datasets acquired offline: the fan consumption and power leakage profiles. The full characterization is automatically performed and only once with a power meter plugged into the computer. Once the characterization ends, the optimization process starts without requiring a power meter to be plugged in. The power profiles are used to determine the effects of the evaluated fan settings on the power consumption. How such data inputs are used to evaluate and optimize the processor power leakage and fan consumption is the goal of the next section.

5.4 DFaCE

5.4.1 Overview

Fans are generally used to cool down any critical part within a computer. To do that more and more efficiently, they have become more and more complex. Current fans generally have an embedded temperature probes and are powered by a Pule Width Modulation (PWM) motor. Both features allow an external resource to control the rotation speed regarding the measured temperature or any other conditions. Generally PWM signals are coded as binary values, allowing a fine grain control on fan speed. For example, our experimental platform allows 256 speed steps for every connected fan. If several fans are available, the number of different settings quickly increases. For example, the experimental platform has three fans, leading to nearly 17 million different possible settings. Considering all the possible fan settings, the objective is to determine which one leads to minimal energy consumption. Depending on the processor usage, the temperature may evolves, forcing the fans speeds to change. DFaCE aims at finding the *optimal* fan setting leading to the minimal energy consumption, for the current workload. To avoid hardware failures, it has to ensure that extreme temperatures cannot be reached.

The impact of a fan on the temperature can be determined as some simple rules

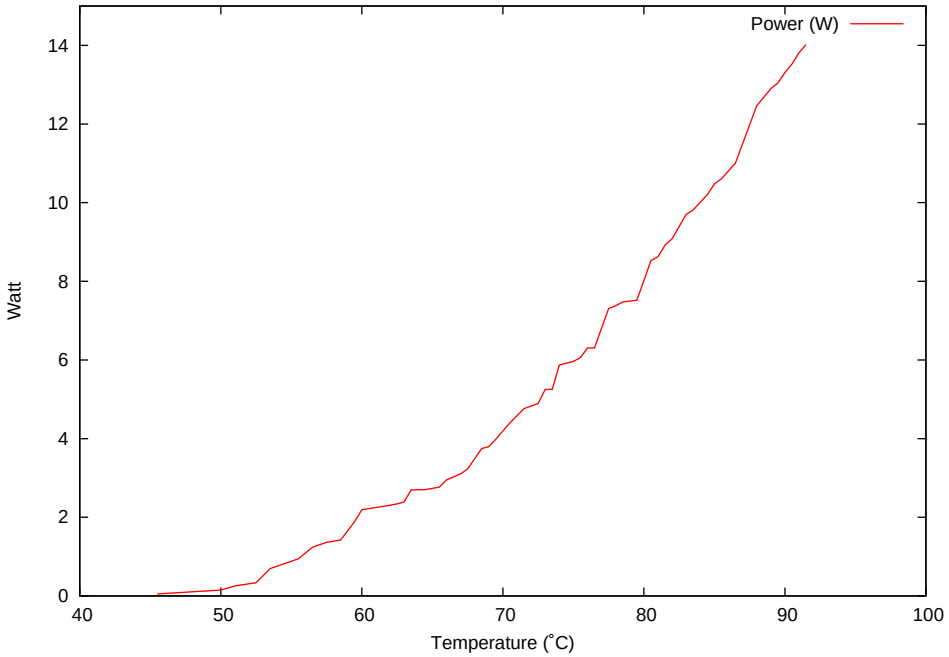


Figure 5.5: Power leakage of a Intel Core i5 2380P processor at different temperatures.

govern the relationship between fan speed, power consumption, and temperature. First, fan power consumption monotonically increases with its speed as shown in Figure 5.2. A fan cannot run faster with less energy. Second, for a fixed workload, the temperature decreases as the fan speed increases. Finally, power leakage decreases as temperature decreases. Provided the three rules are respected, the space of the possible solutions is convex and contains only one minimum. The optimal fan setting is when the total system power consumption is minimal.

To achieve its objective, DFaCE is composed of two phases. During the first one, the training phase, DFaCE learns the optimal fan setting according to processor load. The different steps within the training phase are shown in Figure 5.6. First DFaCE observes the processor load level and measures the die temperature. Once DFaCE has acknowledged the load level, it searches within its known load level and temperature which fan speed setting is the optimum. If known, it applies it. If not or if the load level is new, DFaCE will search for the best solution using the hill climbing technique detailed in the next Section 5.4.2. Some cases can be found where the load level does not last long enough, preventing DFaCE from finding the best solution, and forcing it to stop the optimization process and resume the load level measurements. However, DFaCE stores the preempted load level convergence state, and when acknowledged again, the hill-climbing procedure is restarted where it was stopped and not from the beginning.

The second phase is the fans setting monitoring; it consists in verifying that, at any time, the fan setting remains at its best. Once the best setting is found and applied, DFaCE has to ensure that it is the best. The system load can vary, or fan failure can happen, inducing a change in processor temperature. DFaCE must acknowledge such external event, and resume the training phase to find the best new fan speed. The interaction between the two phases is illustrated in Figure 5.6.

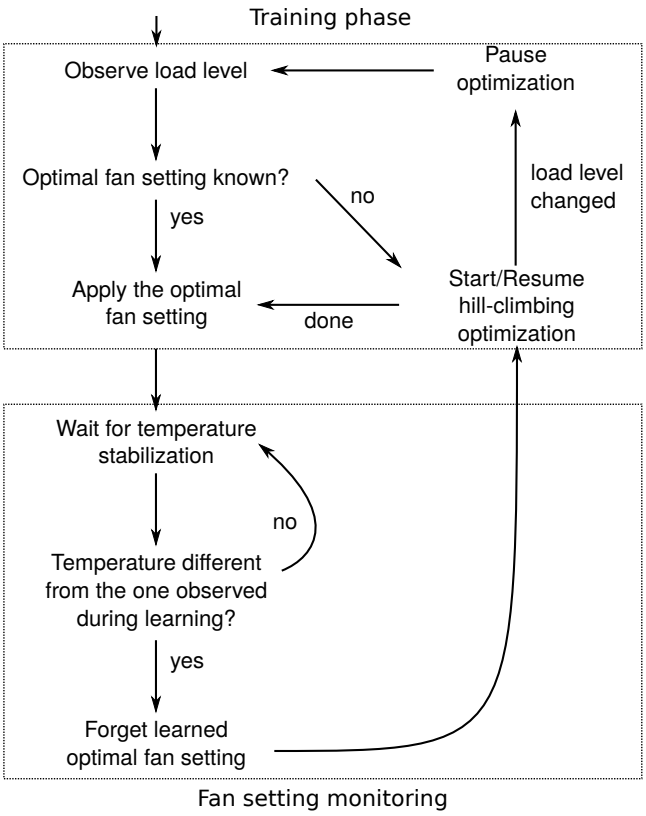


Figure 5.6: Overview of the general system's algorithm.

Though DFaCE adaptivity is important to face external event, the Hill-Climbing algorithm is DFaCE cornerstone. Its ability to quickly converge to a solution defines DFaCE reactivity.

5.4.2 Hill-Climbing

Hill-climbing is an algorithm for determining the global maximum, and conversely minimum, in a convex space. Hill-climbing optimizers first chooses an arbitrary solution then evaluates the surrounding ones. As soon as the algorithm detects a better solution, it uses the new one as the reference point. If none of the surrounding solutions are better, the evaluation restarts on closer points. Hill-climbing allows a fast convergence towards the optimal solution in a convex space.

DFaCE has to find the best fan speed while minimizing the sum of the processor leakage and fan power consumptions. As a reminder, both power consumption were measured beforehand as described in Section 5.3.1 and Section 5.3.2. Based on measurements, DFaCE builds a table associating a power leakage to each observed temperature. Both power leakage and fan consumption are then added when a given temperature is reached during the hill-climbing to estimate the power consumption of evaluated fan settings. The estimated power consumption drives the evolution and hill climbing ultimately determines the setting leading to minimal estimated power. Figure 5.7 exposes the shape of the total power consumption which includes the CPU leakage and the fan power consumption. It also shows how the hill-climbing algorithm described above, and illustrated in Figure 5.8, will converge to the minimum power consumption, thus the best fan speed.

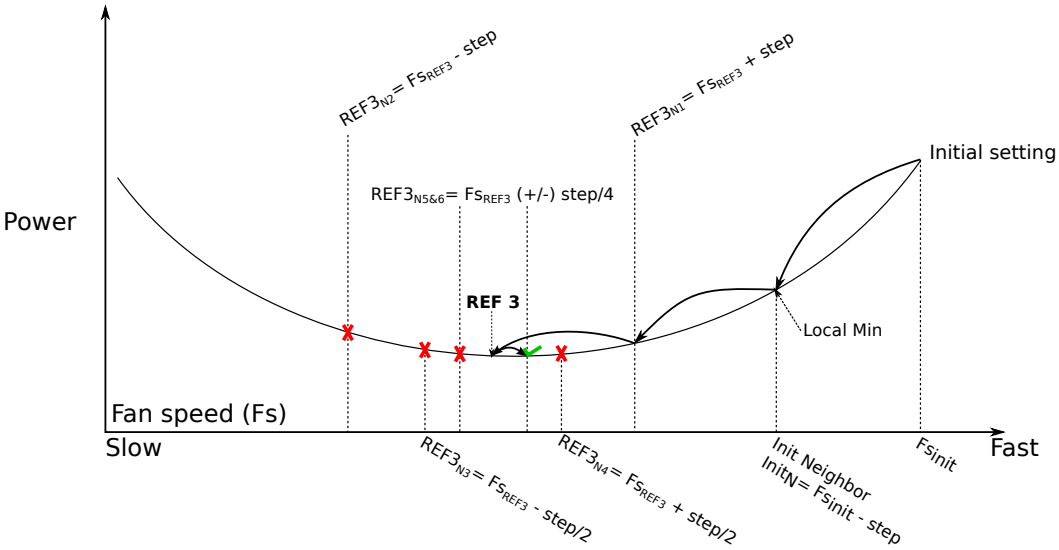


Figure 5.7: The optimizer evaluates several solutions while always progressing toward the optimum.

Figures 5.7 and 5.8 describe in details all the steps needed to find the best fan speed. First DFaCE picks up an *initial setting*, here the highest fan speed. However, the space is convex and choosing an initial point different from the extrema allows a faster convergence. Still, DFaCE is conservative and prevents the processor from reaching its critical temperature, explaining why the highest fan setting is chosen to start the hill climbing algorithm. Once the initial setting is picked up, DFaCE evaluates the power consumption of its adjacent fans speeds. To do so, the hill climbing has to wait for temperature stabilization to retrieve the corresponding power leakage from the associated table and adds it to the fan power consumption. If one of them grants a better power consumption, it is considered as the new reference point here designated as ref. If none of them grants power consumption reduction, a new set of neighbors is computed while using the same ref. The algorithm iterates likewise until it does not find any new ref points within all its adjacent fan speed. The last one is then considered as the global minimum.

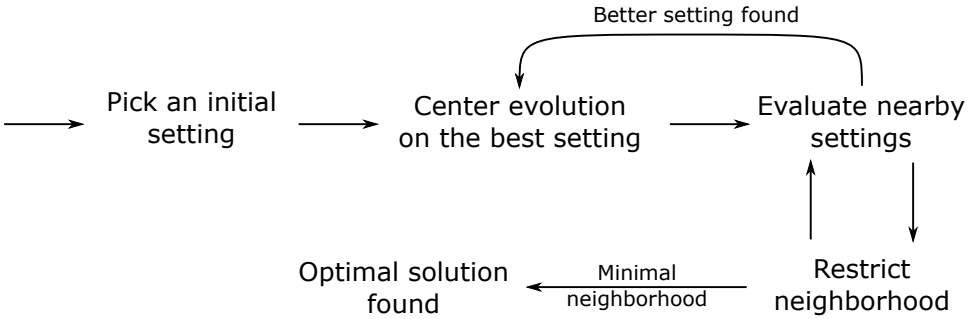


Figure 5.8: Overview of the learning strategy.

The adjacent fan speeds are called neighbors. The considered neighbors are a pair of fan speeds. Each neighbor is *step* distant from their ref point. The neighbors are computed as $Fs_{ref}(+/-)step$. As an example, consider **REF3** in Figure 5.7,

its first neighbors $REF3_{N1}$ and $REF3_{N2}$ are evaluated. However, either of them do not give a better power consumption. The distance *step* is then divided by two, and a new pair of neighbors is considered. If the new neighbors do not give better solution than the one achieved in $REF3$, *step/4* neighbors are considered. It continues until a better new neighbor is found. If none, then $REF3$ is acknowledged as the best solution.

Figure 5.7 displays the convergence mechanism when only one fan is considered. But DFaCE was designed to handle numerous fans. Let n be the number of fans, fs_i be a specific speed of the fan i with $1 \leq i \leq n$ and ref be a fan setting such that $ref = (fs_1, ..., fs_n)$. Every neighbor of the setting ref is defined as $(fs_1 + \alpha_1 \times step, ..., fs_n + \alpha_n \times step)$ with $\alpha_i \in [-1, 1], 1 \leq i \leq n$ and *step* being a distance between ref and its neighbors. Every time DFaCE finds a better setting, it evaluates the power consumption of each neighbor. The value *step* is divided by 2 when closer settings have to be evaluated. Though more complex with a multidimensional space, the convergence technique remains the same as the one exposed in Figure 5.7.

Nonetheless, the Hill climbing algorithm relies on temperature measurements to find the corresponding power leakage measured at DFaCE first start previously presented. The temperature evolves slowly when a new fan setting is applied. If the load level too quickly shifts or ends, DFaCE will always try to find the best fan setting. This potentially results in applying none. DFaCE must then be resistant to fast CPU load transition. A fastest way to extract the processor power leakage from the known profile has to be designed. It is shown before, in Figure 5.4, that there is a link between the processor load and the temperature. The next section shows how it can be used to approximate the processor temperature.

5.4.3 System Load

As a reminder, each actor: the processor temperature, the processor power, the fan speed, and the fan power are linked together. If the processor temperature changes, its power leakage changes, the fan speed is then adapted changing its power consumption. Even though theses four actors are linked together, it is the processor temperature that forces the others to adapt. Then, a fan setting can only be optimal for a given amount of heat generated by the CPU.

That is why DFaCE learns a different fan setting depending on the amount of heat generated by the CPU. As shown in Figure 5.3, different CPU loads are used to traverse the temperature spectrum. Here, the same relationship between the CPU load and reachable temperature is used to allow DFaCE to approximate the processor temperature. One could argue that, only one CPU load could reach multiple temperatures, making such a criterion not suitable for temperature estimation. As a reminder, for Figure 5.3 ,the fans were shut down. The heat calories were not removed from the processor die, making the heat necessarily increasing. It permitted a single processor load to reach multiple temperatures. In the current setup, the fans are powered, preventing the heat calories to stack up, allowing a specific CPU load to reach a single temperature point. Obviously, the CPU load metric does not reflect the exact stress applied on hardware but is sufficient to roughly distinguish various heat generation levels. DFaCE, then, uses the processor activity to estimate the amount of heat it generated and learns different fan settings depending on it.

When considering single CPU load values, the scope of possibilities that must be

tested by DFaCE can be significant. To accelerate the time to solution, it was decided to consider a range of processor loads. The best fan setting is then determined for a given load range. Considering load ranges instead of exact load values leads to a slightly sub-optimal result as the amount of heat generated is not constant for all the possible activities in the same range. In the current implementation, DFaCE considers load ranges of 10%, meaning that it learns 10 different settings to cover all the possible cases.

To measure the impact of one of the 10 different fan settings on the power consumption, DFaCE must wait for the temperature to be stable. If not, DFaCE only has a biased vision of the real impact of the fans speed on both processor leakage and fan power consumptions. The temperature stabilization can take several minutes. Meanwhile, the system load must remain steady. However, a scientific application rarely produces a constant load state. Therefore, if the CPU load changes before temperature stabilizes, the measurement is paused. Though a scientific application does not produce a constant load level, it generally cycles on a fixed range of different load levels. DFaCE resumes the power evaluation, i.e. the learning phase, as soon as an already known load level is noticed and from that setting. In the presence of variable system loads, the learning process could be longer. Nonetheless, as it can resume the evaluation from where it paused, DFaCE still limits the effects of variable workloads on hill climbing convergence time.

The result of the learning phase is then a set of optimal fan settings for different workload level ranges. Once DFaCE has determined the optimal fan setting for one workload level, it immediately applies it once the workload level is observed. However, the link between the load level and the temperature is valid provided that no external factor occurs and changes that relation. As a matter of fact, if a fan fails, the processor generates more heat. Then all previously computed couple CPU load and temperature are not valid anymore. Thus, DFaCE has to be also resistant to any external factor changing the processor temperature to prevent it from reaching its critical temperature as explained in the next Section.

5.4.4 Temperature Stability And Critical Heat

Unexpected external events impact the system temperature and therefore the optimal fan setting. Two major events lead to such a situation. First, the external temperature of the computer might vary, due to local hotspots appearing in the cluster or to a weakness of the air cooling system. Second, one of the fans under control might fail and slow down, or even stop. In both cases, the system temperature is different from the one measured during the evaluation phase. As the temperature is not the expected one, power consumption is sub-optimal: either the system is colder than expected and the fans should be slowed down, or the temperature is higher than what the optimizer learned and the fan speed may have to be increased. Considering the consequences on energy consumption, such external events have to be considered.

In DFaCE, the memorized fan settings are not permanently fixed. Instead, when a previously learned fan setting is applied, DFaCE waits for the temperature to stabilize and compares it to the temperature observed when learning the fan setting during the evaluation phase. If the difference is greater than 3 °C, the fan setting is not considered as optimal anymore and a new learning phase is started, using the previous optimal setting as the starting point for hill-climbing. The 3 °C difference

was arbitrary set, based on real temperature evolution through an application execution. The ability to restart the hill climbing algorithm ensures system reactivity to unexpected events such as a fan failure or a local hotspot.

Temperature instability is also used to the advantage of DFaCE. When a previously learned fan setting is applied, it takes several minutes before the temperature settles. As long as the temperature is below the one observed during the learning phase, all fans are shut down to decrease power consumption. Once temperature reaches the expected value, DFaCE starts the fans and sets them to the learned optimal speed minimizing the power consumption and maintaining temperature stability.

Although DFaCE estimates the generated heat based on system activity, the processor temperature cannot be totally ignored: any cooling system has to ensure no hardware failure occurs due to overheating. Most of the components have critical temperatures above which the hardware lifetime is greatly shortened. The critical temperature is generally given on processors data sheet. It is essential for any cooling system to prevent the hardware from reaching its critical temperature. DFaCE is then designed to react quickly when a setting provokes overheating. Notice that CPU lifetime can also be impacted by DFaCE if it increases the operating temperature [159]. During the evaluation phase, a fan setting is immediately rejected if it leads to a temperature just below the critical threshold. Additionally, if a previously learned setting leads to near overheating situations, DFaCE considers the setting as invalid and starts new Hill climbing iterations using higher fan speeds. Thus, DFaCE never sets the fans to speeds that cause overheating. It has been seen how DFaCE converges to a solution, approximates the processor temperature with the system load, ensures the temperature stability and avoids critical temperatures. However, one last question remains: How fast DFaCE converge to a solution.

5.4.5 Convergence Speed And Optimal Temperature

DFaCE was implemented and tested on an experimental desktop in order to evaluate its accuracy and convergence time. Yet, the transposition to server blades is purely transparent provided it is equipped with compatible hardware: RPM controlled fans. The experimental platform is made of a desktop computer with three different controllable fans: a Scythe Mugen 3 CPU fan and two Alpenföhn Wing Boost 120 chassis fans. One of the chassis fans is located in front of the box, while the other one extracts air at the rear, near the CPU. The CPU is an Intel Core i5 2380P quad-core processor with one thread per core. The running operating system is Linux 3.4, using an experimental driver for the Nuvoton NCT 6775 fan controller chip [142] embedded on the ASUS P8Z77V PRO motherboard.

As a reminder, DFaCE controls the CPU heat through fan speed adaptation regarding each workload. The experimental computer executes synthetic micro benchmarks performing CPU intensive operations such as square root operations or random number generations, and uses an increasing number of processes. The same execution setup used for processor leakage and fan power study. It generates various load levels that last long enough to let DFaCE determine the ideal fan setting for every generated load level.

The experimental platform is made of three fans, implying a three-dimensional domain space. Before starting, the optimizer was set to its initial default state and, using the artificial workloads, it automatically learned the best fan settings. During

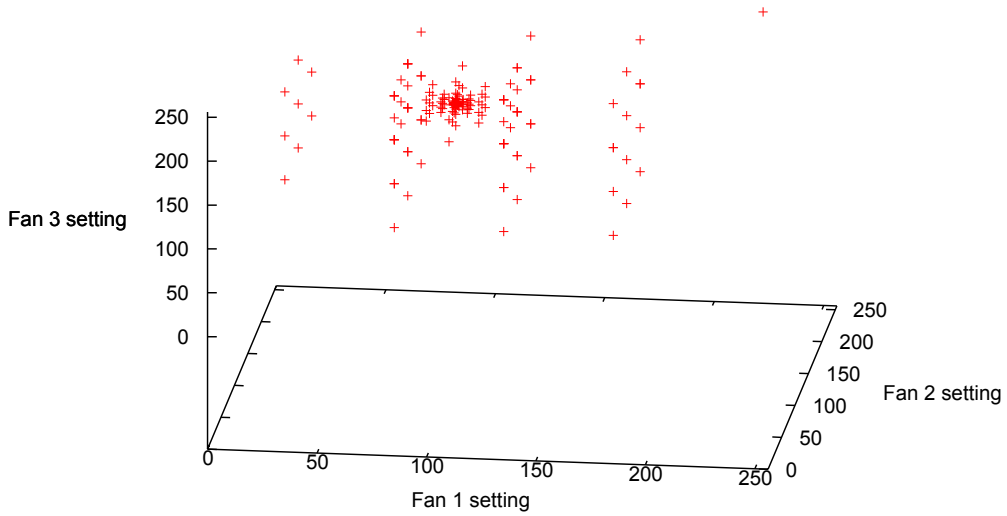


Figure 5.9: DFaCE evaluates only a subset of the fan settings before converging on the optimum.

the experiment, DFaCE converged towards the optimum solution and a large amount of the evaluated settings were close to optimal, which leads to an already lower power consumption during a major part of the execution, illustrated in Figure 5.9. The figure represents the evaluated fan speed solutions by DFaCE during the learning phase for a given workload. The rightmost point in the figure is the first evaluated setting. It corresponds to the full speed setting on the three fans. It is clear the initial default setting is too aggressive, hence the optimizer quickly considers lower fan speeds. The concentration of the evaluated points shows the ability of DFaCE to convergence towards a solution.

The settings closest to the optimum solution were all evaluated in the last phase of the hill-climbing algorithm where the optimizer searched for the optimum solution. The density of the cloud of points shows a large amount of the evolution time is spent near to the optimum setting. If Figures 5.10 and 5.11 are considered, the power consumption after $1,200$ minutes is close to the power consumption induced by the optimum setting. Meaning that respectively 30% and 52% of the total hill-climbing evolution time is spent in evaluating setting close to the optimal one. Figure 5.9 illustrates the efficiency of the hill climbing optimization as only a few settings far from the optimum are evaluated. In fact, most of the hill-climbing time is spent near the optimum setting. DFaCE quickly converges towards the lowest power consumption during the evolution. Moreover, one can notice on Figures 5.10 and 5.11 that power consumption is close to minimal long before the optimization ends, especially in Figure 5.11. Even if DFaCE needs a long time before determining the optimal fan setting because of the slow temperature stabilization, it can achieve near-optimal fan control long before the learning process finishes.

To understand fan and power leakage, and the corresponding CPU temperature DFaCE observed during the optimization procedure, consider two different artificial loads presented with Figures 5.10 and 5.11. The convergence of the optimizer is observed on both the power and temperature curves, although the duration to convergence depends on the initial setting. When only a single core is loaded, the initial default fan setting is close to optimum, leading to less evaluations before converging

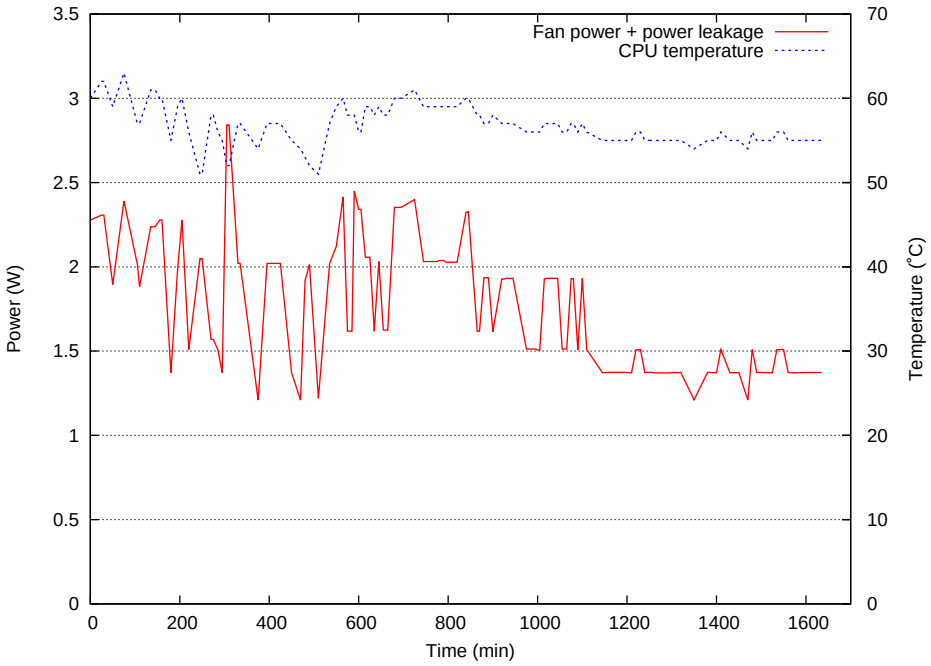


Figure 5.10: Fan power consumption plus power leakage, and CPU temperature converge towards the optimal solution with a 25% load level.

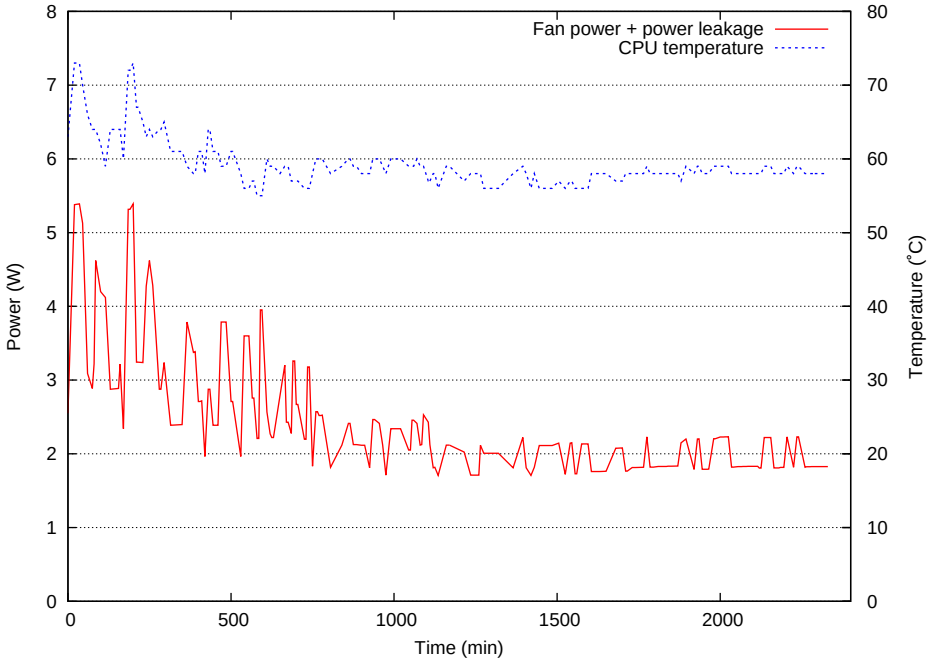


Figure 5.11: Fan power consumption plus power leakage, and CPU temperature converge towards the optimal solution with an half-loaded CPU.

then in the second presented case. Compared to Figure 5.10, the evolution time nearly doubles in Figure 5.11. In case of two loaded cores illustrated with Figure 5.11, the default setting is far from the optimum solution, forcing the optimizer to evaluate more settings before converging. The two presented evolutions illustrate the importance of the initial setting to accelerate the evolution; enhancing the initial fan setting is future work. For clarity purpose, the cases with 75% and 100% processor usage are not presented. They expose the same behavior as Figures 5.10 and 5.11. However, the temperature acknowledged as optimal by DFaCE for such configuration are presented in Table 5.2. One can have concerns about the credibility of such system taking more than 20h to converge. As said above, the first fan setting is far from optimal, and choosing more wisely the Hill-climbing starting point could greatly reduce the convergence time. However, it is left for future work. Furthermore, it has to be kept in mind that the search for optimal temperature regarding different processor loads is performed to build a knowledge base to be used by DFaCE. For different range of loads it knows the optimal temperature and the corresponding fan settings. Therefore when DFaCE is running, it can use that knowledge base to instantly set the optimal fans settings to match a specific load and temperature.

Load level (%)	0	25	50	75	100
Optimal temperature (°C)	36	54	56	64	65

Table 5.2: Optimal CPU temperature for different workload levels.

Table 5.2 shows the temperature found as optimal by DFaCE regarding a specific CPU load. It can be seen that there is at least three temperature modes. One when the machine is idle, one when the processor is partially loaded, and another one when it is heavily stressed. However, the usual method for managing fans consists in defining a single temperature threshold and adapting a fan speed to remain as close as possible to the threshold. When the temperature increases, the fan speed is also increased, when the temperature is below the threshold, the fans are stopped. The temperature threshold has to be low enough to prevent the processor from leaking too much power. It also has to be high enough for fans to operate at a low speed for any temperature, preventing them from over consuming power. In any ways a single threshold cannot efficiently satisfy theses constraints. Thus, thermal-directed fan controllers cannot reach optimal power consumption because they consider a single temperature threshold for every system load where DFaCE shows there is at least three. DFaCE is then superior to classical thermal-directed controllers. Nonetheless, its superiority compared to thermal-directed controllers does not state its real ability to reduce processor power leakage and fan consumption.

5.5 Power Savings

Once DFaCE discovers the best fan setting for all possible load levels, it immediately applies it as soon as a new load level is observed. Load level variations are detected thanks to periodic checks that do not induce any measurable CPU overhead. As explained before, to accelerate the hill climbing convergence, DFaCE only learned the optimal fan settings for ten different load level. However such partial knowledge

is still sufficient to evaluate the power savings achieved at the learned load levels.

An arbitrary sequence of programs, built from the NAS-OMP 3.0 benchmarks, was run using a different number of threads for each program using DFaCE and thermal-directed fan control. The experimental procedure executes a realistic scenario made of existing programs, while the variable number of threads provokes variable load levels, trying to fool the optimizer. The load level seen by DFaCE roughly corresponds to a fourth of the number of threads as the tasks are compute intensive and as there are four cores on the processor. The programs and corresponding number of threads used for each program are presented in Table 5.3. The optimal fan setting discovered by DFaCE was compared to the default thermal-directed fan controller targeting 50 °C, the factory setting, or 60 °C, arbitrarily determined as a relevant value. Such thermal-directed temperature is provided by the experimental motherboard, an ASUS P8Z77V PRO, and consists, as any thermal-directed controllers, in increasing fan speeds, when the temperature exceed the threshold, in order to maintain the system as much as possible below the threshold temperature.

Program	BT	CG	EP	FT
# threads used	1	2	4	2
Savings over 50 °C (fan + leakage)	17 %	40 %	46 %	33 %
Savings over 60 °C (fan + leakage)	29 %	28 %	34 %	31 %
Program	IS	LU	MG	SP
# threads used	1	4	4	4
Savings over 50 °C (fan + leakage)	16 %	0 %	9 %	3 %
Savings over 60 °C (fan + leakage)	32 %	31 %	0 %	30 %

Table 5.3: Power savings achieved by DFaCE compared to thermal-directed cooling with a target temperature of 50 °C or 60 °C.

During the experiments, the power saved by DFaCE at the overall system scale were measured by using a Yokogawa WT-210 power meter plugged to it. In order to determine how significant are the power savings compared to existing mechanisms, the power savings are expressed as a percentage of the maximal fan power consumption and power leakage induced by the default thermal-directed cooling system targeting 50 °C or 60 °C. They are displayed in Table 5.3. The maximal fan power consumption is the value induced by the maximal fan speed and the considered power leakage is the one observed at the temperature targeted by the cooling system, i.e. 50 °C or 60 °C. As the maximal power consumption is considered, the savings expressed as percentage of the fan consumption and power leakage are in fact a pessimistic lower bound. Additionally, power savings can be expressed relatively to the overall system consumption. Table 5.3 shows two distinct cases. Either the saving obtained by DFaCE versus the thermal-directed system targeting 50 °C are higher than the saving obtained when the thermal-direct target 60 °C or the contrary.

In the first case, the temperature generated by the application is greater than 60 °C. It forces the thermal-directed to maintain the fan at full speed longer than if the target was 60 °C. It explains why DFaCE achieves more power savings against

the 50 °C target than against the 60 °C one.

In the second case, the temperatures generated by the applications are comprised between 50 °C and 60 °C. DFaCE achieves almost no power saving on LU or SP when considering 50 °C thermal-directed target. It is because DFaCE learned fan speed are almost the same as the average fan speed obtained by the thermal-directed system. However, DFaCE still is able, to obtain power savings when considering the 60 °C target, mainly because it can save more power leakage than the 50 °C target. Indeed, if the application generated temperature is comprised between 50 °C and 60 °C there is no need for the thermal-directed policy to speed the fans more often than with the 50 °C target. The savings are then not obtained on the fan speed slowing down but rather on the lowered power leakage since a lower temperature is maintained by DFaCE.

The presented numbers show that DFaCE is able to significantly improve the power consumption of the fans and power leakage. However, because the cooling sub-system does not represent a major part of the overall system power consumption, the achieved savings only represents an average of 3% of the full system consumption. It is reasonable to expect more gains in computers where the fans account for a larger part of the total power consumption, such as the ones used by [166]. Even though the power savings regarding a single machine are marginal, the fans are able to limit the processor power leakage. On the used hardware it can represent at most 23% of the total CPU power consumption, purely wasted. It is why manufacturers designed multiple states where some processor parts are shutdown to limit the leakage. Shutdown electronic does not draw power, thus do not leak. The different states, presented in the next chapter, are either idle states or different performance states. The idle states aim to shutdown unused or non mandatory elements, regarding the current processor usage. This allows to reduce the leakage. The performance states, by lowering the stress put on the processor, reduce the heat generation thus the leakage. Even though, performance states are mainly used to adapt the processor running frequencies, as shown in Part II and Part III, they transparently reduce the impact of leakage on the overall processor power consumption. The current chapter introduced the knowledge of processor power leakage and hints on processor dynamic power. The next chapter pursue in that direction. It exposes the most common model for processor power consumption, as well as all the different CPU state and how they impact the processor power model. Finally, small assembly instructions benchmarks are executed to acknowledge how processors actually consume power.

Chapter 6

CPU And Its Environment

Until 2006, manufacturers doubled each eighteen months the transistor density on their processor families. The increase reached a point where any standard cooling system, like fans presented above, were not able to keep the CPU in an acceptable temperature range. As an example, Pentium D 960 had a 130W TDP alone. After three generations of hardware design, P5, P6 and Netburst, Intel decided to re-design the Pentium M family for multi-core. It gave birth to the Core family processors. The new family was offering a Thermal Design Power (TDP) ranging between 10 to 150W. The 150W was obtained on the extreme editions of the family which were quad-core processors and no longer single core as the Pentium D. Among the enhancement and addition done to the Core processor family, power management features were added. Under the name of Intel SpeedStep [4] and TurboBoost [1], leverages were offered to the operating system to manage the CPU operating frequency and efficiently encounter a wide range of situations regarding power consumption. All the means offered to the OS to manage the CPU power consumption are presented below. Though a lot of enhancements were performed on CPU thermal dissipation and power consumption, it still is acknowledged as a the main consumer [44, 50] in most configurations. Therefore a thorough study on how the CPU consumes power and energy is conducted. Such an insight will help designing techniques, presented in Part II and Part III, using the means exposed to the OS to efficiently manage energy consumption regarding the CPU activity.

6.1 Processor Power Model

Before presenting the means exposed to the Operating System to regulate power consumption, the CPU power consumption has to be understood. According to, [27, 42, 50, 69, 73, 162, 165], the most common model for CPU power consumption is as follows :

$$P = P_{dynamic} + P_{static} \tag{6.1}$$

Where $P_{dynamic}$ is mainly induced by the use of the CPU, the higher the usage the higher the consumption is. On the other hand P_{static} comes from hardware imperfections such as leakage, shoot through current or parasitic wire capacitance. Generally the P_{static} is considered static and invariant whereas the $P_{dynamic}$ varies regarding the CPU activity as shown in Equation 6.2.

$$P_{dynamic} = A \times C \times V^2 \times f \tag{6.2}$$

The activity variation is expressed by A which is the activity factor, quantified as the percentage of active gates. For example if a parallel application is using all the cores, the resulting activity factor will be higher than a sequential application using a sole core. V and f are respectively the voltage supply and the operating

frequency. Finally C is the total capacitance. C can be expressed as the sum of all the gates capacitance. It can be seen in Equation 6.2 that $P_{dynamic}$ is quadratic to the voltage. Lower the voltage means important power saving. It will be seen in the next section that the OS sees the operating frequency and voltage as a couple called P-state. Therefore changing the P-state changes the frequency and voltage, impacting the overall power consumption P because $P_{dynamic}$ is strongly impacted.

Yet the dynamic power is not the sole actor in the overall processor power consumption. It has been shown in Chapter 5 that the static power is linear to the temperature. Breaking down the overall CPU power consumption means leveraging significant savings on the dynamic power to compensate the static impact on energy. That is why reducing the static power consumption is also important when seeking processor power reduction. As it will be explained in the next section, as P-state above, processors expose additional states, where sub-parts of the fabric are shutdown. Parts that are not powered cannot leak, inducing power saving on P_{static}.

As hinted above, the Operating System has a set of leverage, which are presented in the next section, to control either the dynamic power or the static power consumption. By taking advantage on both, the OS is able to control the overall power consumption regarding different usage scenario.

6.2 Advanced Configuration and Power Interface (ACPI)

The ACPI [5] provides an open standard for devices configurations and power management by the operating system. Initiated by Intel, Microsoft and Toshiba, it defines platform independent interface for hardware discovery, configuration and power monitoring and management. It gives end users the ability to control, for example CPU operating frequency, fan speed or to monitor CPU or GPU temperatures. In addition to expose hardware feature through dedicate API, the ACPI allows Operating System directed Power Management (OSPM).

6.2.1 OSPM States

OSPM allows the operating system to manage its power consumption regarding the state it is in. For example, if the user is not using its computer for a long period of time, there is no need to operate at full speed. The OS can decide to put itself in a transient state allowing him to consumes less power by shutting down non mandatory features and let it decides to resume the full speed state or go deeper in features deactivation. To do so, a tree of different states is exposed to the OS as shown in Figure 6.1.

The global states describe the whole system power modes. $G0$ corresponds to the working state and $G1$ corresponds to sleep states when no one is using the system. While being used, the CPU faces different utilization. $C0$ corresponds to the state where the processor is executing instructions. The other states $C1$ to $C7$ correspond to idle states.

Idle states must not be mistaken with the Sleep states. Idle states only concern the processor whereas Sleep states impact the global machines.

As an example, for Intel Core i7 family [81], in $C1$, the processor cores are halted and processor cache coherence is maintained. In $C3$, each core flushes the contents of their first level of instruction and data cache, along with the second level of cache

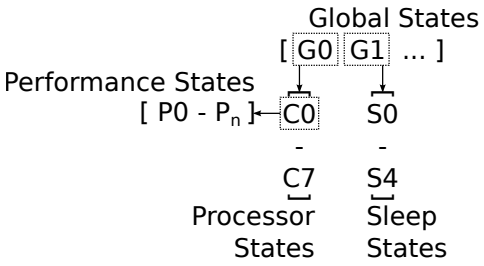


Figure 6.1: ACPI state tree.

Frequency	Voltage	P-State
1.6 GHz	1.484 V	P0
1.4 GHz	1.420 V	P1
1.2 GHz	1.276 V	P2
1.0 GHz	1.164 V	P3
800 MHz	1.036 V	P4
600 MHz	0.956 V	P5

Table 6.1: Intel Pentium M at 1.6GHz P-state detail.

into the shared last level of cache. The cores maintain their architectural states. All core clocks are stopped. Finally, in $C6$, the cores architectural states are saved in a dedicated SRAM on the chip. Once it is completed, the cores power supplies are shutdown. One can notice that the Intel Core i7 family does not implement each C-state, each manufacturer decide whether or not they implement each state or only a subset. But the deeper the C-state, the longer the latency is to put the processor back in $C0$.

If the processor is executing instructions, different performance states can be used. Generally called P-states, they correspond to a CPU operating point. A P-state is a pair of an operating frequency and a voltage. The frequency-voltage matching pair comes from transistor physics. Lower voltage implies slower transistor commutation speed leading to an increased latency of CPU operations, inducing lower operating frequency. Generally each P-state associates a unique frequency and a unique voltage as shown in Table 6.1 for an Intel Pentium M at 1.6GHz [83].

However a unique voltage can enable several operating frequencies, for example, in Table 6.1, $1.420V$ is the efficient lower bound needed to sustain the $1.4GHz$ operating frequency. The CPU could also be powered with $1.484V$ and still use the $1.4GHz$ CPU frequency. To shift between P-states, the processor asks the voltage regulator to scale to the correct voltage according to the selected frequency. If it is an ascending shift, meaning switching to higher frequencies, the voltage has to scale up to meet the requirements as shown in Figure 6.2. If it is a descending shift, the frequency can be immediately switched without waiting for the voltage to reach the correct level as shown in Figure 6.2. However, as power consumption efficiency is sought by manufacturers, the voltage scales down to the matching voltage defined by the P-state. Each voltage transitions are not instantaneous since voltage regulators are controlled systems. A finer study on commutations times between frequencies is conducted in Section 8.3. It shows that the commutation time must not be neglected. The frequency cannot be switched too frequently otherwise the toll on

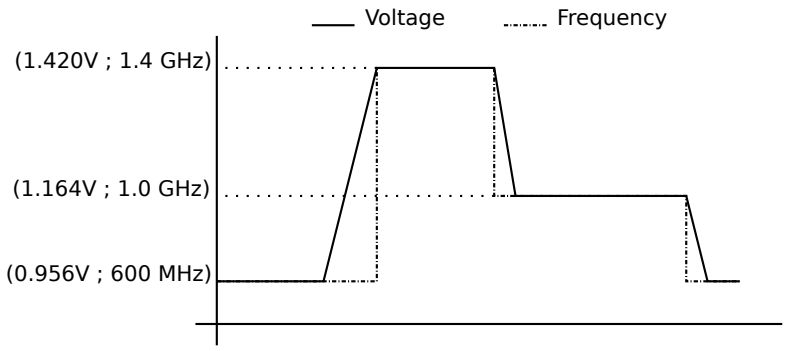


Figure 6.2: Voltage and Frequency up-scaling/down-scaling behavior.

processor power consumption is huge. In addition, the study also shows that the execution pipeline has to be stopped to take the new frequency into account. It enforces that frequency shifts have to be carefully performed.

Figure 6.2 displays voltage-frequency theoretical scaling during an application execution. A more realistic voltage and frequency scaling can be found in [96].

As for Idle states and Sleep states, P-states must not be confused with T-states. The T-state, called throttling state, is the first attempt to let the hardware modulate its operating frequency. Originally designed to prevent the processor from reaching critical temperatures, the T-state allowed a logical division of the base clock. For example, if the base clock is divided by two, only half of the clock ticks will trig the processor logic. It insures lower levels of stress, making it dissipates less energy and reducing the die temperature. Though efficient to manage processor temperature, it was not intended to modulate the processor power consumptions, as it can be done with the P-state and Dynamic voltage Frequency Scaling drivers in Part II.

The operating system has several ways to tweak on the fly the processor energy consumption. It can even decide to shutdown the whole system. Processor states (C-state) and performance states (P-state) can be set regarding the processor/core usage. To illustrate how both P-state and C-state are linked, PowerTop [82] was used to monitor the processor activity while literally writing this thesis for $20s$. The PowerTop report is displayed in Table 6.2. The processor was an Intel Core i3-3227U. During PowerTop monitoring, several applications were running. The set-up induced an overall processor usage of 15%.

The tested processor exposes fourteen different P-states and three processor family specific C-states, in addition to $C0$ and $C1$. The usage of the different P-states and C-states are exposed per core. There is only two columns Core 0 and 1 since the studied processor only has two physical cores. It will be seen below how P-states and C-state are handled in multi-core chip environment. As only thesis writing was performed, the processor was not put in huge stress explaining why only the lower frequency was used. Even though the processor was used at 15% in average, processor cores were in idle state most of the time. And while the cores were idle, the deepest idle state was used most of the time, meaning core consumption shutdown [79]. By using the means to lower power consumption the Operating system was able to, transparently, limit the battery draining to 10W. If applications more CPU demanding are run, like one of the arithmetic benchmark used later in Section 6.5, the operating system is forced to put each core in $C0$ using the highest frequency for 98.2% and 99% of the profiling period respectively for Core 0 and 1.

P-state		
Name	Core 0	Core 1
Turbo Mode	2,5%	1,7%
1,91 GHz	0,0%	0,0%
1,80 GHz	0,0%	0,0%
1,71 GHz	0,1%	0,1%
1,60 GHz	0,0%	0,0%
1500 MHz	0,0%	0,0%
1400 MHz	0,0%	0,0%
1300 MHz	0,0%	0,0%
1200 MHz	0,0%	0,0%
1100 MHz	0,0%	0,0%
1000 MHz	0,0%	0,0%
900 MHz	0,0%	0,0%
800 MHz	0,0%	0,0%
779 MHz	22,3%	11,4%
Idle	75,0%	86,7%

C-state		
Name	Core 0	Core 1
C3 (cc3)	0,7%	0,9%
C6 (cc6)	0,0%	0,0%
C7 (cc7)	74.8%	72.6%

Table 6.2: P-state and C-state usage while writing this PhD thesis.

The intense need of computation prevents the OS from using idle states, making the processor drains $18W$ from the battery.

In Table 6.2 both processor cores spent a different amount of time in the different states. There is only one voltage line managed by the voltage regulator for the entire processor. A unique voltage supplies then the different processor cores. It means that only one P-state can be used at a time for all processor core. One question arises: How does the operating system manage different P-states for the different cores. The same question can be translated to C-states, how does the OS decide to shutdown cores shared resources?. Decision mechanism are implemented in the frequency driver to answer those questions. They are presented in the next section.

6.2.2 P-state, C-state and Multi-core Chip

After 2006 to stop the frequency race and break down the unsustainable processor Thermal Design Power (TDP), manufacturers shifted from a unique complex execution pipeline to multiple cores. By compensating the frequency drop with parallelism, the trend shows that manufacturers produce processors with an increasing number of cores, even maximizing the parallelism by using multiple physical thread per processor cores. Yet only one voltage regulator remains, allowing only one voltage and frequency operating point for the overall processor.

P-states Previously in PowerTop report, showed in Table 6.2, Core 0 and Core 1 have a different usage of the Turbo Mode and the 779 Mhz frequencies. It is possible for PowerTop to expose different usage per core for one frequency because the *ACPI-cpufreq* driver *sysfs* interface, exposes means to manage the frequency per cores. As said above, only one voltage regulator feeds a single voltage level to the processor. It implies a single frequency common to all processor cores at a time. That is why the *ACPI-cpufreq* driver have the final word. It selects the maximum

between the requested frequency shift and the one already used. Regarding the P-state, it is the lowest P-state which will be selected. For example, if a CPU has four cores and each are requesting respectively $1.2GHz$ - $P2$, $0.9GHz$ - $P4$, $0.9GHz$ - $P4$, $1.0GHz$ - $P3$, the frequency which will be set will be $1.2GHz$ - $P2$. Therefore, if a system wants to modulate the frequency per processor core, it must be aware that the applied frequency is not the pure reflection of the asked ones.

Machines or processors where multiple frequency domains coexist, can be created. The best examples are machines with multiple processors or many-core architectures like the Intel SCC platform [71, 119]. A frequency domain is composed by the cores bearing the same voltage/frequency operating points. In the multiprocessor case, each core belonging to the same processor are in the same frequency domain. On the Intel SCC, 48 cores are on the same die, but the on-chip regulator allows the user to independently control the frequency of 8 cores clusters [136]. However, within each frequency domain, the frequency is chosen as described above.

C-states Contrary to P-states, each core can have a different C-state. However, there is a restriction with hyper-threaded processors. An hyper-threaded processor implies two hardware threads per core. Though each thread can choose the best C-state regarding its state, only one C-state can be used for both threads. Halting the execution, flushing caches or even power off cores are actions that can be performed as a result of choosing a C-state. De facto, one thread cannot ask to power off its execution core while the other thread is still computing instructions. Hence, both threads C-states are compared and only the minimum is applied to the core, which is the least aggressive in terms of feature shutdown.

The operating system and the user, with the ACPI, have a wide range of means to optimize the CPU power consumption. However, the different means have to be separated in two categories. The ones directly usable by the users and the ones transparent to him. On the one hand, a user can force a specific P-state through the frequency driver. Part II shows how it can be efficiently used through the presentation of several systems. With a different degree of intelligence, they try to adapt the processor P-state through time regarding processor load to lower processor energy consumption. On the other hand, the OS decides which C-state to use for each processor core without notifying it to the user. Though a user can find a way to modify the different C-states, he must have a complete view of the processor state and what is to come to prevent his choices from strongly impacting the processor execution flow.

Modulating processor P-states regarding computation needs is only meaningful if the processors cores are not in idle state. Indeed, writing this thesis does not put enough stress on the CPU to force most of the cores out of $C7$, where the cores are powered off [79]. That is why optimization solutions presented in Part II and Part III are performed during application executions.

In a nutshell, P-state modulation is used when the processor is in $C0$ state, i.e while performing work. It will then directly impact $P_{dynamic}$. The C-states aim to shutdown processor part that are not needed regarding its state. Parts that are shutdown cannot leak, it then directly impact P_{static}.

The processor power model, exposed in equations 6.1 and 6.2, helped understand why utilizing the different P-states or C-state will help reducing the CPU power consumption. However, they do not state how the processor consumes power and energy

when executing applications. Moreover, having a complete understanding of application power and energy consumption is complex due to the obfuscation brought by different level of abstractions inherent to scientific applications. Consequently, to focus on processor consumption, a fine grain power and energy consumption study is conducted in the next Sections 6.4 and 6.5. It will give an insight on how the processor consumes power regarding different types of instructions. Such insight can be later used to better understand application energy footprint as show in Sections 8.1 and 8.2.

However, to produce the different measurements a robust evaluation methodology is needed. Therefore, before jumping to the insight on power and energy consumption of multiple instructions, the next section presents the methodology used to achieve precise power and energy characterization.

6.3 Micro-benchmarking Characterization

Application performance or energy optimizations are complex processes that can take multiple forms. By fully understanding the application, and then performing optimizations to ease the computational process. For example, designing good heuristics is a solution. Another technique starts by characterizing in detail the hardware used to execute the application. Then, the next step is to tweak the existing application by taking full advantage of features offered by the underlying architecture. Micro-benchmarking characterizations are used in such a way.

Micro-benchmarking systems [122, 125] intend to test and analyze very specific features offered by the hardware. For example, measuring the impact of data prefetchers or the power cost of an addition instruction. Gathering such insights on the hardware can help developers optimize applications or predict code snippets performances [124]. However, achieving fine grain measurement needs a robust methodology as the one presented in 6.3.1 to ensure the data quality.

As a recall, Section 6.1 exposed the processor power consumption model. However, the model does not clearly state how a CPU will consume power and energy regarding a real application execution. The power model also does not show how P-state can impact the overall CPU power and energy consumption. The goal of the benchmarking study is to gather such an insight. An application generally oscillate between data movements and computations. It was then natural to focus the micro-benchmarking study upon memory instructions, as well as arithmetic instructions. Results and observation are presented in Sections 6.4 and 6.5.

6.3.1 Measurement Methodology

Micro-benchmarking uses the same process as any measurement method. For example, when a function execution time has to be known, one puts a probing system at function start-up and exit and by subtraction obtains the function execution time. However, micro-benchmarking is targeting very specific hardware features, any outside noise or poor conception in the test environment system can lead to huge variation on the measured data. To prevent that it exists several ways to ensure measured information sanity by stabilizing the measurement environment.

The first thing to do is to repeat multiple times the benchmark as shown in Algorithm 3. Gathering measurement on several executions easily exposes any instability. Table 6.3 shows different executions of an synthetic benchmark build with

Algorithm 3 Several function execution timing procedure

```
for meta = 1 → nb_meta do
    start ← probe()
    run_function()
    stop ← probe()
    function_exectime[meta] ← stop − start
end for
```

ADDPS instructions as the one shown in Figure 6.3 or 6.6. The first row exposes ten repetitions of the benchmark as described in Algorithm 3. It can be clearly seen that it is not stable since each measurements is very different from the other. There is no way to know whether 6.35 or 9.2 seconds is the real benchmark execution time. Using statistical tools such as the standard deviation, solves that uncertainty. The standard variation roughly represents 20% of variation on the data set which is not good, a mean to stabilize each execution is then needed. The multiple iterations for gathering measurements are called *meta-iterations* in the remainder of the section.

ADDPS benchmark execution time in seconds										Standard Deviation
With Algorithm 3										
6.35	7.12	7.69	8.05	8.69	9.05	9.2	6.49	8.92	6.35	1.15
With Algorithm 4										
6.31	6.32	6.33	6.31	6.34	6.36	6.32	6.32	6.35	6.31	0.02

Table 6.3: ADDPS benchmark execution time for several consecutive executions.

Algorithm 4 Noise reduction

```
for meta = 1 → nb_meta do
    start ← probe()
    for repeat = 1 → nb_repeat do
        run_function()
    end for
    stop ← probe()
    function_exectime[meta] ← (stop−start) / nb_repeat
end for
```

There is no way to perfectly control the test environment, therefore outside events can disrupt the function execution. For example, the Operating System can move the function execution from one core to another, forcing cache misses, or provoke context switches. If variations exist, as in Table 6.3 first row, increasing the number of benchmark repetitions between the measurements point is a good way to solve the problem. It will make that potential additional time tend to zero. Provided that the number of repetitions is high enough, the new Algorithm 4 ensures that outside noises have limited impact on the measurements. Algorithm 4 was then used to re-evaluate the *ADDPS* benchmark execution time and the results are displayed in Table 6.3 second row. The associated standard deviation represents around 1% of variation on the data set which gives confidence on the quality of the measurements.

Performing good measurements is a complex and a long process to ensure mea-

sured data quality. Algorithm 4 presents the generic method to ensure that. It is implemented in MicroTools [20]. MicroTools is used for the micro-benchmarking studies presented in Sections 6.4 and 6.5. MicroTools compiles, executes, and monitors functions written in assembly code or binaries. As said above micro-benchmarking is used to test specific features which need very fine grain measurements. Therefore, MicroTools was chosen to execute designed functions because its capacity to launch assembly codes. It ensures that each wanted hardware features are triggered independently one at a time.

6.3.2 Test Environment

Model Number	X5650	E3-1240	D510
Architecture	Whestmere	SandyBridge	Bonnel
Processors	2	1	1
Cores/Proc.	6	4	2
Memory	8 Gb	4 Gb	2Gb
Measurement Granularity	Entire Machine	CPU & Entire Machine	Entire Machine
Use case	Memory Benchmarks	Arithmetic Benchmarks	Arithmetic Benchmarks

Table 6.4: Experimental Testbed.

As presented above, it was decided to focus the micro-benchmarking study upon memory instructions and arithmetic instructions since an application generally oscillates between data movements and computations.

Generally, accessing memory is considered as a common bottleneck. If huge amounts of data are fetched from the different level of caches or from the RAM, the memory bandwidth can be saturated. Consequently, the throughput is strongly impacted forcing the application to wait longer for its data. Therefore, the execution time and energy consumption regarding data movements from any level of the memory hierarchy, whether its bandwidth is saturated or not, is measured. The gathered insights are presented in Section 6.4.

On some configurations, saturating memory bandwidth was not possible. Table 6.4 shows the different configurations used for the micro-benchmarking study. The first configuration used was the single SandyBridge E3-1240 processor machine. However, the memory bandwidth was over-sized and even by executing up to four micro-benchmark in parallel the memory could not be saturated. The dual Westmere X5650 processors was then considered. By executing one micro-benchmark instance per CPU core bandwidth saturation was achieved.

After the memory study, the focus was put on arithmetic instructions, and how they consume power and energy regarding two different granularities. The measurements were performed at the processor and the entire machine scale as shown in Table 6.4. It will be seen in Section 6.5 that each granularity drastically reports different energy behaviors. Energy optimization decisions can then be different depending on the considered granularity. The focus was also put on potential difference between energy behaviors from the HPC and the embedded worlds. As they share the same desire, to reach the maximum computation capability within their con-

straints, it was then interesting to compare how both trends of processors consume power. The arithmetic instructions power consumption of the SandyBridge E3-1240 was compared to the low power Atom D510 and their differences are presented in Section 6.5.

6.4 Memory

The goal of the section is to present the energy consumption of the memory subsystem. It helps gathering insight on how memory intensive kernels scale regarding frequency and how they are consuming energy. In the past decades, CPU speed grew faster than the memory, that is why, in general, accessing the memory is considered as a bottleneck.

To test the memory energy consumption, the methodology previously presented in Section 6.3.1 was used. It was decided to directly use assembly code in order to have a complete control over the execution. The idea behind the benchmark is to test read-and-write energy cost when targeting different levels of the memory hierarchy. The benchmark iterates on multiple consecutive elements in a vector, and when all vector elements are either read or stored back in the vector the benchmark ends.

```
.L6:
    movaps 0(%rsi), %xmm0    #Load
    movaps 16(%rsi), %xmm1   #Load
    movaps %xmm2, 32(%rsi)   #Store
    movaps 48(%rsi), %xmm3   #Load
    movaps 64(%rsi), %xmm4   #Load
    # 20 elments of 4 bit are consumed,
    # jump to the next unread elements
    add $80, %rsi
    # remove 20 elemts from the iterator
    sub $20, %rdi
    # if the iterator > 0 continue,
    # otherwise exit
    jge .L6
```

Figure 6.3: Kernel Assembly instructions

Each instruction within the benchmark presented in Figure 6.3 was carefully selected. The *movaps* instructions are vectorized memory operations. As they put the maximum stress upon the memory subsystem, they were selected in order to allow the measurement procedure to saturate the memory bandwidth. The combination of load and store operations was done to represent a general program behavior. Generally, programs perform more reads than writes. As an example, a program performing a map-reduce, will perform more reads than writes. Finally the last three instructions, *add*, *sub* and *jge* are needed to iterate on vector elements. Their impact is measured before launching the real benchmark and subtracted to the full benchmark measurement. To do so, the benchmark is run without the *movaps* instructions and measured. The time and the energy obtained are then subtracted from the ones obtained when running the entire benchmark.

As a reminder, the goal of the section is to display the energy consumption

behavior of the memory hierarchy. Therefore, two cases have to be considered. Either the memory bandwidth is saturated or not. Each Figure 6.4 or 6.5 shows the energy cost per memory instruction for the different level of cache and the RAM. It was insured that the accessed vector only fit in the considered level of the memory hierarchy.

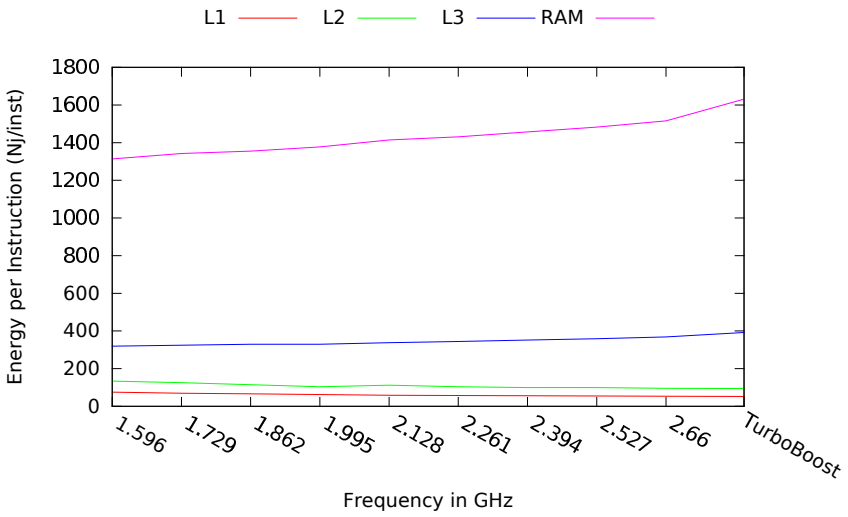


Figure 6.4: Energy consumption per memory instruction depending on the data location when the memory is saturated

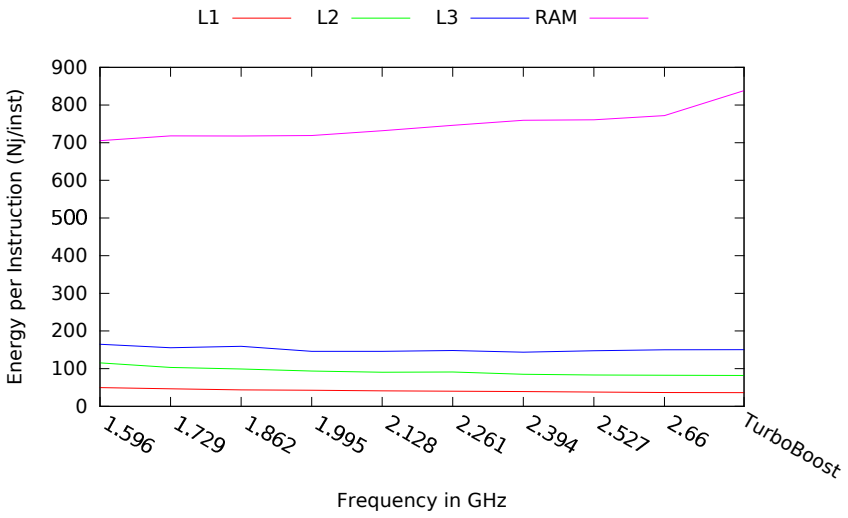


Figure 6.5: Energy consumption per memory instruction depending on the data location when the memory is not saturated

Figure 6.4 and 6.5 display the energy consumption while executing the micro-benchmark. The energy measurements were performed on the entire machine. For both figures, the x-axis represents processor frequencies and the y-axis the energy per instruction. Both figures convey two information. Firstly, the evolution of the energy cost across the processor frequency spectrum. Secondly, the instant energy cost.

It can be noticed in Figure 6.4 that accessing L3 and RAM consumes more energy when using high CPU frequencies than lower ones. It comes from the fact that the overall power consumption is linked to the P-states. Increasing the P-states, increases the voltage and the frequency, inducing a higher power consumption. Finally, as $E = P \times T$, if P increases the energy may also raise. When the memory is saturated, the data fetching latency for L3 and RAM remains constant across the processor frequencies, validating the energy increase. However, it is not the case for L1 and L2, the energy cost decreases with high frequencies. The decrease on the execution time counters the increasing power cost, explaining why the overall energy consumption decreases. Other observations are noticeable when the memory bandwidth is not saturated. The RAM energy cost still follows the same trend as the one seen in Figure 6.4. However, L3, L2 and L1 energy costs decrease with the increased frequencies. The root of that behavior is the same as the one explained above, the performance speed-up counters the increased power cost. When looking at instant energy cost, it cost almost twice the energy to access data when the memory is saturated.

In the end, when creating an application, developers must carefully design memory access phases because if the memory hierarchy is saturated, the overall application energy cost can be dramatically increased. Furthermore, the application energy cost can be also lowered by selecting a frequency best matching the memory access scenario as shown in Figure 6.4 and 6.5. For example, if the application is accessing high level of the memory, like L3 or RAM, the lowest frequency should be targeted whether it is saturated or not. Based on the current observations, the lowest frequency selection will be the default energy optimization when facing an application strongly dependent on the memory hierarchy as shown later in Section 8.1.

6.5 Arithmetic

```
.L6:
    mulpd %xmm0, %xmm0
    mulpd %xmm1, %xmm1
    mulpd %xmm2, %xmm2
    mulpd %xmm3, %xmm3
    mulpd %xmm4, %xmm4
    mulpd %xmm5, %xmm5
    mulpd %xmm6, %xmm6
    mulpd %xmm7, %xmm7
    # remove 8 iteration from the iterator
    sub $8, %rdi
    # if the iterator > 0 continue,
    # otherwise exit
    jge .L6
```

Figure 6.6: Micro-benchmark for arithmetic intensive execution

As said previously, scientific applications usually, load data, perform computations and store the computation results. They oscillate then between data movements and pure computation. It was previously shown how data movements can impact the machine energy consumption and how features along the data path can

help to reduce or worsen it. The focus is then put on the arithmetic instructions. Testing all possible arithmetic instructions to have a complete view is not the goal of the study, it rather is to test most common instructions that can be found in any applications. It was then decided to pick instructions from each category and measure their power and energy consumption. In total twelve instructions were chosen. Integer operations as *add*,*imul*,*idiv*, packed single precision operations as *addps* and *mulps*, packed double precision like *addpd* or *mulpd*, bitwise logic as *and* or *or* and finally jump operations *jmp*, *jnz*, and *ja*. The same micro-benchmarking methodology used in the two previous sections is applied for each selected instruction. Figure 6.6 shows the benchmark kernel for the instruction *mulpd*.

6.5.1 Instruction Clustering

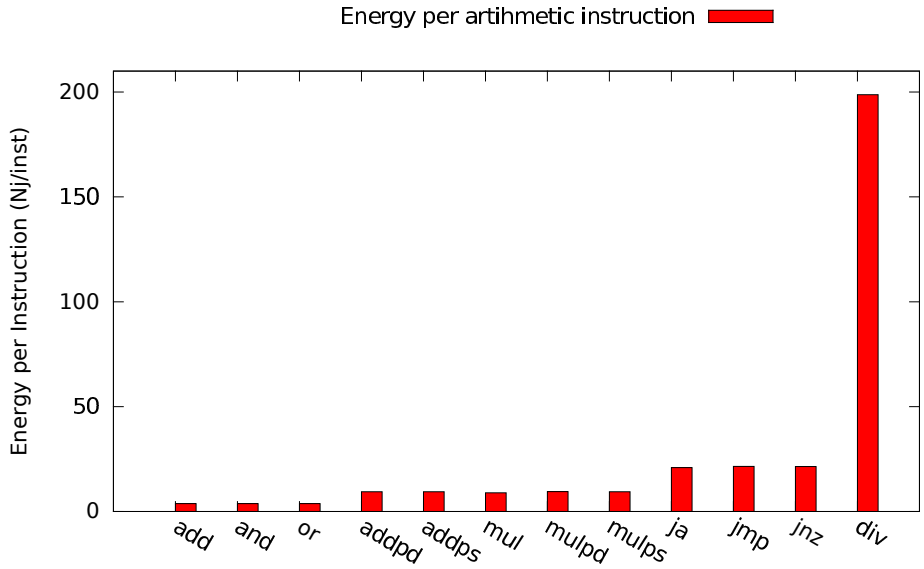


Figure 6.7: Arithmetic instruction energy consumption on E3-1240 at 3.3GHz

Contrary to the memory study, the measurements are performed at the scale of the processor using the SandyBridge dedicated power monitoring. It means that the power measured is only consumed by the processor and not by the entire machine. Figure 6.7 shows the energy consumption of the selected instruction set at the highest processor frequency. The x-axis represents the arithmetic instructions and the y-axis displays the energy per instruction. Indubitably the *div* operation is the one that costs the most energy. It is common knowledge that the *div* operation takes a very long time to be processed then costing to the CPU an important amount of energy. Apart from that outlier, three groups can be determined. One group comprised of the operations consuming the lowest amount of energy: *add*, *and*, *or*. Another, with the packed operations: *mulps*, *mulpd*, *addps* and *addpd*. Finally, a last one with the jump operations : *ja*, *jmp*, *jnz*. As a reminder, the energy consumption is the product of power and time. The root of that possible clustering comes either from the power consumptions or instructions execution times. However, the measured power consumption for each instruction on the same SandyBridge machine, displayed in table 6.6, is the same, even for the division instruction. The instruction execution time is then the root of the clustering. In [45], the author presents the

reciprocal throughput of the entire x86 instructions set measured on a wide range of architectures. The reciprocal throughput is defined as the average number of core clock per instruction for a series of independent instructions of the same kind in the same thread. The same procedure is performed during the micro-benchmarks execution. Achieved reciprocal throughputs are then computed for the twelve instructions and compared with Agner Fog's measurements [45]. Both throughputs are displayed in Table 6.5. The comparison with Agner Fog's measurements validates the evaluation method. It also shows that the energy clustering was only due to the different cycles per instruction.

Unit: cycle per instruction						
Instructions	add	and	or	imul	addps	addpd
Micro-benchmarking	0.49	0.49	0.49	1.01	1.02	1.01
From Agner Foh	0.5	0.5	0.5	1	1	1
Instructions	mulps	mulpd	ja	jmp	jnz	idiv
Micro-benchmarking	1.01	1.02	2	1.99	2	15.42
From Agner Foh	1	1	2	2	2	11-18

Table 6.5: Arithmetic instructions reciprocal throughput.

Unit: nJ per cycle						
Instructions	add	and	or	imul	addps	addpd
SandyBridge E3-1240	21.83	21.85	21.25	21.02	21.82	21.79
Atom D510	18.29	18.42	18.61	18.46	18.15	18.31
Instructions	mulps	mulpd	ja	jmp	jnz	idiv
SandyBridge E3-1240	21.56	21.60	22.04	22.42	22.44	21.34
Atom D510	18.11	18.59	18.17	18.14	18.21	18.11

Table 6.6: Arithmetic instructions power consumption

Though the instruction execution time explains the presented clustering, it is surprising that the power consumption is constant disregarding the instructions used. It means that when executing an addition or a jump at each processor cycle, all the CPU features are powered on, even though both instructions use drastically different paths. During instruction execution some processors use clock gating to switch off untriggered processor parts. Different instructions have then different power costs and by taking advantage of that, application can be build in a power efficient way. Such feature seems not to be available on the SandyBridge since the power cost of the different instruction is almost constant. It can be understandble since the processor is intended for the HPC world, and performances must not be impacted in any way. However, even with a low power processor, as shown in Table 6.6, seems not to use clock gating. The conditional tense was used in the previous sentences, because the power probe resolution can be the reason to the lack of power difference between each instruction. Nonetheless, energy optimization regarding arithmetic instruction selection can be performed. Using an integer instead of a float cost, less energy since only integer operation will be used. By Limiting the number of conditions, apart from potential branching error, the impact of jump

operation on energy costs is lowered.

Finally, Figure 6.7 helps understanding why it exists disparities in term of energy consumption among computational intensive applications.

In the end, power consumptions for different arithmetic instructions are constant for a fixed frequency. If the frequency is increased, the various instructions power consumptions increase at the same rate. Then, if energy has to be saved, and based on the trends exposed in Figure 6.8, people would tend to target low frequency. However, the energy is a tradeoff between time and power. If the speed-up on time is able to counter the power increase, then high frequencies has to be targeted. However, if the power decreases is able to counter the time increase, then, low frequencies have to be used. A crucial question remains: race to finish or not ?

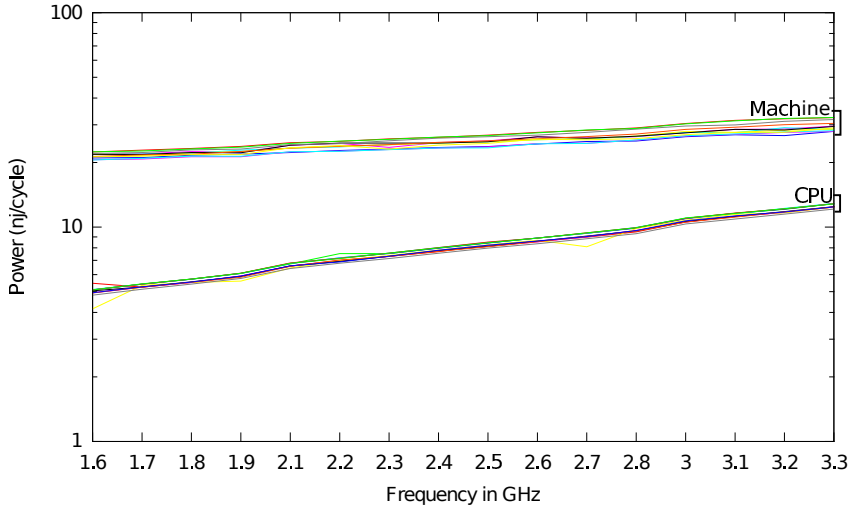
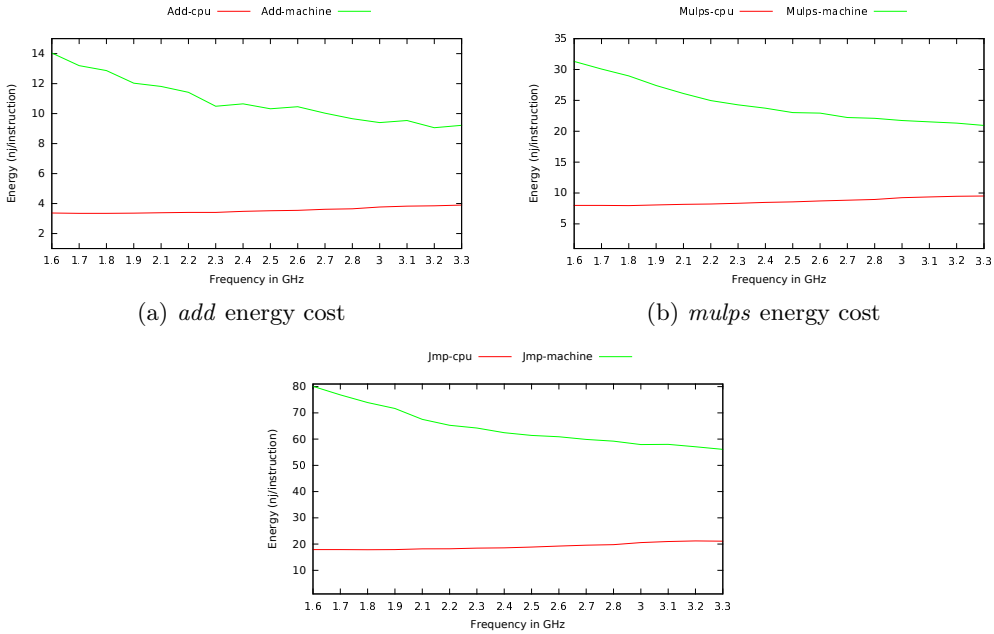


Figure 6.8: Arithmetic instruction power consumption evolution on SandyBridge E3-1240

6.5.2 To Speed Or Not To Speed ?

Starting with the SandyBridge architecture, new registers were added allowing CPU power consumption monitoring. For anterior architectures, as Westmere, using an external digital power meter is the only way to measure power and energy consumption. However, it measures the entire machine power and energy consumption. Yet, using both at the same time allows to acknowledge the real impact of the processor consumption on the full machine. Figure 6.8 was generated while using both granularity. The different instruction power costs rather stay similar across the different frequencies and the same behavior can be noticed with the different probes. The higher the frequency, the higher the power cost. Figure 6.8 also shows that the processor consumption directly influences the overall machine power consumption. It confirms that the CPU is determinant when targeting application power or energy optimizations.

According also to Figure 6.8 high frequency should never be selected when aiming for power reduction. However it is not always the case for energy optimization. The power is not the sole actor in energy consumption. As previously hinted, if the execution time speed-up, when selecting a higher frequency, is able to counter the power increase, then energy savings are possible. It can be clearly seen in Figure



(a) *add* energy cost

(b) *mulps* energy cost

(c) *jmp* energy cost

Figure 6.9: Energy consumption of one instruction per class.

6.9 where the machine energy consumptions decrease with higher frequencies. Three different instructions, one from each cluster defined above, were chosen to show that such behavior is the general behavior of heavy computational applications. It will be used later in Section 8.1 to classify applications and used in Chapters 9, 10 and 11 to design energy optimization mechanism.

However, targeting higher processor speed does not always translate in energy consumption as shown by the processor energy trend for the three instructions as displayed in 6.9. Indeed, the speed-ups obtained by switching from the lowest to the highest frequency are 2.06, 2.06, and 2.07 respectively for *add*, *mulps*, and *jmp* and the power consumptions scaling are 2.38,2.44, and 2.43. The speed ups are not high enough to counter the power increase forcing the energy consumption to scale up over the frequencies. That behavior can also be demonstrated as follows. First, as a remainder, Equation 6.3 shows the processor's dynamic power model. Changing the processor frequency is obtained when the P-state changes. A P-state is a couple of a frequency and a voltage. Increasing the frequency increases the voltage.

$$P_{dynamic} = A \times C \times V^2 \times f \tag{6.3}$$

If two P-states are considered, $P1$ and $P2$ with $P1 > P2$, it means that the voltages and frequencies follow the relation: $V1 > V2$ and $f1 > f2$ with $V1, F1 \in P1$ and $V2, F2 \in P2$. Using these relation, Equation 6.4 represents the power scaling factor.

$$\frac{P_2}{P_1} = \frac{A \times C \times V2^2 \times f2}{A \times C \times V1^2 \times f1} \tag{6.4}$$

As the same benchmark is executed on the same processor, triggering the same arithmetic units, the activity factor A and the capacitance C can be considered equal for different executions of the same benchmark. The power scaling factor can be reduced to Equation 6.5.

$$\frac{P_2}{P_1} = \frac{V2^2 \times f2}{V1^2 \times f1} \tag{6.5}$$

The micro-benchmarks were designed not to have external dependencies, their execution times are fully bounded to the CPU operating frequency. It means, the micro-benchmark execution time speed-up can be at most equal to the frequency ratio, as shown in Equation 6.6

$$speedup = \frac{ExecTime_2}{ExecTime_1} \approx \frac{f2}{f1} \tag{6.6}$$

If, Equation eq:SpeedUPvsPowerRatio is created by injecting 6.6 into 6.5, it can be seen that the power scaling factor is greater than the speed-up factor, since $\frac{V2^2}{V1^2} > 1$.

$$\frac{P_2}{P_1} = \frac{V2^2}{V1^2} * speedup \tag{6.7}$$

However the demonstration is only true if processor static power is negligible compared to the dynamic power. To produce Figure 6.9, one instance of micro-benchmark per processor core were executed, putting the CPU under a significant stress to remove the static power from the equation. However if it is not possible, the energy consumption will behave like the overall machine trend shows in Figure 6.9. Such a behavior validates the conclusion of Yuki *et. al.* [169] where the authors state that compiling for speed is compiling for energy. However, it only validates if the considered application is purely cpu-bound, as shown in Figure 6.9, and if the power static is not negligible. Compiling for speed on a memory bound applications will not drastically change their energy consumption as discussed in the previous section, then questioning the validity of the race to finish policy advocated by Yuki *et. al.* [169].

Though the processor consumes the same amount of power disregarding the arithmetic instruction type, the energy clustering can still be used to bring hints to the developer to create energy efficient application. Like for memory instruction, where the lowest frequency offers energy saving, for arithmetic instruction it is the highest one that grants energy savings in most cases. More over the impact of the processor power on the overall machine power, confirms that the CPU is a strong actor in power and energy consumption but trying to optimize its consumption is not always a matter of racing the application to finish, or slowing the processor as much as possible.

Chapter 7

Conclusion

This thesis started with a simple assessment, each piece of hardware consumes power and energy. Depending on the usage scenario, one piece of hardware can be subject to more pressure than the others. For example, for a data copy, only the disk will be used, then its energy consumption can be greater than what the other components consume, when in another case scenario it would be the element with the lowest energy consumption. An analysis of different hardware power consumers within a simple machine was then presented and, for each of them, possible energy optimization scenarios were described. It was demonstrated for RAM dimms and HDD, either the technology state does not allow to design practical optimization or all possible optimizations already are embedded in the hardware. Nevertheless, the fast growing need for check-pointing, data redundancy or burst buffers will dramatically increase the RAM and disk space. It will force manufacturers, in the future, to reconsider the optimization opportunities for RAM dimms and Hard drives. Contrary to RAM dimms and HDD, for the fan, opportunities for energy reduction exist. Though the system takes a long time to converge to the optimal fan settings, it enlightens an important fact: the leakage power must not be neglected. It represents 23% of the overall processor consumption at high temperature. By reducing the temperature power leakage is reduced, lowering the processor energy consumption which is the main interest of the presented fan speeds modulation technique. The last but not least, the processor is acknowledged to be the main power consumer explaining why multiple possibilities to optimize its power consumption exist. Originally these opportunities were designed to reduce the die temperature since the fans could not cool enough the processor to prevent it from hardware failure due to the heat. Nowadays, processors expose a range of possible operating points to allow the Operating System or any user to adapt them regarding the processor usage. Depending on the scenario, the processor does not have the same energy consumption, as it was shown with the micro-benchmarking study. Memory oriented instructions tend to consume less energy at lower frequencies. The study showed that the energy behaviors of arithmetic instructions under the frequency spectrum strongly change depending on the measure granularity. At the processor scale, it can be seen that the highest frequency induces the highest energy consumption, when at machine scale it induces the lowest. A question naturally arises: should an application always be run at the highest frequency to reduce its energy consumption or not ? That question cannot be answered with only assembly instructions, the overall application has to be studied and depending on its resources usage scenario, dedicated energy optimization can be designed. This is why the next part starts by studying the application types and tries to derive a classification to better understand their energy behavior. Based on these observations multiple systems are designed to reduce any application energy consumption.

Part II

DVFS single chip

Chapter 8

Introduction

Each piece of hardware consumes a certain amount of energy. Unfortunately, the majority does not expose leverage in order to optimize their energy consumption. For example, RAM dims do not support run-time frequency changes. They have to be changed during the boot phase of the machine, which is not feasible for huge HPC systems. For other components, such as fans or hard drives, complex systems must be created, when sometimes better hardware solutions already exist. Indeed, finding the sweet spot for fans is time consuming since the configuration time raises exponentially with the number of fans. For hard-drives, hardware optimization are already proposed by all manufacturers, such as lower platter ration speed coupled with optimized transfer rates and placing algorithms [40] or the Solid State Disk technology which has the best throughput per watt ratio [152]. Hence, further energy optimization study for these pieces of hardware is not automatically the priority.

As seen before, the energy used by the CPU is accounted for almost half of the machine energy consumption. Moreover, for the current processor generation, many tools and measuring probes are available, as presented in Chapter 6, to develop energy reduction methods. It was natural to first focus on energy optimizations for the CPU.

Even though the study of energy consumption of a single instruction gives a good idea on how the energy is consumed, there is a major difference with a full application. It misses the instruction execution density. In Figure 6.8, the instructions are believed to be executed alone in the pipeline, but recent architectures decode and push in the execution pipeline up to four different instructions simultaneously. Such parallelism drastically changes an application energy footprint.

Instruction level parallelism is not the only factor that impacts the energy consumption. It is also important to consider what the application is performing. Generally an application intends to produce results based on a data set. The classic operation is to first load the data set, then perform all the needed computation, and finally store the results. An application can thus be divided roughly into two aspects: one related to computation and one to data movement. These states are respectively called CPU-bound and memory-bound phases.

To get a better idea on how a complete application behaves in terms of energy consumption, two benchmark programs from the SPEC2006 benchmark suite [157] were executed. The selected benchmarks Libquantum and Gromacs respectively are memory-bound and CPU-bound. The results on the different frequencies are summarized in Figures 8.2 and 8.3. Both figures show the execution time and energy consumption obtained by each application on different frequencies. By looking at both Figures 8.2 and 8.3, one can easily find a way to optimize the energy consumption of each application. For Libquantum, a 26% saving of the energy consumption is obtained by running the application at the lowest frequency. For Gromacs, it is a 20% reduction when selecting the frequency 2.66Ghz.

As an example, measurements were performed on each frequency. In general, the information is not available before hand and the best frequency has to be computed during the execution of the application. Finding the lowest energy consumption is here straightforward, but later in the document, additional constraints has to be considered to get a realistic decision as exposed in Chapters 9, 10, and 11. The selected frequency is not always the highest or the lowest frequency.

A wide range of possibilities to perform energy and power reductions at the scale of a single processor exists. Application source code energy predictions [103, 34], code optimizing during the compilation [75, 32, 88], or optimization at run-time using DVFS [65, 28, 168, 70, 33, 116, 95]. Run-time optimizations are the most widely used, since it allows the optimization process to access actual power and energy consumption and react accordingly. That is why all the presented work within this section: REST, UtoPeak and FoREST are dynamic optimizations. Indeed, it is very difficult to perform optimization or prediction without accurate information on how an application execution consumes power and energy. Furthermore, as hinted in the previous chapter, optimizing the energy consumption is not only a matter of race to finish or not, depending on the application behavior different decisions can be made. This is why the application energy consumption has to be understood either by trying to find similarities between their energy consumption, or by finding inside their source code what is the reason behind their energy consumption.

8.1 Application Trends

Scientific applications as a whole try to tackle defined problems, which generally are composed of smaller issues. Some are common to several big applications. To prevent developers from writing different algorithms to solve the same problem, sometimes not in the optimal manner, a collection of the most frequent problems has been constituted: the numerical recipes [156]. It also provides the most efficient algorithms to solve them.

Even if a scientific application is solved in an efficient way by taking advantage of all the possible numerical recipes, it still has to be adapted to the hardware to leverage the maximum performance and provide the fastest time to solution. The adaptation can be time consuming and in some cases not affordable. To break down the study time, applications with similar execution flows may be compared. To take advantage of the similarities, developers can then apply the previously used optimizations for their own applications. By organizing applications by particular trend, developers have a better understanding of their programs and hardware.

But application classifications usually are intended to increase performance and not to optimize energy consumption. The goal of this section is to find common trends in application execution time as well as in energy consumption. Classification can strongly help in the quest of finding the sweet spot combining the best execution time and the lowest energy consumption. By executing the whole sequential benchmark suite SPEC2006 [157] and an industrial code [11, 17], three kinds of trends are found: external resources boundness, compute boundness, and balanced boundness.

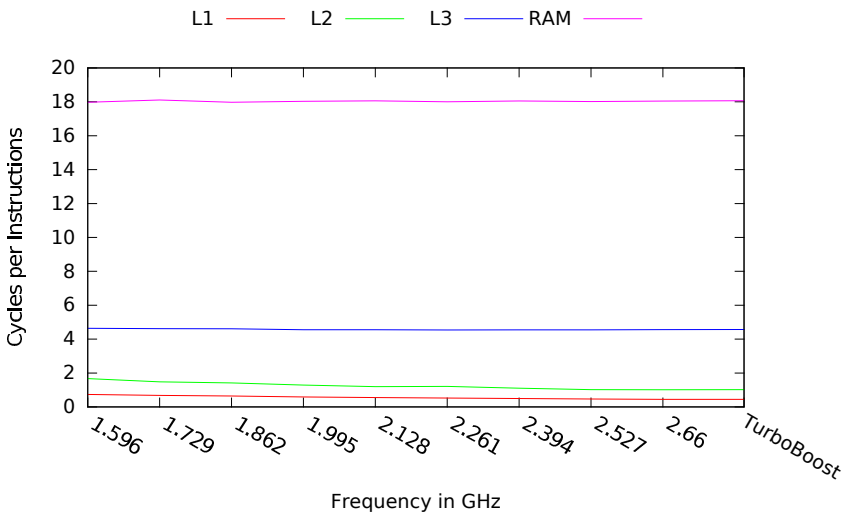


Figure 8.1: Data fetching latency from each cache level.

8.1.1 External Resources Boundness

External resource boundness characterizes applications using data stored in any place out of the processor caches. For example the RAM, hard-drives or the network are external resources. As the stored data locations are not controlled by the processor clock, any change on it will not impact the fetching latency.

A good example was shown in Chapter 6 with Figure 8.1. The data fetching latencies from the first two cache levels scale with the processor frequency whereas the last level of cache does not. When facing such a behavior, the pipeline stalls, basically doing nothing, waiting for the data, so there is no need for the processor to be idling fast. Lowering the frequency will not impact the pipeline throughput. In Figure 8.2, lowering the frequency results in the same execution time. Moreover, as a reminder, a power setting is associated to a frequency, so the power consumption is lowered when reducing frequency. As the execution time remains constant and $E = P \times T$, lowering the operating frequency also means lowering the energy consumption. At the end, when considering such an application, it is easy to find the best frequency setting to get the lowest energy consumption: always use the lowest possible frequency.

This is only true in environments where the external data fetching frequency is lower than the lowest processor frequency. If it is not the case, the execution time will be affected since the external resource no longer is the bottleneck. An example is shown in Figure 8.5a. The considered function, *full_verify*, can be considered as memory-bounded, since, when looking at the source code, it only performs vector traversal. Though *full_verify* is acknowledged as a memory-bounded function, it does not saturate the memory subsystem, as in the different execution time trends in Figures 8.5a and 8.2. The *full_verify* execution time scales down with increasing frequencies instead of remaining constant.

Memory-bound applications have a very specific trend, making them easily noticeable and easy to optimize regarding energy consumption. As shown with *full_verify*, the source code alone is not sufficient to prove that a specific code region belongs to a specific trend. It can, still be used to corroborate with the trend obtained when this code region was executed over the different CPU frequencies.

8.1.2 Compute Boundness

CPU-bound applications are generally characterized by a high computation instruction throughput. They also have very low dependency to the memory sub-system, since in CPU-bound applications, all the needed data are fetched from the lowest level of cache. The modification of the frequency will strongly impact the application execution time as shown in Figure 8.3. If the 1.596GHz frequency is chosen instead of the 2.66GHz frequency, the application execution time is multiplied by a factor 1.64 this is almost equal to the frequency ratio: $\frac{2.66}{1.596} = 1.66$. This connection between the frequency ratio and the execution time is the unique characteristic of a CPU-bound application.

It can be noticed that the energy consumption trend and execution time one follow each other. It is mainly because the decrease in time overcomes the increase in power consumption. Sometimes, the power consumptions overcomes the speedup. This can be observed in figure 8.3 when using the frequency tagged as Turbo Boost.

Turbo Boost [1] is a technology allowing Intel CPUs to overclock themselves under strict conditions. By using Turbo Boost, an application execution can benefit from even higher frequencies. For example, the CPU, used to run Gromacs in Figure 8.3, is allowed to increase its operating frequency up to $3067MHz$ and $2933MHz$ respectively when 1 or 2 and 3 or 4 cores are used. Using Turbo Boost means increasing the power consumption. In the case of Gromacs, the increased Turbo Boost power consumption overcomes the speedup on execution time, resulting in the noticeably huge increase on energy usage.

The behavior of Gromacs on the Turbo Boost frequency suggests that the rise in execution time could overcome the power increase because of the characteristics of the underlying hardware. One can argue that such trend is noticeable only on most recent hardware and cannot be used as a determinant factor for the application trend classification.

Previously to P-state as exposed in Chapter 6, manufacturers implemented T-state in order to deal with CPU overheat. T-state provided a logical division of the base clock. The frequency was divided without regulating the power supply

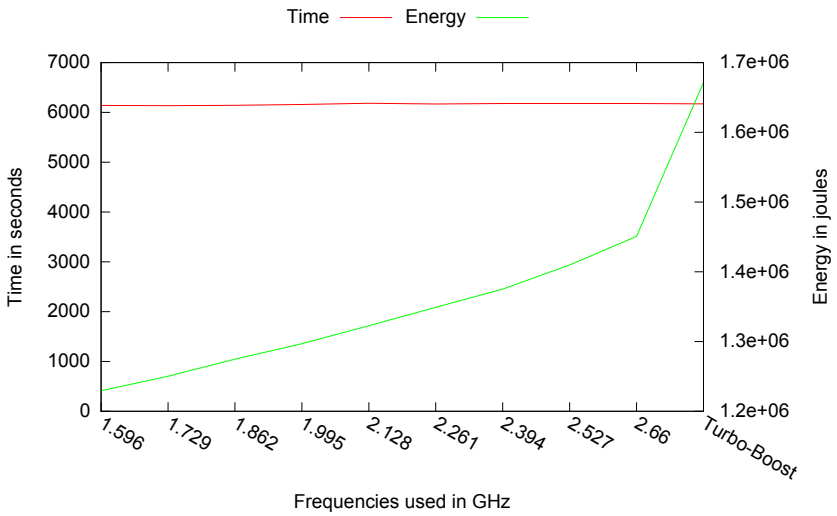


Figure 8.2: Wall energy consumption and time execution for the SPEC program libquantum depending on frequencies.

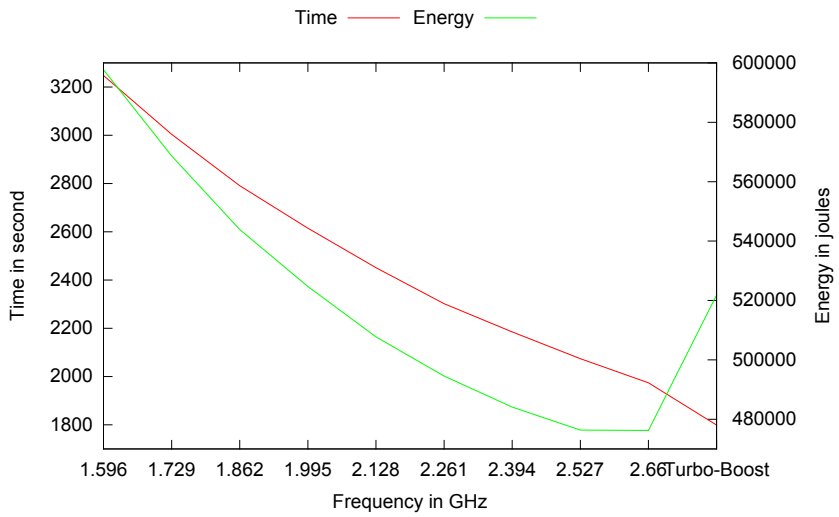


Figure 8.3: Wall energy consumption and time execution for the SPEC program gromacs depending on frequencies.

as performed by the P-state. In such a configuration, a CPU-bound application execution time will scale up according to the clock division ratio. As the power supply remains constant, and $E = P \times T$, the energy consumption will scale down as the frequency increase, inducing the same trend as the one shown in Figure 8.3.

Decreasing energy consumption and execution time generally are the characteristics of a CPU-bound applicaiton.

8.1.3 Balanced Boundness

As suggested by the name of this last category, targeted applications are neither purely CPU-bound nor memory bound, they are constructed with some specificity of each world. As observed in Figure 8.4. From 1.596GHz to 1.995GHz the application behaves as if it was CPU-bound. Starting from 1.995GHz, it behaves as a memory bound code which is not saturating the memory subsystem.

Understanding the behavior of the balanced bounded applications in terms of energy is more tricky since they interleave CPU oriented and memory oriented codes. It is very difficult to know before running an application which boundness will dominate and if a frequency setting can alter the domination. To properly understand the situation, both phases have to be extracted and studied to get a good grasp of the energy trend. The next section explains how to detect each phase.

8.2 Phase Detection

This thesis is about finding frequency configurations where the CPU power consumption and the application execution time give the lowest energy consumption. Even though, application energy trends are summarized into only three categories, each application presents a unique trend actually making the choice of the adequate frequency to obtain the lowest energy consumption a very difficult prediction.

The applications trends showed in Figure 8.3 or Figure 8.4 are influenced by how the applications were developed. In general, each application needs to load

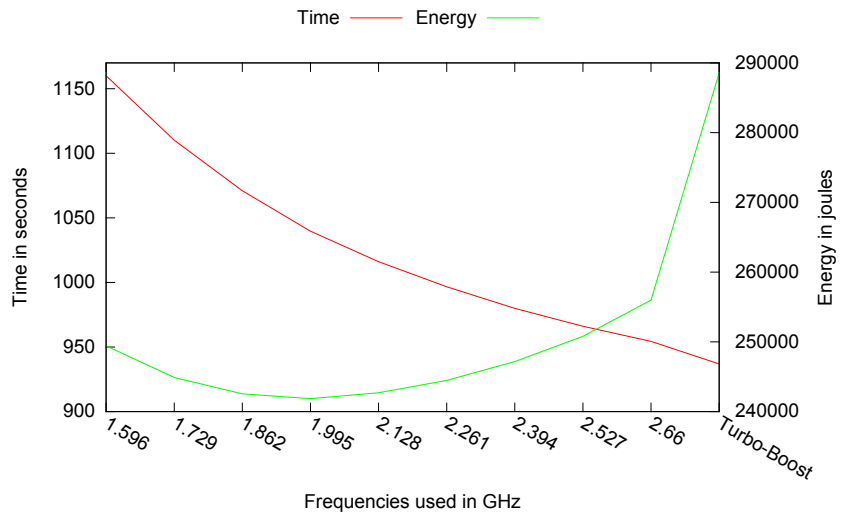


Figure 8.4: Wall energy consumption and time execution for the RTM program depending on frequencies

some data, perform computations, and store the results. Any application is composed of the same three phases. The difference between applications is the ratio of time spent in each phase. Being able to identify which phases are preponderant in terms of energy consumption will help finding the best frequency for the full application. As in any optimization process, it is better to optimize hotspots or resolve huge bottlenecks to obtain a maximum speedup. There are multiple ways to identify hotspots or bottlenecks: statically extracting important code blocks from the source code, or dynamically profiling the application to identify application execution phase. Application phase identification is complex and different methods were proposed [10, 39, 85, 87] but unfortunately none of them take the energy into account as a discriminant factor. Therefore the goal of this section is to show how the energy consumption can be used to identify application phase as well as explain the overall application energy trend.

8.2.1 Static Phase Detection

Static phase detection and analysis can take many forms [9, 19, 26, 36]. As an example, Akel *et. al.* [9] and De Oliveira Castro *et. al.* [36] propose solutions that statically slice an application in multiple regions and them as small benchmarks for performance analysis optimizations. This section use the result of the proposed slicing method. However, when executing the extracted code regions, energy consumption is considered instead of performance.

Static phase detection extracts code snippets from the application source code. To ease the extraction process, the Codelet Tunning Infrastructure (CTI) [160] is used. It is generally used to ease application profiling processes. By relying on a set of automated tools, an application can be decomposed into unique segments of code, called codelets. Each codelet can be run on various architectures and the results can be shared with other users through CTI's sharing system. For example, users can use data mining techniques, provided by CTI, to search through codelet information and find optimization hints that have previously provided a benefit for another similar application.

In the static phase detection, only CTI's ability to automatically decompose a set of application in unique segment of code is used. The specification of each codelet extracted from two NAS benchmarks [14] is displayed in Table 8.1. Each codelet displayed in Table 8.1 corresponds to a certain amount of the original application source code. The codelet selection is based on the amount of execution time they capture, which is expressed as a percentage of the full application execution. The remaining application source code, not comprised in the code coverage is mainly transitions between the selected codelets, or not significant enough in terms of execution time.

Application name	Codelet name	Coverage in percentage
BT	x_solve_	27.32%
	y_solve_	29.36%
	z_solve_	30.1%
Total		86.78%
IS	create_seq	65.19%
	full_verify	32.30%
Total		97.49%

Table 8.1: BT and IS codelets overview

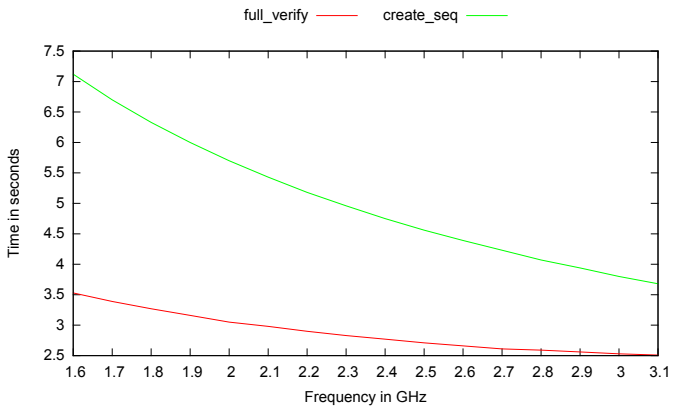
When an application is studied in order to understand its performance, the study generally focuses on the hot-spot or subsection of the application source code that captures most of the execution time. The same approach applies here: to understand the energy trend of each application it is necessary to capture the most of the application execution time. For both applications BT and IS, the execution time coverage of the extracted function respectively represents 86.78% and 97.49% of the total execution time. Once there is a high confidence on the codelet application coverage, running each codelet while measuring the execution time and energy consumption will help understand what their trend is and how they influence the overall application behavior in terms of energy consumption.

The source code of each codelet represents only one call to the extracted function or loop-nest. To have consistent measures to later compare the measured information with those of the full application, the number of calls to each codelet has to be found. It was done by inserting probes at the start and end of each extracted code segment within the original application. The coverage numbers presented above in Table 8.1 were measured during this step aside to the number of calls. The number displayed in Table 8.1 were obtained while the full application were executed.

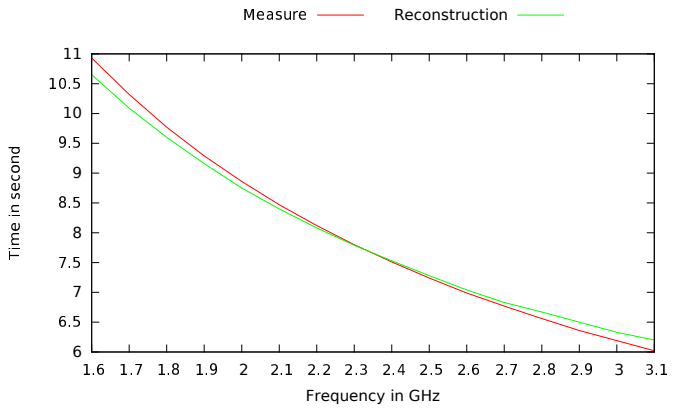
Once the number of repetitions of each codelet is known, they are run separately while measuring their execution time and energy consumption. Figures 8.5 and 8.6 show each codelet execution time for the different considered application, the corresponding energy consumption is shown in Figure 8.7 and 8.8

Figure 8.5 displays IS codelet and the full application execution times. The x-axis shows the different CPU frequencies and the y-axis represents the execution time in seconds.

Figure 8.5a only displays IS codelets execution time. As presented in Table 8.1, the function *create_seq* indeed captures the most execution time. It also lasts roughly twice the time of the *full_verify* method on each frequency. The measurements obtained while executing each standalone codelet matches the one performed



(a) IS' predominant codelets execution time



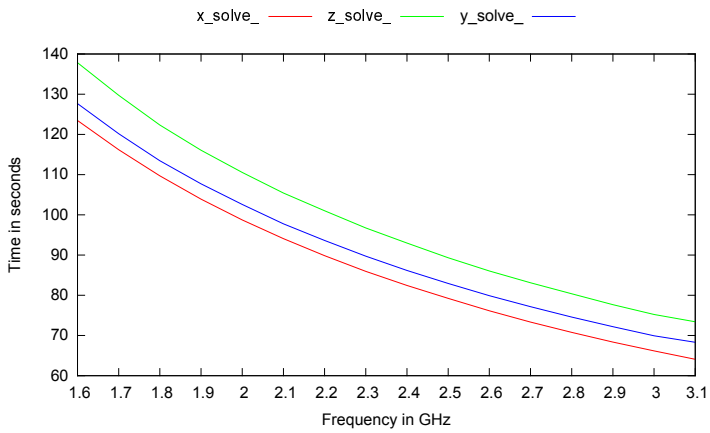
(b) IS recalculated and real execution time

Figure 8.5: Execution time of each codelet of IS and the difference between the re-calculated and the measured ones.

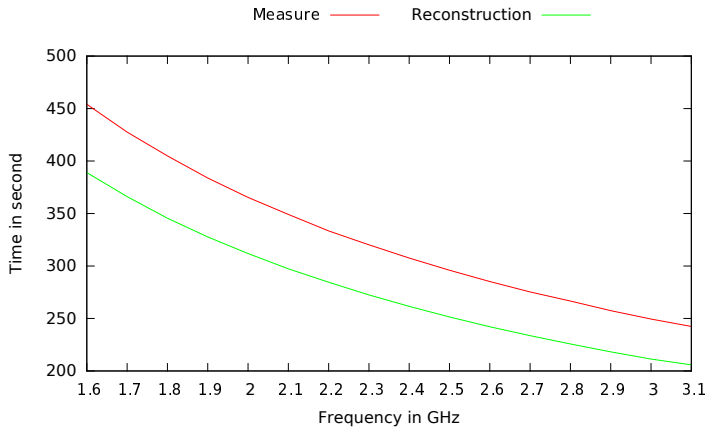
while the full application is executed. It confirms that the codelet extraction is correctly and accurately performed. Another way to check the sanity of the static code extraction is to compare the total execution time covered by the extracted codelet and by the real one. Figure 8.5b shows such a comparison. As stated in Table 8.1 *create_seq* and *full_verify* captures 97% of the application time almost matching the full application one. The fact that the reconstruction curve is above the actual application execution time curve on the five last frequencies comes from the small overhead obtained by separately running both codelets. In addition, the codelet with the highest code coverage, truly edicts the trend of the application. Indeed, *create_seq* edicts the tendency, and *full_verify* acts almost as an offset as the reconstructed tendency almost perfectly matches the measure. In the case of IS, there is no need to track with precision every application phases to get the general behavior of the application across the frequency spectrum. With nearly only half of the application coverage, via the *create_seq* function, a good estimation is produced.

Figure 8.6 displays BT codelets and the full application execution times. The x-axis shows the different CPU frequencies, the y-axis represents the execution times in seconds.

As opposed to IS where one codelet is the trend driver of the application, each



(a) BT predominant codelets execution times



(b) BT recalculated and real execution times

Figure 8.6: Execution time of each codelet of BT and the difference between the re-calculated and the measured ones.

BT standalone codelet captures the same amount of execution time as is seen in Table 8.1. The reconstructed execution time based on the extracted codelet shown in Figure 8.6b also validates the total coverage from Table 8.1. In the case of BT, each codelet is roughly equally weighted, making them all important when studying the general application tendency. This is the major difference with IS. Since all BT codelets are almost equally weighted and have the same trend, only one can be considered to determine the overall application tendency.

In the end, there is no need to precisely track all the application phases. With only the major application phase a good estimation can be produced. However, every codelet among the major application's phases has its importance, because each has an real impact on the energy consumption.

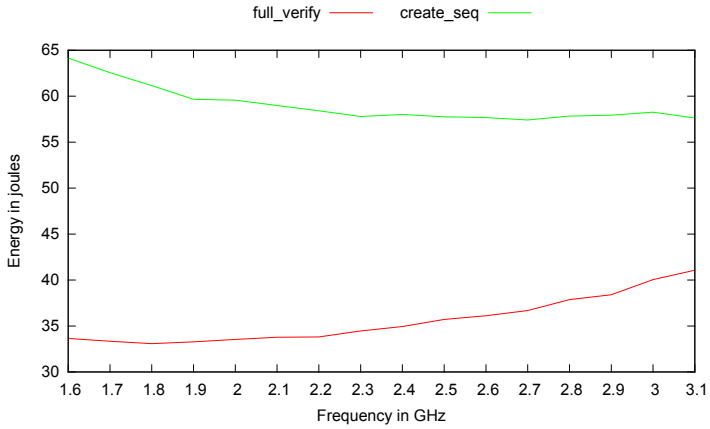
Figure 8.7 displays IS codelets and the full application from an energy consumption point of view. The x-axis shows the different CPU frequencies and the y-axis represents the consumed energy in Joule.

It can be noticed that both extracted functions, *create_seq* and *full_verify* have opposite behaviors that are not noticeable when only looking at their execution time. Indeed, both functions execution times scale with the frequency, even if *create_seq* scales more *than full_verify*. The difference comes from the fact that *create_seq*

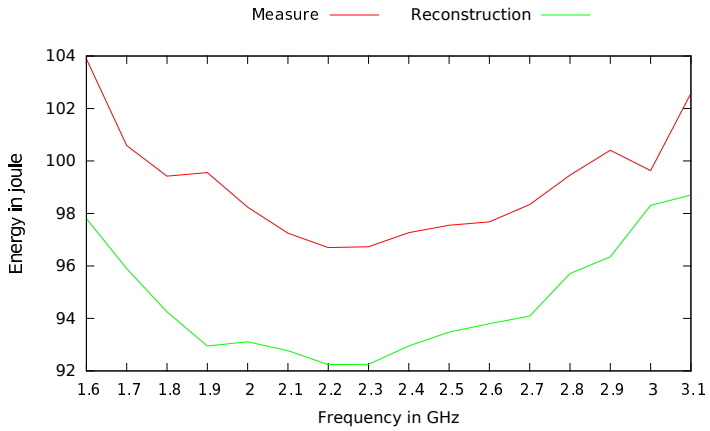
uses a vector traversal, additions, and multiplications whereas *full_verify* only does a vector traversal and comparisons. So *full_verify* is more memory bounded than *create_seq*. Both functions have the energy behavior corresponding to their respective trends as explained in Section 8.1. One could object that *full_verify* does not have the typical memory bound behavior as displayed in Figure 8.2. It is mainly due to the fact that the memory subsystem does not saturate, because the selected codelets are sequential. It allows the execution time to scale with frequencies, but the energy trend remains the same.

By summing both codelet energy trends, the reconstruction curves presented in 8.7b are built. It represents 97.19% of the total measured energy consumption. The noticeable gap in Figure 8.7b is mainly due to the scaling, they should be very close to one another as in Figure 8.5b. It can be noticed that both curves have the same bowl shape. The bowl shape is derived from the sum of two opposed trends. The *create_seq* function imprints its decreasing trend from $1.6GHz$ to $2.2GHz$. *full_verify* is the one influencing the general behavior from $2.2GHz$ to $3.1GHz$.

Lastly, picking the frequency inducing the lowest energy for each codelet should imply the lowest energy consumption for the overall application. As *create_seq* is CPU-bound, the race to finish policy exposed in Section 8.1 to get the lowest



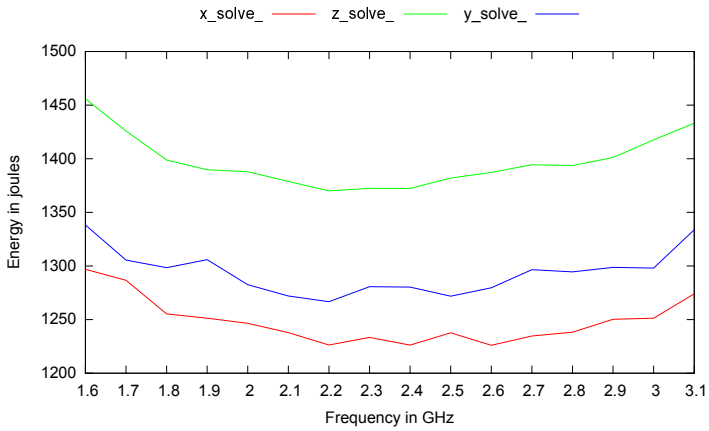
(a) IS predominant codelets energy consumption



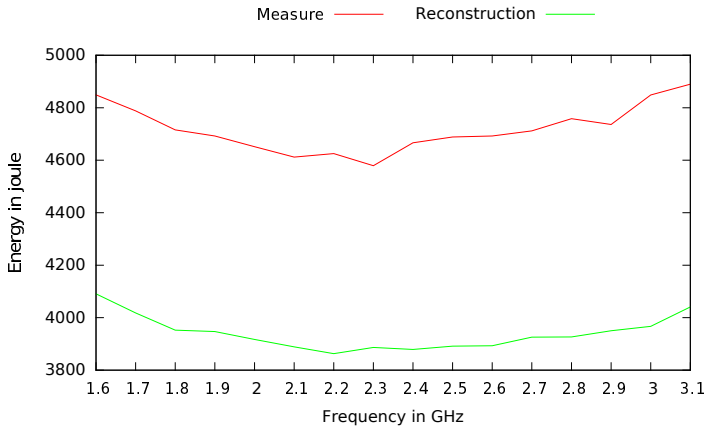
(b) IS recalculated and real energy consumption

Figure 8.7: Energy consumption of each IS codelet and the difference between the re-calculated and the measured ones.

energy consumption can be used. For *full_verify*, on the other hand, the lowest frequency has to be used to achieve the lowest energy. With both selected frequencies, the energy consumption obtained is equal to 91.28 Joule which is the lowest energy achieved on frequency $2.2GHz$ for the full application. One can object that, in the case of IS, designing a specific optimization for each codelet to achieve minimal energy it not necessary. When looking at Figure 8.7, the trend of *full_verify* and *create_seq* are respectively flat from $1.6GHz$ to $2.2GHz$ and from $2.2GHz$ to $3.1GHz$. Picking any couple of frequency ($[1.6-2.2], [2.2-3.1]$), will result in the same solution as the one stated above. For example by selecting $2.2GHz$ frequency for both functions, the recalculated energy consumption is equal to 92.23 Joules which is close to the solution obtain with dedicated optimization. Still, such observation was only possible since all the data were measured and could not certainly be computed while dynamically discovering the application.



(a) BT predominant codelets energy consumption



(b) BT recalculated and real energy consumption

Figure 8.8: Energy consumption of each codelet of BT and the difference between the re-calculated one and the measured one.

Figure 8.8 displays BT's codelets and the full application energy consumption. The x-axis shows the different CPU frequencies and the y-axis represents the consumed energy in Joule.

As for the execution time, all three BT codelets have the same energy consumption trend. It is understandable since all three codelets do basically the same things.

Each one uses a 3D matrix and performs computation on it. The only difference seems to be the number of accessed matrix cells, which also drive the number of performed computations. Each codelet is not purely CPU-bound since it has to load each matrix cell prior to the computation. Further more the codelets are not fully memory bound, as was the case for IS, because the memory subsystem is not fully saturated. It corresponds to the balanced behavior explained in Section 8.1.

Like IS, by summing all the codelet energy trends, the reconstruction curves from 8.8b is built. It represents 84.38% of the total measured energy consumption, therefore, the noticeable gap in Figure 8.8b is mainly due to the scaling. A zoom is needed, otherwise the classic bowl shape would not be visible.

The full BT trend is easier to understand than the IS one, because all the major codelets have the same energy trend. Searching for the frequency giving the lowest energy for the entire application is also more straightforward than for IS, after all the same frequency gives the lowest energy consumption for all three codelets.

Static codelet extraction is a good way to study an overall application behavior. By only studying the predominant application phase and picking the best frequency for each of them, the best frequency setting can be derived for the entire application. But in some cases the codelet extraction either fails or the code coverage is poor. In addition for applications where the coverage is good, the number of calls to the codelet is still needed to get consistent insight on its behavior. Unfortunately to retrieve that call number, code instrumentation is often needed. If the source code is not available, complex binary manipulation [25] are needed which is not always affordable. To bypass these difficulties, application behavior can be dynamically studied by using profiling along with being executed.

8.2.2 Dynamic Phase Detection

In the previous section, functions or loop nests were statically extracted. The generally is no way to know whether or not they represent a significant part of the application execution. A process is then needed to isolate the predominant loop before going any further in the analysis. Dynamic phase detection requires the same process: profiling. Profiling systems use information collected during the actual execution of the program. It allows to discover which part of the program is time consuming. It is important in this case where predominant functions or loops have to be isolated. For example Jimborean *et. al.* [89] designed a framework that allows any user to perform code analysis at different granularity. It can for example trace all the memory addresses that are accessed during the execution of a loop nest. Barthou *et. al.* [15] uses MAQAO to perform performance analysis of openMP applications. However, the simplest way to identify where the code spends time is *Gnu gprof*. Table 8.2 shows the information retrieved by using the *Gnu gprof* profiler on the full benchmark BT from the NAS benchmarks. Only the data regarding the three predominant phases identified in the previous subsections are shown for clarity purpose.

Compared to static code extraction, the dynamic profiling only needs one application execution to get the same amount of information as the used in the previous section. In addition, profiling systems usually use sampling method to measure the needed information, for example the information displayed in Table 8.2 were measured by using a 10ms sampling period. Profiling technique indeed allows smaller granularity to be targeted. Also, making the phase detection no longer bound to

%time	#called	name
29.6	201	z_solve_
29.2	201	y_solve_
26.2	201	x_solve_

Table 8.2: BT Gprof condensed summary.

function or loop-nest level.

In the previous subsection, the source code was used to explain the behavior of the energy consumption trend of IS or BT. It was assumed that, because IS' function *create_seq* was using additions and multiplications, it was CPU-bound. With a profiling system, the whole application workflow is studied and the profiler determines whether *create_seq* really is CPU-bound. To achieve that, the profiling system can rely on hardware counters. They are registers that show the user what the hardware is really doing. Using such insight on the execution flow allows anyone to understand the stress of the computational units as well as the stress on the memory sub-system. In a nutshell, a profiling method identifies CPU-bound, memory-bound, or balanced application's phases. In Section 8.2.1 the characterization was done by looking at the execution times or at the energy consumption trends in Figures 8.6, 8.8, 8.5, 8.7, but profiling enable a much finer grain.

Hardware counter profiling is a common technique for phase detection [10, 86, 154]. But as the profiling measures the application execution, it must have the smallest possible impact on the execution flow. If not, the profiling system will measure its own impact. Further details can be found in Chapter 9. The implemented profiler uses the smallest number of hardware counters to limit the impact of the dynamic phase detection.

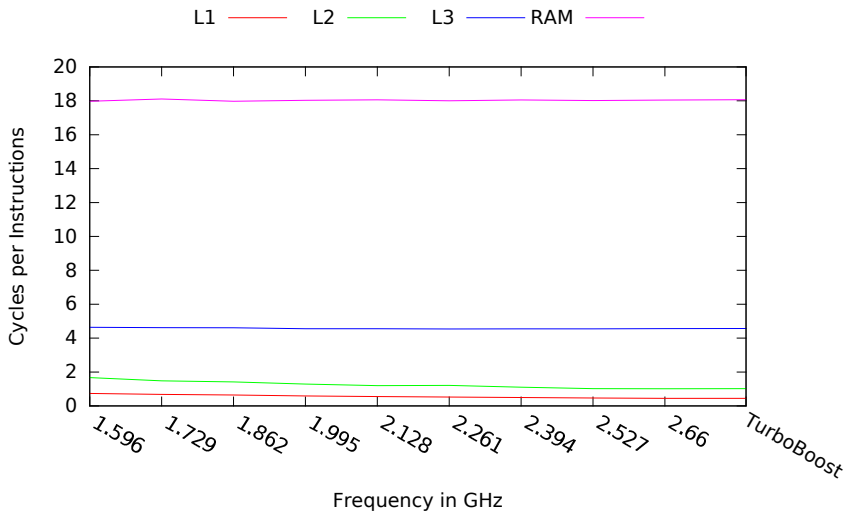


Figure 8.9: Data fetching latency from each cache level.

The profiling system tries to find all the application phases and classify them as CPU-bound, memory-bound or balanced. The set of hardware counters has to allow the phase discovery and classification in addition to be as small as possible.

To get a good idea of the CPU stress intensity, the quantity of executed instruc-

tions per sampling period could be measured. It quantifies the density of executed instructions and the higher the metric, the faster the instruction execution. However a lower value does not necessarily mean that the program is making more memory accesses. It can be due to bad branching predictions, or to numerous execution of division instructions which have to be emulated on certain processors, and imply thousands of cycles to be completed. Although the quantity of executed instruction is necessary, it is not sufficient, since its variation can be either due to memory access or just slow pipeline execution.

To solve the previously described uncertainty, monitoring the memory subsystem is necessary. Consider Figure 8.9, it shows that the data access latency strongly depends on the level where the data have to be fetched from. On most architectures, the Last Level of Cache (LLC) is usually shared by all the CPU cores, and is located on the "off-core" CPU region [104]. The LLC has its own independent working frequency leading to a slower data access latency as shown in Figure 8.9. The figure also shows the different latencies for each level of the memory subsystem. The first two levels have the smallest latency and the LLC and RAM have the highest ones. To see if the application will be penalized by memory operations, accesses to the LLC and upper have to be monitored.

To determine if the CPU is intensively computing or waiting for the memory sub-system, a metric is needed to determine how many instruction are executed and how many data accesses are performed in the LLC and higher. However, the selection of good hardware counters strongly relies on the architecture used and can change from one to another. In the study case, a Nehalem architecture was used. Figure 8.10 shows how the set of chosen hardware counters behave when executing a synthetic benchmark. The synthetic benchmark was created to alternate memory bound phases and CPU-bound ones. The memory phase randomly accesses elements in a vector dimensioned to not to fit in any level of cache except the RAM. The CPU-bound phase is designed intensively perform additions. To ensure that the memory subsystem is saturated, several instances of the benchmark are launched at the same time. The phase alternation is easily noticeable in Figure 8.10.

The figure displays a lot of information. The x-axis represents the application execution flow expressed as a number of samples. Each sample represents one hundred milliseconds of the application execution time. The right y-axis displays the instant values read in the hardware counters per one hundred milliseconds. The left y-axis represents a finer sampling, it counts the number of milliseconds spent in any function executed on one hundred millisecond of the application execution, that is to say one x-axis sample. As the synthetic benchmark alternates between a CPU and a memory phase, it is then logical that the memory function capture all the $100ms$ samples for the five first second. The same happens for the next five second with the CPU function and so on. It allows anyone to clearly acknowledge the benchmark workflow and its impact on the underlying hardware. They are measured through three different hardware counters: $UNALTED\_CORE\_CYCLES$, $L2\_RQST\_MISS$ and $SQ\_FULL\_STALL\_CYCLES$. For clarity purpose, each hardware counter will respectively be given an alias: $CORE_{CYCLES}$, $L2_{MISS}$, SQ_{CYCLES}.

The $CORE_{CYCLES}$ counts the number of cycles spent by the CPU cores in issuing micro-op into the execution pipeline. The variation of this counter clearly states the degree of activity of the execution pipeline. There is no drastic drop, apart from the beginning of the application, which means that the cores are always

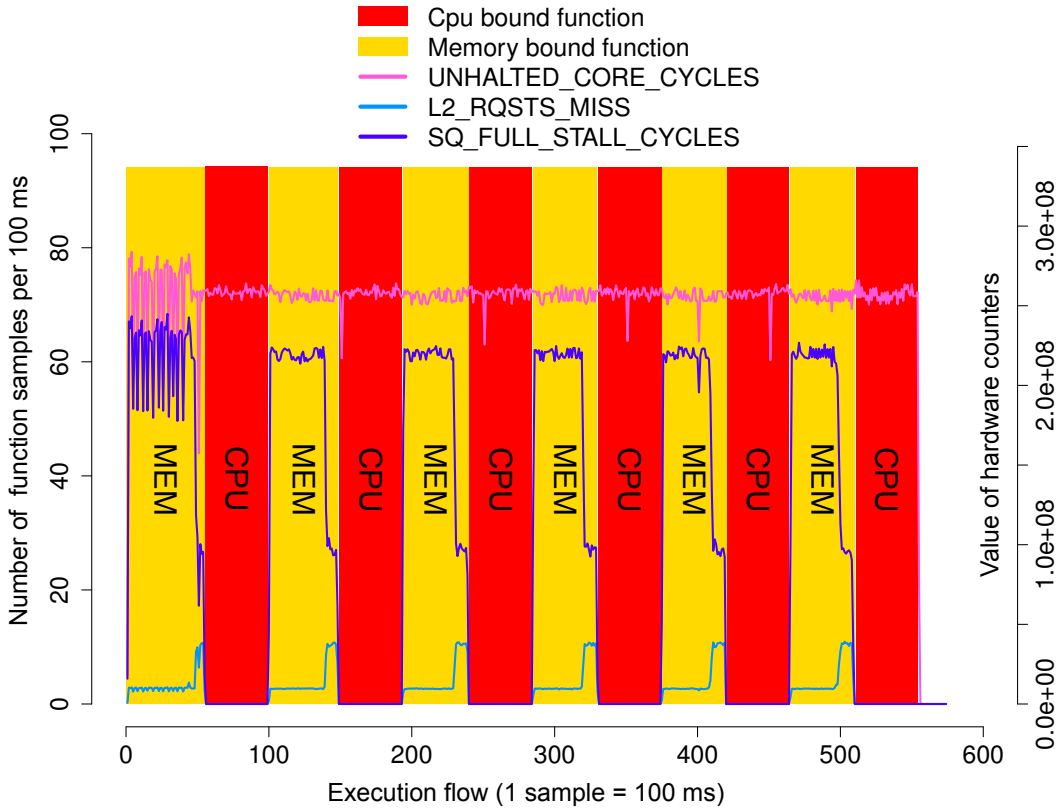


Figure 8.10: Hardware counters during the execution of a synthetic benchmark: showing how the counters evolve between a memory bound and a compute bound execution

working. As stated before, only looking at this counter will not be sufficient to identify the alternation between the phases. There is no drastic change either when the application shifts from a CPU phase to a memory phase.

The $L2_{MISS}$ measures the number of data requests higher than the L2 cache level. A high value means a lot of miss in the L2, implying an increased number of accesses in the LLC.

The SQ_{CYCLES} quantifies the number of cycles spent by a new request before being served when arriving in the already full queue of the LLC. As said above in the synthetic benchmark, the memory phase consists in randomly accessing data out of any level of cache. This roughly implies that any request has to wait for all the previously buffered requests to be handled. It explains why a data request has to spend so many cycles in the queue.

Since the memory related counters drop to zero when the memory hierarchy is not stressed, the selected hardware counters clearly show, on the synthetic benchmark, the alternating phases. Either SQ_{CYCLES} or $L2_{MISS}$ alone, coupled with $UNALTED_{CYCLES}$ could do the job of noticing the different phases. The use of both at the same time is useful when considering real world applications. In addition, notice the drop/increase in both SQ_{CYCLES} and $L2_{MISS}$ just before changing from MEM to CPU. It is due to the decrease in the number of memory accesses to handle, freeing up some space in the buffer of the LLC. This leads to lower servicing time, and allows an increase in the density of requests going outside of the L2 cache.

However, a synthetic benchmark is not representative of real-world applications. Generally they do not have such discrete phases but instead have continuous phase transition. The set of counter values is closer to the ones in Figure 8.11.

The considered program is part of a larger application called RTM, *Reverse Time Migration* [11, 17]. RTM is an high-end two-way wave-equation migration for accurate geological imaging. Here the kernel is extracted from the full application belonging to Total, but other corporation as cggveritas [24] or Schlumberger [149] have their own implementations.

Figure 8.11 shows more subtle variations but they still are noticeable between processing and memory oriented phases, especially in the zoomed box on the top right.

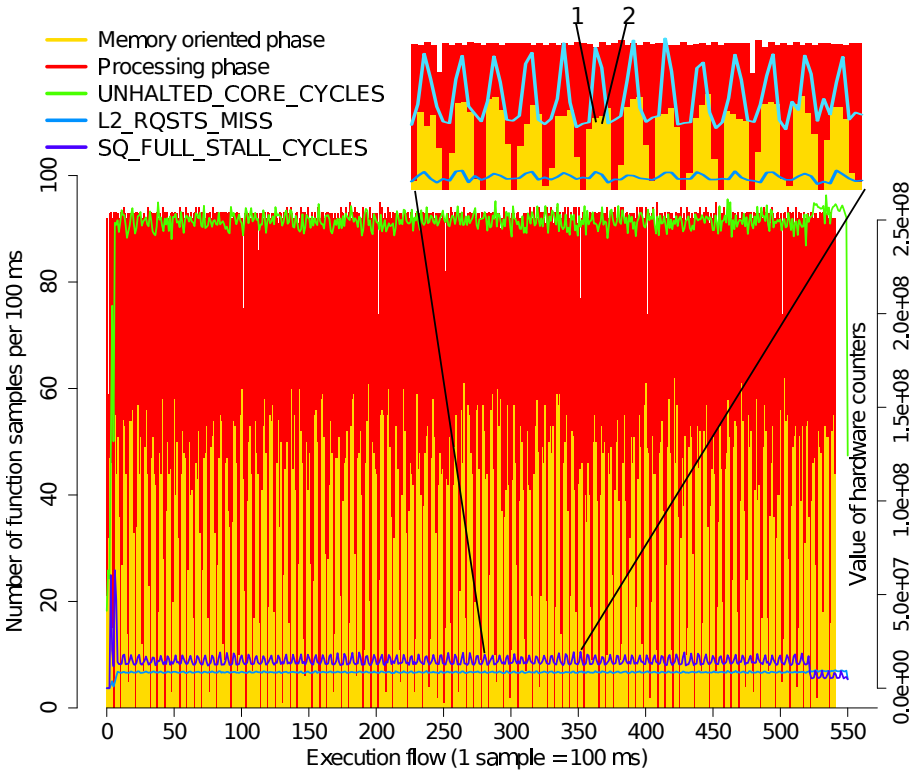


Figure 8.11: Hardware counters during the execution of a real world application (RTM)

In both worlds, static or dynamic, the used techniques are able to identify the application phases and their boundness with a different degree of granularity. Based on this observation, tools can be built to take advantage of each application trend and phase to predict the best frequency to execute a specific phase. But the frequency selection for each phase strongly depends on the capacity to quickly switch the frequencies between two phases. Consider that a memory phase is being executed, and a more CPU intensive phase is going to be executed next. A different operating frequency will then be needed and the latency of switching to the best frequency for the next phase must be greatly shorter than the next phase.

8.3 Dynamic Voltage Frequency Scaling Latency

As a reminder from the first part, the idea behind DVFS is to dynamically adapt the P-state to reduce power consumption and hopefully energy. On one hand it was seen in Section 8.1 that programs intensively using memory can be run at a low frequency as it will not impact their execution time. They provide significant energy savings as shown in Figure 8.2 . On the other hand, CPU intensive programs are very sensitive to frequency as their execution time is heavily impacted by any frequency switch. It induces a negative impact on energy consumption as shown in Figure 8.3 from Section 8.1. At the end of Section 8.2 it was concluded that DVFS controllers should be able to set the best P-state for each application phase to minimize their energy consumption. However, changing a P-state is not a free process, it takes time as introduced in Figure 6.2 from Chapter 6. Currently, the frequency transition latency is hard to obtain as processor manufacturers often do not provide the information in the product documentation or only provide approximate values [84]. On the operating system side, Linux provides an estimated transition latency in a file named $cpuinfo\_transition\_latency$. However, the provided latency is a unique estimated value whereas, as shown in Figure 8.14, 8.15, 8.16, the transition latency depends on the current and desired frequencies. Hence, neither the operating system nor the manufacturers provide reliable transition latencies.

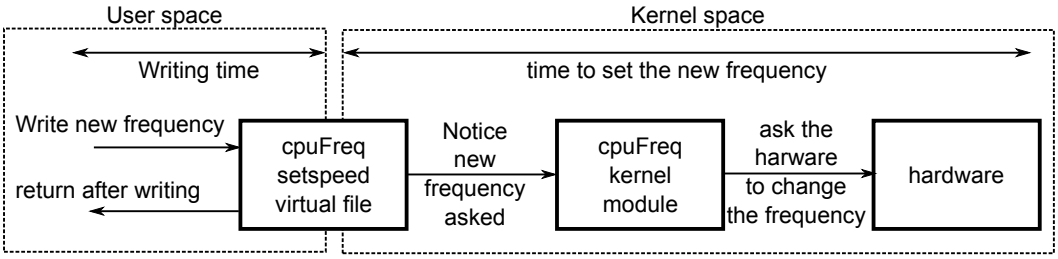


Figure 8.12: Step before actual frequency switch

For their defense, trying to estimate it from the operating system side is a complex operation since there is a lot of abstraction between the frequency switch asked by the DVFS controller and the actual frequency switch on the hardware side. The figure 8.12 shows the different step from the operating system to the hardware. Each step in the kernel space and in the voltage regulator is done asynchronously to the user space. There is no easy way to acknowledge when the actual frequency shift is performed.

In order to get a good estimation of frequency switch latencies a tool was developed : FTaLaT (Frequency Transition Latency) [120]. For each pair of CPU frequencies, it measures the transition latency, or switch delay. In fact, it aims at measuring the time between the request for a new frequency and the actual frequency transition. FTaLat's approach relies on the measurement of a micro-benchmark kernel made of a set of assembly instructions. The kernel has to be CPU bound in order to be as much as possible affected by the frequency change. The kernel consists in a set of consecutive add assembly instructions resembling the one displayed in Figure 6.6 .

The experimental methodology, used by the authors, consists of two main steps: initialization and frequency transition latency measurement. In the initialization phase, FTaLaT measures the execution time of the kernel when it runs using the

Algorithm 5 FTaLat's Frequency latency measure procedure

```
Init
set start frequency
startTime ← kernel exec time
set target frequency
targetTime ← kernel exec time

Latency measure
set start frequency
latency ← 0
set target frequency
while PeriodMeasure ! = targetTime do
  increase latency
  PeriodMeasure ← kernel exec time
end while
```

target and the *start* CPU frequency. In the second phase, FTaLaT sets the CPU frequency to the *target* one and waits for the kernel execution time to match with the one measured in the initialization phase. The frequency switch delay is then the number of cycles elapsed between the frequency shift order and the kernel execution time change. To be more reliable, FTaLaT uses a statistical approach to estimate when CPU frequency transitions truly occur.

In order to have a better view of the transition latency, Figure 8.13 extracted from [120], represents the measured kernel execution times on an IvyBridge machine when switching the CPU frequency from $1.6GHz$ to $3.4GHz$. While the vertical axis reports the execution time of the kernel, the horizontal axis represents different kernel execution. Figure 8.13 visually displays the moment when the new frequency is actually changed. The new frequency shift order was issued at iteration 1 and is effective at iteration 50. The delay between the order and the actual shift represents in the example 45 us. Additionally to the delay, the execution pipeline seems to be paused and flushed to take into account the new operating frequency as shown by the dramatic increase on the benchmark iteration 49. Hence, too frequent frequency shifts can have a huge impact on the execution pipeline and strongly impact the overall application execution. However, that specific problem was acknowledged using as much CPU bound benchmark as possible with very precise measurement set-up. Unless the application to be optimized is perfectly CPU bound, other bottlenecks will prevent that pipeline pause to be the major limiting factor. Moreover, if the application is perfectly CPU bound, any DVFS system will recommend to only use the highest frequency, as shown later on, hence preventing the pipeline pause from occurring.

Though the mechanism to change the operating frequency is identical, as shown in Figure 8.12, each architecture exposes different behaviors. Figures 8.14, 8.15 and 8.16 show the latency of the frequency switches exposed by FTaLaT on three different Intel architectures. For each figure, the x-axis shows the frequency spectrum available on each architecture. It can be noticed that the Ivybridge and SandyBridge architecture have fifteen different frequency settings. The Westmere, which is the oldest architecture, has only ten. In addition the frequency padding between each architecture is different. The SandyBridge has a strict padding of $0.1Ghz$ between

each frequency whereas on the Westmere and the IvyBridge the padding varies between $0.1Ghz$ and $0.2Ghz$. The y-axis on each figure represents the frequency switching latency needed to change the frequency setting.

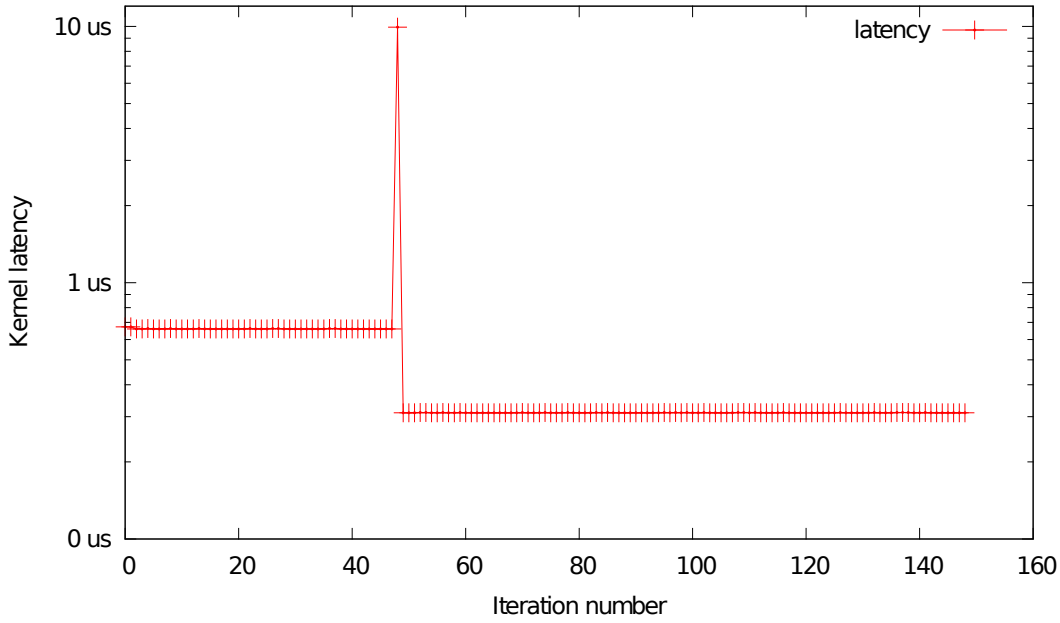


Figure 8.13: Observed execution times of the assembly kernel for the pair (1.6 GHz, 3.4 GHz) of CPU frequencies on the *IvyBridge* machine

On the three machines, the transition delay is not constant. The transition latency is observed between 22 μs and 45 μs on IvyBridge, between 20 μs and 70 μs on SandyBridge, and between 10 μs and 70 μs on Westmere. It can also be noticed that newer processor generations have smaller latency ranges. The transition latency increases whenever the target frequency is higher than the start frequency. For each start frequency higher than the target, the transition latency falls in very tight range of latency values: between 20 μs and 25 μs on the SandyBridge and IvyBridge machines, and almost 10 μs on the Westmere machine. These observations shows that changing frequency upwards is much more costly than changing it downward, validating the suggestion made in Chapter 6 with Figure 6.2. The transition latency increase does not follow a similar trend on all machines. Indeed, while the transition latency increases linearly when CPU frequency is increased on the SandyBridge machine, at least three levels of transition latency increase on the IvyBridge and the Westmere machines can be identified. However the voltage regulator is located outside the processor die, on the mother board [96]. It is then difficult to say if the different behavior comes from optimizations inside the processor die, or on the voltage regulators.

In order to set the correct frequency for each application phase, several parameters have then to be taken into account. Each application phase has to last a sufficient amount of time to benefit from the best frequency. For example, consider a theoretical frequency latency of 22μs and an application phase durations of 11μs and 440μs. The new frequency is asked at the beginning of the phase. When considering 11μs duration, the phase end before the new frequency is applied. It will not benefit from that frequency as illustrated in Figure 8.17a. When considering 440μs the frequency transition latency will only represents 5% of the phase duration,

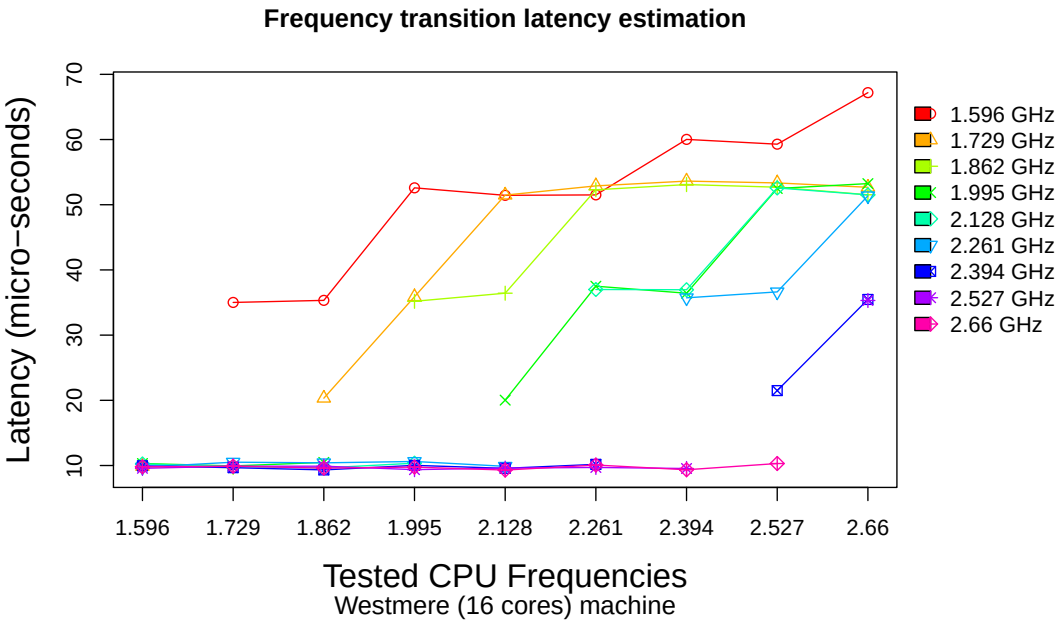


Figure 8.14: Latency to change frequency on an Westmere architecture

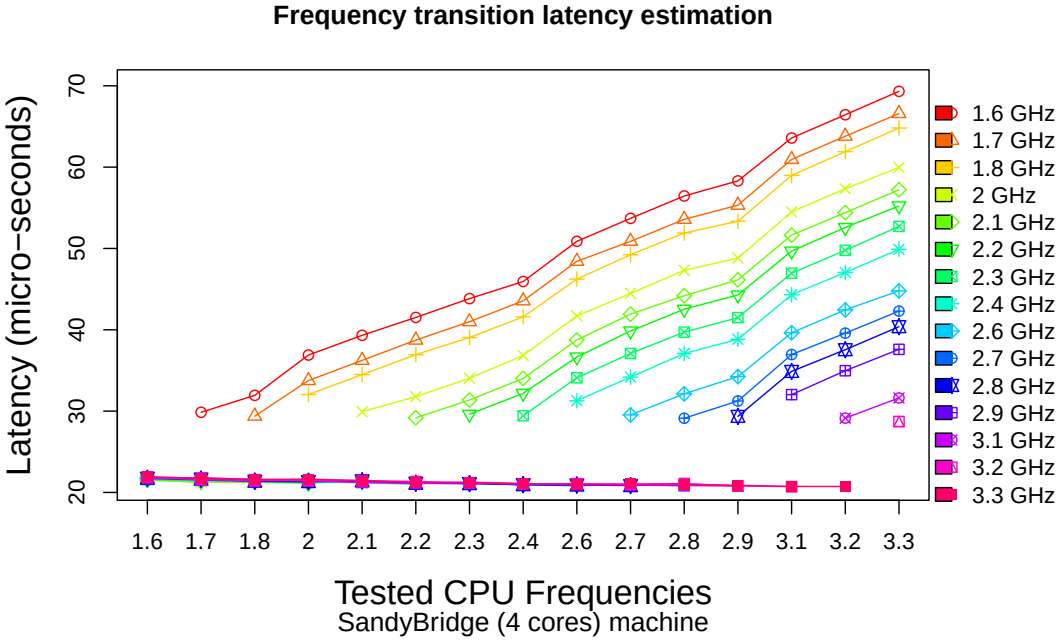


Figure 8.15: Latency to change frequency on an SandyBridge architecture

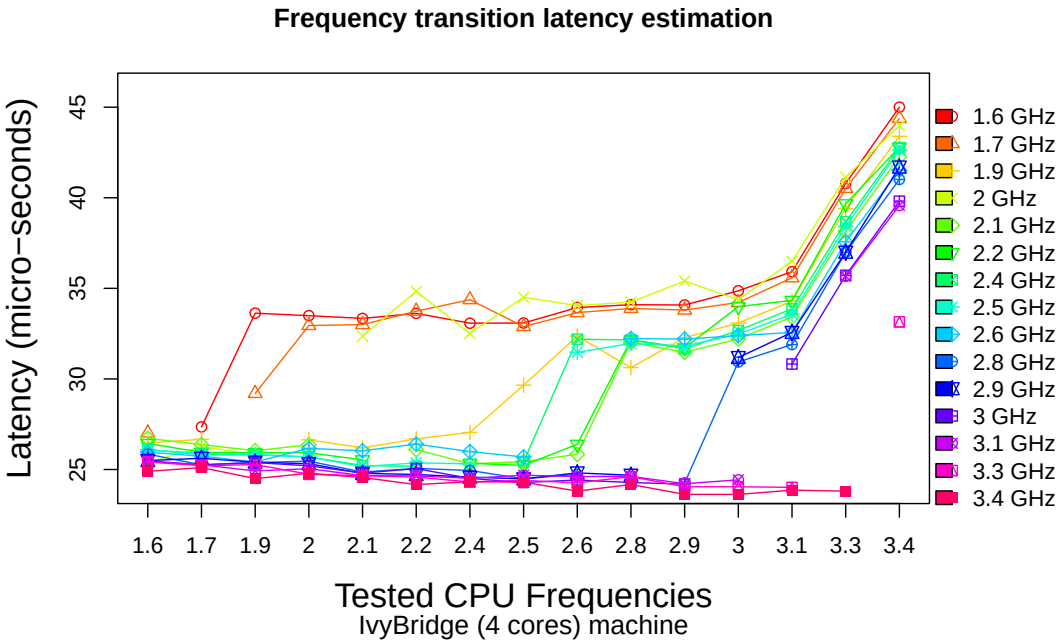
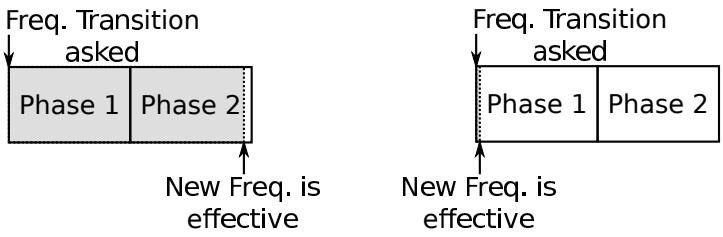


Figure 8.16: Latency to change frequency on an IvyBridge architecture

allowing the phase to benefit from the use of that specific frequency. Figure 8.17 summarizes both cases.

Finally, only three different categories summarize the various applications execution trend: CPU-bound, memory bound, and balanced. On the one hand, it was easy to find optimization policy to reduce the energy consumption of CPU-bound and memory bound application by either race to finish or lowering at maximum the operating frequency. On the other hand, the balanced application interleaves phases with different behaviors, resulting in non predictable energy trends. Still it was shown that an energy optimization process could be done by extracting and individually studying the behavior of each unique application phase. Finding a frequency setting for each individual phase allowed to guess the full application optimized energy consumption. It was also shown, that static phase extraction from application source code has huge limitation. Therefore, another way to identify phases is necessary, making the scope of study shifts from static world to a more dynamic one. Application phases are then identified during the application execution. With a small set of metrics, the boundness of each application can be easily diagnosed. That helps to



(a) Under sized application phase

(b) Good application phase size

Figure 8.17: Frequency transition latency versus phase duration

understand application trend and derive a strategy to reduce energy consumption, which is the goal of the next three chapters.

The following three chapters present three different DVFS mechanisms relying on the previous observation. Chapter 9 presents the Runtime Energy Saving Technology (REST), which is a purely on-the-fly DVFS system. For example, REST uses the boundness evaluation presented in Section 8.2.2 to determine the overall application boundness and converge to the frequency granting the lowest energy consumption . However, though Chapter 9 shows good gains in energy consumption, one question still arises: is REST doing good enough?

The quality of any optimization technique is always an important question to answer and Chapter 10 tries to answer it. The chapter presents a second tool called Utopeak, which is a static profiling tool. Utopeak analyzes the application and determines the best frequency sequence a dynamic system such as REST should choose to achieve optimal energy consumption.

Utopeak's analysis shows that there is more to do in the DVFS domain. REST, though a good first tool, is a bit naive in its decision making and requires hardware counters specific to the Nehalem architecture. Both issues led to the creation of Forest presented in Chapter 11. Forest presents a mean to handle the CPU DVFS issue in a more interactive way. Instead of trying to create a function that calculates the best frequency depending on current hardware counter values, Forest uses an iterative approach to adjust on the fly the frequency and obtain better energy consumption. Additionally, Forest has the big advantage of being able to offer the user a predetermined *acceptable* slow-down.

All three techniques help in understanding the advantages but also limitations of Dynamic Voltage and Frequency Scaling related tools. Finally, the three tools only consider a uni-node configuration. It makes sense to start to crawl before attempting to run and, as a proof, Part III considers the more complex DVFS in a multi-node configuration problems.

Chapter 9

Runtime Energy Saving Technology (REST)

In the previous sections it was shown that applications have different trends, meaning different ways to optimize the energy consumption. Moreover, though an application belongs to a specific trend, each application phase can have a different behavior. Therefore, the use of different frequencies through the execution is needed to optimize the overall application energy consumption. It was also shown that application phase identification could be efficiently performed at run-time. Therefore, selecting the optimal frequency regarding the application phase trend can be performed while the application is running.

By compiling all the previous insights, a first attempt to create a dynamic system that dynamically changes the frequency while applications are running was made with REST. The motivation of REST, is to make energy consumption reduce by selecting frequencies that best fit applications phases trend.

9.1 State of The Art

DVFS techniques are definitely not new [72, 85, 96, 114]. Dynamic systems [73, 86] profile the code at runtime using low overhead techniques. Some require modifications to the code base [50, 70], to the hardware [96, 114], use tools such as VTune [10], or use a simulator [113, 115]. REST, implemented on modern systems, provides a software layer that utilizes hardware performance counters and gives users a plausible energy consumption improvement with their current hardware set-up.

Isci *et. al.* [86] provide a similar hardware counter-based system to REST. The authors propose a phase prediction system using a *Global Phase History Table*, which produces next-phase behavior deductions based on previous samples. Once a prediction is performed, the framework is linked to a DVFS method, which reduces the frequency if memory-bound and raises it if cpu-bound. REST is similar in the concepts of frequency decision making but differs from their approach by handling multi-process applications. Also showing, in the DVFS scenario, predictions are not required; a simple naïve decision maker suffices and reduces the incurred decision making overhead. REST was evaluated using the energy consumption from the wall and not from the CPU, a methodology more widely used.

Hsu et Feng *et. al.* [73] created a power-aware runtime system taking into account the maximum slow-down the user may desire. The authors use a system to detect the global number of executed instructions per second. Therefore, if the system is executing multiple applications, specific per core information is lost in the global tracking. REST considers the application alone and the user can attach REST to the most important application. Also, Hsu and Feng's algorithm provides a means to reduce the maximum slowdown obtained, REST's goal is to always maintain as much the performance as possible while hopping to lower the energy consumption.

Gotz *et. al.* [56] has the same vision of an application energy consumption as REST. They also validate the presence of a sweet spot frequency allowing the lowest energy consumption. Their study is only performed on three different sort algorithms, but they show that for each sort, a single frequency grants the minimum energy consumption disregarding the data set size. It could be interesting to investigate that further as future works.

9.2 General Presentation

In the simplest explanation, REST selects a frequency regarding an application phase behavior. Therefore, REST first need is a way to measure the application activity. Then based on the activity, a trend has to be selected, either CPU-bound, balanced or memory-bound. Based on the trend a frequency is selected. Finally, once the frequency is chosen, it has to be applied. In a nutshell, REST is composed of three steps. The link between each of them is shown in Figure 9.1.

Figure 9.1 shows how REST's steps are linked and how they are implemented. The activity monitoring was performed via sampling based profiling as shown in Section 8.2. In REST overseeing the application activity relies on hardware counters monitoring. At each profiler wake up, the hardware counters evolution are sent to the application phase trend selection. Based on the gathered information, a trend is selected. REST uses different methods to achieve that. Called Naïve, Branch-predict or Markov, each of them relies on different level of complexity to produce the solution. Finally, a frequency is deduced from the selected trend, and sent to the frequency changer to be the new CPU operating frequency.

REST is started when launching an application to be optimized. The different steps are repeated at each profiler wake-up. Once the application is finished, REST stops.

REST impact on energy consumption relies on the efficiency of the first two steps. The profiling system must accurately grasp the application activity; this is far from simple as show in Section 9.2.1. The trend selection has to correctly

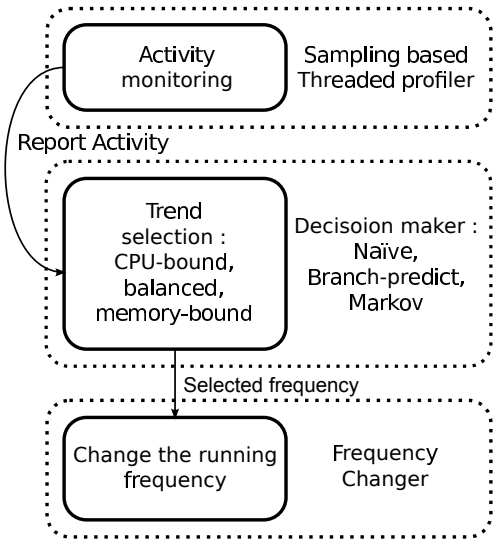


Figure 9.1: REST system overview

interpret the measured activity to select the best frequency. That is why different levels of complexity are tested to see if more intelligent system can produce better solution as presented in Sections 9.2.2.1,9.2.2.2, and 9.2.2.3.

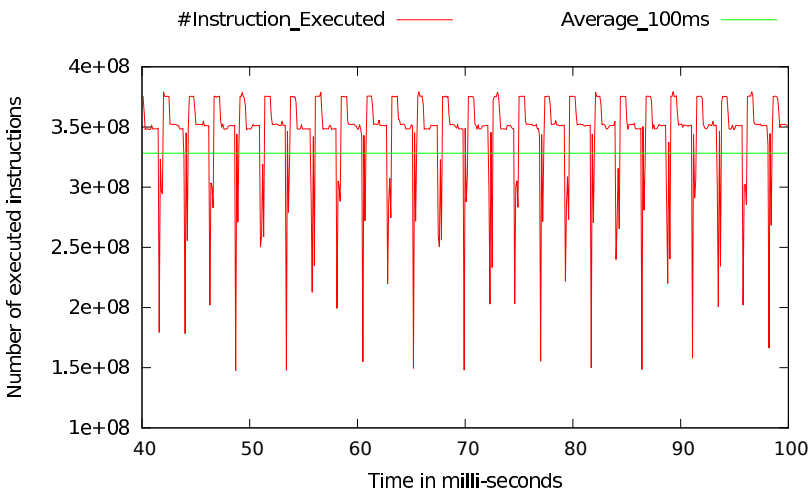
9.2.1 Dynamic Profiler

A way to accurately identify the different applications phase through their activity on the processor was shown in Section 8.2. It was decided to implement the same hardware counters interrupt-based sampling in REST. As a recall, the Figures 8.10 and 8.11 demonstrate the system ability to detect changes in the application execution flow.

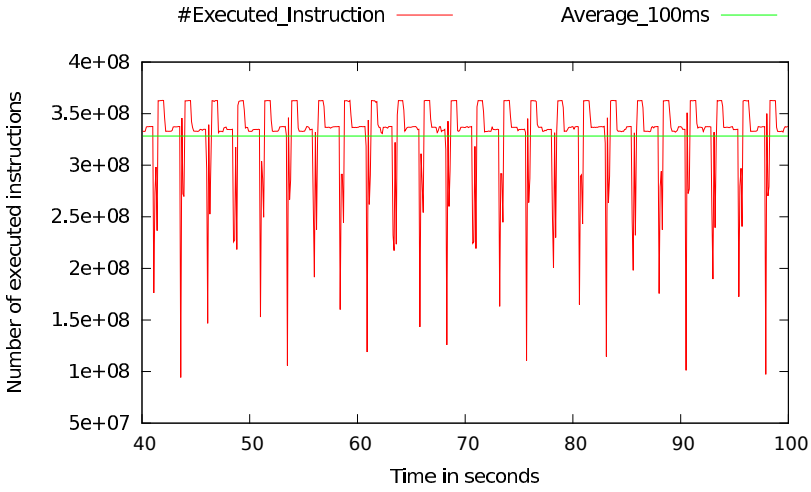
REST profiler periodically wakes up to measure the activity of the application via selected hardware counters from Section 8.2. As a reminder, the $CORE_{CYCLES}$ measures the CPU activity, $L2_{MISS}$ and SQ_{CYCLES} capture the level of memory saturation. At each wake up, the profiler measures the instant value of each counter, and passes them down to the decision makers. As for each profiling technique [46, 144], correctly designing the sampling period ensures the sanity of the measured information. If the sampling period is too small, the profiler wakes up too ofter; this might forces frequent context switches. It could slow down the studied application's execution and increase the energy consumption. The too frequent profiler wake-up will also tamper with the hardware counters values since the profiler also measures its activity. At the opposite, a too low sampling frequency makes the profiler miss application phase shifting, feeding biased information to the decision units. Figure 9.2 shows the impact of too small and too long sampling period on the measured activity. The following example, shown in Figure 9.2, displays the impact of bad sampling period on the CPU activty as the number of executed instructions are monitored. It was decided to choose such counter, because there is a direct relation between its evolution and the fact that more computation has to be handled by the processor. If the profiler wakes too much, it generates additional work for the CPU, therefore, increasing the number of instruction to be executed. The same study could be performed with the hardware counters used by REST but the difference would not be as clear as shown in Figure 9.2.

Figure 9.2 shows the impact of different sampling periods on the measurement done by a profiling technique while executing the benchmark program BT from the NAS benchmarks suite [14]. All three figures display the evolution through time of the number of executed instructions, consequently the x-axis represents the execution time in seconds and the y-axis the number of executed instructions.

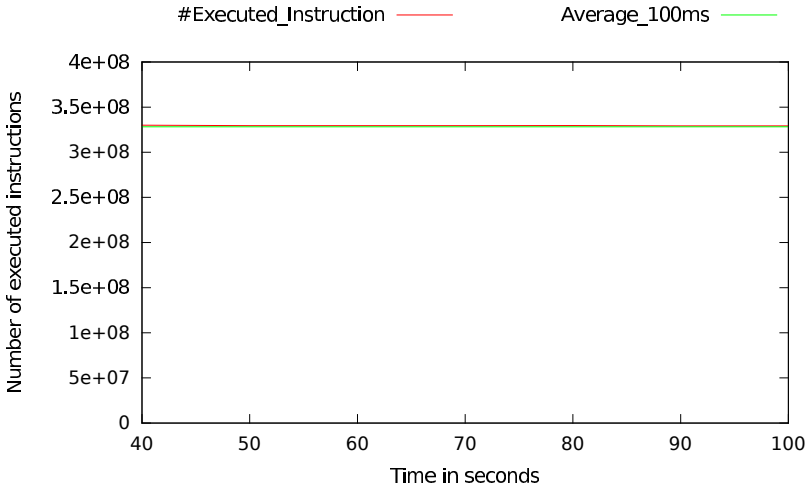
Figure 9.2b is used as a reference point for the two others. It clearly appears in Figure 9.2a that too frequent wake-ups generate a huge number of context switches, therefore additional instructions are executed. Figure 9.2a displays the case where the measuring tool identifies its own impact on the system in addition to the studied application behavior. The difference between both case is 4.58% in average. The profiler is minimalistic, it only polls the hardware counters and store its readings. If heavier computation were performed during the waking period, the impact on the number of executed instructions would have been more dramatic. At the opposite side of the spectrum, slow wakes up, misses too much information. In the case displayed in Figure 9.2c, the profiler misses all application phases, making the user believe the application only has one phase, which is far from true when comparing it to the reference case.



(a) Too frequent profiling wake up



(b) Reference case



(c) Too few profiling wake up

Figure 9.2: Different profiler waking period

The sampling period used as a reference in 9.2b is the one used in REST. It is based on a set of reference benchmarks, chosen in each category of trends (CPU, memory, balanced) shown in Section 8.1. It is used to evaluate the accuracy of different sampling period. The period giving the lowest overhead while correctly capturing all the application phase shifts is used as default sampling period. Conversely for the next chapters, REST does not use energy probes, so the sampling period sizing is purely based on application phases and not on the best tradeoff between the application's phase duration and energy probes resolution.

In the end, the REST sampling period is dimensioned to correctly profile any kind of application. It ensures that the read information shipped to the decision makers at each profiler wake up is correctly capturing the application execution. Finally, as there is strong confidence on the application activity monitoring, the decision makers can then fully dedicate themselves to finding the best frequency regarding the measured information.

9.2.2 Decision Makers

REST implements several decision makers. The first and the most naive, considers only the present and makes his decisions only on the current sample. The second makes its decisions on passed and current samples. The last one, does not make decisions, but predictions. It tries to predict the application future activity and anticipates the frequency setting. Each version has advantages and drawbacks, addressing different needs.

The role of each decision maker, is to find a frequency regarding the trend of each application phase. Or more precisely, the trend of the slice of activity measured during one profiler sample. The application trend classification performed in Section 8.1 is based on the application activity on each processors' frequency. In the case of REST, there is no possible way to stop the execution, run the piece of code that was monitored during one profiler sample on each frequency and then derive its trend. A ratio based on the measured information can the be created. The ratio, called boundness evaluation, expresses whether the current application activity is more CPU-bound, balanced or memory-bound. It is comprised between 0 and 1. If the ratio is equal to 0 the current sample acknowledged a purely CPU-bound behavior. At the opposite, if the ratio is equal to 1, the measurement is performed during an intensive memory bound application phase. Any value between the two extremities represents a more balance trend. Depending on the complexity of the desired decision makers, different decisions are taken on the basis of on the boundness ratio value.

9.2.2.1 Naïve Decisions

The naïve decision maker was developed in order to check the viability of the boundness ratio to correctly express what is being executed. Therefore, the naïve decision maker simply accepts all ratios as correct.

As the ratio expresses the application trend, it is natural to map a certain value range to a specific frequency. Indeed, as explained in Section 8.2.1, when facing a fully CPU-bound phase or memory bound phase, the best way to reduce the energy consumption is respectively select the highest and the lowest speed. So in the case of the naïve decision maker, the highest frequency is mapped to the ratio value 1 and the lowest frequency to 0. For the other frequencies in-between, a simple

interpolation is performed. The ratio value segment is divided by the total number of frequencies as shown in Figure 9.3.

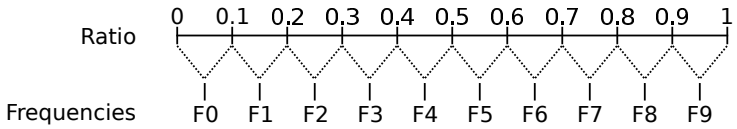


Figure 9.3: Naïve frequency mapping

So the frequency selection based on the boundness ratio is straightforward. After the ratio is computed, the proper range and the corresponding frequency are selected.

Not keeping passed samples, is the major drawback of the method. To illustrate the problem, consider the profiled execution displayed in Figure 9.4. The profiling samples labeled *1* and *2* expose the same quantity of executed memory and processing oriented phases, but have slightly different boundness ratios. The computed boundness are respectively 0.50 and 0.503. By construction, as shown in Figure 9.3, the boundness ratio interval for the frequency $F4$ is $]0.4; 0.5]$ and for $F5$ is $]0.5; 0.6]$. Therefore, though samples are equivalent, two different frequencies are selected. The slight variation of the boundness ratio around any frequency frontier forces different consecutive frequencies to be selected. Such a variation is called *constant-shift* in the remainder of the section. One can easily suppose that constant-shift can tamper with the execution time and energy consumption. Building a decision maker able to detect small variations of the bounding ratio around a frequency frontier prevents the system from undesired frequency shifts. It is the purpose of the next presented decision maker: the branch-predictor decider.

9.2.2.2 Branch-predict Decisions

Branch predict mechanism, tries to predict which branch of a future branching is likely to be taken. The prediction mechanism is based upon an history table to track past events to help future decision. Using such history tracking can strongly help detecting constant-shift and preventing them. Based on an history table the branch-predict decider builds a confidence level for each frequency, helping it choosing the correct frequency for the current application phase.

The confidence level is based on two observations. The first is the number of calls to a specific frequency. If a frequency is selected more often than others, it should be applied. The second is the distance between two frequency selections. A valid frequency shift occurs when the distance between the newly selected frequency and the current one is more than one. For example if the current applied frequency is $F1$, and the newly desired frequency is $F3$, the shift would be considered as valid. Adding both constraints solve the constant-shift described in Section 9.2.2.1 for the naïve decider.

To illustrate the need to have both constraints to solve the constant-shift, consider Figure 9.5. It shows the frequency selection tracking, i.e. the chosen frequency at profiling samples. From iteration 1 to 5, it can be see that the frequencies $F1$ and $F2$ are alternatively selected, then from iteration 6 to 9 only $F3$ is selected. If only the most selected frequency is chosen to be applied, the constant-shift remains present from iteration 1 to 5. $F1$ is applied at iteration 1, then at iteration 3 $F2$

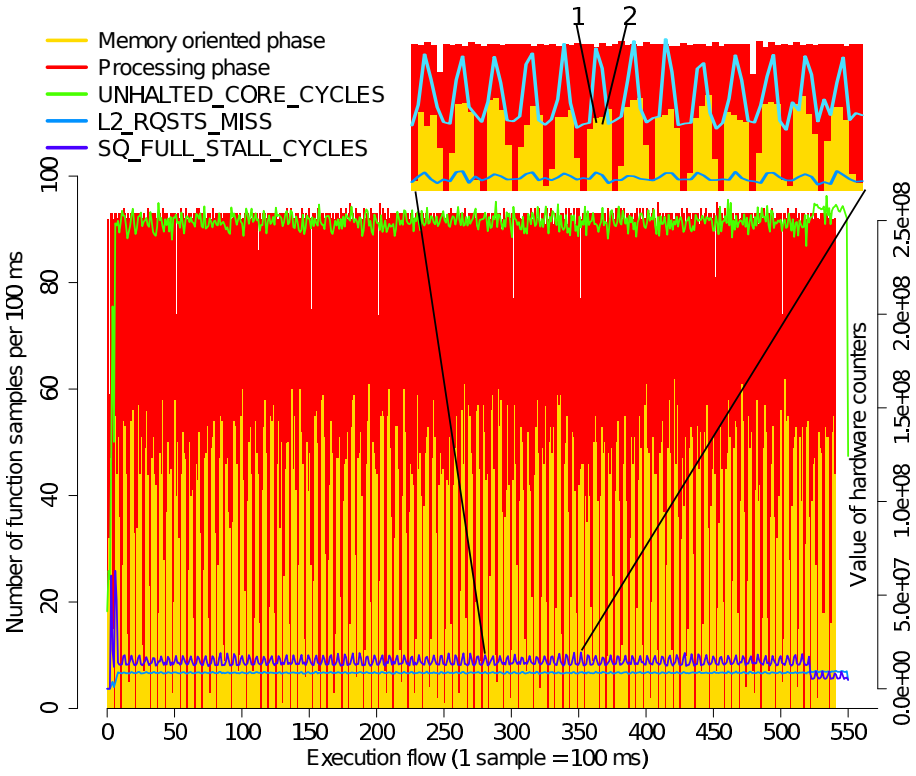


Figure 9.4: Hardware counters during the execution of a real world application (RTM)

is selected and finally $F0$ is resumed at iteration 5. In the current example the frequency alternation sequence is stopped at iteration 5. However, one could find an application where the pattern is repeated during the entire application, letting the constant-shift happening even though the decision are based upon past events. Adding the constraint distance solve the problem, letting frequency $F1$ applied from iteration 0 to 9 until $F3$ is illegible to be applied as the new processor operating frequency.

Composing both observations solve, constant-shift. Once a frequency shift is decided to be valid, the frequency usage history is cleared in order to detect a new phase and when to apply a different frequency.

The disadvantage of the prediction system is twofold. First, even though it solves the major disadvantage of the naïve decision maker, it forces the branch-predict decision maker to be over conservative. It must wait for a few samples to confirm the change and finally modify the frequency. As shown in Figure 9.5, the system waits for three iterations, from iteration 6 to 9, to be sure that F3 is the new frequency to apply. In addition, the overhead of calculating and maintaining a history is not free. Second, the slow change in frequencies requires larger phases, since the system needs several samples within the same phase to acknowledge it as legitimate. By construction it will skip small phase shifts. As an example, consider the synthetic benchmark used in Figure 8.10, with a ratio of 10:1 for the CPU phase. The over-conservative frequency switch will only authorize frequencies regarding the CPU boundness ratio for the entire application execution. It will neglect all the memory phases since REST will not be confident about them, leading to a non optimal energy reduction.

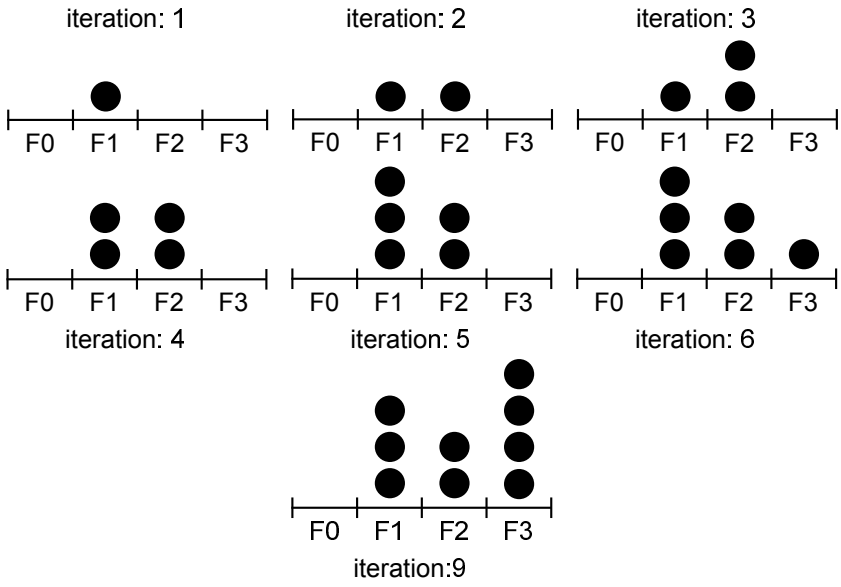


Figure 9.5: Frequency confidence level evolution

To conclude, the Branch-predict decision maker solves the constant-shift at the cost of lower reactivity. The frequency is only changed when the system is confident about the current executed phase, that also solves the problem of different frequencies for the same phase. But during the time needed by the decision maker to evaluate the executed phase, a non optimal frequency is applied. It is the same for not long enough phases. Though the system has a strong confidence on the applied frequencies, it leads to a sub-optimal energy consumption. But one can consider that the small phase or the time needed to evaluate the frequency confidence is negligible and does not impact much the energy consumption. As the system has no idea of the energy consumption, the impact of a skipped phase or the evaluation time on the energy consumption cannot easily be known.

9.2.2.3 Markovian Prediction

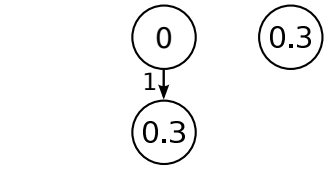
Scientific applications are usually iterative applications, meaning that the same phase sequence is repeated several time. So instead of re-learning the frequency settings for each phase, a decision maker can try to predict at each sampling step, which will be the next frequency to apply and when it will occur. In order to achieve that, a markov predictor based on Esodyp [21] is used. It takes as input the phase boundness ratio and it tries to predict the next ones. The presented algorithm is the same as used for Esodyp but instead of predicting memory strides, it predicts the next frequency to apply based on the boundness ratio.

To illustrate the used algorithm, consider the following example which represents phase boundness ratio sequence that a running program could have given:

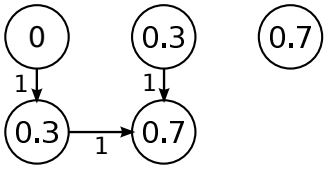
$$0, 0.3, 0.7, 0.3, 1, 0.3, 0.7, 0.3, 1$$

.

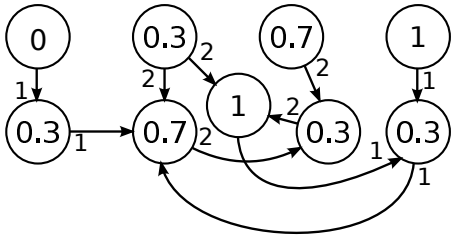
Markov model uses the past to predict the future. In the example, if the ratio 0 is followed by 0.3, the next phase ratio that will be measured is 0.7 with the assumption that what occurred previously is likely to be repeated. The used algorithm implements the backward dependencies as a graph as shown in Figure 9.6.



(a) After two phases



(b) After three phases



(c) After the full sequence

Figure 9.6: The Markovian Graph construction evolution

To explain how the graph works, Figure 9.6a shows what happens as the two first phases boundness ratio are caught. The markovian decision maker uses that algorithm with a depth of two as shown in Figure 9.6. The meaning of the single edge is that after a phase boundness of 0, a boundness of 0.3 will occur. In the case of nodes without successor, it is not known what could happen next. For example, the connex group $(0, 0.3)$ is not attached to anything since information is missing on the next phase. The node 0.3 alone symbolizes that, if the only information at hand is a stride of 0.3, nothing can be predicted since it can lead to anything. The edge label 1 means that this edge has been followed once. Such labels are used to select the most followed edges.

It can be seen how the graph construction evolves while another ratio is added on Figure 9.6b. Two more nodes have been added to the graph. The first node 0.7 is attached to both nodes 0.3. This symbolizes that after a 0.3 ratio as well as after the sequence (0,0.3) a 0.7 occurs. This value only indicates that the last phase ratio is 0.7 What happens next is not known.

Eight nodes are used in the graph when the whole sequence has been processed as shown in Figure 9.6c. Now some edge labels present a value at 2 meaning that those edges were followed twice.

The graph construction process stops after the graph starts to be used in the prediction phase. For REST markovian prediction, the graph construction is stopped after the addition of 100 nodes. As the construction is stopped, the optimizer points to the last created node and starts prediction. When receiving a new boundness ratio, it checks whether there is an edge from the current node leading to that ratio. If so, a prediction can be produced. On some nodes several paths have to be considered, and can lead to miss-prediction. A huge amount of miss-prediction can

state that the graph is not representing the reality. The used algorithm defines a threshold of miss-prediction before flushing the graph and starting over. For REST, the limit is fixed to 100 errors before starting over.

Benchmark	Valid phase shift	Undesired phase shift phase shift	Percentage of valid shift
Astar	20	3573	0.56%
Cactus	8	18788	0.042%
Calculx	6	11714	0.051%
gmss	8	2639	0.30%
gcc	47	764	6.15%
gobmk	7	964	0.72%
gromacs	2	7692	0.020%
h264ref	1	1156	0.086%
hmmer	2	2094	0.095%
lbm	2	24119	0.0082%
ls3D	18172	49957	36.37%
libquantum	12348	53478	23.09%
namd	4	6436	0.0621%
omntpp	1	7854	0.012%
povray	28	3112	0.89%
sjeng	4	7749	0.051%
soplex	419	6103	6.86%
sphinx3	2	16848	0.011%
tonto	2	9292	0.021%
xlcbmk	4	5849	0.068%
zsmp	72	8752	0.82%

Table 9.1: Valid frequency shift versus non valid ones

However, the overhead of the Markovian decision maker is high. The system has to wait until the prediction graph is built, and if facing unpredictable behavior, the graph can be flushed and rebuilt during the entire application execution not giving frequencies to apply. In addition, in the case of predictable behavior, boundness ratio has to be the same each time each the same single phase is executed. This is generally not the case. As presented in the naïve decision maker the same phase can be characterized by a range a value. Each value within that range leads to a different node in the prediction graph. Making impossible the prediction mechanism since multiple path in the graph could be walked. A mean is needed to prevent the undesired node from being added in the prediction graph. One way to perform that is using the mechanism presented in Section 9.2.2.2, therefore only nodes representing valid phase shifts will be added into the prediction graph. Table 9.1 shows the number of valid phase shift against the undesired ones identified when using the branch-predict phase identification procedure. It can be seen that the number of valid phase shift is rather small. Even though the markovian system is able to correctly predict the frequency shift and set the best frequency at the start of each new phase, on average the energy reduction granted by this system would have been identical to the branch-predict one. The period of time between the actual

phase shift and the correct frequency application is the major difference between the markovian and branch prediction decision maker. During that time slice, the markovian will prevent the system in wasting energy while using a non optimal frequency, leading to a more efficient energy reduction. But one can object that the difference in energy consumption from two consecutive frequencies is not significant. Therefore, the overall energy consumption obtained from the use of the markovian prediction will not be drastically better than the one obtained from the use of the branch-predictor. That is why in the next section, only the energy reduction and the impact on application performance of the naïve and branch-prediction decision maker will be discussed.

Finally, each decision maker had a common point, if the same frequency were chosen across several profiling samples, the sampling frequency was decreased. As the decider is selecting the same frequency, it means that the application is still executing the same phase. There is no need to stills rapidly wake up REST, but as soon as a new phase is spotted the original sampling period is resumed. By doing so, REST's activity has a very small impact on the application's execution. Of course small phases can be missed, but they are not the source of great energy consumption reduction. And if the application is only composed of small phases, the feed back mechanism for the sampling period will not be triggered.

Once the decider mechanism has selected the new frequency to apply, it is sent to REST's frequency driver which is in charge to ask the hardware to change the operating frequency.

9.2.2.4 Frequency Driver

The last component needed by REST to apply the selected frequency is a frequency driver. As presented in Chapter 6, the Linux operating system uses the *cpufreq* module to provide an interface for managing CPU frequencies. REST frequency driver is on top of *cpufreq* and uses the *sysfs* interface to change the frequency.

REST assumes that each core has an independent voltage supply, making a frequency decision for each available core. The used architecture does not have an independent voltage supply per core, it is only available at the processor scale. When facing several requests to frequency switch, the *cpufreq* module applies the highest frequency among those of the requests. For example, if a processor has 4 cores, and each one respectively asks for 2.1GHz,2.0GHz,2.0GHz and 1.9GHz, only the 2.1GHz will be applied. Another behavior must also be considered. If the number of requests is lower than the number of available cores, the frequency used on the core not requesting a new frequency is considered when determining the highest. In this case, if three cores upon the four ask for the lowest frequency whereas the last core uses a higher frequency, the requests won't be taken into account and the higher frequency will be used.

The energy savings presented in Section 9.3 are achieved while using all the cores. The applied frequency indeed reflect REST's frequency selections and not a undefined behavior.

9.3 The Cost of Energy Savings

REST was developed to be application independent. It has to start when the application starts. REST transparently initializes itself by the use of the LD_PRELOAD

environment variable at the start of a program and configures all necessary decision makers through REST specific environment variables.

Two benchmark suites were used to show REST's energy saving capability. First the sequential SPEC2006 benchmark suite [157] was used. The application, though sequential, was executed on each core simultaneously to simulate a full workload. An internally developed tool, MicroLauncher [20], ensured all processes were pinned, synchronized, and uninterrupted to attain the most stable results. The SPEC programs are relatively complex and complete programs, and for some of them, the simulation required more memory than was physically available and forced paging to occur. Such cases were dropped from the study because correctly tuned programs should always fit into the available physical memory. Second, to prove that even with real parallel applications REST can perform energy savings the NAS benchmarks suite [14] was used.

Finally, to prove REST is fully capable to adapt to different architecture and software environment, two experimental platforms with different tool chains were selected as summarized in Table 9.2.

Model Number	X5650	E3-1240
Cores	2 x 6	4
Memory	8 Gb	4 Gb
PowerMeter	Yokogawa WT210	Hardware counters
Operating System	Linux 2.6.38	Linux 2.6.38
Compilers	Gcc 4.6 - Ifort 12.1	Gcc 4.6

Table 9.2: Experimental Testbed

Table 9.2 shows that the Westmere X5650 architecture uses two processors. REST is able to transparently perform energy savings either with one or more physical CPU within the same machine as it only considers cores. REST sends its decision to the *cpufreq* module to be transparently condensed into one frequency shift per processor as explained above. REST also adapts itself to different energy probing systems. For the Westmere X5650 architecture, a digital power meter was used to measure the full machine energy consumption. For the SandyBridge E3-1240 machine, the CPU's dedicated hardware counter was used. The modularity makes REST able to provide frequency decisions for energy reduction, whether its is measured on a full system or only on a CPU.

The choice of both architectures was also driven by the fact that some NAS benchmarks needed either a power of two or a quadratic number of cores. So it would have led to parasitic frequency selection since less than the twelve cores would have been used. The parallel execution of the SPEC2006 was performed on the twelve core machine whereas the NAS was executed on the four core machine.

Figure 9.7 shows how REST is able to reduce the energy consumption for the SPEC2006 benchmark suite executed on the dual processor setup with the digital power meter. As explained above, all the benchmarks within the suite are not displayed in the figure, since the parallel execution forcing paging to occur.

Each bar represents energy savings or performance degradation. Energy saving is obtained with almost no performance slowdown, for sphinx3, lbm, or libquantum(libq). REST is achieving energy reduction without degrading the performance

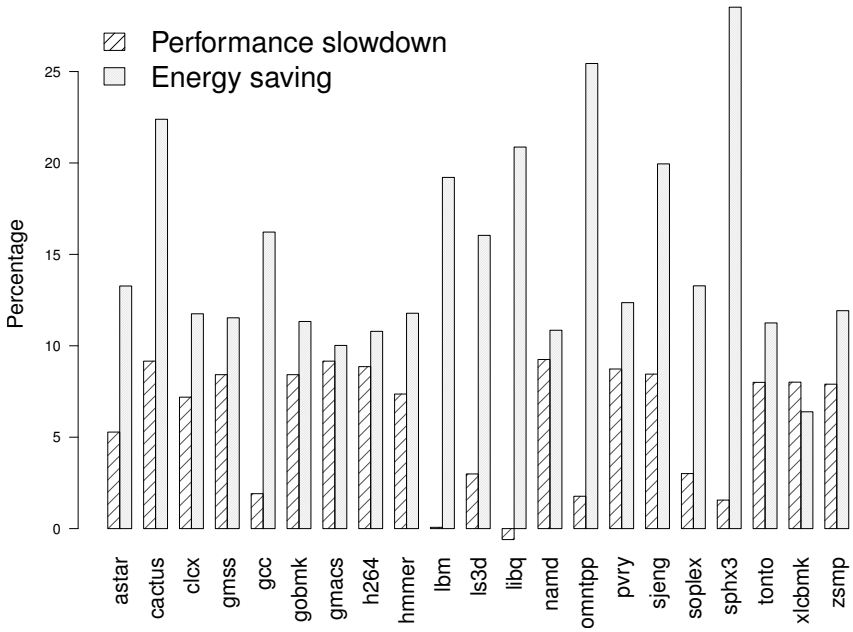


Figure 9.7: REST energy savings and performance degradation on the SPEC 2006 benchmark suite, with the most naïve decision maker

since their execution behavior is equivalent to the external boundness trend as diagnosed in Section 8.1. As a recall, external resource boundness, exposes a constant execution time over the different frequencies whereas the energy consumption decrease when selecting lower frequencies. Therefore, selecting the lowest frequencies exposes significant energy reduction without arming performances.

In the majority of cases, the energy saving are greater than the performance degradation, which validates the fact that even by blindly slowing down the execution, energy reduction can be performed. Note that in some cases like libquantum (libq), throttling the frequencies actually increases performance, likely due to reduced conflicts in buffers and coherence buses. It can also be due to a borderline effect, since it is within the error margin.

It can also be seen that aside the assumed memory bound applications all other energy optimization were obtained with performance degradation. Indeed, in Section 8.1, for CPU-bound and balanced applications, lowering the frequencies means degrading the application's execution time. As REST achieves energy reduction on each application, they are not CPU-bound. Indeed, if one was purely CPU-bound REST could have done nothing, which implies letting the higher frequency for the entire application execution. The benchmarks were executed in parallel to completely occupy the system. Therefore, some bottlenecks happen, lowering the application CPU stress making them shift from CPU-bound trend to a more balanced one.

One feature, which has not been mentioned yet, is the case when REST only selects fixed frequencies, i.e. it does not select the Turbo Boost frequency. Had the system allowed Turbo Boost frequencies, the results would have varied slightly. For memory bound programs, the results are identical because REST never selects the

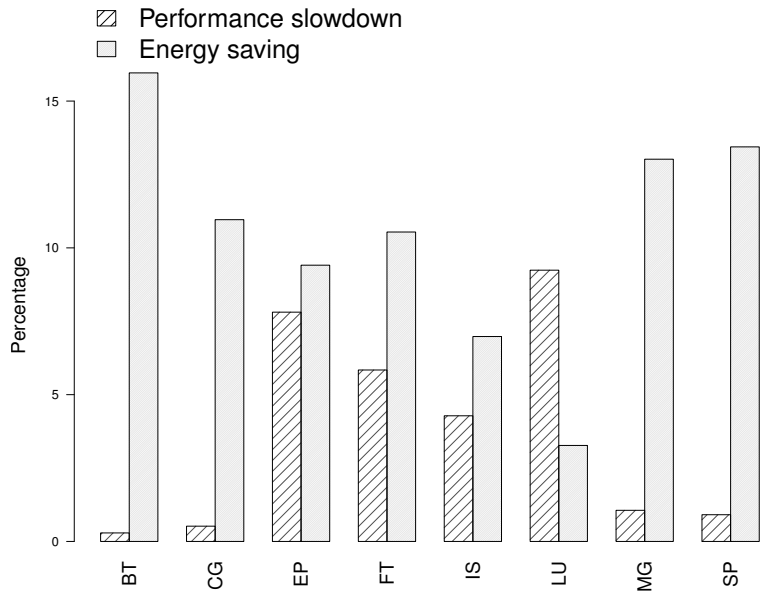


Figure 9.8: REST energy savings and performance degradation on the parallel NAS benchmarks using the naïve decision maker

upper frequencies. However, for compute bound frequencies, the results would have been close to 0% for both energy savings and performance degradation. Indeed, for such programs, the decision makers select the highest frequencies and maintain them for the rest of the execution. Since the OnDemand Linux governor would also go into Turbo Boost mode, both REST and OnDemand would have identical behavior therefore, identical results. The reason REST restricted itself to static frequencies was to provide insight on the cost effectiveness of the Turbo Boost mode. If a user considers energy savings though, it is moot to select Turbo Boost since, according to the results, it gains in performance but actually reduces the power/performance ratio. On the dual processor platform, using Turbo Boost means dramatic increase in power consumption. Overclocking the execution frequency, apart from increasing the operating voltage accordingly, increases the die temperature. As explained in Chapter 5, the increase in temperature leads to higher power leakage and fans consumptions. This power consumption increase cannot be countered by the speedup of the overclocked frequency. The dramatic power increase can be noticed on Figures 8.2, 8.3, 8.4.

In addition to SPEC results, Figure 9.8 presents the results obtained on the parallel NAS benchmark suite. The runs are performed using Class C benchmark sizes. As BT, CG, MG, and SP in Figure 9.8 show, there is an opportunity for large energy savings at minimal performance degradation when MPI communications overlap the slower processing. However, LU, which is highly coupled to its messaging and scheduling suffers from skew introduced by the REST runtime. As REST independently slows down each processing phase between communications without taking into account their overall impact on the application, it generates a significant amount of slack time. Slack time, is a time slice where a process is waiting for a message to arrive. If the sending process is slowed down by REST before sending the needed message, the receiving process will have to wait for it

longer. That unbalance can take dramatic proportion as almost all processes are interlinked. LU is the best example of what happens when skew are introduced in a distributed application.

9.4 The More The Better ?

As exposed previously, REST implements several decision makers, each one of them using an increased level of complexity. Therefore, any one can expect that the more intelligent decider exposes better energy savings. But sometime the difference between the solution given by the moire complex system is not worth the cost when compared with a simpler system. It is the case for REST.

The previously presented results are obtained by running REST with the naïve decision maker. Figure 9.9 presents the results obtained by the predictive decision maker, and one could notice that there is no drastic change between them. The variations on energy savings and performance slow-down are within 2% as shown in Figure 9.10 and 9.11.

Figures 9.10 and 9.11 present a percentage point comparison in energy savings and performance degradation between the naïve and the branch-predictor decision makers. The values are obtained by calculating the difference between percentage gains or losses between each predictor. The difference between the two is limited. Though the branch predictor may achieve better results in certain cases, the effort required is not automatically worth the complexity.

Branch prediction on the majority of the tested benchmarks does not significantly exhibit more energy reduction than the naïve version. It has less impact on the application execution time, since the branch-predictor decision prevents the

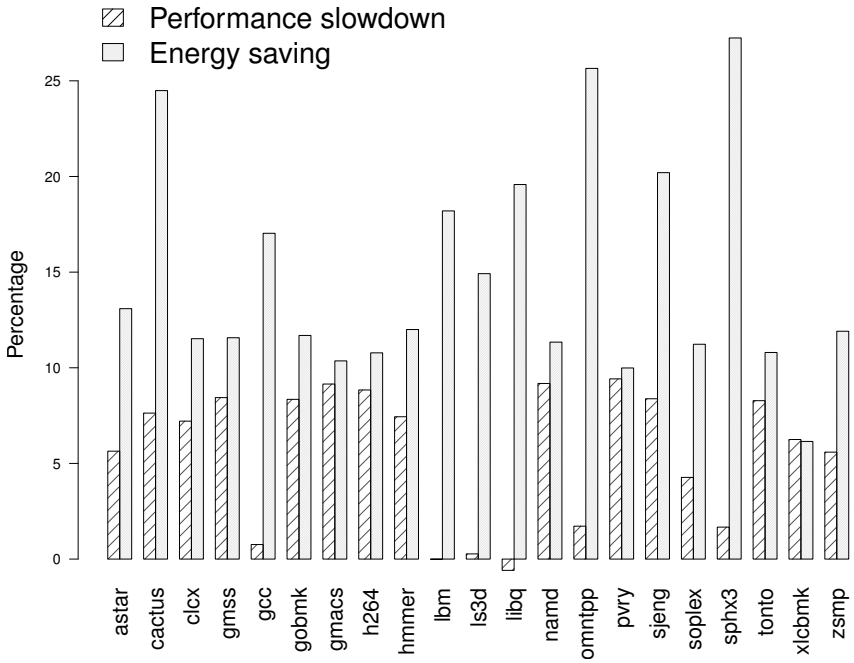


Figure 9.9: REST energy savings and performance degradation on the SPEC benchmarks using branch prediction decision maker

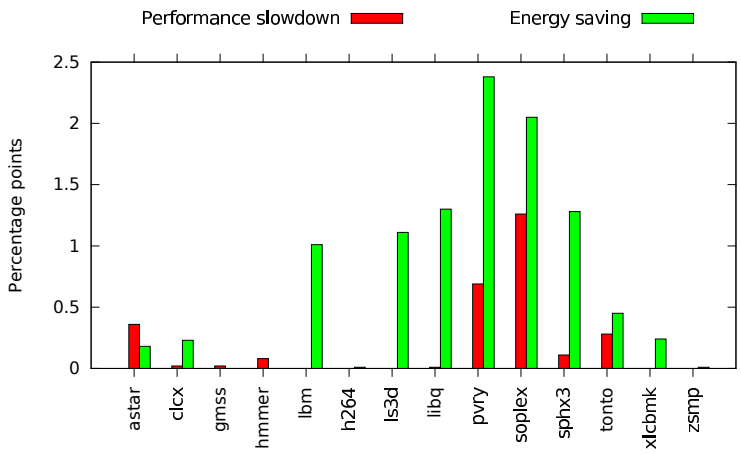


Figure 9.10: Naïve decisions lead to better energy saving and lower performance slowdown on a subset of tested applications

frequency selection from constant-shifts. In the end, to invest in complex decision system, does not always mean better solution. In REST case, using the naïve version gives a good approximation on energy savings.

REST can perform good energy saving with decent performance degradation. Unfortunately as there is no way to be sure that all the energy saving can be achieved, one can legitimately question REST efficiency or potential. The next chapter answers the question. One can also wonder about performance degradation. Indeed, REST cannot estimate the real impact of each frequency selection on energy consumption and execution time. Such trade offs are discussed in Chapter 11 where a study is performed to enhance the energy reduction while strictly limiting the performance degradation.

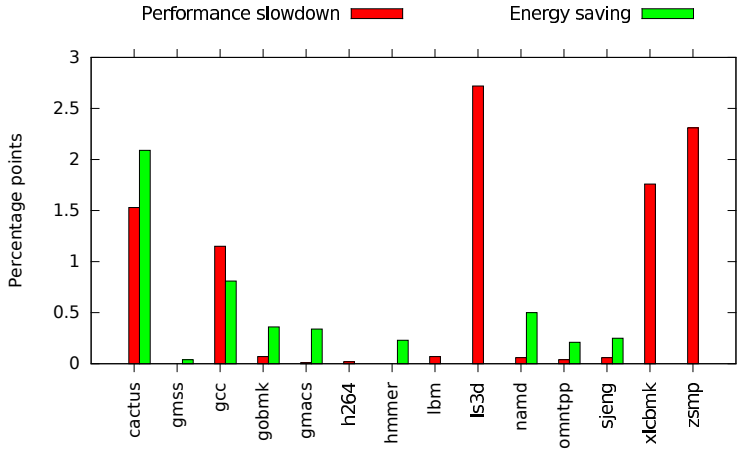


Figure 9.11: Branch prediction decisions lead to better energy saving and lower performance slowdown on a subset of tested applications

Chapter 10

UtoPeak

The previous chapter described a dynamic tool intending to reduce the energy consumption of an application execution. While the application is running, REST determines the trend of each application phase in order to select the best frequency. REST achieves a significant amount of energy reduction even with the naive control over the frequency. Therefore one can question the efficiency of such a tool regarding energy consumption reduction. Moreover, multiple DVFS controllers exist [86, 47, 143, 145] with various levels of complexity leveraging different amounts of energy consumption reduction, but none of them evaluate the real efficiency of their solution.

UtoPeak was designed in order to primarily evaluate the efficiency of REST. The goal is to compute the maximum reduction of energy consumption that one can expect from the use of any DVFS controller. If the higher bound on energy reduction is known, it gives other DVFS solutions a reference point to compare their own reduction with the maximum. It gives the users a mean to compare all the DVFS systems at hand and then to select the most appropriate regarding their constraints.

10.1 State of The Art

Even though UtoPeak shares common characteristics with run-time DVFS system like REST, [86, 87, 158], such as the frequency selection regarding the program phases, or the use of different frequencies during application execution, UtoPeak is a static method using an offline energy study to build a frequency sequence as it is presented below.

Hotta *et. al.* [70] present a system close to UtoPeak. A frequency sequence is build based on information gathered during an application profiling step. The evaluated application is then run while using the sequence of frequency. Unlike UtoPeak, the application execution performed during the profiling step contains instrumentation code in order to help application phases identification. The granularity of such instrumentation is function based because finer grain could induce perturbation on the profiling information. Utopeak uses a helper thread to track application phases, thus no modification to the original application binary is needed and the overhead remains limited. In addition, as the helper thread measurement sampling rate often represents hundreds of milliseconds, UtoPeak can track program phases at finer grain such as loops.

Freeh *et. al.* [46] also describe a system strongly related to UtoPeak. It splits the evaluated application in different phases, and for each one, a frequency setting is chosen in order to satisfy a constraint. The major difference with UtoPeak comes from the number of profiling runs involved during the profiling step. Indeed, to achieve the optimal heuristic, the evaluated application is run $n \times f$ times, where n is the number of phases and f is the number of frequencies. UtoPeak only needs n

runs to compute the optimal frequency sequence. Moreover, in the parallel context $n \gg f$, therefore UtoPeak has a lower overhead when producing the frequency sequence.

Ge *et. al.* propose in [50] several techniques to optimize energy consumption. Two of them are related to UtoPeak as they involve prior application energy consumption study. The first one requires application profiling over different frequencies and sets the best static frequency for the entire application execution, unlike UtoPeak which uses a sequence. The second approach identifies application phases through instrumentation in addition to energy consumption measurements. As for Hotta *et. al.*'s system, API function calls are injected around identified code blocks to set the correct frequency setting. The evaluated application is then recompiled. However, too many API function calls injection can greatly modify the application behavior by preventing the compiler from performing some optimizations, potentially adding significant overhead on the energy consumption. UtoPeak, by using an helper thread, operates the needed frequency switch outside of the application binary, thus has a lower impact on the overall application energy consumption.

Kolpe *et. al.* [97] propose a system also very close to UtoPeak. It slice the application into multiple step of fixed durations. Unlike UtoPeak, for a time step the Kolpe *et. al.*'s approach considers all the combination of frequency. For example, at step N, if only two frequencies are available, 2^N combination are evaluated, inducing an combinatorial explosion.

10.2 Under the Hood

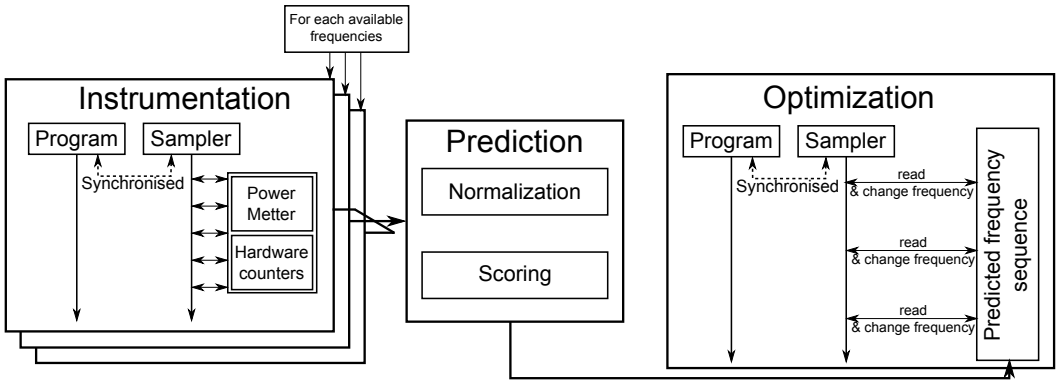


Figure 10.1: UtoPeak's general overview.

As said previously, UtoPeak intends to compute the maximum energy reduction possible for one application's execution. UtoPeak takes advantage of the boundness of each application's phase to select the frequency giving the lowest energy consumption disregarding the execution time's degradation. To do so, it has to identify each application phase and record the impact of each frequency on their energy needs. The recording is achieved through application profiling of each CPU frequency. Once all the application phases are identified, UtoPeak searches for the best frequency for each one and predicts their impact on the overall application's energy consumption. Once it is done, all the frequencies are gathered as a sequence and used while the application is run one last time to identify the energy savings. Figure 10.1 shows the interaction between the three described steps. How and why profiling is per-

formed, how the application phase identification and frequency matching is done are explained in the next subsections.

10.2.1 The Necessity of Profiling

It has been shown in Section 8.2 how application phase identification can be performed. Either via static code analysis or via dynamic code instrumentation. The static code analysis is generally limited to functions or loops whereas dynamic code instrumentation is not as limited. It can target basic blocks or even smaller code blocks provided that probes resolution is precise enough. UtoPeak phase energy study could have been performed with both solutions but the dynamic profiling has two major advantages.

First, performing the full application phase study via static phase extraction needs $N \times F$ execution. All N phases have to be executed on the F CPU frequencies, which can be time consuming. Even more in a parallel context with $N \gg F$ it can lead to a combinatorial explosion as discussed in Part III.

Secondly, once phases are extracted, the application has to be run to retrieve the phase execution sequence as well as the number of calls for each one of them. Retrieving the sequence of phase execution is important in order to correctly schedule the frequency shifts for the frequency sequence evaluation. The number of calls is also important to evaluate the weight of the application phases and accurately predict its influence on the overall optimized energy consumption.

In addition to the complexity of phase extraction, numerous application executions are required to retrieve all the needed information while dynamic profiling only needs F application executions without prior application knowledge.

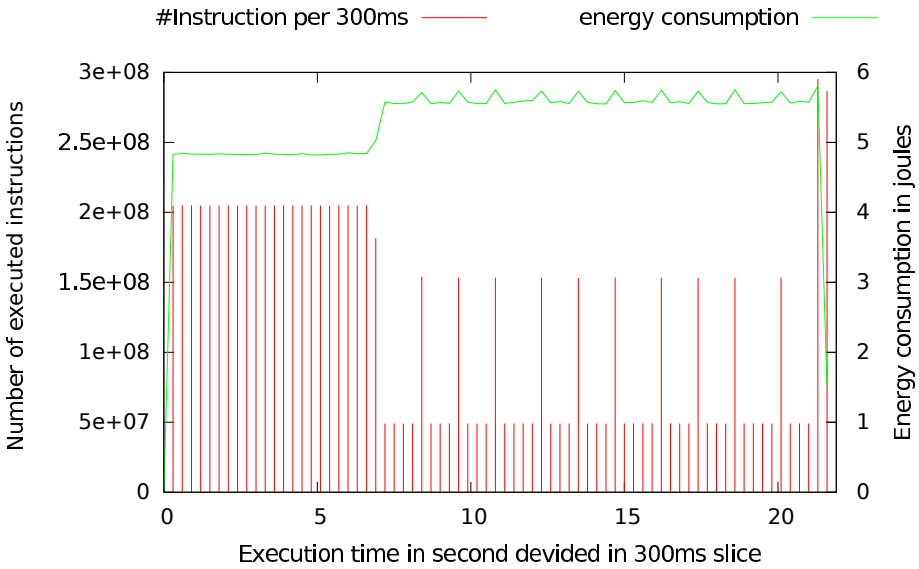


Figure 10.2: UtoPeak profiling information during IS execution at 1.6GHz.

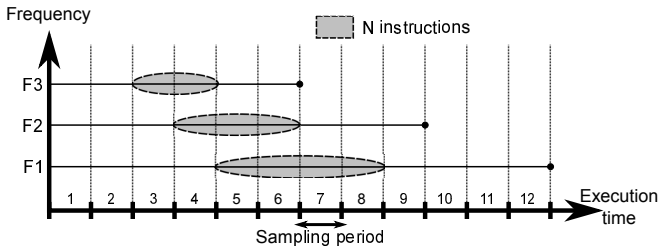
Figure 10.2 shows all the information needed by UtoPeak. The IS benchmark program was run on all the CPU frequencies while periodically monitoring the number of executed instructions and the CPU energy consumption. Figure 10.2 shows the profiling information for only one frequency. As said above, there is no need to have prior knowledge to identify different application phases. Here, in the case of IS, four distinct phases are identified as the CPU undergoes different levels of stress. It

starts by intensively executing instructions. It is then followed by ten alternations of higher and lower CPU stress, and finally finishes by the highest CPU stress.

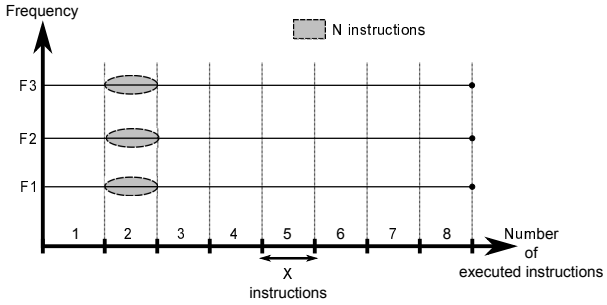
The explanation of the varying CPU stress can be found in the IS source code. As for the sequential version, there are three main functions : *create_seq*, *rank* and, *full_verify*. However, there is a major difference between the execution exposed in Figure 10.2 and the one studied in the Section 8.2. The version here is the parallel version of IS using openMP. The injection of openMP pragma for transparent parallelism is the only difference with the sequential source code. The function *create_seq* is called at the beginning of the application, followed by 11 calls to *rank*, and ended by a call to *full_verify*. The eleven calls to *rank* are due to the selected size problem.

One can see that the execution sequence found by looking in the source code almost matches the evolution displayed in Figure 10.2. By looking deeper in the IS source code, it can be seen that *rank* is not fully parallel, a sub part of the function remains sequential. It explains why during the eleven calls to *rank*, there is a spike followed by a lower amount of executed instructions. In the case of the static phase extraction explained in Section 8.2, only the openMP region is considered.

Without prior knowledge of the application, dynamic profiling is a good way to catch important application phases.



(a) Application time-based sampling.



(b) Application instruction-based sampling.

Figure 10.3: Difference between time-based and instruction-based sampling.

The dynamic profiling, as shown in Figure 10.2 is performed by sampling execution time. As said above, the profiling has to be performed on each CPU frequency. Unfortunately, as exposed in Section 8.1, each application has a different sensitivity to frequency. Indeed, the execution time generally varies with the frequency. With constant time sampling, a shorter execution time implies fewer samples.

Figure 14.6a shows differences in the number of time samples for a theoretical application execution. The gray area represents one application phase. When using different frequencies, the application phase is executed on a different number of time samples. Moreover, the fifth time sample in Figure 14.6a does not represent the same grey area segment under the different frequencies. It is impossible to

directly compare time samples energy consumption under different frequencies.

To solve the problem, one could dump the dynamic profiling to get back to static phase extraction and comparison. Comparing the same phase under different frequency executions, means comparing the same code section, in other words, the same number of instructions. Though one application phase has different execution times under different frequencies, the number of executed instructions remains the same. To solve the problem illustrated in Figure 14.6a, the application profiling has then to be performed on the number of executed instructions as shown in Figure 14.6b.

Unfortunately, instruction-based sampling is not easily done out of the box. Instead of a fixed period of time, a fixed number of instructions has to be chosen to trigger the probe reading. Each application phase puts the CPU in a different level of stress, meaning a varying number of executed instructions per cycle (IPC) while the application is executed. A varying IPC implies a varying execution time for each instruction sample. Yet, each probe has a specific resolution which is expressed as a period of time, therefore the execution time of each instruction sample has to be at least equal to each probe time resolution. The IPC evolution through the application execution has then to be known. Since UtoPeak has a significant profiling overhead, adding further profiling is not affordable. Application sampling is performed based on time and UtoPeak performs a conversion from time-based measurements to instruction-based ones. The conversion is detailed in the next subsection and corresponds to the normalization step in Figure 10.1.

10.2.2 Normalization and Prediction

The normalization step is needed because the instruction based sampling is not possible out of the box. As the number of samples is linked to the application execution time, the longer the application lasts, the larger the sample vector is. The normalization process has to be kept simple to be fast and lightweight. The gathered data inputs correspond to the the first three columns from Table 10.1. Here only two time samples are used to demonstrate how the normalization process is done. The *Inst.* column and $e(TS)$ represent the number of executed instructions and their corresponding energy consumption measured on a time sample *TS*.

Based on the available information, the following equation shows how to compute the average energy consumption $e^{TS}(i)$ per single instruction i for a time sample TS:

$$e^{TS}(i) = \frac{e(TS)}{\#Instruction(TS)} \tag{10.1}$$

TS	Inst.	$e(TS)$	$e^{TS}(i)$	IS	Inst.	$e(IS)$
1	6	3	0.5	1	5	5×0.5
2	4	1	0.25	2	5	$0.5 + 4 \times 0.25$

Table 10.1: Theoretical program sampling and normalization results

$e(TS)$ represents the energy consumed during a time sample and $\#Instruction(TS)$ is the number of instructions executed on the same time sample.

The $e^{TS}(i)$ column of Table 10.1 shows the average energy per instruction computed by using the Equation 10.1 on each time sample. The same process is repeated

for every frequency. By comparing each executed instruction under the frequencies based on their average energy consumption, a frequency sequence can be built.

However changing frequencies takes time, as shown in Subsection 8.3, and the delay to set a new frequency is longer than executing a single instruction. To ensure that the delay for changing frequencies will be negligible, the normalization process is done for several instructions, defining an instruction sample. By summing the energy consumption per instruction $e(i)$, for each instruction belonging to an instruction sample IS, one can compute its energy consumption $e(IS)$:

$$e(IS) = \sum_{i \in IS} e(i) \tag{10.2}$$

By using Equation 10.2 and considering the instruction sample size is five instructions in our example, the time samples are now converted into instruction samples as shown in the last column of Table 10.1. Notice that, in many cases, instructions belonging to an instruction sample do not come from a unique time sample, thus $e(IS)$ is computed from several time samples involved in the instruction sample composition.

One can object, as for time-based sampling, that deterioration on the normalization process can be induced by not correctly sizing the instruction sample. If the instruction samples are too small or too huge the normalized energy consumption will no longer reflect the consumption evolution noticed during the profiling step. Therefore, it was decided to select for each benchmark, the smallest amount of instructions executed on a time sample. It ensures that even a small phase or a CPU not intensive phase is taken into account. One could have used an arbitrary instruction sample size for all the benchmarks, but each one of them differently interacts with the hardware, and using different sizes help UtoPeak to grasp the uniqueness of each application and hardware.

After the normalization process, UtoPeak knows, for every frequency, the energy consumption per instruction sample. By comparing all the samples over the different frequencies, the tool selects the one granting the lowest energy consumption. Utopeak repeats the process for each instruction sample and builds a sequence of frequencies. Table 10.2 shows an example. The predicted energy consumption is obtained by summing $e(IS)$ for every selected instruction sample.

IS	1	2	3	4	5	6	7	8
Frequency	F1	F1	F2	F3	F2	F1	F3	F3

Table 10.2: Theroritical application's frequency sequence.

Once the frequency sequence and the best theoretical energy consumption for the evaluated application are computed, the sequence is used in the last step to evaluate the energy prediction precision.

In order to evaluate precision, the frequency sequence player starts as a new thread at the beginning of the application. The goal of this new thread is to potentially change the frequency every time a full instruction sample has been executed. As instruction-based profiling is not possible on our testbed, the watcher thread periodically wakes up to retrieve the number of executed instructions and determines if the current instruction sample has been fully executed. It then sets the next fre-

quency if needed. It repeats the process for each instruction sample until the end of the application execution.

At the end of the evaluation, the overall application execution is measured and compared to the prediction to ascertain the precision of the prediction. The final step is only done to prove the accuracy of UtoPeak and to demonstrate to users that they can positively rely on its predictions to evaluate the best energy consumption they can expect when using any DVFS controller.

10.3 UtoPeak Assumptions

The following section presents the different assumptions made by UtoPeak. First, the section shows the variations in the number of executed instructions over different runs and how to maintain it as low as possible. Then, the section presents how the frequency transition latency can impact the sequence evaluation.

10.3.1 Constant Number of Executed Instructions

In order to compare the energy consumption of the same program phase under different frequencies, UtoPeak assumes the number of instructions to be executed remains constant across runs. Furthermore, UtoPeak supposes the application execution to be fully reproducible.

The variation in number of executed instructions between runs is in average 0.05% and 0.16%, respectively for SPEC2006 [157] and NAS-OMP [14]. Therefore, for the considered benchmark suites, there is little or no variation between runs, validating the assumption.

As a caveat, to achieve such a low variation, even in the parallel context, the execution environment is controlled in order to get the most deterministic execution as possible. To prevent the operating system from moving the different processes or threads around available cores, each one is pinned on distinct cores. Moreover, in the case of parallel benchmarks, barriers can induce variable number of instructions to be executed due to the active polling. Thus, for OpenMP applications, a passive barrier implementation is used.

All the previously performed optimizations can have an impact on the execution time and on energy consumption. However, the variation between the application execution time between all optimizations or without them is measured to be 0.06% and 0.16%, respectively on SPEC2006 and NAS-OMP. One can indeed question the chosen optimizations since they have so little impact on the environment. However, drawing that conclusion has only been possible after hand. Preventing potential alteration of the measurements, insures a stable test environment.

Application executions can then be considered as fully reproducible.

10.3.2 Frequency Switch Latency

The second issue to consider is the time taken to switch frequencies. As explained in Section 8.3, it is expressed as the number of micro-seconds between the request of a new frequency and its actual setting. The latency on the experimental platform used to evaluate UtoPeak as described in Section 10.4 is comprised between 20 μs and 70 μs.

The time sampling, as explained above, is sized regarding probe resolution. To be able to use UtoPeak even on hardware architecture prior to SandyBridge, a

digital power metter is used, forcing UtoPeak's sampling period to be $300ms$. The frequency latency then represents at most 0.7% of the instruction sample execution time. As the impact of the frequency shift on each instruction sample is negligible, UtoPeak does not implement a specific mechanism to take that latency into account.

One can find or design an architecture where the frequency latency is huge enough not to be neglected. UtoPeak can solve that issue by simply prefetching the frequency in this way ensuring the switch is effective for the next instruction sample.

10.4 Prediction Versus Real World

The experiments are run on an Intel Core i5 2380P quad-core processor, running Linux 3.5.3. The sixteen processor frequencies range between 1.6 GHz and 3.1 GHz, plus a turbo mode. The benchmark programs consist in the NAS OpenMP parallel programs 3.0 running the C class datasets [14] and the sequential benchmark programs SPEC2006. All sequential and OpenMP programs are compiled using the GNU compiler (version 4.7) and `O3` optimization flags. Energy measurements are performed using energy probes embedded in the processor [80].

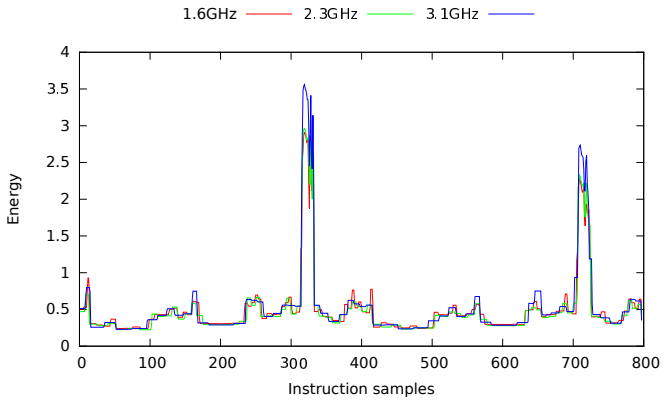
For REST, each SPEC2006 benchmark program are executed in order to simulate parallel execution running one instance per CPU core. However it as been seen after REST that simulating parallel executions in such a way generates an artificial stress on the memory or other resources. It changes the real application trend. In UtoPeak's case, only one instance of each SPEC2006 benchmark program are run to prevent artificial bottlenecks from helping the tool in its energy saving effort. For example, the ray-tracer *Povray* from the SPEC2006 benchmark suite, is known to be CPU bounded. However, when looking at REST energy savings, it exposes almost 13% of energy savings where UtoPeak only reach 2.58%.

Before comparing the maximum energy reduction to any other DVFS driver, the precision of UtoPeak prediction has to be evaluated. Table 10.3 shows UtoPeak predictions for different benchmarks programs. The *Prediction* column corresponds to the energy consumption predicted when building the frequency sequence for the evaluated application. The *Measured* column shows the energy consumption when applying the frequency sequence on the evaluated application. The last column presents the prediction precision as the difference in percentage between the two previous columns. In our test environment, UtoPeak reaches a prediction precision of 96.15% in average on the sequential benchmarks. UtoPeak obtains similar precisions on parallel benchmarks with 96.43% for NAS-OMP. The high percentages show UtoPeak accuracy. This means it is able to correctly predict the expected energy consumption when using the computed frequency sequence.

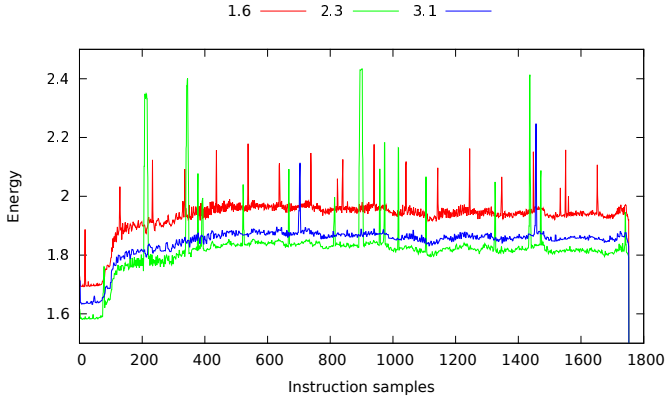
UtoPeak is able to accurately predict the application optimized energy consumption because it is the only one executed on the experimental setup. Though the measured hardware counters reflect only the execution of the profiled application and are not impacted by potential application run aside, the energy measurement is accounted for the entire CPU. So if any other application is run it will greatly skew the energy reading, tempering the energy normalization resulting in false predictions. The best example can be found for very short benchmarks as IS.C or 403.gcc. Any stress variation on the CPU in addition to the profiled application, for example the operating system, can induce sufficient variation on the energy probe readings to modify the energy normalization leading to the selection of non optimal frequency.

SPEC2006			
Benchmark	Prediction (Joule)	Measured (Joule)	Precision
453.povray	3 146	3 150	99.87%
464.h264ref	1 151	1 158	99.36%
456.hmmer	2 540	2 559	99.26%
471.omnetpp	4 219	4 258	99.07%
458.sjeng	8 702	8 805	98.82%
437.leslie3d	11 817	11 978	98.64%
470.lbm	5 619	5 713	98.32%
444.namd	7 097	7 223	98.23%
435.gromacs	8 081	8 276	97.59%
450.soplex	2 312	2 370	97.48%
482.sphinx3	10 349	10 613	97.45%
416.gamess	2 911	3 001	96.91%
483.xalancbmk	3 725	3 841	96.86%
436.cactusADM	11 302	11 668	96.76%
433.milc	5 428	5 616	96.54%
401.bzip2	1 594	1 652	96.42%
429.mcf	4 411	4 573	96.32%
447.dealII	5 343	5 540	96.31%
462.libquantum	6 543	6 800	96.06%
454.calculix	13 703	14 269	95.87%
445.gobmk	1 093	1 140	95.72%
410.bwaves	9 960	10 391	95.67%
400.perlbench	3 180	3 341	95.2%
465.tonto	8 446	8 946	94.08%
434.zeusmp	6 739	7 161	93.74%
459.GemsFDTD	9 039	9 690	92.80%
473.astar	2 331	2 507	92.42%
481.wrf	10 801	11 827	90.86%
403.gcc	336	381	86.72%
Average Precision		96.15%	
NAS-OMP			
EP.C	2 394	2 397	99.87%
CG.C	2 341	2 350	99.64%
BT.C	11 463	11 768	97.34%
MG.B	862	891	96.60%
LU.C	9 857	10 293	95.58%
SP.C	7 267	7 619	95.16%
FT.B	2 671	2 820	94.42%
IS.C	363	398	90.36%
Average Precision		96.12%	

Table 10.3: UtoPeak energy guessing precision for SPEC2006 and NAS-OMP sorted by decreasing precision



(a) GCC normalized energy



(b) POVRAY normalized energy

Figure 10.4: Normalized energy per instruction samples for *GCC* and *POVRAY*.

Therefore any external effects, can impact UtoPeak precision.

The difference between *GCC* and *POVRAY* prediction's accuracy finds its root in their application trend. *POVRAY* is a ray tracer, highly demanding for computing resources. *GCC*, on the other hand, is closer to the memory bound trend since it has to access files on disk. As shown in Section 8.1, a CPU-bound application, is extremely sensitive to frequency whether considering execution time or energy consumption. Hence its execution on the spectrum of frequency generates a wide range of energy levels, explaining the clear difference between the three frequencies displayed in Figure 10.4b. On the other hand, memory bound applications executions generate constant energy over the frequency space, explaining why *GCC*'s different energy levels are overlapping for the entire execution as shown in Figure 10.4a. However GCC is not fully memory bound, since the energy per instruction sample is not constant for the entire program.

If UtoPeak has to produce its prediction based on the information displayed in Figure 10.4, it chooses $2.3GHz$ for the entire execution of *POVRAY*, except for the different energy picks, where $3.1GHz$ is chosen. The resulting frequency sequence reflects the actual reality where $2.3GHz$ grants for each instruction sample the best energy consumption. It explains the 0.13% prediction error reported in Table 10.3. For *GCC*, the choice is more complex since there are no visible differences between the different frequencies' energy consumption. A different frequency can be chosen for each instruction sample inducing a more biased vision of the reality as shown in

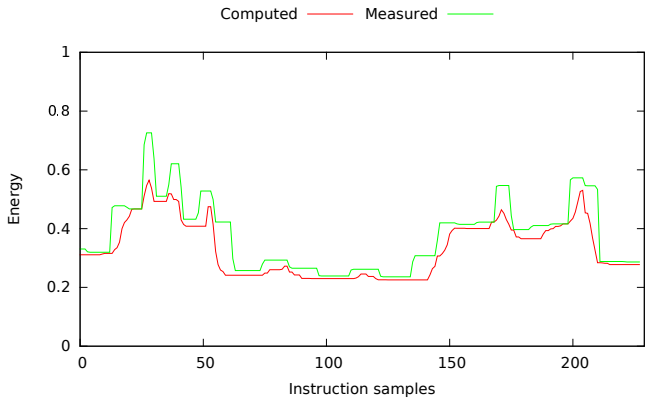


Figure 10.5: Energy consumption prediction error on GCC benchmark

Figure 10.5.

Figure 10.5 shows, the difference between the computed energy consumption per instruction samples, and the one actually monitored during the frequency sequence evaluation. In a nutshell, Figure 10.5 displays how well UtoPeak grasps the reality of energy consumption tendency of the predicted frequency sequence. Though UtoPeak masters the overall trend, it lacks in precision, explaining the 13.28% of error reported in Table 10.3.

The root of UtoPeak's lack in precision on *GCC* is twofold: the profiling period and its application type. For Table 10.3 the used period was $300ms$ for reasons exposed in Section 10.3. However it is too coarse a grain for *GCC*'s energy monitoring. It leads to the too overlapping energy per instruction sample displayed in Figure 10.4a. Lowering UtoPeak's profiling period to $50ms$, allows a finer energy monitoring reducing in the end the prediction error from 13.28% down to 7.90%. The remaining prediction error comes from the fact that the *GCC* benchmark program is sequential. Unlike *POVRAY* it does not put the processor under enough stress to allow UtoPeak to precisely distinguish each frequency's energy consumption trend. One theoretical solution can be to launch one instance of the program per processor core. It magnifies the CPU stress as well as the energy consumption. However, as exposed above, it creates a congestion on some shared resources, modifying the benchmark program execution. A more practical solution would be to adapt the profiling period to the application behavior. At that point UtoPeak needs additional runs to acknowledge the application behavior before computing the best profiling period. For some applications, it dramatically increases the time to solution which is not affordable.

This shows that UtoPeak has a good prediction precision in average on all tested benchmark programs. Therefore, in the experimental environment used, UtoPeak produces realistic DVFS energy consumption predictions, that can be later used with a high degree of confidence.

10.5 UtoPeak DVFS Potential

The DVFS potential is the maximum amount of energy that can be saved while using DVFS controllers. It is a key feature to decide whether to apply a DVFS mechanism.

SPEC2006			
Benchmark	DVFS potential	Benchmark	DVFS potential
445.gobmk	0.40%	447.dealII	2.98%
435.gromacs	1.25%	470.lbm	3.39%
401.bzip2	1.35%	434.zeusmp	3.50%
400.perlbech	1.43%	481.wrf	5.20%
416.gamess	1.48%	473.astar	6.11%
456.hmmer	1.97%	403.gcc	6.76%
437.leslie3d	2.06%	459.GemsFDTD	7.47%
454.calculix	2.23%	462.libquantum	8.62%
465.tonto	2.27%	436.cactusADM	8.79%
444.namd	2.38%	483.xalancbmk	8.80%
458.sjeng	2.47%	450.soplex	9.92%
464.h264ref	2.57%	433.milc	14.24%
482.sphinx3	2.67%	429.mcf	14.47%
453.povray	2.69%	471.omnetpp	16.07%
410.bwaves	2.72%		
average energy reduction potential		5.04%	

NAS-OMP	
Benchmark	DVFS potential
EP.C	15.29%
BT.C	25.19%
FT.C	26.45%
CG.C	27.46%
SP.C	38.70%
IS.C	41.34%
MG.C	44.28%
LU.C	45.29%
average energy reduction potential	33%

Table 10.4: DVFS energy reduction potential for SPEC2006 and NAS-OMP sorted by increasing DVFS potential

Tables 10.4 shows the energy reduction potential on sequential and parallel applications. UtoPeak energy reduction potential is computed as the difference, in percentage, between the highest frequency energy consumption and UtoPeak's one. The comparison is performed with the highest frequency, because on standard clusters, the Linux Ondemand frequency governor [129] is used by default and once it spots intense CPU activity, it applies the highest frequency including TurboBoost if activated. Here, the energy consumption induced by UtoPeak is compared to the one induces by highest frequency non TurboBoost. If it was compared with TurboBoost, the DVFS potential would only be greater, since the frequency is overclocked consuming more energy as shown in Figures 8.2,8.3,8.4 from Section 8.1.

Both tables show large differences between energy reduction potential one can

expect from sequential and parallel programs. On sequential application, UtoPeak average energy reduction is never greater than 10% whereas on parallel application it is never below 15%. The major difference between both kinds of application is the slack time [144]. In parallel application, slack time is a period during which a process is doing nothing else than waiting for other processes in barriers or communications. Lowering frequency during these phases provides significant energy reduction. Therefore, parallel applications have phases with higher potential of energy reduction.

Furthermore, DVFS potential of energy reduction, is linked to the used hardware. Even when processing the same application on a CPU with different voltages or various frequencies, one can note different potential of energy reduction.

Even if DVFS techniques do not grant significant energy savings on sequential applications, the potential of reduction is not negligible for parallel benchmarks. On some benchmarks, almost half of the CPU energy consumption can be saved thanks to DVFS.

It has been clearly stated that UtoPeak can accurately quantify energy reduction potential of any DVFS solutions. The previous section was dedicated to a naive DVFS mechanism only relying on CPU boundness to reduce energy consumption. Now that the user knows the lower bound on energy consumption, it can use it to verify REST efficiency.

10.6 UtoPeak Versus The World

The previous sections exposed how UtoPeak is able to accurately predict the lower bound in energy consumption for a various range of applications. The previous Chapter described REST, a naive attempt to optimize application energy consumption through dynamic voltage frequency scaling. Though it suffers from several flaws, the system was able to perform energy consumption reductions. But as there is no point of comparison no one could evaluate REST efficiency. UtoPeak was designed to give that reference point. Therefore REST energy savings are compared to the maximum possible ones as shown in Figure 10.6. It can clearly be seen that REST is far from optimal and still offers many opportunities for optimization.

Both UtoPeak and REST do not set limitations on the execution time degradation. But, REST tends to select frequency regarding the trend of the application. If an application is CPU-bound, high frequencies are selected, and for more memory bound applications, lower frequencies are more likely to be selected as explained in Section 8.1. It will indirectly be more conservative on performance degradation than UtoPeak. Therefore, REST is less aggressive on power consumption reduction, leading to lower energy reduction as it is shown in Figure 10.6. REST conservatism regarding application performance is clearly stated by the huge difference in energy reduction between both system on the LU benchmark. As said previously, letting REST evaluate the energy gain offered by each frequency as well as the boundness ratio, is one possible optimization to enhance its energy optimization.

UtoPeak is able to accurately predict the lowest energy consumption one can expect from the use of any DVFS system. By doing so, it gives users the possibility to evaluate the efficiency of such systems regarding energy reduction. REST with its naive frequency management was only able to optimize, on each benchmark program, a small portion of the full potential of energy reduction leveraged by UtoPeak. However, some control is better than no control at all. By acknowledging

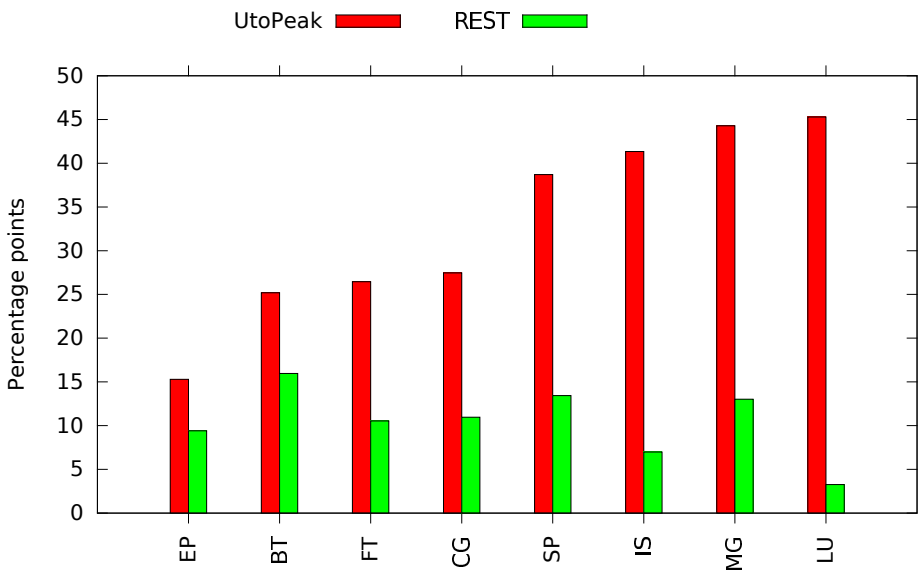


Figure 10.6: REST compared to UtoPeak

the room for optimization, up to 40% on LU, it allows the user to evaluates the interest in further optimizing the current solution. In the light of Figure 10.6, REST was reworked and the new system is presented in the next chapter.

Chapter 11

FoREST

11.1 State of The Art

As it was shown previously, many DVFS controllers were proposed in the past. Some of them focus on reducing energy consumption of a specific program during its execution while others consider the processor workload and do not require any program-specific knowledge. The latter predict the impact of frequency transition to decide on the frequency to apply. They exploit models correlating hardware counters to CPUboundness, and then CPUboundness to the sensitivity of the workload to frequency transitions.

Closer to FoREST work, Semeraro *et. al.* [153] proposed to periodically reduce CPU frequency until an impact on execution time is suspected from hardware observation. The proposed mechanism is not able to control its impact on slowdown as opposed to more recent solutions line in [52, 73].

Hsu *et. al.* [73] proposed *beta-adaptive*, a runtime DVFS controller that periodically evaluates the impact of frequencies on performance to deduce the best frequency to use under performance constraints. It shares several features with FoREST as it directly evaluates the impact of a frequency transition on Instruction Per Seconds (IPS) and reacts accordingly. In general, the existing dynamic DVFS controllers suffer from several limitations. First, several existing controllers exploit a complex model to estimate the impact of a frequency transition on energy. Such models heavily depends on the target hardware and may be quickly outdated. For instance, a recent study shows that memory bandwidth is now impacted by frequency transitions since the SandyBridge generation of Intel x86 CPUs [151]. Such subtle evolution, even within a micro-architecture, leads most of the existing models to fail. Moreover, all the presented systems are ignoring energy when selecting the frequency to apply. Indeed, most of them assume energy gains when reducing frequency while ensuring a relatively small slowdown. Such hypothesis is wrong on modern processors where energy consumption may increase when decreasing the frequency, depending on programs CPU usage. When FoREST chooses which frequency to apply, it considers impacts on both power and execution time. While other systems try to reach as much as possible the user requested slowdown, FoREST estimates what slowdown allows maximal energy gains at the system scale. Finally, multicore support is unclear for several systems and, in some cases, the method cannot fit current multicore processors where frequency has to be applied simultaneously to several cores. Thus, compared to existing DVFS controllers, FoREST is more suited to modern processors and, beyond compatibility, FoREST also takes advantage of recent hardware evolutions such as processors energy probes to effectively reduce the system energy consumption.

Rong *et. al.*[52] extend the *beta-adaptive* and various strategies are employed to predict the performance of the next time step. However, the presented method suffers from the same flaws presented above.

11.2 Motivation

Chapter 9 presented a DVFS controller, named REST, relying on application resource saturation monitoring. Depending on the boundedness criterion, a frequency was applied. REST demonstrated that even with a low intelligence some significant reduction of energy consumption could be achieved with an execution time degradation between zero and ten percent. Chapter 10, described UtoPeak. It was designed to produce the optimal frequency sequence in order to expose the maximum energy reduction one can expect for a given application execution. Using its ability to find the lower energy consumption bound, one could find that REST reduction of energy consumption was far from optimum, leaving opportunities for optimizations. The following chapter exposes the next version of REST, named FoREST, which intends to enhance energy optimization while fixing all REST's drawbacks.

Adding a strict limit on performance degradation is the first enhancement that could be done to REST. The advantages of such a limit is twofold. First, it will give users the control of the performance degradation. Second, in some cases, it will give the system more room to potentially do more aggressive energy optimizations. For example, take Astar or Soplex from Figure 9.7, displaying REST energy optimization on SPEC2006, both of them respectively have a 5% and 7% penalty on execution time. In the case where the user specify a strict limit of 10%, the system will be granted an additional 5% and 3% on the performance degradation. It will allow FoREST to target a wider range of frequency to potentially achieve more energy reduction. However, slowing down the application, does not automatically imply energy saving. For many programs, the highest frequency provides the largest energy savings and any slowdown actually leads to increase the energy consumption.

The ability to detect an efficient frequency is a significant improvement over REST. Indeed, existing run-time controllers usually do not consider power when predicting the frequency to use. It is then impossible to determine if a slowdown is profitable for energy or not. To be able to target the correct frequency, the system will need a feedback loop in order to evaluate the efficiency of each frequency. Because each frequency does not give the same power consumption and performance degradation, the feed-back loop is decomposed into two components: relative slowdown and power consumption estimation. The frequencies power consumption evaluation is explained in Subsection 11.2.1. The slowdown computation of each frequency is described in Subsection 11.2.2.

11.2.1 Power Ratio

The previous system applied a frequency for a phase only based on the boundedness criteria. Without feedback regarding the impact of the selected frequency on the power consumption, REST was not capable to say if the chosen frequency was the best to use. In order to correct that, and help FoREST selecting the best frequency, relative power consumption are computed. The relative power consumption will show which frequency offers the greater reduction of power consumption. The relative comparison is called power ratio. The power consumption is linked to the hardware and the application. The power ratios will then have to be computed prior to each application's execution, based on the application's thorough power consumption characterization, which is not affordable. To have a better idea on application power consumption, both the *NAS Parallel Benchmarks 3.0* suite and the

SPEC CPU 2006 are run on the same hardware, resulting in power consumptions per frequency displayed in Figure 11.1. Though NAS benchmark programs power needs are higher than the SPEC ones, the evolution across the different frequencies for each application is identical. So the evolution the power consumption seems to be primarily influenced by the hardware and not by the application.

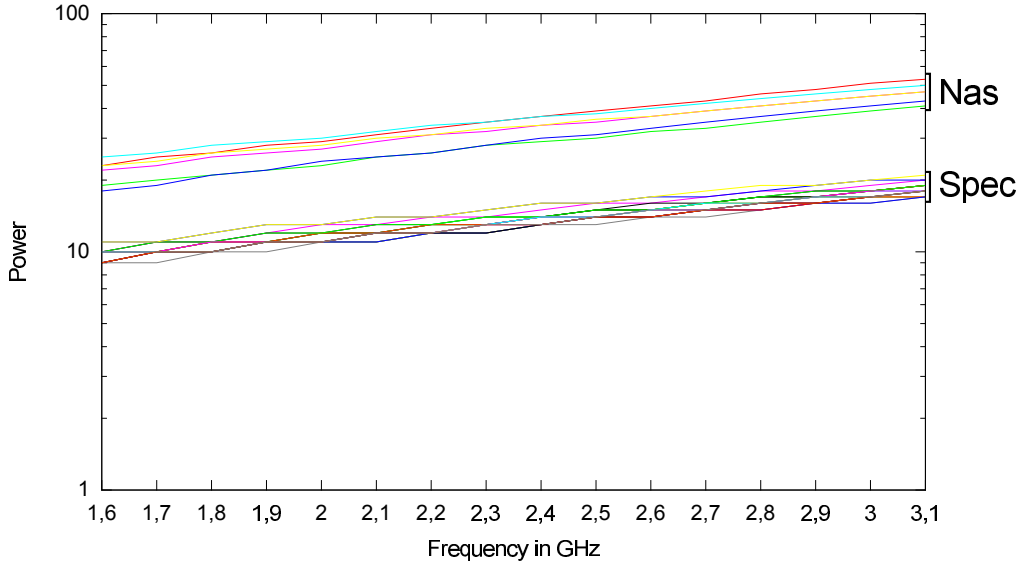


Figure 11.1: Power consumption evolution for SPEC and NAS benchmarks program

As presented in Chapter 6, the power consumption can be expressed as:

$$P = P_{static} + P_{dynamic} \tag{11.1}$$

Also, the dynamic power is expressed as:

$$P_{dynamic} \simeq A \times C \times V^2 \times f \tag{11.2}$$

where A is the percentage of active gates, C is the total capacitance load, V is the supply voltage, and f is the processor frequency. Note that the power depends on the machine characteristics (V, f, and C) and the program (A).

Many studies assume that $P_{dynamic}$ is proportional to P_{static}, [106, 133, 165] ,in other words:

$$P_{dynamic} = k \times P_{static} \tag{11.3}$$

By injecting Equations 11.2 and 11.3 into Equation 11.1 the total power is then proportional to the dynamic power:

$$P \simeq (k+1) \times P_{dynamic} \tag{11.4}$$

Based on the total power formulation, let P_1 and P_2 be the power induced after executing the same program at two different frequencies f_1 and f_2 provided the two executions were done on the same hardware. It is possible to compute the power ratio between P_1 and P_2 as follows:

$$\begin{aligned} P_1 &= (k_1+1) \times (A \times C_1 \times V_1^2 \times f_1) \\ P_2 &= (k_2+1) \times (A \times C_2 \times V_2^2 \times f_2) \\ \frac{P_1}{P_2} &= \frac{k_1+1}{k_2+1} \times \frac{C_1 \times V_1^2 \times f_1}{C_2 \times V_2^2 \times f_2} \end{aligned} \tag{11.5}$$

Formula 11.5 shows that the power ratio of two executions at different frequencies is independent from A, hence independent from the program. Besides, in [133], the authors showed that k does not depend on the program either, meaning that the ratio remains unchanged for all programs at frequencies f_1 and f_2. Therefore, it is possible to evaluate the power gain of all possible frequencies, over a reference frequency and reuse the information for any program. However, A is not really independent from the frequency. It is in fact an approximation as subtle variations can appear depending on the frequency. For instance, a memory-intensive program can saturate some resources such as store queues at high frequencies, leading different activities to occur on the processor depending on the frequency. Such variations were assumed to be negligible and consider the average number of active gates to be stable for a given program, independently from the frequency.

In order to verify the program independence of power consumption ratios, the *NAS Parallel Benchmarks 3.0* suite and *SPEC CPU 2006* were run using every processor frequency while measuring power consumption. It resulted in 688 different runs. Then, for any pair of frequencies, the power ratio induced by each program is computed. Finally, the standard deviation of the power ratios involving the same frequencies were computed while different programs were running on the same number of cores. The standard deviation expresses the average error of the theoretical calculation for the evaluated programs. Table 11.1 shows the power evolution of a subset of the run benchmark programs. It can be noticed that the evolutions of each program power needs are indeed equivalent.

Table 11.1 shows a subset of all the power ratio space. Like the difference noticed in power consumption in Figure 11.1, the parallel benchmark programs consume more power. But that difference does not impact power ratios, since the focus is only put on relative increase or decrease in power consumption. Here, if the frequency $1.7GHz$ is chosen to replace $1.6GHz$, the new setup will consume 4% more power. However, if the frequency $1.6GHz$ is chosen to replace $1.7GHz$, the frequency shift will reduce the power consumption by 4%.

Results showed a maximal standard deviation of 0.66 % for power ratios whereas power itself has a standard deviation of more than 5.1 W for different programs using the same frequency. Thus, even if power consumption obviously depends on the program itself, considering ratios of power consumption for different frequencies as being program-independent is a realistic hypothesis.

FoREST exploits power ratios to estimate the power gains achieved by any frequency. As the ratios are proven to be program-independent, the measurements can be transposed to any program. Alike insight in the power scaling implied by each frequency will help FoREST correctly select a frequency granting the highest energy optimization for each application phase.

Benchmark	Power at 1,6GHz (W)	Power at 1,7GHz (W)	Power Ratio $\frac{1,7GHz}{1,6GHz}$	Power Ratio $\frac{1,6GHz}{1,7GHz}$
400.perlbench	9,87	10,23	1,04	0,96
401.bzip2	9,69	10,12	1,04	0,96
403.gcc	10,06	10,48	1,04	0,96
410.bwaves	10,97	11,45	1,04	0,96
416.gamess	10,35	10,76	1,04	0,96
429.mcf	9,86	10,28	1,04	0,96
433.milc	10,77	11,27	1,05	0,96
434.zeusmp	9,94	10,39	1,05	0,96
435.gromacs	9,62	9,97	1,04	0,97
436.cactusADM	10,26	10,69	1,04	0,96
437.leslie3d	10,25	10,77	1,05	0,95
444.namd	9,81	10,17	1,04	0,96
445.gobmk	9,94	10,32	1,04	0,96
447.dealII	10,16	10,59	1,04	0,96
450.soplex	10,31	10,72	1,04	0,96
453.povray	10,23	10,71	1,05	0,96
454.calculix	10,21	10,60	1,04	0,96
456.hmmer	10,13	10,54	1,04	0,96
458.sjeng	9,89	10,27	1,04	0,96
459.GemsFDTD	10,64	11,05	1,04	0,96
462.libquantum	11,47	11,97	1,04	0,96
464.h264ref	10,23	10,61	1,04	0,96
465.tonto	10,06	10,48	1,04	0,96
470.lbm	11,39	11,90	1,04	0,96
471.omnetpp	9,84	10,20	1,04	0,97
473.astar	9,90	10,33	1,04	0,96
481.wrf	10,09	10,51	1,04	0,96
bt.C	23,64	25,02	1,06	0,94
cg.C	19,10	20,13	1,05	0,95
ep.C	18,84	19,93	1,06	0,95
sp.C	22,44	23,68	1,06	0,95
mg.C	25,56	26,81	1,05	0,95
lu.C	23,49	24,65	1,05	0,95
is.C	18,07	19,06	1,05	0,95
ft.C	24,45	25,86	1,06	0,95
Standard deviation	5,18	5,51	0,65%	0,65%

Table 11.1: Example of power ratios for two frequencies across all SPEC and NAS benchmark programs

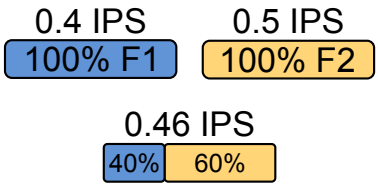


Figure 11.2: An example of a frequency pair and the associated IPS.

11.2.2 Continuous Frequency

As explained above, FoREST implements a slowdown limit. The system insures that the overall application performance will not be impacted beyond the desired slowdown limitation. However, applying a unique frequency, may not provide enough flexibility to meet the performance constraint while performing energy savings. Indeed, in some cases, all the frequencies lead to a slowdown greater than what the user tolerates. As an example, consider consider a theoretical application composed of only one task run on a processor using three frequencies, $F0$,$F1$, and $F2$. One wants to reduce the application's energy consumption without degrading its performances more than 5%. $F0$ is the highest frequency and is used as the reference. To achieve energy consumption reduction, the user will then choose a lower frequency between $F1$ and $F2$. If, between the two, one frequency grants energy reduction while degrading the application performances no more then 5% it will then be selected. If not, no frequency can be chosen meaning no energy reduction. However, it is possible to compose both $F1$ and $F2$ during the application execution to perform energy reduction within the performance constraint.

To evaluate the slowdown applied at each program phase, FoREST uses the Instruction Per Second (IPS) criterion to evaluate the *speed* of the computation. The user slowdown limitation is then transposed as a lower limit on the IPS to achieve at each program phase. Going under that limit means that the application execution time will be degraded more than the user limit. So when none of discrete frequencies can be used to ensure that IPS limitation, multiple frequencies can be used at the same time to emulate in average the IPS limit . When two frequencies are applied, the number of instruction achieved per second is proportional to the one achieved by every frequency [52, 72]. For example, it is possible to achieve 0.46 IPS using two frequencies able to perform respectively 0.4 IPS and 0.5 IPS, as illustrated in Figure 11.2.

Consider again the example presented at the beginning of the section. Say that the $F0$, $F1$, and $F2$ have respectively an IPS of 0.3,0.4 and 0.5 and the user specifies a limit at 53% of the measured performance on the highest frequency $F0$. The target IPS to achieve is then 0.46. $F1$ respect the performance constraint with a limited energy reduction. $F2$ on the other hand, though it ensures maximum energy reduction, cannot be selected as it will degrade the performance too much to meet the constraint. The possible solution, as shown in Figure 11.2, consists in

Algorithm 6 Best frequency pair generation

1: Build all possible frequency pairs
2: Compute the associated durations
3: Pick the pair with maximal energy savings
return $(bestPair, bestEGain)$

composing both $F1$ and $F2$ to ensure maximum energy reduction while meeting the performance constraint.

FoREST exploits this property and actually selects a couple of frequencies for every CPU's core to achieve any desired IPS compliant with the user requirements. For a specified slowdown, FoREST determines the best frequency couple to use in three successive steps, as illustrated in Algorithm 6.

Table 11.2 shows the different variables, and definitions, used in each Algorithm 6 step presented in Algorithm 7, 8, and 9.

Variable	Description
F	List of evaluated frequencies
d	Requested slowdown
$totalPairTime$	Duration of the overall pair execution
$lowerF$	lower pair's frequency
$greaterF$	higher pair's frequency
$PGain(f)$	Power gain of the frequency f over maximal frequency
$SPD(f,c)$	Speedup of frequency f on core c compared to the maximal frequency
$IPS(f,c)$	IPS measured for frequency f on core c
$maxIPS(c)$	Maximal IPS measured on core c for all frequencies
$bestPair$	Frequency pair with minimal energy consumption
$bestEGain$	Energy gain of the pair returned

Table 11.2: Best Frequency Pair Generation variable definition

First, as shown in Algorithm 7, all the frequencies pairs granting the desired *computation speed* to meet user's slowdown degradation have to be computed. The *computation speed* for each processor core, a.k.a. target IPS, is computed as the maximal IPS observed on the core among the evaluated frequencies, minus the desired slowdown. For example, consider a processor core with three different frequencies. If the measured IPS are 0.4,0.5, and 0.55, and the slowdown allowed by the user is 10%, the target IPS will then be $0.55 - 10\% \times 0.55 = 0.495$. The target IPS represents the objective IPS for a given processor core. To achieve such a target IPS, as presented above, a couple of frequency is built. The couple is composed of two frequencies $lowerF$ and a $higherF$ giving IPS surrounding the target, in Algorithm 7, it corresponds to lines 6 and 7. Multiple frequencies couples can produce IPS surrounding the target, each of them are considered valid. However, the cores may run various workloads that react differently to frequency transition. Every core then has a different target IPS. As a remainder of the first part, each core has to share the same frequency. Couples build for each core that do not share the same $lowerF$

Algorithm 7 All Possible Frequency Pairs Building

1: // ***1:*** *build all possible frequency pairs*
2: $lowerF \leftarrow F$
3: $greatrF \leftarrow F$
4: **for all** $c \in$ cores sharing the frequency setting **do**
5: $\quad target \leftarrow d \times maxIPS(c)$
6: $\quad lowerF \leftarrow lowerF \cap \{\forall f \in F,\ f \mid IPS(f,c) < target\}$
7: $\quad greatrF \leftarrow greatrF \cap \{\forall f \in F,\ f \mid IPS(f,c) \geq target\}$
8: **end for**

and $higherF$ are removed from the list. In a nutshell, the goal of the first step is to eliminate frequency couples that cannot be used on every core sharing the same frequency setting.

Algorithm 8 Pairs Durations Computation

1: $allPairs \leftarrow \emptyset$
2: **for all** (f_1, f_2) $\in lowerF \times greatrF$ **do**
3: $t_2^{max} \leftarrow$ maximal duration among all cores for f_2
4: $t_1 \leftarrow totalPairTime - t_2^{max}$
5: $allPairs \leftarrow allPairs \cup (f_1, f_2, t_1, t_2^{max})$
6: **end for**

Second, as shown in Algorithm 8, FoREST computes the duration associated to each frequency in couples obtained at the first step. As shown in Figure 11.2, to emulate a specific IPS, both $lowerF$ and $higherF$ are run during a specific period of time. The execution times depend on the target IPS to achieve on each core. Hence, different cores associate different durations to the same frequencies in pairs. Finally, within the frequency couples space, only the couples with the longest execution at the highest frequency are kept. Indeed, FoREST is conservative in order to ensure the slowdown limitation. At the end of the second step, couples that can be used on each processor cores with the highest frequency executed the longest are passed down to the last step.

Algorithm 9 Pair With Maximum Energy Savings

1: $bestEGain \leftarrow \infty$
2: **for all** $(f_1, f_2, t_1, t_2) \in allPairs$ **do**
3: $procEGain \leftarrow 0$
4: **for all** $c \in$ cores sharing the frequency setting **do**
5: $f_1EGain \leftarrow t_1 \times SPD(f_1, c) \times PGain(f_1)$
6: $f_2EGain \leftarrow t_2 \times SPD(f_2, c) \times PGain(f_2)$
7: $coreEGain \leftarrow (f_1EGain + f_2EGain)/2$
8: $procEGain \leftarrow procEGain + coreEGain$
9: **end for**
10: $procEGain \leftarrow procEGain/nbCores$
11: **if** $procEGain < bestEGain$ **then**
12: $bestEGain \leftarrow procEGain$
13: $bestPair \leftarrow (f_1, f_2, t_1, t_2)$
14: **end if**
15: **end for**

Finally, FoREST has to choose one frequency pair among all the possible ones. To do so, as shown in Algorithm 9, it computes the energy gain achieved by every couple and only selects the one providing maximal energy savings. However, in recent multicore processors, a frequency is necessarily set simultaneously on all the cores. FoREST considers this limitation and computes the energy gains achieved by the couples on each individual processor core. Then, the overall processor energy gain for a given couple is computed as the average gain over all the cores sharing the same frequency setting. By doing so, FoREST assumes that all the cores equally participate to the total energy consumption. Then, once energy gains are known for all the considered frequency couples, FoREST picks the one achieving maximal

energy savings.

In conclusion, by composing discrete frequencies, continuous frequencies can be emulated. As shown by the different steps of the algorithm 6, the emulated frequency ensure energy savings within the user's performance constraint.

11.2.3 Multicore Processors

Multicore processors have been introduced in the past few years and quickly became a standard in computers. No DVFS controller can ignore anymore this fact and has to be compatible with multicore processors, especially regarding the frequency settings shared among several cores.

As a reminder, for REST in Chapter 9, two different processors were used in order to ensure that all used applications where using all processors cores. Letting some cores unused could strongly temper with REST computed frequency shift, since they are taken into account by the Cpufreq module as explained in Chapter 6.

FoREST considers shared frequency domains of multicore processors at every stage. First, FoREST is only replicated once per group of cores sharing the same frequency transition. Then, during the frequency evaluation, IPS is measured synchronously on all the cores of a group. Finally, the selected frequency couple is specifically constructed to ensure the desired slowdown on every core. Thus, FoREST is unique in its ability to work on the recent multicore processors.

In addition, by construction FoREST monitors activity of every core sharing the same frequency domains, it cannot dissociate cores running the application to be optimized and unused ones. Therefore, it ignores the cores whose activity is below 30% in order not to pollute IPS evaluation with non relevant data.

11.2.4 Frequency Transition Overhead

Finally, considering the frequency transition overhead is the last enhancement. In order to achieve any desired slowdown, FoREST uses frequency pairs instead of unique frequencies. Thus, at every time step, the number of frequency transitions is at least doubled, potentially harming performance. In order to evaluate the overheads induced by frequency transitions, their effects on performance were measured using micro-benchmarks while switching frequencies. As shown in Section 8.3, an overhead of approximately 10 μs on execution time was measured on the experimental setup. During that time, the processor pauses the execution, slightly increasing the workload runtime. Such impact on performance is negligible compared to the time step durations considered by FoREST. An additional issue related to frequency transition is the latency required to perform the transition. It will be seen in Section 11.3.2, that the IPS evaluation is performed on 100 μs time period to have the minimum impact on the application execution and to ensure the best reactivity to application phase shifts. Alas, the measured frequency transition latency lies between 10 μs and 80 μs. Hence, in order to increase the measurement precision, FoREST starts measuring the IPS only after having waited for the frequency to change. FoREST is then aware of the frequency transition overheads and takes them into account.

11.3 Overview

FoREST is a dynamic DVFS controller running as a deamon on the host operating system. The general strategy employed by FoREST is to periodically evaluate the impact of a frequency transition on energy consumption. FoREST is then not based on any cost-model but rather on run-time measurements to determine which frequency to apply at any time. FoREST's algorithm is made of two main phases, *Evaluation* and *Execution*, in charge of the frequency evaluation and application, as illustrated in Figure 11.3.

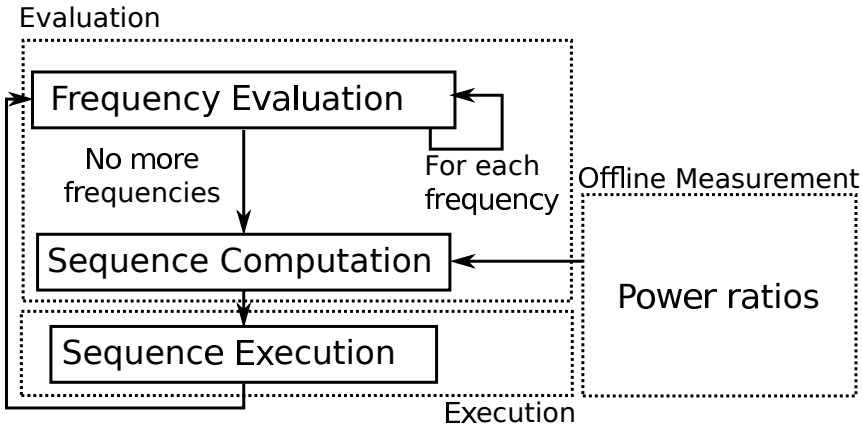


Figure 11.3: FoREST's architecture overview.

The main phase is the evaluation phase. Its goals consist in evaluating frequencies and providing a sequence of frequencies leading to the minimal energy consumption for the next execution step. In the execution phase, the frequency sequence is applied for a short period of time, before restarting the evaluation.

The evaluation segment consists in measuring the average number of executed Instructions Per Second (IPS) for every frequency. The maximal IPS measured in this phase is then used as a reference, defining the best performance that can be achieved with the current workload. FoREST then determines couples of frequency able to ensure the desired slowdown and selects the couple leading to the minimal energy consumption. FoREST design allows an efficient frequency selection driven by observations on energy consumption and allows users to define the maximal tolerated slowdown.

11.3.1 Offline Power Measurement

The offline analysis aims to provide FoREST with the impact of frequencies on power consumption. The impact is expressed as a power ratio between the power consumption of all processor frequencies over a reference frequency. As power ratios are program-independent, as explained in Section 11.2.1, one just needs to run one program in the offline analysis and compute the ratios. Moreover, the analysis is performed only once, before FoREST is launched.

In order to be compared, power ratios have to be computed over the same power: the reference power. The reference is chosen to provide power gain. In FoREST, the reference is the highest frequency power consumption since any other frequency will provide lower power consumption as shown is Equation 11.2 from Section 11.2.1.

To obtain the different frequencies power, CPU intensive benchmark are run

while measuring power consumption. The micro-benchmark used in this offline phase consists of a sequence of *add* operations very similar to the micro-benchmarks used in Section 6.5. Note that all runs are performed on the same number of cores in order to guarantee that P_{static} remains unchanged.

11.3.2 Frequency Evaluation

In order to determine which frequency to use, FoREST evaluates the energy gains achieved when choosing one frequency rather than the maximal one. To do so, FoREST combines power and execution time gains. Power gains are computed for every possible workload during the offline profiling. The impact of a frequency transition on execution time still remains to be determined. As opposed to power, the execution time gains heavily depend on the program itself, and more specifically on its CPU boundness [125]. FoREST must then evaluate the speedups induced by frequency transitions at runtime.

To measure the speedup achieved for the current workload depending on the frequency, FoREST applies the frequencies during short periods of time while measuring the number of Instructions executed Per Second (IPS). Although it is not perfectly representative of execution time, IPS can be considered as a precise-enough metric for evaluating speedups. Thus, FoREST measures the current workload IPS during periods of 100 μs, using the maximal frequency plus a few others close to the one previously chosen. The measurement is performed synchronously on all the cores sharing the same frequency setting. Then, every measured IPS is divided by the one achieved by the maximal frequency in order to deduce speedups of every frequency compared to the highest frequency. FoREST assumes the IPS to remain constant during the whole evaluation, which may not be correct, leading to potentially incorrect frequency selection. As for many dynamic systems, such mispredictions are tolerated, considering that a new evaluation will be performed soon after the incorrect one. Sample IPS measurements for one processor core are presented in Table 11.3, associated to the corresponding speedup over the highest frequency. Using the IPS evaluation, FoREST is then able to determine the speedup required to compute the energy gains of each frequency.

Using the information measured from both offline profiling and runtime IPS evaluation, FoREST directly deduces the energy gain that can be achieved by any frequency compared to the gain with the highest one. Indeed, as $e = P \times t$, the energy gain e_i and e_h achieved by any frequency f_i relatively to the highest frequency f_h is the product of the speedup achieved by f_h over f_i and the power gain measured for f_i relatively to f_h. The energy gains of the example are also presented in Table 11.3. Table 11.3 enlighten an important fact: lower frequency does not mean higher

Frequency	f_1	f_2	f_3	f_4
IPS ($\times 10^9$)	1.7	2.0	2.5	3
Speedup of f_4 (t_i/t_4)	1.8	1.5	1.2	1
Power gain vs. f_4 (P_i/P_4)	0.4	0.6	0.7	1
Energy gain vs. f_4 (e_i/e_4)	0.72	0.9	0.84	1

Table 11.3: Sample measurement results from offline profiling and online evaluation for one processor core

energy savings. Here $f2$ offers less energy gain (10%), than $f3$ (16%) because the ratio of speed-up and power gain are more in favor of $f3$ than $f2$. FoREST computes the energy gain achieved by every evaluated frequency on each individual processor core in order to decide later which frequency to use. Once energy gains are known for all the cores sharing the same frequency setting, the overall energy gain for a given frequency is computed as the average energy gain over all the cores. Since all the cores are powered with the same voltage and are composed of the same hardware units, hence when facing the same stress they will consume the same amount of energy. At the scale of a scientific parallel application, each processor cores goes on average through the same stress. Therefore, assuming that each core is equally responsible for the overall energy consumption, is a good approximation. Of course, if cores have individual voltage source, such approximation is no longer possible. By doing so, FoREST assumes all cores equally participate to the total energy consumption. Then, once energy gains are known for all the evaluated frequencies, FoREST can simply pick the one achieving maximal energy savings, provided it respects the user-defined performance constraint.

11.3.3 Sequence Execution

The sequence execution step consists in applying the frequency couple previously built. In couples, every frequency is associated to a duration. Each frequency is then applied sequentially with no specific order for the computed duration. FoREST is a dynamic system that periodically evaluates which frequencies should be applied. The main risk with such periodic behavior is to miss phase changes in programs. In order to minimize the risk, the total frequency pair execution time changes depending on the workload stability. During the frequency pair construction described in Algorithm 6 the main frequency *mainFreq* of a couple is defined as the one executed for the longest duration. If *mainFreq* is the same as during the previous sequence execution, the overall workload is assumed to be stable and FoREST doubles the total couple execution time. As soon as *mainFreq* changes, the total execution time is reset to an initial, arbitrary value of 1ms. Hence, when the workload behavior changes, FoREST re-evaluates it more frequently, trying to keep up with workload phases. If lower execution time was chosen to allow FoREST to faster adapt phase shift, most of the time would have been spent in frequency shift. At the opposite, if larger execution time were selected, FoREST would miss numerous program phases.

In conclusion, on one hand, such adaptive execution time reduces the number of evaluations during stable phases while, on the the other hand, ensuring reactive decisions when phase changes occur.

The main frequency is also used to limit the overheads of the evaluation process performed by FoREST. Indeed, once the frequency couple is effectively applied, FoREST goes back to the evaluation step as shown in Figure 11.3. The goal is then to restrict the number of evaluated frequencies. Evaluating all the available frequency is time consuming, even though if the evaluation period is set to 100 mus, it would take 1.6ms. If the application was composed of small programs, the frequency couple execution time would remain at 1ms. It means, in such condition, that the evaluation represents 60% of the application execution time.

By restricting the frequency range to those having a good chance of being executed afterward, the time spent in the evaluation process is drastically reduced . The *mainFreq* is then considered as the center of a frequency subset made of

all the frequencies near *mainFreq*. Only the frequencies in this subset are evaluated in FoREST evaluation step. In FoREST implementation, only one higher and one lower frequencies are considered. Reducing the range of evaluated frequencies substantially reduces FoREST overheads, because low frequencies are not evaluated when only high frequencies should be considered. However, it may take several steps to reach the optimal frequency as not all of them are evaluated. In fact, as it is combined with the adaptive execution time presented before, the limited number of evaluated frequencies is much more beneficial than harmful. FoREST sequentially applies the two frequencies previously chosen. It also adapts its behavior depending on which frequency is executed for the longest duration, enabling it to adapt to workload phase changes and reduce the evaluation overheads.

11.4 FoREST Versus the World

FoREST is implemented for x86_64 processors on Linux, in order to evaluate its main features on a real environment. The energy savings achieved by FoREST and the associated slowdowns are measured in order to check its ability to reduce energy consumption and guarantee the requested maximal slowdown.

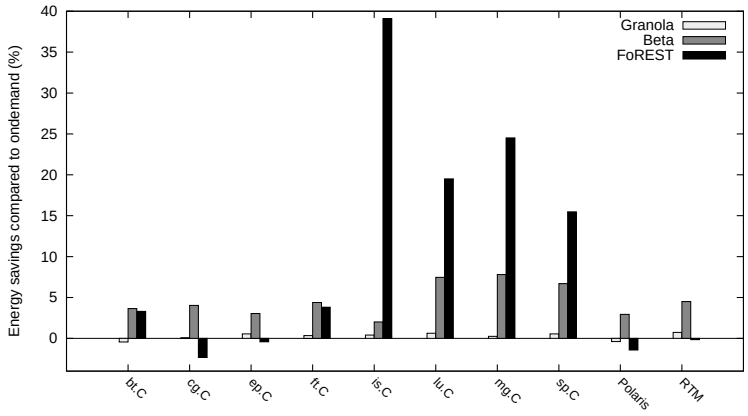
The experiments are run on an Intel Core i5 2380P quad-core processor, running Linux 3.5.3. The sixteen processor frequencies range between 1.6 GHz and 3.1 GHz, plus a turbo mode. The benchmark programs consist in the NAS OpenMP parallel programs 3.0 running the C class datasets [14]. Additionally, two industrial programs are considered: RTM [17, 11]. Only the forward kernel is extracted out of TOTAL implementation used to perform reverse time migration, and Polaris, a molecular dynamics program from CEA [140] . Measurements are performed using energy probes embedded in the processor and using a Yokogawa WT210 power meter plugged to the computer electrical socket in order to measure both processor and overall system energy consumption. Results are the median value of 5 executions, normalized relatively to ondemand. Existing DVFS controllers as Granola, a commercial DVFS controller [78] and beta-adpative [73] are used in order to provide a comparison with FoREST.

11.4.1 Energy Gains

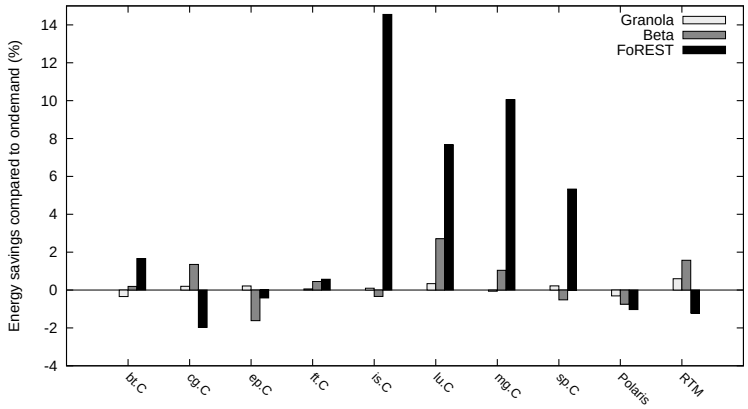
The benchmark programs are run on the experimental platform using different DVFS controllers. The first one is Granola, a commercial DVFS controller designed by MiserWare Inc. [78]. The second one is beta-adaptive [73]. Finally, the last one is FoREST. Granola uses its default configuration, while FoREST and beta-adaptive are allowed at most a 5% slowdown. The beta-adaptive system described by Hsu *et. al.* in[73], as for REST, is not designed to support multicore processors where all the cores share the same frequency. Beta-adaptive sees each processor core as an independent core, allowing it to compute a different frequency for each core, event though it is not the case for all recent Intel x86 processors.

The energy consumption induced both at processor and at system levels for every DVFS controller is presented in Figure 11.4. In the figure, positive values are energy savings compared to an execution when using Ondemand. Conversely, negative values represent additional energy consumption.

Some programs contain large memory intense phases. It is therefore possible to decrease the CPU frequency without impacting the execution time. Such programs



(a) CPU



(b) Whole system

Figure 11.4: Energy consumption normalized to that achieved by ondemand. 5% slowdown required for FoREST and beta-adaptive.

are perfect targets for DVFS controllers, achieving major energy savings. On the other hand, some programs are CPU intense and reducing CPU frequency often increases their energy consumption. Therefore, no significant energy savings can be expected from such programs. Granola is able to achieve light energy savings in many cases but did not significantly outperform ondemand. In fact, from the words of Granola's authors, Granola is not designed to outperform ondemand but rather to save as much energy as possible without harming performance. Beta adaptive is able to save more energy in general, at the cost of an increased execution time. However, FoREST clearly outperforms both DVFS controllers with memory-bound programs while maintaining a decent consumption with other programs. Indeed, 39%, 25%, 20%, and 16% of energy saving is achieved respectively for *is.C*, *mg.C*, *lu.C*, and *sp.C* at the processor level. It illustrates the ability of FoREST to detect even short memory phase in programs and to exploit them to save energy.

Even at the whole system scale, FoREST outperforms both DVFS driver. The only difference between Figure 11.4a and 11.4b is the relative amount of energy saved. It can be noticed that the energy savings exposed at the whole system scale are lower than the one exposed at processor scale event if the instant energy reduction remains the same. As a reminder from Section **??**, the processor energy consumption only account for a sub-part of the overall machine energy consumption.

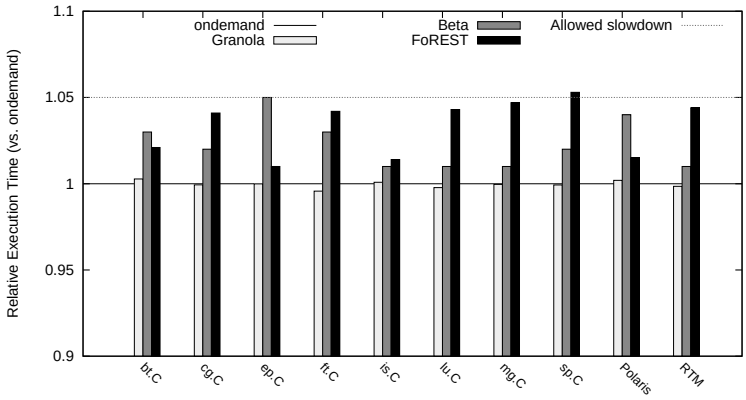


Figure 11.5: Execution time normalized to that achieved by ondemand. 5% slowdown required for FoREST and beta-adaptive.

For example, in the case where an entire machine consumes $50J$ and the processor $23J$, the processor is accounted for 46% of the overall energy consumption. So even if a DVFS driver is capable to optimize 100% of the processor energy consumption it will only represent 46% of the overall system. That is why huge energy saving exposed at processor scale are less significant at machine scale.

For CPU intense programs, FoREST sometimes achieves slight energy over consumption. One can notice a similar behavior for the beta-adaptive method. In fact, it is mostly due to the adaptive method chosen by FoREST and beta-adaptive. Both systems evaluate the impact of a frequency transition on performance and, in the case of FoREST, on energy consumption. It implies periodic evaluation of frequencies, including inefficient ones. Such evaluation on CPU intense program immediately leads to an increased energy consumption, the importance of which depends on how frequently and for how long the evaluations are performed. Additionally, dynamic systems may pick incorrect frequencies for short durations before correcting their mistakes at the next evaluation, increasing in some rare cases their overheads. Moreover, like energy saving is reduced when the scale is changed, the energy overheads are increased. The impact of incorrect choices leading to small energy overheads at processor scale are amplified at the overall machine scale as seen in Figure 11.4b.

As suggested in Figure 11.4, the energy savings are achieved when allowing 5% of degradation of the overall applications performances. As presented above, a specific mechanism is implemented to prevent FoREST from degrading more than the specified limit. Section 11.4.2 shows how FoREST actually performs regarding performance degradations.

11.4.2 Performance Degradation

FoREST directly measures the impact of frequency transitions on IPS to guarantee a maximal slowdown afterwards. In order to determine if it actually enforces the requested maximal slowdown, the execution time of all the benchmark programs are measured when using ondemand, Granola, beta-adaptive, and FoREST. The execution times are normalized regarding ondemand in Figure 11.5. FoREST is able to enforce the maximal requested slowdown as it never provokes more than 5% slowdown. Granola leads to execution times similar to what ondemand achieves.

Compared to ondemand, beta adaptive and FoREST increases programs execution time but the resulting slowdown is always in the range tolerated by the user. When considering both slowdowns and energy savings, the presented results indicate that FoREST takes relevant decisions as it can trade slowdown for energy all the while not exceeding the requested slowdown threshold.

FoREST has the ability to automatically determine what slowdown must be applied at anytime in order to save as much energy as possible. As opposed to many other mechanisms, it does not systematically choose the maximal tolerated slowdown if other slowdowns offer more energy reduction. It is ensured when selecting the best frequency couple since the energy reduction is the decisive criterion in the third phase from Algorithm 6. This ability is reflected in Figure 11.5 as the measured slowdown is often much lower than what the user tolerates.

11.4.3 Frequency Sequence

In order to illustrate the decisions taken by FoREST, we present in Figure 11.6 the frequencies chosen by FoREST during the execution of *ic.C* when 5% slowdown is tolerated. Although FoREST generates frequency pairs, only the frequency set during the maximal duration is represented in the figure. The program clearly exhibits several phases and FoREST quickly adapts the frequency accordingly. Figure 11.6 can be compared to Figure 10.2 on page 97 to see that FoREST correctly reacts to the application phase shift. FoREST chooses high frequencies during the initialization and termination of the program, while low frequencies are preferred for the main part of the execution. FoREST adapts it frequencies choices to the different application phases. It is also able to maintain stable settings when the program behavior is constant. FoREST is then able to adapt to program phases and, as proven by the energy savings achieved, the frequency transitions it applies are relevant.

Users are allowed to select any level of performance degradation. Specifying a higher performance degradation allows FoREST to be more aggressive on its frequency choices to leverage more energy savings. Section 11.4.4 shows how FoREST performs when 100% of application's performance degradation is allowed.

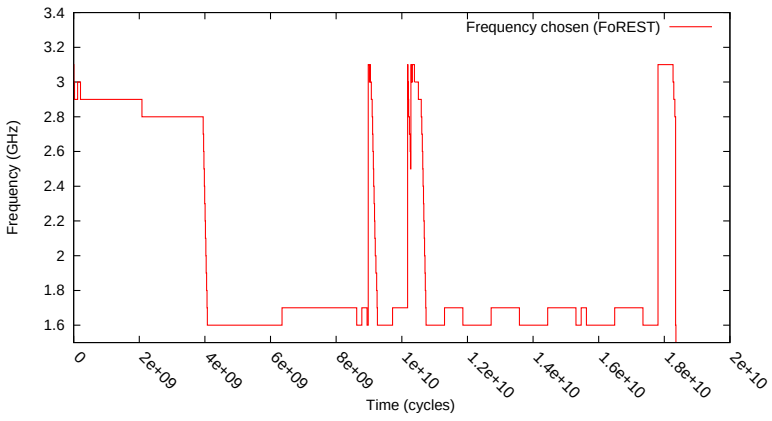


Figure 11.6: Forest frequency selection when running the `is` program with 5% slowdown.

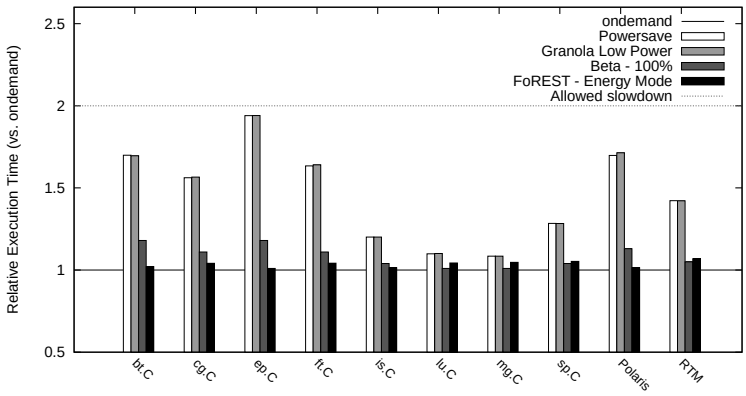


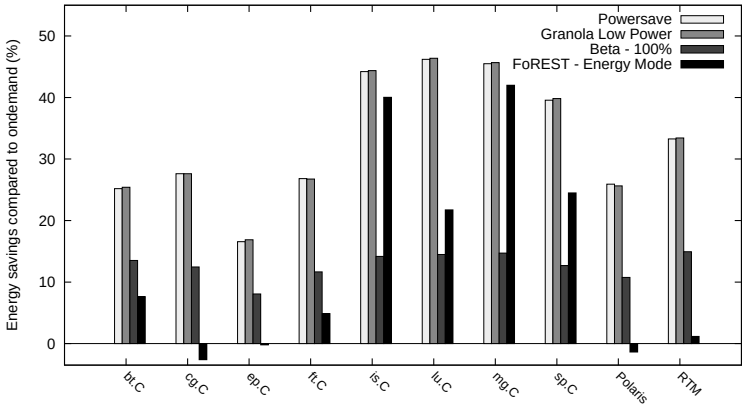
Figure 11.7: Execution time normalized to that achieved by `ondemand`. 100% slowdown allowed for Forest and beta-adaptive.

11.4.4 Energy saving mode

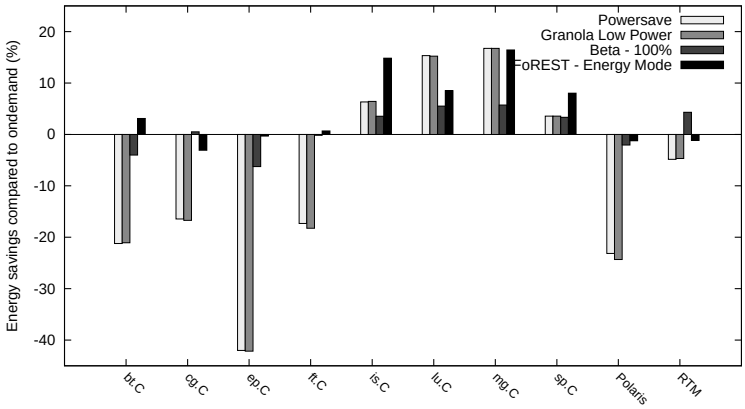
In some cases, energy consumption is a major concern for the user and execution time does not count. For instance, when working on battery-powered devices, autonomy becomes a critical criteria for the user. For that purpose, FoREST was configured to ensure a maximal slowdown of 100% and the previous experiments were run again. The resulting energy consumption and execution time are presented respectively in Figure 11.8 and Figure 11.7. During the experiments, FoREST was compared to the *ondemand* and *powersave* settings. *Powersave* is the Linux DVFS policy that systematically sets the lowest frequency. In the experiment, Granola uses its low power mode, beta-adaptive is also targeting a 100 % slowdown.

Figure 11.7 shows that, with extreme setting, FoREST is able to achieve processor energy savings while provoking slowdowns within the user requirements. One can note by comparing Figure 11.5 and 11.7 that the performance degradation of FoREST are similar. As explained above, FoREST does not always target the frequency allowing the highest slowdown if others expose more energy savings. Therefore on the tested benchmarks there is no real use to authorize such performance slowdown if FoREST induces a slowdown equivalent to those observed when the limit is set to 5%. It can be noticed on Figure 11.8a and Figure 11.8b, that FoREST is able to reduce the whole system and the CPU energy consumption. However, if Figures 11.8 and 11.4 are compared, one can note that, as for performance slowdown, no significant additional energy savings can be spotted apart for *mg.C*. One can note that the relative reductions of energy consumption achieved on the CPU are reduced when considering the whole system. As explained above the static power of the system plays a crucial role into the energy consumption. If the application last too long the energy savings at processor scale are not sufficient to counterbalance the static energy, inducing over consumption as it can be seen for *bt*, *cg*, *ep*, *ft*, *RTM*, and Polaris in Figure 11.8. The only way FoREST can limit the static power influence is by limiting the application execution time degradation.

In conclusion, DVFS systems have to be carefully designed in order to leverage energy reduction disregarding the granularity. The best example, FoREST, can leverage significant energy savings both at processor scale and for the entire machine. Therefore, targeting low frequencies to reduce energy is not always a good idea even



(a) CPU



(b) Whole system

Figure 11.8: Energy savings over what `ondemand` achieves. 100% slowdown allowed for Forest and beta-adaptive.

if it helps the processor to reduce its energy consumption. As for REST, and despite all the optimizations performed in FoREST, one can still question its efficiency. Table 11.4 shows the difference in percentage between UtoPeak and FoREST for the NAS benchmarks. FoREST with the 100% performance degradation constraint was compared to UtoPeak in order to be consistent. As a reminder, UtoPeak seeks for maximum energy savings without caring for application execution time. It can be noticed that FoREST is very close to the maximum energy savings and has little room of improvement. FoREST even exposes even more energy reduction than UtoPeak on is.C. The comparisons only involves processor energy consumptions. However FoREST might not be the most efficient DVFS controller when considering the full machine, even though it outperforms the other tested DVFS drivers.

Benchmark	bt.C	cg.C	ep.C	ft.C	is.C	lu.C	mg.C	sp.C
Difference	1.78	3.42	6.23	5.87	-0.50	0.47	2.34	1.26

Table 11.4: FoREST energy savings compared to UtoPeak.

Chapter 12

Conclusion

At the beginning of this part, a classification of applications was presented to get a better vision of the different applications types regarding energy consumption and execution time behaviors. Though a wide applications range exists, they are classified in only three different categories. The first one, named external resources bound or memory bound, exposes a specific behavior where the execution time remains constant across the processor frequency spectrum. The energy consumption is the lowest at the lowest frequency. Choosing the lowest frequency is then the easiest way to reduce the energy consumption. At the opposite, the CPU-bound applications trend, exposes a decreasing energy and execution time tendency when increasing the operating frequency. As for the memory bound trend, the energy optimization is straightforward, the application has to be executed with the highest frequency. Finally, the balanced application trend comprise all the application which are neither bound to memory nor CPU. Unlike, the two others trends, there is no single optimization to apply on each application belonging to the balanced trend.

A more precise study was then needed to understand the relationship between the application and its execution time and energy. The focus was put on the different phases that can be found inside an application. It was shown that studying all the application phases helps to understand the general energy behavior and to derive potential energy optimizations. Selecting the frequency granting the lowest energy for each application phase, helped to find the lowest energy for the overall application. However, the major difficulty is to isolate each application's phase and derive its boundess.

Two systems are presented, REST and FoREST to propose a solution to the application phase identification. REST as the first iteration, is based on the assumption that if a frequency is chosen regarding the stress the application put on the hardware, energy savings could be obtained. The results shows that it is not a bad assumption though it lacks efficiency as demonstrated by UtoPeak. As for any optimization procedure, if there is no way to quantify the optimization added value, the optimization is hardly justifiable. Consequently, UtoPeak was created to fulfill that purpose. By through-fully studding each application's phases boundness and choosing the best frequency to target their minimal energy consumption, the application energy savings upper bound is computed. By comparing the upper bound to the savings granted by either REST or FoREST, it gives an estimation of their efficiency. As it is presented, FoREST is the next iteration of REST, granting the tool the capacity to quantify the energy saving of every frequencies on every phases, and always choosing the one granting the lowest energy consumption. That feedback loop granted to FoREST the capacity of achieving almost the maximum energy savings exposed by UtoPeak.

Unfortunately the previous studies only considered a single processor. Yet, scientific application always tackles bigger problems and their needs for computational power always are increasing. Currently, TITAN, one of the most powerful machines

listed in the top500, uses more than sixteen thousand processors. The next big challenge is then to take all the knowledge presented in the current part and adapt it to that multi-processor environment. Sadly, it will be seen in the next Part that the presented optimization mechanisms cannot be applied out of the box, and a totally new approach has to be considered.

Part III

DVFS multi chip

Chapter 13

Introduction

In the previous part, multiple solutions to reduce program energy consumption on a single CPU were presented. Each of them exposed significant reductions in energy consumption. Thanks to UtoPeak, it was possible to evaluate their efficiency and demonstrate that FoREST almost provides optimum energy consumption savings. Hence, the problem of energy consumption reduction on a single processor is considered as solved. Nonetheless, scientific applications are not using only one processor, and their constant needs of performance drive the top500 [161] machine sizing. Therefore in the current Part, the search for optimal energy reduction will move to multiple processors environments.

One could think of an easy way to solve the problem. If UtoPeak or FoREST is used on each processor involved in an application execution, it will theoretically give the best energy consumption on each CPU. When considering the entire set of processors, if each CPU exposes the maximum possible energy consumption reduction, the best solution is found. However, it is not fully true since additional application constraints have to be taken into account as shown in Section 13.2 which presents the execution context. The problematic derived from this execution context is expressed in Section 13.3. Section 13.4 explains Utopeak incapacity to adapt to distributed environments when Chapter 14 presents different solutions to the problematic.

13.1 State of The Art

During the previous Part, all the optimization mechanisms took place on a single processor with the inherent constraint of one frequency for the total package at a time. Here, multiple processors are considered meaning multiple frequencies regions. One could associate that to chip-multiprocessor (CMP) which exposes multiple frequency domains [30, 31, 135]. Though, the technique described in this chapter, OUTREAch, targets applications using the Message Parsing Interface (MPI) and traditional cluster set-up, it can be transparently transposed to CMPs since it does not rely on any power model. The sole requirement is to have the possibility to abstract the application using a task graph.

When considering multiprocessors and task graphs, a close domain is the energy efficient task scheduling [102, 105, 111, 132, 141]. They present similarities with OUTREAch. They handle tasks graphs, each task having dependencies to other tasks that must be taken into account. Moreover, as for OUTREAch, the energy efficient schedule generally uses linear programming to minimize the energy consumption of the schedule. However, OUTREAch performs its work after the scheduling operation.

Rizvandi *et. al.* [141] show a classic energy optimization algorithm used to schedule a bag of tasks on a set of processors. It is fully static algorithm which finally relies on power models the authors considers as fit. On the opposite OUTREAch bases each of its decisions on real power and energy measures. Furthermore, Riz-

vandi *et. al.* consider variables difficult to be measured in the real world due to their magnitude. For example, the power cost of a frequency switch: it can last at most 100μs when the resolution of current processor energy probe generally is $1ms$. Finally, they only consider the processor as an entity when OUTREAch takes all processor cores into account to perform frequency decisions.

Lui *et. al.* [111] propose, in addition to a scheduling technique, to perform DVFS on the different processors used for the computed schedule. It first determines the scheduling and the deadline to meet, and then performs DFVS technique to reclaim potential slack or extend task execution to meet the deadline. However their technique does not take into account parallel tasks execution per processors since they only attribute one task to each processor. Furthermore they do not consider the time needed to perform a frequency switch. Ignoring hardware constraints is a major limitation, as presented below. OUTREAch takes them into account. Finally, even though they were to take into account for example the frequency shift delay, it will add additional binary variables. Having too many binary variables can be troublesome for the solver to converge to a solution, as presented below.

Pierson *et. al.* [132], also use linear programing to schedule tasks on a set of processors. Like Lui *et. al.*, they rely on power models, therefore the systems are bounded to a specific set of machines. This is not the case for OUTREAch but they consider relative power consumption between the different CPU to handle non homogeneous processors. However, in their study, the authors only considered a set of homogeneous processors.

The common point of all the described scheduling methods is that they do not state the actual convergence time of their linear problem. As they are all considering linear programing or even mixed integer programming, the convergence time can become huge when dealing with a significant number of tasks. It can be a downfall for scheduling techniques, if they themselves, take more time to schedule than the actual application execution time.

Li *et. al.* [105] present a solution closer to OUTREAch than the others because they consider MPI applications and not just pure bag of tasks. They propose to aggregate MPI processes as regards communications between them. Indeed, small messages benefit from the latency of shared memory while huge messages benefit from the high bandwidth of the network. However, they limit their level of tasks to MPI processes as tasks, when OUTREAch considers tasks at a finer grain. Furthermore, they compare their aggregation methodology to a case where one MPI process is occupying one node, therefore necessarily exposing energy reductions since less machines are used. They do not perform further energy optimization once the processes are aggregated, contrary to OUTREAch. One could use OUTREAch in addition to the aggregation system to perform additional energy consumption.

In addition to scheduling techniques, systems that try to reduce the energy consumption of an MPI application while being executed, do exist. The simplest form of DVFS controllers for parallel program are those reducing the frequency during the communication phases [107, 109]. OUTREAch aims to find the lower bound in energy consumption for an application execution. Therefore, it cannot restrict itself to only target the communications. Even more complex systems exist [63, 92, 145].

Kappiah *et. al.* [92] describe a system that reduces the frequency of nodes proportionally to the time they spent executing tasks out of the critical path. However, compared to OUTREAch, they do not consider the processor frequency limitation

that forces all the cores to be run under the same frequency. Rountree *et. al.* [145] propose a similar solution to Kappiah *et. al.*, however they consider tasks instead of CPU. Still, they also do not take into account the hardware limitations.

Halimi *et. al.* [63] present a method to optimize at run-time, the frequency for each task to lower the energy consumption of the overall application across the different processors. Similarly to OUTREAch, they take into account the hardware limitations and the possibility to limit the application execution time degradation, however their system does not search for a lower bound on the energy consumption as OUTREAch does.

It exists systems that search for the lower bound on energy consumption considering a distributed environment. To the best of our knowledge, only Rountree *et. al.* [143] expose a system using, as for OUTREAch, profiled information and linear programing with the intent of finding the lower bound in energy consumption for one application execution. However, unlike OUTREAch, they do not consider hardware constraints and execution time limitation. A comparison between the two system is proposed in Section 14.6.

13.2 Execution Context

Parallelism is one efficient way to reduce the time needed to solve a problem, that is why scientific applications use it to its full extent. The scientific world appetite for parallelism can be demonstrated by the always increasing number of cores in the most powerful machines of the top500, or by the apparition of many cores processors [119] or acceleration cards [137]. Other signs of this trend are the efforts of some people to create tools in order to better use these massively parallel machines or to simplify the development of parallel applications. Among them, it can be found some means to allow communication between different application processes like Message Passing Interface (MPI) [48], compilation framework for automatic code parallelism like openMP [127] or PLUTO for polyhedral code [22], high-level parallel abstraction like charm++ [90] and many others dedicated to many-core processors or acceleration cards. Even though a wide range of solutions exists to develop parallel applications, the most widely used still is the MPI. The search for the lower bound lower bound of energy consumption will therefore be performed on applications using the Message Passing Interface.

The execution context is then MPI applications running on a multi-node platform. A node can be defined as a set of processors with a shared memory and a network card for communication between the nodes. The application is seen as a set of processes all concurrently running on the collection of available cores on the different used nodes. Finally, an application process executes a set of tasks. Each task is data dependent. Indeed, some tasks need previously computed results to operate. Considered applications can be abstracted using Direct Acyclic Graph as shown in Figure 13.1.

Each considered application has a beginning phase where the different application processes are created, and an end where they are destroyed. Each application tasks is then organized regarding their precedence constraints. For example, task labeled 5 cannot start right after the initialization phase is finished, it has to wait for task 1 to finish. Considering an MPI application, a task, denoted T_i, is defined as the computation between two communications. The application execution is then represented as task graph, like the one in Figure 13.1, where the tasks are vertices

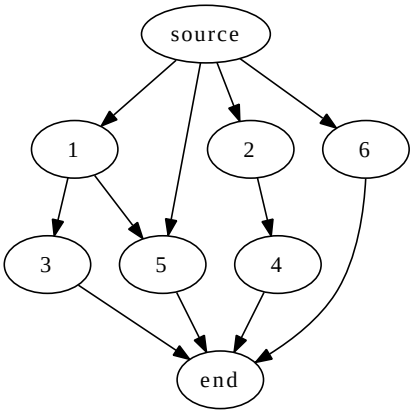


Figure 13.1: Parallel Application Tasks Abstraction.

when the edges are the messages between the tasks. Figure 13.2 is an example of the task graph running on two processes. Compared to Figure 13.1, Figure 13.2 is a two dimension representation of an application task dependencies. The x-axis represents the different processes involved in the application execution. Here, one process executes tasks T_1 and T_2 while the other one executes tasks T_3 and T_4. The y-axis represents the execution time. Accordingly, the longer the task on the y-axis, the longer its execution time. For example, in Figure 13.2, tasks T_3 last longer than T_1,T_2 and, T_4.

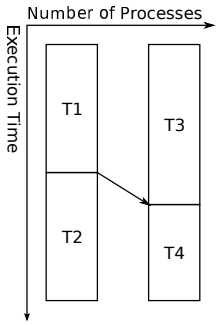


Figure 13.2: Task graph

The representation of the tasks duration and dependencies showed in Figure 13.2 will be the default task graph representation for the entire Part III.

Tasks within a core are totally ordered. If a task T_i ends with a send event, then the following task T_j starts exactly at the end of T_i. On Figure 13.2, task T_2 starts exactly after T_1 ends. Moreover, when a task is created by a message reception, T_4 on Figure 13.2, it cannot start before all the tasks it depends on, finish. Fortunately in Figure 13.2, task T_3 ended right on time to receive the message. However, if the message arrives after the end of the task which is supposed to receive it, the receiving task will have to wait for the message to arrive as shown in Figure 13.3b. The time between the end of the task and the actual reception is known as *slack* time.

The considered applications executions contexts are seen as a group of tasks scheduled on different processors that are organized regarding their precedence constraints. Each task is processing between communications and can include period of

time spent doing nothing other than waiting for a communication to arrive. Regarding the described execution context, reducing the global energy consumption means reducing the energy consumption of each task regarding a set of constraints related to the considered application architecture and hardware limitations as explained in the next Section.

13.3 Problematic

The problematic to be solved is the search of the highest bound in energy saving one can expect from using DVFS on the different processors used by a distributed application.

A task energy consumption E_i is defined as the product of its execution time $exec_i$ and its power consumption P_i. Since the application is composed of several tasks, its global energy consumption can be expressed as the sum of the energy consumptions of all the tasks. Hence one can calculate the application energy consumption as:

$$E = \sum_i (E_i) = \sum_i (exec_i \times P_i) \tag{13.1}$$

Minimizing the energy consumption of the application is equivalent to minimizing E in equation (13.1). It was seen in the previous parts that an application execution and power consumption is relative to the used frequency. It was also shown that application phases follow the same relation,consequently it is the same for application tasks. For each task T_i, as both $exec_i$ and P_i depend the frequency, the problem shifts to finding, for each tasks, the frequency that best minimize the overall application consumption. Consider the example provided in Figure 13.3. A theoretical application is executed on two different processors. Each processor has only one core. The example exposes two possible executions. Either the application is perfectly optimized and the communication times are perfectly overlapped with processing in Figure 13.3a or slack time exists in the application execution as shown in Figure 13.3b. Slack time is the time spent in a communication event waiting for the message to arrive. The execution is paused until the message arrives.

Suppose that in the ideal case, all the different tasks are performing memory operations. As stressed in Section 8.2, significant energy consumption reductions can be leverage without hurting the performances when lowering the operating frequency. A lower frequency is then chosen for each task reducing the overall application energy consumption without modifying the application execution as shown in Figure 13.3c. Now consider the case where some slack time exists as shown in Figure 13.3b. On the one hand, T_1,T_2, and T_4 are CPU intensive tasks meaning that choosing a lower frequency results in increasing their energy consumption. Hence their frequency setting are not changed. On the other hand, T_3 is less CPU intensive than the others and exposes energy saving if the frequency is lowered. The frequency setting of T_3 is then lowered, reducing its energy consumption. Its execution time also is impacted, forcing T_3 to last longer, removing the slack time period as shown in Figure 13.3d. The optimization of T_3 allows two things. Firstly, it allows T_3 to reduce its energy footprint. Secondly, as it removes the slack period, it completely subtracts its impact on the overall energy consumption. However, recall that, on processors, only discrete frequencies are available and fine control over the execution degradation is not possible. The new selected frequency for T_3 may degrade

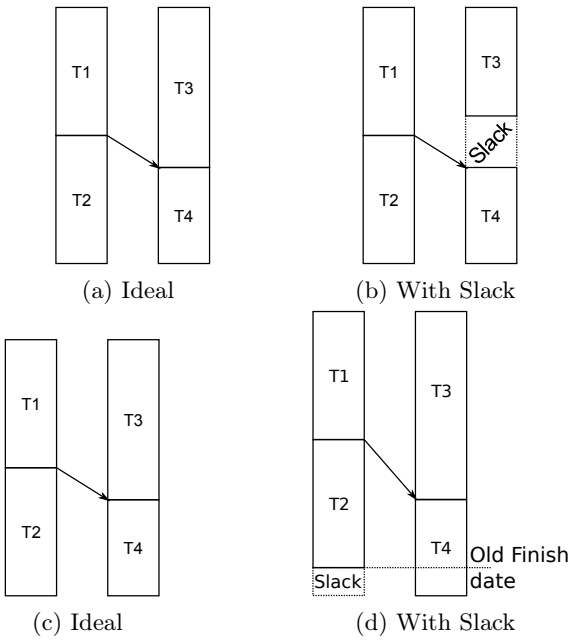


Figure 13.3: Different execution scenario and their potential optimization

too much the execution forcing T_3 to delay the start of T_4. T_4 will then start latter, forcing the application to last longer, to consume more energy. Selecting a frequency for each task to reduce the overall application energy consumption, is in appearance a simple optimization solution to the stated problem. However, constraints from the application, like the impact of a single task modification on all the others, or constraints from the hardware, as the discrete range of frequency, transform the problem into a more complex one as explained below.

13.3.1 Application Constraint

As described previously the different tasks composing an application are linked to eachother. So changing the execution time of one task can strongly impact all the tasks that depend on it. It can totally unbalance the application and generate a lot of slack time, inducing over consumption. As a matter of fact, energy savings can be countered by the consumption of newly appeared slack and/or by the fact the application last longer, increasing its energy consumption. As an example, Figure 13.4 shows the same execution scenario as the one exposed in Figure 13.3b. In this case, only the energy optimization of T_1 is considered. Suppose that the lowest frequency offers the highest energy reduction for T_1. It also forces task T_1 execution time to double, strongly delaying T_4 message sending. As T_4 has to wait for the message, the task start will be delayed forcing the apparition of additional Slack time. T_2 is also dependent on T_1 and cannot start before it finishes. This also delays the application end. Even though only T_1 execution time is modified, the entire application is impacted. The question that arises is: was it worth the costs, i.e, are the energy savings obtained on T_1 high enough to overcome the energy consumption of the new slack section? If the answer is yes, the optimization for T_1 can be considered as valid. However, recall that the goal is to minimize the overall application energy consumption. Then suppose that another frequency for T_1 could have limited the slack time increase. Even though that specific frequency

does not grant as much energy reduction as the lowest frequency, it could grant greater overall energy savings since the toll for slack is reduced. As hinted in the problematic section, global energy optimization cannot be performed independently from other tasks since any change can strongly impact all successive tasks. Each modification has to be taken into account to obtain the highest energy reduction.

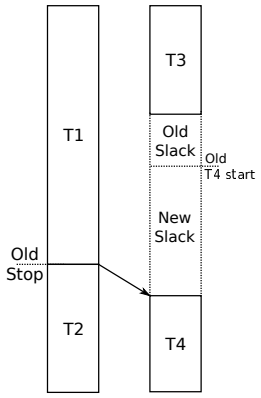


Figure 13.4: Task optimization impact on overall graph

Naively, in order to find the lowest energy consumption, each combination of frequency per task can be tested to cover the entire scope of possibilities and choose the combination that grant the highest overall energy saving. Suppose that for each task in Figure 13.4, fifteen frequencies are available; it would need $50,625$ combination in the worst case to find the optimal one. It can be easily understood, that it cannot be performed for applications with thousands of tasks. Special mechanism are then needed to prevent the optimization procedure to walk through the entire space of solutions to find the best one. However, task dependency is not the sole constraint to be taken into account. Architecture constraints can limit the use of frequencies for certain tasks obfuscating the search for the optimal frequency combination as it will be explained in the next section.

13.3.2 Hardware Constraints

In addition to the application constraints, numerous hardware constraints must also be taken into account when optimizing the energy consumption. As stressed in previous parts, multiple limitations exist on the current hardwares: non instantaneous frequency transition latency, discrete processor frequency, and, frequencies shared by the entire set of processor cores. Each limitation drastically impacts the search for the best frequency per task, which has to be performed to achieve optimal energy savings for a whole application across the used processors.

As stressed in Part II Section 8.3, it takes time to shift frequencies. In the previous part, for UtoPeak and FoREST, it was not a limiting factor since both had the control over the granularity of the considered phase. A coarse enough phase size was always chosen to mitigate the impact of frequency shift delay. However it is no longer the case, tasks are now the base elements. There is no control on their durations since they are defined by the application during its execution. It may then exist tasks shorter than the time needed to shift frequencies. That possibility must be taken into account when designing the optimization mechanism, otherwise, the task optimal frequency will not be applied when the considered task is executed. It will induce a difference between what is found to be optimal and what is actually

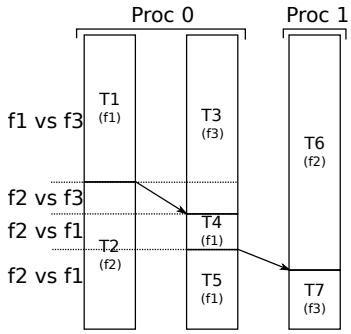


Figure 13.5: Frequency shift constraint

measured to be optimal as it will be shown later in Chapter 14.

However, before having to consider frequency transition delay, the frequency for each task on each involved processor has to be computed. Consider the example provided in Figure 13.5. The application is executed on 3 cores, 2 in the same processor and one in another processor. Tasks T_1, T_2, T_3,T_4 and T_5 are executed on processor 0 while tasks $T65$ and T_7 are executed on processor 1. Consider that the best frequencies to use in order to minimize the application energy consumption are $f1$,$f2$,$f3$,$f1$,$f1$,$f2$,$f3$ respectively for T_1,T_2,T_3,T_4,T_5,T_6,T_7 with $f3 > f2 > f1$. It can be seen that T_1 and T_3 are running concurrently on the same CPU while requesting different frequencies. As described in Chapter 6, only the highest frequency among the requested is applied, forcing T_1 to run at a frequency non optimal regarding energy savings. It is the same scenario when considering T_2 in parallel of T_3, T_4 and T_5. As T_6 and T_7 are on a different processor, their frequencies settings do not directly impact those of processor 0.

In a nutshell, selecting a frequency for each task in order to target optimal energy savings, is not the best strategy. As shown in Figure 13.5, within the set of concurrent tasks run on the same processor, a task may ask for the highest frequency while the others are benefiting from a lower one, forcing these latter to be run at the highest frequency. If it is repeated for each set of concurrent tasks, no optimization will be performed. This also induces a difference between what is found and what is actually measured to be optimal.

Finally, as suggested previously with Figure 13.3d and 13.4 discrete frequencies can imbalance the application or generate non wanted slack time. For example, recovering slack time with task execution is a great opportunity for energy savings. The goal is to select, for the task prior to a slack time section, a lower frequency that reduces the task energy consumption and extends as well its execution time. This should remove the slack time from the application execution. As a result, the task energy consumption will be reduced, and the energy consumption induced by the slack section will be saved. However, as frequencies are generally discreet the execution time either cannot be extended enough to recover the entire slack time section, or is extended too much, generating slack time elsewhere in the application as illustrated in Figure13.3d. For that reason, couple of frequencies, as the one used in FoREST, are used to have a fine control over the execution time degradation to perfectly recover slack time sections and/or limit their generation during the optimization procedure. .

Multiple constraints make the search of the best energy saving on a distributed application very complex. Tasks are dependent on eachother. Some are run concur-

rently on the same processor forcing them to share the same frequency setting. On top of that, any change in tasks duration can impact the overall task graph forcing slack time to appear and producing additional energy instead of saving it.

13.4 This is not UtoPeak you are looking for

As explained at the beginning of the chapter, UtoPeak can be naively used to optimize the energy consumption of distributed applications. One instance of UtoPeak can be spawned on each processor involved into the application execution. Each UtoPeak instance will profile the execution on each processor and, based on that, determine the best frequency sequence to achieve the lowest energy consumption per processor. However, the UtoPeak does not take into account all the previous quoted constraints. All the decisions taken by UotPeak on one processor are taken without being aware of the decisions made by other UtoPeak instances on the other processors. Communications will then be delayed, generating slack time, preventing UtoPeak from inducing the lowest energy consumption as shown in Table 13.1.

Benchmarks	UtoPeak vs Best Static Frequency
IS.C.16	-20%
EP.C.16	-10%
FT.C.16	-28%
BT.C.16	-24%
CG.C.16	-27%
MG.C.16	-35%
SP.C.16	-40%
LU.C.16	-38%

Table 13.1: Energy consumption comparison between multiple UtoPeak and the best static frequency

Table 13.1 was obtained by running the NAS-MPI benchmark suite on a dual processor machine while monitoring the energy consumption at processor granularity by using the embedded energy probe. One instance of UtoPeak was run per processor, trying to optimize the energy consumption. The results given by each UtoPeak instance were summed up and then compared to the lowest energy consumption given by a static frequency. A static frequency, is a frequency used for the entire application execution. The best static frequency corresponds to the static frequency granting the lowest energy consumption. As UtoPeak seeks for the maximum energy reduction, it should at least grant the same energy consumption at the best static frequency. However, in this case, UtoPeak is far from the optimum. Then using UtoPeak out of the box is not the correct answer to compute the best energy consumption when facing distributed applications. One naive solution to solve that problem is to allow UtoPeak instances to synchronize their decisions, therefore each instance will be aware of the potential impacts induced by the other UtoPeak and react accordingly. However, that solution is similar to computing all possible frequency combinations. Indeed, for each instruction sample, each UtoPeak program has to send its choice to the other instances, and potentially updates its choice regarding what was decided on the other processor. Then it has to iterate

until a solution is found for each UtoPeak instance on the considered instruction sample. At worse for each application sample, all the frequency combinations would have to be tested which is not affordable. A much more complex solution has then to be defined to tackle the problem of finding the lowest energy consumption for a distributed application. The next Chapter presents the different heuristics built to find an almost optimal solution in a decent period of time regarding the constraints presented above. The energy gain obtained while using this solution are then presented and compared to a concurrent method.

Chapter 14

OUTREAch : One Utopeak To Rule Them All

As explained above, OUTREAch is an attempt to transpose UtoPeak to multi-node machines. It was clearly demonstrated that UtoPeak cannot be used out of the box to obtain the minimum energy consumption for a distributed application. OUTREAch is based on UtoPeak and takes into account all the constraints described previously to find the lowest energy consumption for one application execution. OUTREAch follows the same three steps architecture as UtoPeak as shown in Figure 14.1. In step one, OUTREAch gathers all the needed information related to related to the execution time, relations and energy consumption of the different application tasks. The measurement mechanisms are inspired by UtoPeak. In step two, like UtoPeak, OUTREAch builds an internal representation of the application using the gathered data. By using that internal representation, OUTREAch builds a linear programming problem and calls a state of the art solver to work out the linear program. The solver, Gurobi [60], will then compute the theoretical energy consumption lower bound and the related frequency sequence per processor. Lastly, the solution is played to validate the theoretical lower bound and measure the precision of OUTREAch prediction. In a nutshell, OUTREAch performs the same steps as UtoPeak, but adapts each of them to the new considered problem and constraints. How the application profiling, how the linear programming problem are built and how the solution is evaluated are explained in the next subsections.

14.1 Application Profiling

Figure 14.2 shows that numerous information on the application execution and structure are needed by OUTREAch. It was explained above that the targeted applications were MPI applications. It was also explained that MPI applications are alternating between computations and communication phases. The profiling step

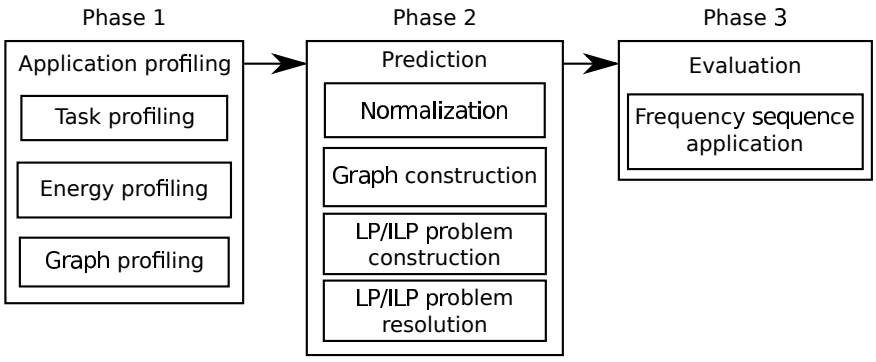


Figure 14.1: OUTREAch's steps overview

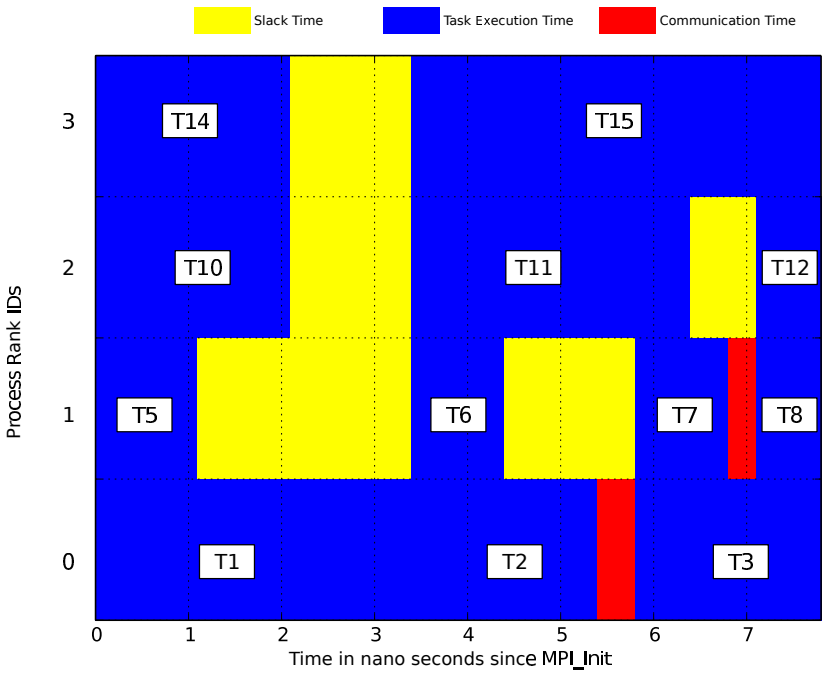


Figure 14.2: Task and communication timing

then needs to gather information on the two phases. Moreover, OUTREAch needs to acknowledge the different task dependencies to be able to build on internal representation of the application execution. Without this internal representation OUTREAch would not be able to acknowledge the potential task modification impact on the entire application as explained in Section 13.3.1. It was also the main drawback of the solution using multiple UtoPeak as explained in the previous chapter. Finally, OUTREAch also measures the energy consumption during the application execution. Without it, finding the lowest energy consumption for the considered application would be hard. As for UtoPeak all the measurements are repeated on the different processor frequencies expect for the task dependencies reconstruction. Indeed, like UtoPeak the targeted applications have to be deterministic, the task dependencies are then identical for each application execution, then only one profiling pass is needed. Application that cannot ensure that are dropped from the scope of the study. Figures 14.2 and 14.3 show the different information measured during OUTREAch first step. For clarity purpose, all the explanation performed during this section are based on a small theoretical application and not a real life example. Indeed, the smallest real life application at hand is IS.A.2; it is composed of 77 tasks and is therefore not practical for explanations purpose.

Figure 14.2 shows the information obtained after the Task profiling. OUTREAch measures the start date and duration of each task and each communication. Both information are required by the linear program as will be explained in Section 14.4. For communications, the measured duration is broken down into two different periods. The full communication time, regarding Figure 14.2 is obtained by summing up the yellow and red area. The red area stands for the actual time spent inside the communication, the yellow area, is the slack time spent in waiting for other tasks to handle their end of the communication. As an example, at the first communication involving the four application processes, T_5, T_{10} and T_{14} are waiting for T_1 to fin-

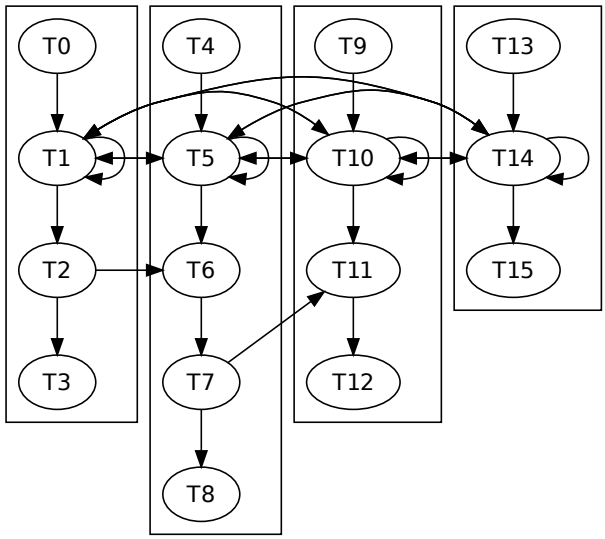


Figure 14.3: Task dependencies

ish its computation and reach the communication. Other communications are only composed of slack time or communication time. As it will be explained in Section 14.2, for some communications like simple send operations, OUTREAch considers that no slack time is possible. For other communication like receive operations, OUTREAch considers that the communication is only composed of slack time. As an example in Figure 14.2, task T_7 waits for the message from T_2 then only slack time is measured between T_6 and T_7 when only communication time is measured between T_2 and T_3 . It is the same for T_{12} waiting for a message from T_7.

As said previously, OUTRAch needs to acknowledge the different relations between each task. When considering only Figure 14.2 it is hard to see that T_1,T_5,T_{10} and T_{14} are all linked together or that T_2 is sending a message to the end of T_6 as it is shown in Figure 14.3. Having that information let OUTREAch understand how each task is linked to one another. In a nutshell, OUTREAch gathers F time the information displayed in Figure 14.2, F being the number of frequencies available on each processor involved in the application execution. It only gathers one time the information displayed in Figure 14.3 since the task dependencies should not change from one application execution to another. Now that the information measured by OUTREAch are presented, the next section focuses on how they are gathered.

Contrary to UtoPeak which uses only a sampling based profiling technique, OUTREAch uses MPI library instrumentation. Two different instrumentations were performed, one for the tasks and communication timings and one to acknowledge tasks dependencies which are detailed in Section 14.2.

14.2 Tasks And Communication Profiling

Multiple MPI libraries implementations exist [2, 3, 49]. MPI is a standard, therefore each implementation has to be compliant to the standard. The instrumentation was performed using the standard profiling interface primitives. Then if the MPI library implementation changes, the profiling is still operational.

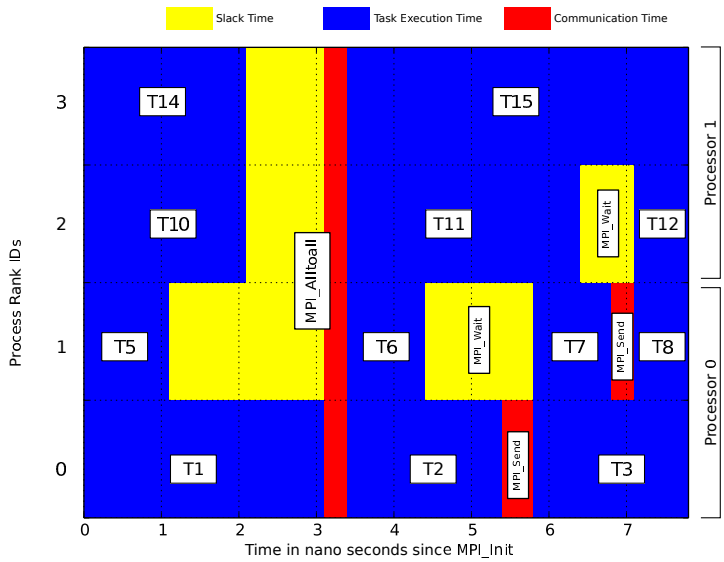


Figure 14.4: OUTREAch Task profiling

To retrieve the timing information, as the ones displayed in Figure 14.2, a probe is inserted at the entry and exit point of each MPI communication operation. At the entry point, the previous task duration and the communication start date are measured. At the exit, the new task start date, communication and slack duration are timed. As an example, consider Figure 14.4, T_2 start date is retrieved when the *MPI_ Alltoall* communication is finished. The duration of T_2 is measured when starting the *MPI_ send* following T_2. Measuring communication information is less straight forward. As hinted above, OUTREAch considers that there are three trends of communications. The first type is communication where no slack can ever exist. Only the point to point send operation *MPI_ send* compose this category. The second type, is communication where slack may exist. OUTREAch considers all the collective operations as *MPI_ Alltoall* to be part of the second category. Lastly, OUTREAch considers all the synchronization operations, as *MPI_ Barrier* or *MPI_ Wait* as purely composed of slack time. Indeed, their role is purely to wait for other tasks to reach the same point of the execution. The timing method of the first and third category is straightforward, the start date and duration are measured when getting in and out of the communication. According to the type of communication it is decided whether the duration is considered as slack or communication time. For the second category, a more complex method is needed. Indeed, the basic behavior of each function is slightly altered as shown in Algorithm 10. The algorithm only shows the method to dissociate the slack from the communication time for MPI_Alltoall. It is the same method for all the collective operations. One could state that altering the default behavior of some communications can impact the overall application execution times. However, it will be seen later that the overhead of the measurement method is rather due to disk writes than to communication function modifications. However, the overhead is kept low as will be presented later in this section..

The presented instrumentation method allows OUTREAch to measure the timing information needed for OUTREAch second step. The next instrumentation method to retrieve the task dependencies during the application execution is less straightforward. Indeed, during the instrumentation of the application, only the link between

Algorithm 10 MPI_Alltoall communication instrumentation

1: **MPI_Alltoall(...) instrumentation**
2: $com\_Start \leftarrow date$
3: $PMPI_Barrier(...)$
4: $slack \leftarrow curDate - com_Start$
5: $PMPI_Alltoall(...)$
6: $com \leftarrow curDate - com_Start$

MPI processes are available. This is why, as a first step and for all communications, Outreach gathers the different link between the involved processes. It is performed only once during an application execution since OUTREAch considers that the target applications are deterministic. The communication order and participants will not change from one execution to another. As an example, in Figure 14.5, $COM1$ will always be executed before $COM2$ and $COM3$ and it will always involve $P0$, $P1$ and $P2$. One could still perform multiple runs, however no additional information can be gathered after the first run. After that unique process dependencies profiling run, OUTREAch only has information on processes relations during communication. For each process, it knows to which processes information are sent and from which they are received. Consider *Step 1* from Figure 14.5. At the end of the instrumentation run, OUTREAch knows that during $COM1$, each MPI process $P0$, $P1$ and $P2$ are exchanging messages with all the other processes. It knows that $P0$ sends a message to $P1$ during $COM2$, and during $COM3$, $P1$ sends a message to $P2$. At this step, OUTREAch has no information on tasks. Recall that regarding the execution model defined in Section 13.2 a communication ends a task and a new one

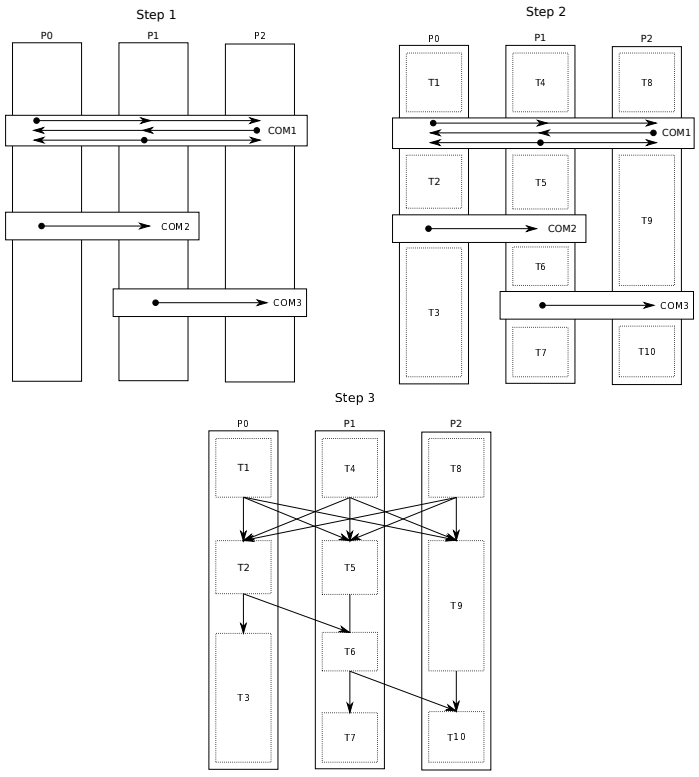


Figure 14.5: Task Graph Reconstruction

is started when it finishes. OUTREAch then assumes that for each process there is a task before and after a communication. Since it also knows the link between the processes during each communication OUTREAch can assume that each task before a communication will have a link with the tasks after the communication. Consider *Step 2* from Figure 14.5, OUTREAch knows that $T1$ will have a link with $T5$ and $T9$. By using the link between the processes PO,$P1$ and $P2$ the relation between $T1$, $T5$, and $T9$ can be inferred as shown in *Step 3* from Figure 14.5. One can note that the relation between $T1$ and $T2$ is not mentioned in the previous example, because such precedence relation were already computed at *Step 2*. There is no need for OUTREAch to look into the communication links information to know that $T2$ is executed after $T1$.

After both task and communication profilings, OUTREAch has all the information needed to start and optimize the application energy consumption. It will be able to select for a given task the frequency that reduces its energy consumption and to propagate any task modification along the task dependency chain. However, it will be seen in Section 14.4 that energy minimization is not as simple. Before looking into energy optimization, one can wonder why the task information profiling is only performed F times and not $F \times N$, F being the number of frequencies and N the number of tasks compsing the application. First, OUTREAch assumes the set of processors to be homogeneous. For the results exposed in Section 14.5, for each application execution either 2, 4 or 8 Xeon E5-2670 processors are used. Each processor has then the same range of frequencies. Furthermore, what actually matters are the task execution durations, thus the energy consumption. Testing all the combinations of task and frequency during the profiling step will not change the fact that a task is executed in t time at frequency f. Finally, even when only performing F application executions if one still want to have all the combinations of task and frequencies, it will have all the necessary information by composing all the measurement per tasks.

Benchmarks	Profiling impact	Benchmarks	Profiling impact
IS.C.16	0.18%	SP.C.16	0.18%
IS.C.32	0.55%	SP.C.64	5.45%
IS.C.64	4.2%	CG.C.16	0.77%
EP.C.16	0.23%	CG.C.32	0.70%
EP.C.32	0.43%	CG.C.64	6.26%
EP.C.64	4.18%	LU.C.16	1.59%
FT.C.16	0.49%	LU.C.32	3.62%
FT.C.32	3.42%	LU.C.64	9.77%
FT.C.64	5.10%	MG.C.16	0.64%
BT.C.16	0.16%	MG.C.32	1.47%
BT.C.64	5.24%	MG.C.64	4.72%

Table 14.1: OUTREAch instrumentation impact

Having to perform F runs or $F \times N$ runs will not change the potential overhead of the technique. The task profiling need to gather information on the execution and stores it on disks. Each process outputs information to disks, so the application execution can be strongly impacted. Table 14.1 shows the impact of OUTREAch on the full set of applications used to evaluate OUTRAch. It can be seen that

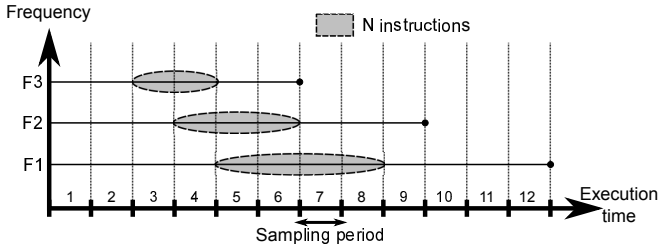
OUTREAch does not degrade the application execution time by more than 5% in most cases. OUTREAch overhead was kept low thank to the General Parallel File System (GPFS) [150] installed on the test machine. GPFS is optimized for large write sequence into different files. However raising the number of processes increases the pressure on the file system, as well as the impact of OUTREAch instrumentation as it is suggested by the degradation on 64 processes for each NAS benchmark programs. If more than 64 processes were to be considered for OUTREAch then a dedicated solution would be needed to keep that overhead acceptable. One of them would be to use burst-buffers [110, 163]. All the data would be pushed into RAM and a dedicated system would write the data back to disks. From OUTREAch point of view the cost to output each task and communication information would be drastically reduced limiting its influence on the application execution. However, the topic of OUTREAch is not to design a burst buffer but to find a solution to compute the minimum energy consumption regarding the application energetic behavior. In the end, OUTREAch overhead is kept low, except for LU.C.64. One cause would be the intensive usage of the GPFS since it is shared between all the machine users and LU is the benchmark containing the most tasks and communications around 31 million against 3 million for BT, SP. However, for EP and IS the writes intensity is not the reason for the degradation of the application execution time. They are the benchmarks with the lowest amount of tasks, respectively 4928 and 960, and any write to the backing file system is very costly as it can be seen in Table 14.1.

The energy measurement methodology is exactly the same as the one used in UtoPeak. It is demonstrated in the chapter dedicated to UtoPeak that the overhead of this method was almost null. Therefore the energy profiling does not generate any additional overhead. In the end, the measured data give a fair report of the application execution and energy consumption. Since the energy profiling is taken from UtoPeak, a normalization step is needed to be able to compute the effective energy costs of each task and each communication. It is the goal of the next Section.

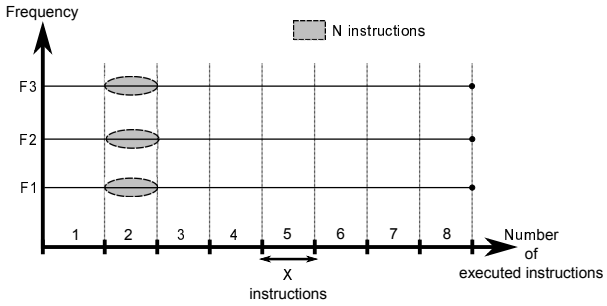
14.3 Energy Normalization

As a reminder, UtoPeak could not use the profiled energy consumption directly for the normalization process as shown in Figure 14.6.

In UtoPeak case, the use of the fixed time sampling provides no possibilities to compute the energy consumption per phase without using the instruction sample. However, there is no need for such a transformation with OUTREAch. Indeed, it knows when all the phases, in the current case task or communication, start and finish. Then if the task of communication last for multiple profiling sample OUTREAch knows where to look. The normalization process is very similar to the one UtoPeak used after the instruction sample transformation. If the task or communication phase is comprised inside a profiled sample, a time ratio is computed as $ratio = T_{duration}/SMPL_{duration}$. With $T_{duration}$ and $SMPL_{duration}$ respectively being the task and the sample durations. The ratio quantifies the task impact on the energy consumption inside the sample. Then to get the task/communication energy consumption, the measured sample energy is multiplied by the ratio. Some task or communication can be between two profile samples. As OUTREAch knows when the task/communication and both profiling sample start and stop, it cuts the task/communication duration into two portions. The portions are then compared to their corresponding profile sample. For each portion the impact ratio is compared



(a) Application time-based sampling.



(b) Application instruction-based sampling.

Figure 14.6: Difference between time-based and instruction-based sampling.

as described above. Both profile samples are multiplied by their respective ratio. By summing the two obtained energies the the overall task/communication energy is obtained.

The energy normalization process is repeated for each task and each communication under the different application processes. Once the normalization is done, OUTREAch constructs an internal representation based on all the measured information. The closest graphical representation of that internal structure can be found in Figure 14.3. For each task it stores the start date, duration and energy consumption under the different frequencies. For the communication, only the impact on the different involved processes are used. As an example, if process $P0$ sends a message to process $P1$, OUTREAch will know when the send operation started and finished on $P0$ and when the receive operation started and finished. By using such a representation, OUTREAch tries to find the maximum energy consumption for an application execution. However, there is no easy solutions since a vast amount of constraints has to be taken into account as explained in the previous chapter. It was then decided to use linear programing in order to leverage the resolution complexity by using tools that are meant for such a complex problem. By using a state of the art solver [60] OUTREAch was able to give a practical answer to the problem. Though it is an approximation coming from multiple problem refinements presented in the next section, it is available fast and with good precision as it is shown in Section 14.5.

14.4 Building The Linear Program

The following paragraphs describe how the energy minimization problem, described in Section 13.3, can be translated into a linear program. First the precedence constraints are expressed, to allow the solver to correctly manage the application dependency graph as described in Section 13.3.1. The major OUTREAch contestant

[143] defines similar constraints, however OUTREAch contributions lies withing the problems refinements presented thereafter. After the precedence constraints, architectural constraints are described, to help the solver producing a realistic solution. Finally, based on these constraints three different formulations are presented. Sections 14.4.3, 14.4.4, and 14.4.6 discuss the feasibility of each solution.

14.4.1 Precedence Constraints

As explained in Section 13.3.1 all the task withing an MPI application are linked to one another. The alteration of one task execution time can greatly imbalance the overall application task graph. Unbalancing the application task graph can lead to degraded application execution time and/or apparition of slack time, generating more energy consumption. A special formulation is needed to allow the solver to grasp these relations between tasks and prevent it from making wrong decisions ending up in degrading the energy consumption instead of reducing it.

Before jumping into the linear programing constraint formulations, consider Table 14.2 which exposes the different variables that will be used in the remainder of the section. Each line explains the different task attributes that must be taken into account to correctly teach the solver how to understand task precedence constraints.

bT_i	Beginning of a task T_i
eT_i	End of a task T_i
bTs_i	Beginning of a slack task Ts_i
eTs_i	End of a slack task Ts_i
$exec_i^f$	The execution time of a task T_i if executed completely at frequency f
tT_i^f	The time during which the task T_i is executed at frequency f
δ_i^f	The fraction of time a task T_i spends at frequency f
M_j^i	Message transmission time from task T_j to task T_i

Table 14.2: Task variables

Let T_i be a task defined by its start time bT_i and its end time eT_i. The beginning of tasks is bounded by the precedence relation between them. As already stressed out, a task cannot start before its direct predecessors complete their execution. As explained in section 13.2, if T_i sends a message, its child task T_j starts exactly when T_i ends since the end of the communication means the beginning of the next task. It can be expressed as:

$$bT_j = eT_i \tag{14.1}$$

Additionally, when the same task T_i ends with a message reception from another task T_k, one has to make sure that its successor task T_j starts after both tasks end. As an example, consider Figure 14.7 where T_4 must wait for T_1 and T_3 to finish before starting. Moreover, as pointed out in section 13.2 and shown in Figure 14.4, when a task receives a message, some slack may be introduced before the reception. Slack is handled the same way tasks are. It has a start and an end time. To ease the presentation, it is assumed that each task T_i receiving a message, from a task T_k, is followed by a slack task, denoted Ts_i. The beginning of Ts_i, denoted bTs_i is equal to the end of T_i:

$$bTs_i = eT_i \tag{14.2}$$

If Figure 14.7 is considered, that the slack task Ts_3 starts when T_3 is waiting for the message to arrive, and the next task T_4 starts just after Ts_3 ends as enacted by Equation 14.1.

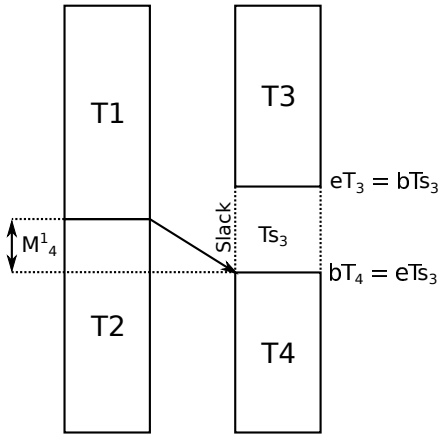


Figure 14.7: Slack time

The slack task end time, denoted eTs_i, is at least equal to the arrival time of the message from T_k. Let M_k^i denote the transmission time from T_k to T_i. Thus:

$$eTs_i \geq eT_k + M_k^i \tag{14.3}$$

If equation 14.3 is considered with Figure 14.7, it means that Ts_3 will end after the message is actually sent by $T1$ plus the transport duration. Note that a task may receive messages from different processes, after a collective communication for example, and equation 14.3 has to be valid for all of them.

Finally, since T_j, the successor task of T_i has to start after T_i and T_k finish, one just needs to make sure that:

$$bT_j = eTs_i$$

For Figure 14.7, it means that the beginning of T_4 happens at the end of Ts_3. In order to compute the end time of a task T_i, eT_i, one has to evaluate the execution time of T_i. As explained in Section 13.3, the search for the optimum energy will be achieved by selecting the best frequency per task. This is why, during OUTREAch profiling step, task information are gathered for different frequencies. The solver must take that into account. Let $exec_i^f$ be the execution time of T_i if executed completely at frequency f. However it was demonstrated in Section 13.3.2 that using a single frequency is very limiting for slack recovery. Every frequency can then be used to run a fraction δ_i^f of the total execution of the task. Let tT_i^f be the fraction of time T_i spends at frequency f. It can be expressed as:

$$tT_i^f = \delta_i^f \times exec_i^f \tag{14.4}$$

For Figure 14.7, Equation 14.4 implies that the task T_3 can be sliced into multiple sections, and for each of them a different frequency can be used. It allows T_3 for example to perfectly recover Ts_3, saving its energy consumption without modifying

the application execution graph. By considering that multiple frequencies can be used to execute a single tasks, it modifies the task end time formulation:

$$eT_i = bT_i + \sum_f tT_i^f$$

If a task can be sliced into multiple fractions, the solver must consider all of them. Equation 14.5 ensures that a task is completely executed:

$$\sum_f \delta_i^f = 1 \tag{14.5}$$

Changing the perception of the solver on how task is executed, also changes the objective function as it was presented in Section 13.3. As a recall it was defined as follows:

$$E = \sum_i (E_i) = \sum_i (Time_i^f \times P_i^f)$$

$Time_i^f$ and P_i^f were the execution time and the power consumption of a task i at a frequency selected for the entire task execution. Since the execution is devised into multiple part, the formulation of the objective function is changed to :

$$min(\sum_{T_i}(\sum_f (tT_i^f \times P_i^f))) \tag{14.6}$$

Solving the new objective function showed in Equation 14.6, provides for each task, the time to spent in the different frequencies tT_i^f. With a dedicated system that ensures that each task is run for the correct amount of time on the computed frequencies, the application execution will theoretically generate the lowest amount of energy. However, nothing constrains parallel tasks on one processor to run at the same frequency, and the threshold of switching frequency is not considered either. As explained in Section 13.3.2 considering theses architectural limitations ensure a realistic solution. The closest solution to OUTREAch [143] did not take that into account, and as it will be seen its prediction precision suffers from that. Finally, OUTREAch add one constraints compared to [143]. That was not discussed previously. Like FoREST, with OUTREAch the user has the possibility to select the quantity of time degradation it allows. OUTREAch will then find the lowest energy consumption regarding that new constraint. The next sections introduce the additional constraints related to the architecture and the execution time limitation.

14.4.2 Execution Time Limitation

It was seen in Part II with FoREST that significant energy consumption reduction could be achieved even by limiting the performance degradation. Moreover, whether the energy consumption is considered or not, the performances of an application is always a major concern. Constraints to control that performance degradation is integrated in OUTREAch. The remainder of the Section describes how such a constraint was translated into linear programming.

In MPI, all programs end with *MPI_Finalize* which is similar to a global barrier. Let lT^i be the last task on core i. Since the application ends with a global communication, every task lT^i is followed by a slack task lTs^i. The difference between

the global communication slack and the other slack tasks lies in the end time: the end times of all slack tasks of a global communication are the same as all processes leave the barrier simultaneously. Hence, for every couple of cores (i, j):

$$elTs^i = elTs^j \tag{14.7}$$

Let $total\_Time$ be the application execution time. It is equal to the end time of the last slack task.

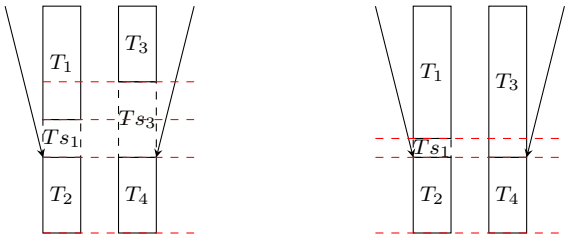
$$total\_Time = elTs^i \tag{14.8}$$

However, in some cases, increasing an application execution time benefits to energy consumption as it was shown during the entire Part II. In order to allow this performance loss to a specified extent, the user controls this performance loss by limiting the degradation to a factor x of the maximal performance, like the one defined in FoREST. Let $exec\_Time$ be the execution time when all tasks run at the maximal frequency, and x the maximum percentage of loss allowed by the user. The following constraint allows performance loss with respect to x:

$$total\_Time \leq exec\_Time + \frac{exec\_Time \times x}{100}$$

As an example, it will be demonstrated, in Section 14.5, that for the NAS benchmark programs suite, more than 50 % of the maximum energy savings are already achieved when only setting the execution time degradation limit at 10%. Then even for users that fears dramatic performance loss, significant energy saving can be performed. However, that performance limitation is not the sole OUTREAch contribution. To be able to realistically find the minimal energy consumption, OUTREAch takes into account all possible task configuration, i.e. all possible combination of tasks working in parallel on the same processor. It then chooses the one minimizing the energy. That method is presented in the next Section.

14.4.3 Architecture Constraints: The Workload Approach



(a) f_max (b) f_min

Figure 14.8: Workloads

As stress out before, on current SandyBridge and IvyBridge architecture, all processors cores share the same operating frequency. Therefore, when selecting a frequency for a task, all other concurrent tasks on the same CPU also have to be taken in account. It is the goal of the workload approach. It is achieved by computing the complete list of tasks that are potentially run concurrently. However, each task within the application task graph can be run at different frequencies, potentially modifying its concurrent neighbors list. A neighbor is a task executed

bW_i	Beginning of a workload W_i
eW_i	End of a workload W_i
tW_i^f	The time a workload W_i is executed at frequency f
dW_i	The duration of a workload
$\overline{tW_i^f}$	A binary variable used to say if a workload is executed at a frequency f or not

Table 14.3: Workload formulation variables

at the same time and on the same processor as the considered task. For a single task multiple possible neighbors can exist. The first step is then to compute the full combination set of possible neighbors. One combination of possible neighbors is called a workload. The full list of possible workloads is generated beforehand by crawling into the data set measured during OUTREAch first step. As an example, consider Figure 14.8, two different executions of the same application performed at the maximal and minimal frequency. Only processes that belong to the same processor are represented. In Figure 14.8a, when the processor runs at f_max, the set of neighbors is: $\{(T_1, T_3), (T_1, Ts_3), (Ts_1, Ts_3), (T_2, T_4)\}$. The horizontal dotted lines represents the separation of each neighbor. When the frequency is set to f_min, shown in Figure 14.8b, the slack after T_3 is completely covered and the set of neighbors becomes: $\{(T_1, T_3), (Ts_1, T_3), (T_2, T_4)\}$. OUTREAch builds the set of workloads based on the list obtained at each frequency . For the example, OUTREAch builds five different workloads : $W_1 = (T_1, T_3), W_2 = (T_1, Ts_3), W_3 = (Ts_1, Ts_3), W_4 = (Ts_1, T_3), W_5 = (T_2, T_4)$. Notice that there are no workloads with the same set of tasks. By construction, a workload is intended to display the tasks that are run concurrently, so that when a task is finished a new workload is started. The process of workload building is repeated on each processor. Once the complete list of possible workloads is computed, it is given to the solver, that will choose the best ones to reach minimum energy consumption. The next section explains how to describe the workloads to the solver and how to handle them in order to let the solver choose the best set.

14.4.3.1 Shared Frequency Constraint

Before going into further details, Table 14.3 explains the different used variables and their meanings. One can find the different variables very similar to the ones used to describe tasks in Table 14.2. A workload will be seen by the solver as a meta-task. It is the bridge between the tasks belonging to the same workload. It helps the solver to understand the potential harm brought to other tasks within the same workload when a frequency is chosen for one of them. Basically the workload are here to force parallel tasks to run at the same frequency.

As defined previously for the tasks, a workload can be sliced into fractions tW_i and, for each fraction, a new frequency can be chosen. However the objective function defined in Equation 14.6 uses task slices tT_i and not workload slice. The following constrain define the relation between the workload and the tasks.

As an illustration consider Figure 14.8. On the left side of the figure there is the list of frequency that was decided by the solver. The succession of workloads is defined on the right side of the figure. As stated above, the challenge is to retrieve, for each task, the portion executed at different frequencies. In the example the

workload $W_1 = (T_1, T_3)$ is executed at frequency f_1 then at frequency f_2. However, $T1$ also belongs to workload $W2$ which is executed at frequency f_1 then at frequency f_2. The execution time of T_1 at frequency f_1 ,$tT_1^{f_1}$, can be calculated by using the fraction of time W_1 and W_2 spend at frequency f_1. In other words, the execution time of a task can be calculated according to the execution time of the workloads it belongs to. Let tW_i^f be the fraction of time the workload W_i spends at frequency f. Thus:

$$tT_i^f = \sum_{W_j, T_i \in W_j} tW_j^f \tag{14.9}$$

As explained above, the workload approach needs that all the possible workloads are described to the solver. However, depending on the optimization performed by the solver, some task combinations are no longer possible. A mechanism to remove the invalid workloads has to be added.

14.4.3.2 Valid Workload Filtering

As explained above, the linear program is provided with all possible workloads. However, all workloads cannot be present in one execution. In Figure 14.8, $W_1 = (T_1, Ts_3)$ and $W_2 = (Ts_1, T_3)$ are two possible workloads, but they cannot be active in the same execution, because if W_1 is being executed, it means that T_3 is over, since Ts_3 comes after T_3. Hence, W_2 cannot appear later since Ts_1 and T_3 never are parallel. Thus, in order to prevent W_1 and W_2 from both existing in one execution, a check is needed to verify whether the tasks of the workload can be parallel or not, depending on the execution context. Two tasks are not parallel if one ends before the beginning of the second. Since workloads are only considered, the focus is put only on the beginning and end time of the workload. Let bW_i and eW_i be the start time and the end time of the workload $W_j = (T_1, \ldots, T_i, \ldots, T_n)$. They are such that:

$$bW_j >= bT_i \tag{14.10}$$

$$eW_j <= eT_i \tag{14.11}$$

Note that although the beginning and the end of the workload are not exactly defined, this definition makes sure that the beginning or the end of a task starts a new workload. Moreover, the complete execution of a task is guaranteed thanks to equations (14.5) and (14.9).

Figure 14.10 is an example of a workload that cannot exist. Consider the execution represented in Figure 14.10, and focus on the workload $W_1 = (T_1, Ts_3)$.

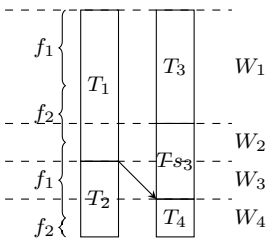


Figure 14.9: Workloads and tasks execution

Assume also that with other frequencies, a possible workload is $W_2 = (T_3, Ts_1)$. As explained above, W_1 and W_2 cannot both exist in the same execution because of precedence constraints. It can be seen, from the example, that T_3 and Ts_1 are not parallel, let see how it translates into workloads. Since W_2 has to start after both T_3 and Ts_1 begins, then it starts after Ts_1 (since $bTs_1 \geq bT_3$). In the same way it ends before eT_3. But since $eT_3 \leq bTs_1$ then the duration of W_2 would be negative which is not possible.

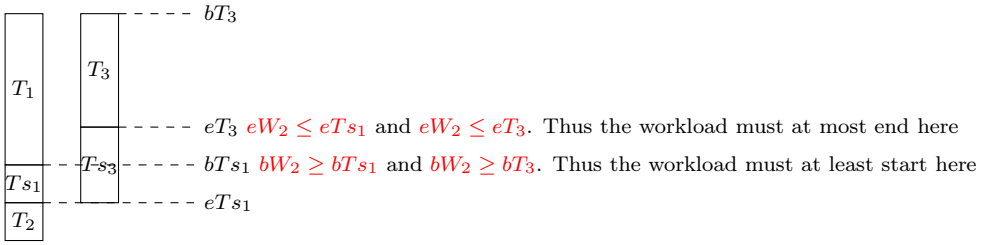


Figure 14.10: Negative workload duration for impossible workloads

Workloads filtering is then performed by identifying workloads which end before they begin. In such a case the solver is instructed to set a null duration forcing it to discard the erroneous workloads. Finally, the duration of a workload is such that:

$$dW_i = \begin{cases} 0 & eW_i < bW_i \\ eW_i - bW_i & otherwise \end{cases} \tag{14.12}$$

14.4.3.3 Handling Frequency Switch Delay

Recall that one of the problems when considering DVFS is the time required to actually set a new frequency. In order to set a new frequency, one has to make sure that the duration of the workload is long enough to tolerate the frequency change since changing a frequency takes some time. In other words, if the frequency f is set in a workload, W_i, tW_i^f must be larger than a user-defined threshold, denoted Th :

$$\forall W_i, \forall f : tW_i^f \geq Th \times \overline{tW_i^f} \tag{14.13}$$

$\overline{tW_i^f}$ is a binary variable used to guarantee that definition (14.13) remains true when $tW_i^f = 0$. The threshold, is not randomly defined, it is set as the worst frequency transition delay noticed on the set of processors. The measurement method was presented in Section 8.3.

$$\overline{tW_i^f} = \begin{cases} 0 & tW_i^f = 0 \\ 1 & otherwise \end{cases} \tag{14.14}$$

The expression of definition 14.14 as a linear programming formulation is expressed in the appendix in [59].

14.4.3.4 Discussion

The appendix in [59] provides a detailed formulation of the energy minimization problem using workloads. The formulation shows the use of two binary variables: one to express the threshold constraint and one to calculate the duration of the workload. With these two variables, the formulation is not linear anymore, which requires more time to solve, especially when the number of workloads is important.

Moreover, the workload generation was tested on IS, one of the NAS parallel benchmarks program on class C with 16 processes. The test machine was equipped with 16 GB of memory. The application task graph is composed of 630 tasks. The generated workloads could not fit in the memory of the machine. Thus, even with no binary variables, providing all possible workloads is not possible when considering real applications.

Though the workload approach could grant the maximum energy reduction, because it considers all possible task configurations, it is not a practical solution. Based on the work presented above, a new formulation is needed to remove the workload from the picture. The next section presents a refinement where no workloads but only the task dependencies are needed.

14.4.4 Architecture Constraints: The Frequency Switch Date Approach

The workload approach, even though it would have given the optimum solution, is not a practical solution since a machine with a tremendous amount of memory would be needed even when considering small applications. A new approach is then required. The previously presented workload constraints are then discarded, however the objective function and all the precedence constraints remain the same. In the next section a refinement of the workload approach is presented. The new solution ask the solver to compute the dates to set a new frequency on the whole processor. As all the task on the same processor share that shift dates, the frequency selection, regarding the needs of the different tasks, will be automatically adjusted by the solver. The hardware limitation about frequency domain is then transparently taken into account. With the new solution, the solver will provide a list of dates when a frequency shift must happen, and the list of frequencies to be set.

14.4.4.1 Frequency Switch Date

Let c_{jp}^{f} be the date when the frequency f is set on the processor p, j being the sequence number of the frequency switching. As an illustration, consider Figure 14.11 representing the execution of four tasks on two cores of the same processor p. In the example, it is assumed that there are only 3 possible frequencies. After the first switch date $c_{1p}^{f_1}$, the frequency $f1$ is set for both tasks T_1 and T_3. At the second shift date $c_{2p}^{f_2}$, the frequency $f2$ is set for T_1 and T_3. And so on for the entire list of switch dates. The solver must be thaught how to order the shift date, a date in the future cannot happen before a date in the past. More precisely a shift date with an sequence index $j+1$ is happening after a shift date with the index j :

$$c_{\{i+1\}p}^{f_2} \geq c_{ip}^{f_1}$$

Consequently the duration of a frequency application f between two consecutive dates is computed as follows :

$$duration^f = c^{f'}_{\{i+1\}p} \geq c^f_{ip}$$

Contrary to the previous approach where the user solely had the control over the execution time degradation, with the new solution, he can also set the number of frequency shift dates the solver has to compute. The user will then be able to go either for a fast solution or a very precise one. However, the solver still has the final word, even though the user chooses to have a lot of shifts date, if the solver considers that the optimum solution is achieved with less shift dates, it will discard the leftovers. As an example, consider Figure 14.11. Five shift dates were allowed, and the solver decided that one of them was not necessary. It is shown by the fact that two shift dates are equal. In the example, the shift date $c^{f_3}_{31}$ is never used.

c^f_{ip}	Date of the i^{th} frequency switch on processor p. The frequency f is the one set
d^f_{ij}	The amount of time a frequency f is set for the task i for the frequency switch j

Table 14.4: Frequency switch formulation variables

Table 14.4 shows the variable used by the solver for the current constraint formulation, it can be seen that $duration^f$ is not directly used. Indeed, recall that the objective function minimize the energy consumption of each task slice. So the duration of a frequency set between two consecutive dates cannot be used out of the box. From that duration, the different task slices tT^f_i have to be computed. As a clarification, consider Figure 14.11 it can be seen that T_3 is executed at $f1$ and $f2$ and T_1 is executed at $f1$ followed by $f2$ and finishes with $f1$. Then the time T_3 spends at frequency f_1, is $c^{f_2}_{21} - c^{f_1}_{11}$ whereas T_1 is $(c^{f_2}_{21} - c^{f_1}_{11}) + (eT_1 - c^{f_1}_{41})$ at frequency f_1. Let d^f_{ij} be the time the task T_i spends at frequency f after the frequency switch j. Back to Figure 14.11, $d^{f_1}_{11} = c^{f_2}_{21} - c^{f_1}_{11}$ and $d^{f_1}_{41} = eT_1 - c^{f_1}_{41}$. Based on both durations, $tT^{f_1}_1$ becomes $tT^{f_1}_1 = d^{f_1}_{11} + d^{f_1}_{41}$. By generalizing, it translates into:

$$tT^f_i = \sum_j d^f_{ji}$$

Note that a task is not impacted by a frequency change if it ends before the change or begins after the next change. In other words, $d^{f_1}_{ij} = 0$ if $eT_i \leq c^{f_1}_{jp}$ or $bT_i \geq c^{f_2}_{\{j+1\}p}$. Otherwise, $d^{f_1}_{ij}$ can be calculated as $min(eT_i, c^{f_2}_{\{j+1\}p}) - max(bT_i, c^{f_1}_{jp})$.

$$d^f_{ji} = \begin{cases} 0 & eT_i \leq c^f_{jp} \; or \; bT_i \geq c^{f'}_{\{i+1\}p} \\ min(eT_i, c^{f'}_{\{j+1\}p}) - max(bT_i, c^f_{jp}) & otherwise \end{cases} \tag{14.15}$$

With this list of constraints the solver will be able to compute the different shift dates, and acknowledge their impact on the objective function. However, the duration between two cut dates can be lower than the time needed to change a frequency. To take that into account, the following set of constraints is added to the solution.

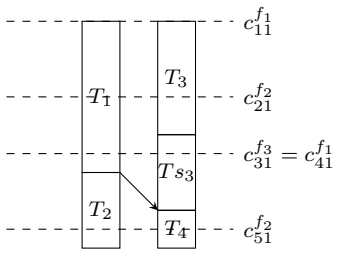


Figure 14.11: Frequency switches example

14.4.4.2 Handling Frequency Switch Delay

As explained earlier, changing frequency takes some time. Thus, for a change to be applied, its duration has to be longer than the user-defined threshold Th. As explained before, the threshold is measured to be the worse case frequency shift duration. Let ζ_{ip}^{f} be a binary variable, that:

$$\zeta_{ip}^{f} = \begin{cases} 0 & c_{\{i+1\}p}^{f'} - c_{ip}^{f} = duration^{f} = 0 \\ 1 & otherwise \end{cases} \tag{14.16}$$

The threshold condition is then expressed as:

$$c_{\{i+1\}p}^{f'} - c_{ip}^{f} \geq Th \times \zeta_{ip}^{f}$$

The binary variable is used here to take out of the picture the shift dates that start at the same time preventing the solver from spending time on on a meaningless comparison. For example, based on Figure 14.11, the comparison $c_{31}^{f3} - c_{41}^{f1} \geq Th$ has no meaning since $c_{31}^{f3} - c_{41}^{f1} = 0$, however it is discarded thanks to ζ.

The translation of each constraint into into their linear programming versions is detailed in the appendix section of [59]. Unfortunately, that new refinement also has drawbacks discussed in the next Section.

14.4.5 Discussion

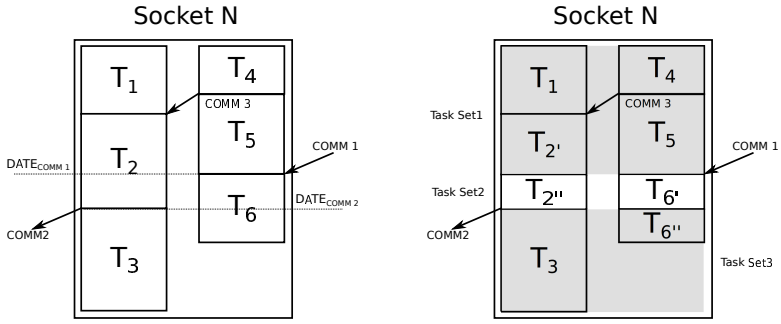
The appendix in [59] provides the complete formulation of the problem using the frequency switch time variables. In addition to the binary variable used to satisfy the frequency switch delay, five additional binary variables are used for each task and for each frequency switch. For n tasks and m frequencies switch, $5 \times n \times m$ binary variables are required. Mixed integer programming is NP-hard [121], therefore with such a number of binary variables, no solution can be provided.

When comparing the workload approach and the frequency switch approach, it can be noticed that the former needs less binary variables and should be able to provide a result. However, because all possible workloads have to be provided to the solver, it is as complex as the second approach because of the amount of memory required. On the one hand, if a extremely large memory is available, the workload solution is the one to be used. On the other hand, if new faster binary resolution techniques are provided, then the frequency switch solution should be used.

Several heuristics can be assumed in order to reduce the time to solve the problem. First, iterative applications can be considered. By solving the problems for only one iteration and then using the solution on the remaining ones, the amount of tasks to be considered by OUTREAch can be drastically reduced. However, this

solution strongly restricts the type of applications that can be optimized. In addition the solution still depends on the number of tasks per iterations: if the number of binaries is too large the problem still remains. It was decided not to reduce the scope of applications considered by OUTREAch but to apply a final refinement to the problem formulation. This new approach can be seen as the fusion of both previous attempts. Instead of finding all possible execution scenarios for a group of concurrent tasks, it is decided to fix that set of tasks. Based on arbitrary decided dates, the application will be divided in multiple sets of tasks. The solver will then try to find the best frequency for each set of tasks, to solve the previously stated objective function. That approach solves the main issues of both previous solutions; No more combinatorial explosion for finding all the workloads, and fewer binary variables. That last refinement is presented in the next section.

14.4.6 Super Tasks



(a) A Task graph (b) The task graph abstraction

Figure 14.12: Task graph to super-task graph

The workload approach tried to consider all possible execution scenarios in order the find the best energy consumption. Theoretically, it should have given the optimum solution. However, generating all possible concurrent tasks execution scenarios over the different frequencies ended up with a combinatorial explosion. However, the workload was first thought to tackle the processor core shared frequency. It was then decided to keep that notion of workload and eliminate the combinatorial explosion. When considering a single frequency, it is straightforward to find if two tasks are concurrently run. Then, instead of simultaneously considering all the frequencies to determine if two tasks are run in parallel, only one is taken into account. The questions that naturally arise are, how to choose the frequency? Is a frequency better than another? To answer these questions, each frequency were tested for each NAS benchmark program presented in Section 14.5. In the end, only an average 1.05% variation on the prediction is spotted. That approximation is considered as valid.

To build the new generation of workloads, called super-task, a criterion to start or finish a super-task had to be defined. Recall from the introduction, when UtoPeak was tried on a two processors set-ups. The major drawback stated then was the incapacity of UtoPeak to communicate its frequency choices to the other UtoPeak instances running on the other processor. Its enlightens the importance of the dependencies between the processors more than the dependencies between the tasks on the same processor. Based on that observation, it was decided to start or finish a

super-task when a dependency to another task belonging to another processor was noticed. As an example consider Figure 14.12a displaying a simple task graph of an application executed on a socket. The other application tasks are executed on other sockets. $COMM1$ and $COMM2$ display the communications coming from and going to other processors. Based on that execution scenario, three super-tasks are built. The result of the construction is displayed in Figure 14.12b. By using the link to other processors, as $COMM1$ and $COMM2$ a super-task graph can be built upon the task graph. Each super-task is linked to one another thank to extra socket communications.

On the solver side, that new abstraction does not change the set of constraints. Indeed, a super-task is strictly identical to a workload, all the constraints described in the Workload approach section are valid. Except the set of constraints for filtering invalid workload since here, each super-task represents an actual execution scenario.

It will be seen in Section 14.5 that, for some NAS parallel benchmarks, the solver needs up to 14 hours to find a solution. Even though the solver now is able to converge unlike in the previous approaches, 14 hours is not affordable in an HPC environment. It was therefore decided to builds groups of super-tasks. Merging small super-tasks together reduces the scope of variables to be tested by the solver. As an example, consider Figure 14.12b. If the super-task 2 is considered as too small, it is merged to the super-task 1. However, the communication $COMM1$ cannot be dropped from the picture. To keep the fact that the super-task 3 can be impacted by any delay on $COMM1$, the communication will then arrive at the end of the super-stask 2. Super task compression though alter a bit the super-task dependencies, this strongly helps to reduce the time to solution without hurting much the prediction quality as will be shown in section 14.5. That approximation is also considered as valid.

In the end, in addition to the application constraints, three different models to take into account the architectural constraints were presented. Each of them has advantages and drawbacks but only the super-task model was able to converge to a solution. The accuracy of the produced predictions and the time needed to achieve them are presented in the next section. As presented above, OUTREAch has a major contestant [143]. As it only uses the same set of constraints defined in the precedence constraint section, it was implemented in parallel to OUTREAch to compare their predictions and times to solution. The comparison between both system is presented in Section 14.6.

14.5 Experimental Results

In order to evaluate OUTREAch precision and potential for energy reduction, the NAS parallel benchmark programs are selected. For each benchmark application the class C is considered, and up to 64 processes are used to execute the different benchmarks. OUTREAch profiling step and frequency evaluation is performed on HPC resources located at Strasbourg University. Depending on the execution scenario, two to eight Xeon E5-2670 processors are used distributed into one to four machine nodes. For the linear programing resolution, a desktop machine with an Intel Core i7-3770 and 16 Go of RAM is used. Ideally every step of OUTREAch could have been performed on Strasbourh HPC ressources, however the Gurobi[60] academic license could only be attached to a single machine. The execution procedure then is as follows. First OUTREAch profiling step is launched with the NAS benchmarks

application programs on the HPC ressources. Once the profiling is finished, the gathered information are sent to the desktop machine to normalize the energy over the application tasks and communications. OUTREAch then builds the internal representation of the application and outputs the linear problem to be solved. The solver is started with the output linear problem, and once the energy consumption and the frequency sequences are made available, they are sent back on the HPC resources to be evaluated.

It can be noted that the used set of processors is homogeneous. It is chosen in such a way that it insures the hardware characteristics and the set of processor frequencies are equal. However, OUTREAch could handle non homogeneous processors set. The major requirement is to have the same application processes pined on the same processor cores, for all the application execution needed by OUTREAch profiling step and frequency sequence evaluation. If it is not the case, additional constraints should be added to OUTREAch, but it is part of future work.

Finally the execution procedure, could not ensure that the nodes used for the profiling step are the same as the ones used for the frequency evaluation. OUTREAch was not the only tool requesting access to Strasbourg HPC resources then to hasten the experiments, it was decided to choose the first available nodes with the same processors for the profiling and the frequency sequence. By doing so, the frequency sequence evaluation can be altered since it was build regarding a determined execution set-up. It has to be taken into account when analyzing OUTREAch accuracy.

	Degradation Limit : 500%			Degradation Limit : 10%		
	Evaluation	Prediction	Precision	Evaluation	Prediction	Precision
IS.16	344.51	348.71	98.79%	387.88	391.62	99.04%
IS.32	487.18	488.53	99.72%	521.44	519.19	99.56%
IS.64	926.28	907.73	97.99%	950.13	916.25	96.43%
FT.16	3491.67	3273.08	93.74%	4202.07	4221.04	99.55%
FT.32	4132.56	4165.44	99.21%	4950.68	4967.63	99.65%
FT.64	4662.68	4659.48	99.93%	5346.53	5239.53	97.99%
EP.16	1169.63	1157.42	98.95%	1259.48	1246.77	98.99%
EP.32	1173.02	1116.46	95.17%	1281.82	1256.73	98.04%
EP.64	1214.17	1174.77	96.75%	1332.12	1271.93	95.48%
BT.16	13098.8	11355.81	86.69%	15534.51	14671.1	94.44%
CG.16	2299.93	2268.00	98.61%	3011.14	2765.59	91.84%
CG.32	2144.71	2033.18	94.79%	2513.31	2471.99	98.35%
MG.16	895.99	866.07	96.66%	1038.95	983.45	94.65%
MG.32	906.78	853.09	94.08%	1104.85	1019.64	92.28%
SP.16	11302.21	10123.62	89.572%	12099.70	11806.77	97.57%
LU.16	8718.75	7427.93	85.18%	10049.25	9399.52	93.53%
LU.32	9617.36	8359.36	86.91%	11157.36	10242.08	91.75%

Table 14.5: OUTREAch accuracy on NAS parallel benchmarks programs with two limit on performance degradation.

As previously described, OUTREAch allows a limitation of the performance degradation. For each benchmark execution, two different limits are then set. One at 500%, it allows an infinite time degradation to allow OUTREAch to expose the

maximum energy reduction. Another limit was set at 10% in order to see how OUTREAch performs with less time degradation. It is interesting to see that with the 10% limit, more than half of the energy reduction achieved with the 500% already is optimized as it will later be seen in Table 14.6.

To evaluate the OUTREAch legitimacy, two important factors have to be taken into account. First, OUTREAch energy prediction has to be determined. Also, the time to reach a solution is as important as the prediction accuracy. Indeed, if the solution is the most accurate but takes years to be computed, it is not a practical solution. However, if the prediction is not very accurate, but if this solution is quickly available, then it is a good practical solution. Therefore when analyzing the next presented results, each prediction errors has to be put in perspective with the time to solution in order to have a good evaluation of the OUTREAch results.

Table 14.5 shows the accuracy of OUTREAch on multiple NAS benchmark programs when considering two different performance degradation limitations. One can notice that BT.64,CG.64,MG.64,SP.64 and LU.64 are not listed in the table. It is because each application generated a huge amount of tasks and communications, making the machine swap when the solver tried to perform its optimization. They are not considered here, but having a machine with more memory would easily solves that issue. In most cases OUTREAch prediction precision is above 90%. However, for LU.16/32 and BT.16, the accuracy drops due to multiple factors. Identifying them with accuracy is difficult since multiple sources of variations exist. The most probable comes from how the applications are implemented and their impacts on the super-tasks. Indeed, both BT and LU uses intensively point to point communications, then a huge number of super-tasks are built. As explained before, super-tasks energy consumption are derived from the portion of tasks comprising them, magnifying potential measurement errors. It forces the solver to produce an over optimistic solution, like for GCC and UtoPeak, because it has difficulties grasping the execution reality. Furthermore, since they are both intensively sending messages, any temporary congestion on the network can alter the frequency sequence evaluation, increasing the application energy consumption since unexpected slack times appear. Even though both application has lower prediction precision, they still offer decent energy reductions as shown in Table 14.6.

The table shows the energy reduction granted by OUTREAch for the the different NAS benchmark applications per degradation limit. The third column shows the portion of the maximum energy reduction achieved at the 10% limit. As for UtoPeak, the potential for energy consumption is computed as the difference between what OUTREAch grants and what the highest frequency grants. The highest frequency is chosen because it is the default frequency chosen by frequency governor once it spots processor activity. It can be noted that the more processes the more energy reduction potential is unveiled by OUTREAch. It can be explained by the Amdahl Law [62]. The Amdahl law is used, in the parallel world, to quantify the maximum speed-up one can expect by increasing the parallel resources in order to accelerate the parallel section of an application. Here, in the case of the NAS parallel benchmarks programs, the granted speed up is not high enough to overcome the increase in power consumption by using more processors. Further more, Table 14.6 only focuses on the processor point of view. The increase in power consumption can be more dramatic if all the other hardware parts are taken into account. One has then to be very careful when adding processors to an application execution, though it can gain in performances, it is not clear about the energy consumption.

Gain Over The Maximum Frequency			
Benchmark	500% Limit	10% Limit	Already Achieved At 10%
IS.16	25.73%	16.38%	63.65%
IS.32	40.20%	35.99%	89.54%
IS.64	51.86%	50.62%	97.61%
FT.16	32.34%	18.58%	57.44%
FT.32	31.54%	17.99%	57.04%
FT.64	39.15%	30.23%	77.20%
EP.16	9.43%	2.47%	26.20%
EP.32	11.57%	3.37%	29.14%
EP.64	23.04%	15.56%	67.55%
BT.16	26.94%	13.35%	49.56%
CG.16	32.10%	11.10%	34.58%
CG.32	33.81%	22.44%	66.36%
MG.16	38.33%	28.49%	74.33%
MG.32	38.64%	25.24%	65.31%
SP.16	45.51%	41.67%	91.55%
LU.16	25.83%	14.51%	56.18%
LU.32	25.43%	13.49%	53.05%

Table 14.6: OUTREAch energy reduction potential

Finally, the last column of Table 14.6 shows the portion of the maximum energy consumption achieved when the degradation is set to 10%. In the case of OUTREAch it was decided not to put a discrimination on the range of frequencies and use what is allowed on the set-up. In the case of the machine in Strasbourg, the turbo-boost frequency was activated. The fact that more than 50% of the energy allowed with the 500% limit is already achieved at the 10% limit confirms the fact that Tubo-Boost has to be deactivated by all means if any one cares about energy consumption. However, it was not expected to be in such proportions. In the end, Table 14.6 shows that adding resources of any kind to speed the execution is not always transposed into energy savings. One has to be careful, because, in the future, HPC centers may change their billing procedure to charge the energy consumed and not just the usage time.

Having a tool that has a good precision and exposes good potential in energy reduction, is of no use if it is not able to converge, or is taking an unrealistic amount of time to produce a solution. Table 14.7 shows the time needed by the solver to converge to a solution. It can be seen that on the short benchmarks that are mainly using collective operations, the solver is fast to produce a solution. However, when the number of tasks and communication drastically grows, the complexity of the problem increases as well as the time to find a solution.

When seeing that more than two hours were needed for some benchmark the compression optimization described in the last model refinement is also considered. By simplifying the super-task graph, the space of possibilities considered by the solver was drastically reduced, hastening the time to solution. By seeing the acceleration granted by the optimization, some could wonder how much accuracy of the prediction and frequency evaluation was abandoned to get this reduction in conver-

Time To Solution In Seconds		
Benchmark	Without Compression	With Compression
IS.16	0.001	0.001
IS.32	0.01	0.001
IS.64	0.06	0.02
FT.16	0.01	0.001
FT.32	0.02	0.01
FT.64	0.06	0.02
EP.16	0.001	0.001
EP.32	0.001	0.001
EP.64	0.01	0.001
BT.16	6354	7.51
CG.16	176.05	3.15
CG.32	6174	13.29
MG.16	56.79	1.65
MG.32	6526.71	0.86
SP.16	56.79	1.65
LU.16	39532	3.14
LU.32	52364	20.54

Table 14.7: OUTREAch converging time with and without the super-task graph compression

gence time. On average, on all the benchmarks in Table 14.7 a variation of 1.43% and 0.29% are noticed for the evaluation and prediction with the 500% limit. For the 10% limit, a variation of 2.19% and 1.42% are respectively noticed for the evaluation and the prediction. Then the compression optimization is able to grant a significant time to solution reduction without hurting the predictions and the measurements performed when replaying the frequency sequence. It will be seen later in Section 14.7 that the compression optimization was able to grant so much speed-up without hurting the predictions only because the tested applications had a very homogenous energy behavior across the different processors.

OUTREAch is able to predict with accuracy the lower bound of processor energy consumption for distributed applications. It is also able to produce the prediction in a reasonable amount of time. However, as said above, OUTREAch has a major contestant [143] and the next section is dedicated to their comparison to validate that OUTREAch better performs since it takes into account more constraints.

14.6 OUTREAch versus the world

As explained above, the authors in [143] only take into account the application dependency constraints and not the hardware constraints. Only considering one side of the token generally gives truncated solutions. As an example, recall REST, it was only taking hardware performances into account to perform energy reduction. It was demonstrated that its savings were far from the optimum. It is the same here, by not taking into account the hardware frequency limitations, OUTREAch constestant will make over-optimistic frequency selection. Then the frequency sequence that will

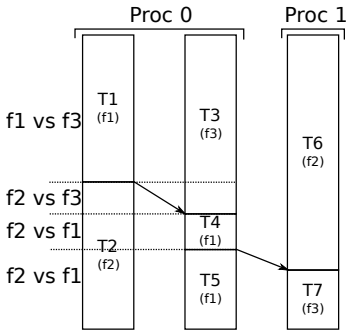


Figure 14.13: Frequency shift constraint

be executed, will not be the real reflection of what was computed. As an illustration consider Figure 14.13. In the figure $f1 > f2 > f3$. Each frequency displayed shows the frequency computed to be optimal for each task. Recall that the $cpufreq$ driver when facing different frequency shift requests always choose the maximum one. So instead of $f3$ for the task $T3$ the frequency $f1$ will be run, generating a difference between what was computed to be optimal and the reality of the frequency application.

For the purpose of the comparison, OUTREAch constestant will be called $SC07$ in the reminder of the section. The other difference between OUTREAch and $SC07$ is the granularity of the base element. On the one hand, OUTREAch considers groups of tasks, reducing the number of variables to consider in the linear program, thus its dependency to RAM. On the other hand $SC07$ considers the simple tasks as the elementary element inducing a strong dependency to the RAM amount available on the machine where the solver is run. For that reason, a solution could not be generated for LU.16/32 in addition to the application already dropped by OUTREAch for the same reasons.

Table 14.8 shows the comparison between OUTREAch and $SC07$ on the energy

Benchmark	OUTREAch vs SC07
IS.16	4.30%
IS.32	4.29%
IS.64	51.86%
FT.16	1.34%
FT.32	1.70%
FT.64	0.92%
EP.16	10.12%
EP.32	12.01%
EP.64	6.57%
BT.16	14.20%
CG.16	0.39%
CG.32	2.56%
MG.16	10.49%
MG.32	5.07%
SP.16	10.86%

Table 14.8: Energy reduction potential comparison between OUTREAch and SC07.

reduction potential they provide. Apart from $IS.64$ or $FT.16$ and $FT.32$, OUTREAch in average grants 10% more energy reduction than $SC07$. To be consistent, the comparison was performed with OUTREAch performance degradation limit set to 500%. Indeed, $SC07$ does not have an explicit degradation factor and lets the linear program decide to degrade the application performances as long as it translates into energy savings which is the default behavior of OUTREAch with the 500% limit.

Speed Up Factor		
Benchmarks	Without Compression	With Compression
IS.16	910	910
IS.32	337	3370
IS.64	90	270
FT.16	65	650
FT.32	112.50	225
FT.64	188.17	564.50
EP.16	110	110
EP.32	370	370
EP.64	170	1700
BT.16	3.78	3201.20
CG.16	140.02	7825.40
CG.32	5.91	2744.92
MG.16	15.35	528.21
MG.32	2.80	21259.30
SP.16	207.46	7140.45

Table 14.9: Time to solution comparison between OUTREAch and SC07.

One could think, that a lot of complex model were design to only get, in the end, only 10% better energy savings. However, it has to be kept in mind that OUTREAch is able to give a solution on LU.16 and LU.32 where $SC07$ could not. $SC07$ could not produce a solution on LU.32 because the problem was exceeding the memory size, the solver could not load the file and could not perform its first steps of test space reduction without forcing the machine to swap. For LU.16, the convergence algorithm was stopped at 53235 seconds because more than 14 hours to converge to a solution is not realistic. The second interest of OUTREAch over $SC07$ is the fact that it can accuratly predict the potential for energy reduction. In average for all the considered benchmark programs, OUTREAch has an prediction error of 3.95% where $SC07$ has 14.72%. The difference mainly comes from the fact, that $SC07$ does not take into account architecture constraints, ending up in producing over-optimistic predictions. It considers that for each task, a sequence of frequency can be set. However, it does not take into account the fact that the *cpufreq* driver always chooses the highest frequency among the shift requests. Moreover, $SC07$ does not take into account the time to switch frequencies. Some tasks are then executed at sub-optimal frequencies, increasing the difference between the prediction and the frequency sequence evaluation.

Finally, the last interest of OUTREAch over $SC07$ is the time it needs to produce a solution. To be fair, $SC07$ is compared to OUTREAch with and without the

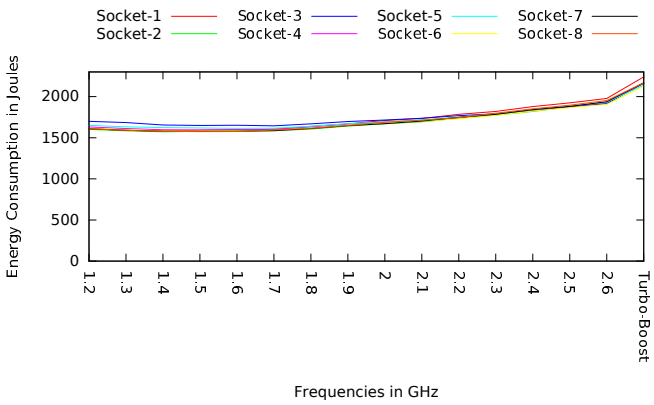
compression optimization. Table 14.9 displays the speed-up factor granted by OUTREAch when comparing it to $SC07$. The two columns show the speed-up factor on the time needed to produce a solution granted by OUTREAch with and without the compression optimization. For example on IS.16, OUTREAch produces a result in 0.001 seconds when $SC07$ takes 0.91 second. OUTREAch then offers a time reduction factor of 910. For some benchmarks as $IS.16$ or $EP.16$ the acceleration factor is identical between the two versions of OUTREAch. Indeed, the solver was not giving an answer with a finer resolution that the millisecond. For both versions on $IS.16$ or $EP.16$ the solver gave 0.001 second for the time to solution explaining the identical speed-up factor. In the end, even without the compression mechanism, OUTREAch is able to produce a solution way faster than $SC07$.

OUTREAch is faster, more accurate and exposes more energy savings than $SC07$. For $SC07$ defense, it was the first attempt to estimate the lower bound of energy consumption for distributed MPI applications. With OUTREAch contributions, that lower bound was decreased by 10% in average in a decent amount of time. However, as it was shown both solutions have a limitation, they are bound to the amount of memory available on the resolution system. Bigger problems mean bigger amount of memory to solve the problem. However, OUTREAch with the super-task graph approach started to move from a task based view to a more task group based view. The compression algorithm pursued in that direction, grouping more tasks into super-tasks. The result was a faster solution without hurting the prediction quality. Then, to break the dependency to the amount of RAM available for the solver one possible way would be to consider processors in addition to tasks, or group of tasks. The next section describes briefly the frame of future work.

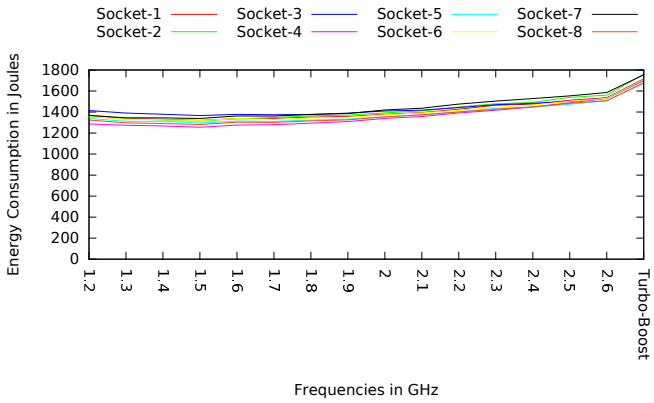
14.7 What Next ?

The goal of that section is to think about potential improvements for the OUTREAch RAM dependency. When facing a huge amount of tasks and communications, around 33 million for LU.64, 5 million for CG.64, and 2 million for BT.64 and SP.64, the solver has to deal with a vast space of variables. A huge space of RAM is then required for the solver to load all the variables and to perform its work. The time to solution is also bounded to the size of the variable space to explore. OUTREAch started an abstraction of the traditional single tasks model, to overcome the RAM dependencies and long time to solution. What if that abstraction can be taken to another level? Consider the Figure 14.14 displaying the energy behavior over the different processors frequencies across the sockets used while executing LU, BT or SP. It can be seen that each socket has the same energy behavior. Such a behavior is easily explained because the NAS benchmark application programs are very regular applications. Each processor is put under the same stress level.

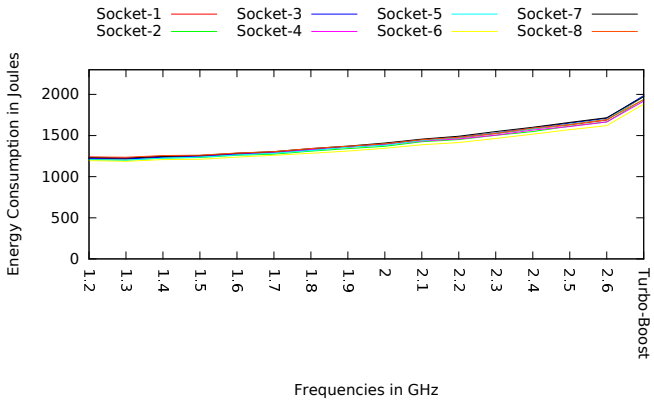
If OUTREAch were to consider an entire socket, a mechanism to detect the sockets energy tendency similarities can be created. Consequently, a very fast approximation could be produced, since the complexity of the potential detection mechanisms is bound to the number of the processor frequencies. However, it can exists applications exposing different energy behaviors for different processors. In such a case, the previously proposed approximation is no longer valid and OUTREAch has to be used. However, clusters of energy behaviors can be created. On each clusters an approximated best frequency can be selected as suggested above. By feeding that hint to the solver OUTREAch can potentially reduce the test space



(a) BT.64 energy behavior on each involved processor



(b) LU.64 energy behavior on each involved processor



(c) SP.64 energy behavior on each involved processor

Figure 14.14: Energy behavior across multiple processors

to consider reducing its dependency to RAM.

In a nutshell, like with the applications trends classification performed in Section 8.1 from Chapter 8.1 where optimizations techniques could be derived from the different trends, OUTREAch could use new insights from such a classification applied to distributed applications to further reduce the space of variables to consider. Every answers to the questions that arise then are part of future work.

Chapter 15

Conclusion

Moving from a single processor to a multi-processor problematic is not easy and simply transposing UtoPeak to that new world appeared not to be the solution. Indeed, a wide new set of constraints has the be taken into account. On the one hand, the application constraints changed. Instead of an array of application phases or tasks, a graph of tasks has to be considered. Changing the frequency of one task can have harmful effects and ultimately force the application to consume more energy. On the other hand, the hardware also exposes limitations that have to be considered. Only one frequency can be applied within an entire processor, then choosing one frequency for a specific task impacts all the others being executed. An unwise choice of a frequency for this task also can force the application to consume more energy. The solution is then simple, each frequency decision has to be validated according to both types of constraints.

However it cannot be solved with brute force, too many cases have to be considered to reach the optimal solution. Furthermore, designing a system to obtain the lower bound on energy consumption would probably cost more time than the one actually needed to converge to a solution. To solve that issue, linear solver was thought as the most appropriate solution. The only remaining problem was then to formulate a linear problem representing the different stated constraints, and reach the lowest energy consumption for one application execution.

To achieve that, three different models were designed but only one was able to reach a solution. The first two models, though they should have granted the optimum solution, were limited either by the computation power or by the RAM space needed. The first model would have needed an infinite memory space, and the second one would have needed an infinite computational power. The first model was computing all the combinations of possible concurrent tasks over a processor cores and frequencies. With the list of all possibilities, it should have been easy to identify which tasks are run concurrently and to choose a frequency accordingly. However, it ended with a combinatorial explosion. As an example for is.C.16 more than 16GB of ram was needed to generate all the combinations of possible concurrent tasks. Based on that teaching, the application execution was sliced in multiple sections, and for each section, the solver had to find the best frequency. However, to specify if a task was or was not inside a specific section, binary variables were needed. However, Mixed Integer Programming is NP-hard. With the considered amount of binary variables even is.C.16 could not converge. Facing that assessment, that new model was also put aside in favor to a simpler refinement. The first model was reconsidered, and instead of generating all possible execution set-ups for each concurrent tasks regarding each processor frequency, only one set-up was taken into account. In a nutshell, only the task schedule at a single frequency was considered to find the different concurrent tasks. By aggregating them inside super tasks, the execution was abstracted to a single process of super-tasks per processor. However, the solver still has the knowledge of each task comprising a super-task in order to evaluate the real impact of each frequency selection. The abstraction drastically reduced

the scope of variables the solver had to consider, allowing it to converge to a good solution. "Good" is used here because OUTREAch has the capacity to converge in a decent period of time and reduce the lower bound in energy consumption by 10% with 10% more accuracy and 3000 times faster than its major contestant.

Nonetheless, OUTREAch still is dependent to RAM space. It is its main limitation for targeting bigger applications. A proposed solution to investigate as future work, would be to consider the energetic behavior of the application at the level of the entire socket, and give the solver more hints to reduce the number of possibilities to test.

OUTREAch demonstrated that it is possible to predict energy consumption of a parallel application in a distributed environment with decent convergence time. It also demonstrated that the task model is too finely grained and a coarser grain must be preferred since it allowed OUTREAch to further reduce its convergence time without impacting the prediction quality.

Chapter 16

Conclusion

16.1 Contribution of This Thesis

This thesis presents some original approach to perform DVFS on single and multi processors environment. It also presents new approach to evaluate the maximum potential for energy savings. The interest of being able to obtain the maximum energy reduction one can expect from the use of any DVFS mechanism is threefold. First, it confirms that DVFS techniques still are valid to reduce any application energy consumption on modern processors when considering either single or multiple processors environment. Second, it shows that using DVFS techniques makes more sense on parallel applications than on sequential applications. Finally, it allows to actually measure the efficiency of any existing DVFS techniques. Combining these three major benefits, implies that a DVFS mechanism can reach the optimal saving, provided it can correctly grasp the architecture power consumption and limitation, even with no prior knowledge of the application.

In the first part of this thesis, different power consumers within a simple machine are analyzed. It appears that with the current state of technology very few hardware components can be controlled from software space. On a simple machine only the fans and the processor are exposing such capability. After that assessment, it was decided to focus the efforts on processors. Indeed, even though fans greatly help reducing the processor leakage, they have a limited impact on the overall system energy consumption. Moreover, processors are acknowledged as the major consumers in a computer or sever blade, therefore, by optimizing its energy consumption, savings should be extended to the entire system. However processors are complex and their energy consumptions have to be demytified. It was performed using simple memory and compute operations. On the one hand, if an application is bounded to the memory hierarchy then it is not required to wait at full speed and the frequency could be decreased to lower the processor energy consumption. On the other hand, if an application intensively performs arithmetic instructions, choosing any other speed except the maximum one, generally translates to an over-consumption of energy. It is then easy to optimize any application energy consumption by choosing either the highest frequency or the lowest one. However that binary vision of energy optimization is not always true. Depending on the influence of the static power consumption, the lowest frequency can be a good choice to reduce energy consumption even for arithmetic instruction. Raising the legitimate question to speed or not to speed ? In short, it depends on the application.

The second part, presents a detailed study of application energy consumption performed. Indeed, it exists as many applications as there are problems to be solved. To clarify and organize the field of research, a classification is performed, exposing three categories. The first and second ones comprise compute intensive and memory bounded applications. The third one is composed of applications that are neither compute intensive nor memory bounded. They expose a complex alternation of com-

pute and memory bounded phases. The focus was put on the different application phases to understand how they influence the overall application energy footprint. It was expected to see that a more preponderant phase was enforcing its behavior to the overall application. However, it is the composition of each phase behavior that decides the energy consumption of the entire application. Furthermore, it was noticed that by choosing the frequency granting the lowest energy consumption for each application phase, the minimum energy consumption at the entire application could be reached. Based on that observation and a dynamic procedure to identify the different application phases and their boundedness, three different tools were built. The first one, was adapting the hardware performance to each application phase boundedness. By reducing the frequency, it was hopped to perform energy savings. Even with that naive use of DVFS, significant energy reduction could be performed ranging from 4% to 27%. It demonstrated that DVFS still is valid to operate significant energy reduction. As it was a naive first step, its efficacy was questioned but no reference point was existing to evaluate it. The second tool was created to fill that blank. It divides an application execution in small sets of instructions and find for each of them the frequency giving the lowest energy consumption. Based on information gathered during previous runs of the tested application, it predicts the lowest energy consumption. It was able to accurately demonstrate, less that 4% of error, that the previous attempt was far from optimal. It also demonstrated that DVFS technique should target parallel applications rather than sequential ones. On sequential applications, the potential for energy savings ranges from 0.40% to 16.07% on the SPEC2006 sequential benchmark applications, when for NAS-OMP benchmarks, it is between 15.29% and 45.29%. The last tool presented in the second part, corrects all the flaws of the first attempt, and is then able to reach the maximum energy consumptions. The question of DVFS legitimacy on a single processor is then answered. Within that domain, it has a real impact on parallel applications energy consumption. What about parallel applications on multiple processors ?.

The last part, presented a first attempt to predict maximum energy reduction of a parallel application executed on multiple processors. By taking into account the constraints originated from the parallel applications and from the hardware, the proposed solution tries to match a specific frequency to each application phase across the processors in order to minimize the overall application energy consumption. It cannot be solved with brute force because an unreasonable amount of combinations has to be taken into account. A solution using linear programming was considered to move the solving complexity to a state of the art solver. Regarding the problem formulation, the solver perform an energy prediction. As for the same system on a single processor, it is able to perform accurate predictions exposing 5% of error in average. The energy saving granted by the system on the tested applications are between 9.43% and 51.86%. The high capacity for energy reduction demonstrates, even on multi-processor environment, that DVFS can still be used on modern processors. The presented tool, OUTREAch has a major contestant. A comparison is performed to demonstrate OUTREAch superiority. The obtained evidence is that it decreases the lower bound on energy consumption by 10% with 10% more accuracy and 3000 times faster in average.

16.2 Future Works

Three major axes of future work can be identified. Firstly, pursue the energy characterization of assembly instructions, secondly reduce the DFaCe convergence time and plug it to REST or FoREST. Finally, reduce OUTREAch dependency to RAM space.

During the first part of the thesis, power and energy characterization of multiple memory and arithmetic instructions is performed. By extending that characterization to a more complete set of arithmetic instructions and memory access patterns, static code energy prediction could be performed. By analyzing the code given by the compiler or by using a disassembly tool [164] an energy prediction could statically be performed for an application on a given architecture. It would grant the possibility to point the best architecture for a given application in terms of energy efficiency.

Still during the first part of the thesis, a fan speed regulation technique was designed to regulate the processor power leakage. In addition to drastically enhance the time to solution, it would be interesting to couple that method to the presented DVFS techniques. Indeed, when REST or FoREST change the operating frequency to a lower one, it induces lower stress on the processor. The temperature generated by the application would be reduced. Reducing the fans speed accordingly could save additional energy when looking at the entire machine.

Finally, the last one, is to further enhance OUTREAch. As explained above by providing hints to the solver based on an analysis of the application energy trend per processor, it could help to reduces the space of considered variables, time to solution and RAM usage.

Bibliography

[1] Intel® turbo boost 2.0. *http : //www.intel.fr/content/www/fr/fr/architecture-and-technology/turbo-boost/turbo-boost-technology.html*. 35, 60

[2] Mpich. *https : //www.mpich.org/publications/*. 143

[3] Mvapich2: High performance mpi over infiniband, iwarp and roce. *http : //mvapich.cse.ohio − state.edu*. 143

[4] Enhanced intel®speedstep®technology for the intel®pentium ®m processor. White Paper, March 2004. 35

[5] Advanced configuration and power interface specification, July 2014. *http : //www.uefi.org/sites/default/files/resources/ACPI_5_1release.pdf*. 36

[6] Scaling up energy efficiency across the data center industry: Evaluating key drivers and barriers, August 2014. http://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf. 1

[7] U. E. I. Administration. Electric power monthly, 2015. *http : //www.eia.gov/electricity/monthly/current_year/january*2015*.pdf*. 11

[8] F. Ahmad and T. N. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 243–256, New York, NY, USA, 2010. ACM. 15

[9] C. Akel, Y. Kashnikov, P. de Oliveira Castro, and W. Jalby. Is source-code isolation viable for performance characterization? In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 977–984. IEEE, 2013. 62

[10] M. Annavaram, R. Rakvic, M. Polito, J. Y. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 93–104, Dec 2004. 62, 69, 79

[11] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza. 3d seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Sci. Program.*, 17(1-2):185–198, Jan. 2009. 58, 72, 121

[12] G. Aupy, A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert. Energy-aware checkpointing of divisible tasks with soft or hard deadlines. In *Green Computing Conference (IGCC), 2013 International*, pages 1–8. IEEE, 2013. 13

[13] R. Ayoub, S. Sharifi, and T. S. Rosing. GentleCool: cooling aware proactive workload scheduling in multi-machine systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 295–298, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. 15

[14] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA, 1994. 62, 81, 90, 101, 102, 121

[15] D. Barthou, A. C. Rubial, W. Jalby, S. Koliai, and C. Valensi. Performance tuning of x86 openmp codes with maqao. In *Tools for High Performance Computing 2009*, pages 95–113. Springer, 2010. 68

[16] C. Bash and G. Forman. Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 29:1–29:6, Berkeley, CA, USA, 2007. USENIX Association. 15

[17] E. Baysal, D. D. Kosloff, and J. W. Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983. 58, 72, 121

[18] C. Belady, A. Rawson, J. Pfleuger, and T. Cader. Green grid data center power efficiency metrics: Pue and dcie. Technical report, Technical report, Green Grid, 2008. 9

[19] Z. Bendifallah, W. Jalby, J. Noudohouenou, E. Oseret, V. Palomares, and A. C. Rubial. Pamda: Performance assessment using maqao toolset and differential analysis. In *Tools for High Performance Computing 2013*, pages 107–127. Springer, 2014. 62

[20] J. Beyler, N. Triquenaux, V. Palomares, F. Chabane, T. Fighiera, J. Halimi, and W. Jalby. Microtools: Automating program generation and performance measurement. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 424–433, Sept 2012. 43, 90

[21] J. C. Beyler and P. Clauss. Esodyp: An entirely software and dynamic data prefetcher based on a markov model. In *In Proc. 12th Workshop on Compilers for Parallel Computers*, 2006. 86

[22] U. Bondhugula, J. Ramanujam, and P.Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, oct 2007. 133

[23] P. Bose, M. Martonosi, and D. Brooks. Modeling and analyzing cpu power and performance: Metrics, methods, and abstractions. *Tutorial, ACM SIGMETRICS*, 2001. 9

[24] CGGVeritas. Reverse time migration, 2014. *http* : *//www.cggveritas.com/default.aspx?cid* = 4-11-2358. 72

[25] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. S. Shende, A. D. Malony, and W. Jalby. Mil: A language to build program analysis tools through static binary instrumentation. In *20th Annual International Conference on High Performance Computing (HiPC'13)*, Hyderabad, India, dec 2013. 68

[26] A. S. Charif-Rubial et al. CQA: A code quality analyzer tool at binary level. *HiPC '14*, 2014. 62

[27] J. C. Charr, R. Couturier, A. Fanfakh, and A. Giersch. Dynamic frequency scaling for energy consumption reduction in synchronous distributed applications. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 225–230, Aug 2014. 20, 35

[28] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 174–179. ACM, 2004. 9, 58

[29] R. Chu, M. Ellsworth, D. Porter, R. Schmidt, and R. Simons. Apparatus and method for facilitating cooling of an electronics rack employing a heat exchange assembly mounted to an outlet door cover of the electronics rack, June 2008. US Patent 7,385,810. 16

[30] P. Cichowski, J. Keller, and C. Kessler. Energy-efficient mapping of task collections onto manycore processors. In *Proc. 5th Swedish Workshop on Mulitcore Computing (MCC 2012)*, 2012. 131

[31] P. Cichowski, J. Keller, and C. Kessler. Modelling power consumption of the intel scc. In *The 6th Many-core Applications Research Community (MARC) Symposium*, pages 46–51. ONERA, The French Aerospace Lab, 2012. 131

[32] P. Clauss, I. Fassi, and A. Jimborean. Software-controlled processor stalls for time and energy efficient data locality optimization. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 199–206. IEEE, 2014. 58

[33] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1396–1410, 2008. 58

[34] M. Dabbagh and H. Hajj. An approach to measuring kernel energy in software applications. In *Energy Aware Computing (ICEAC), 2011 International Conference on*, pages 1–6. IEEE, 2011. 58

[35] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 31–40, New York, NY, USA, 2011. ACM. ix, 11, 12, 13

[36] P. de Oliveira Castro, Y. Kashnikov, C. Akel, M. Popov, and W. Jalby. Fine-grained benchmark subsetting for system selection. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 132. ACM, 2014. 62

[37] D. Delia, T. Gilgert, N. Graham, U. Hwang, P. Ing, J. Kan, R. Kemink, G. Maling, R. Martin, K. Moran, J. Reyes, R. Schmidt, and R. Steinbrecher. System cooling design for the water-cooled ibm enterprise system/9000 processors. *IBM Journal of Research and Development*, 36(4):791–803, July 1992. 16

[38] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. *SIGPLAN Not.*, 47(4):225–238, mar 2011. 12, 13

[39] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 217–227, Dec 2003. 62

[40] W. Digital. Wester digital green hard drives, 2014. *http* : *//www.wdc.com/wdproducts/library/SpecSheet/ENG/*2879-800026*.pdf*. 14, 57

[41] J. S. Dinh and G. K. Korinsky. Temperature dependent fan control circuit for personal computer, 06 1993. US 5526289A. 16

[42] E. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In B. Falsafi and T. Vijaykumar, editors, *Power-Aware Computer Systems*, volume 2325 of *Lecture Notes in Computer Science*, pages 179–197. Springer Berlin Heidelberg, 2003. 20, 35

[43] W.-C. Feng. Making a case for efficient supercomputing. *Queue*, 1(7):54, 2003. 9

[44] X. Feng, R. Ge, and K. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 34–34, April 2005. ix, 7, 8, 35

[45] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus, mar 2014. 47, 48

[46] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173. ACM, 2005. 81, 95

[47] V. W. Freeh, F. Pan, N. Kappiah, D. Lowenthal, and R. Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 4a–4a, April 2005. 95

[48] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 133

[49] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 143

[50] R. Ge, X. Feng, and K. Cameron. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 34–34, Nov 2005. 20, 35, 79, 96

[51] R. Ge, X. Feng, and K. W. Cameron. Improvement of power-performance efficiency for high-end computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005. 9

[52] R. Ge, X. Feng, W. chun Feng, and K. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 18–18, Sept 2007. 9, 109, 114

[53] Y. Ge, P. Malani, and Q. Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 579–584, New York, NY, USA, 2010. ACM. 15

[54] P. J. Giorgio. Method for optimizing the rotational speed of cooling fans, 11 1996. US 5777897A. 16

[55] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996. 9

[56] S. Götz, T. Ilsche, J. Cardoso, J. Spillner, U. Aßmann, W. Nagel, and A. Schill. Energy-efficient data processing at sweet spot frequencies. In *On the Move to Meaningful Internet Systems: OTM 2014 Workshops*, pages 154–171. Springer, 2014. 80

[57] S. Greenberg, E. Mills, B. Tschudi, P. Rumsey, and Myatt. Best Practices for Data Centers: Results from Benchmarking 22 DataCenters. In *Proceedings of the 2006 ACEEE Summer Study on Energy Efficiency in Buildings.*, 2006. 16

[58] K. C. Gross, S. Ho, A. M. Urmanov, and K. Vaidyanathan. Controlling the power utilization of a computer system by adjusting a cooling fan speed, 01 2012. US 8108697B2. 16

[59] A. Guermouche, N. Triquenaux, B. Pradelle, and W. Jalby. Minimizing energy consumption of MPI programs in realistic environment. *CoRR*, abs/1502.06733, 2015. 155, 156, 158

[60] I. Gurobi Optimization. Gurobi optimizer reference manual, 2014. *http : //www.gurobi.com*. 141, 148, 160

[61] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Drpm: dynamic speed control for power management in server class disks. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 169–179, June 2003. 14

[62] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. 162

[63] J.-P. Halimi, B. Pradelle, A. Guermouche, and W. Jalby. Forest-mn: Runtime dvfs beyond communication slack. In *Green Computing Conference (IGCC), 2014 International*, pages 1–6. IEEE, 2014. 132, 133

[64] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014. 2

[65] V. Hanumaiah and S. Vrudhula. Energy-efficient operation of multicore processors by dvfs, task migration, and active cooling. *Computers, IEEE Transactions on*, 63(2):349–360, Feb 2014. 58

[66] D. P. Helmbold, D. D. Long, T. L. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000. 14

[67] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 217–222, New York, NY, USA, 2003. ACM. 15

[68] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pages 217–222, Aug 2003. 23

[69] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, Oct 1994. 9, 35

[70] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a pc cluster. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 298–298, Washington, DC, USA, 2006. IEEE Computer Society. 58, 79, 95

[71] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb 2010. 40

[72] C.-H. Hsu and W. chun Feng. A power-aware run-time system for high-performance computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 1–1, Nov 2005. 79, 114

[73] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 1–, Washington, DC, USA, 2005. IEEE Computer Society. 9, 35, 79, 109, 121

[74] C.-H. Hsu, W.-c. Feng, and J. S. Archuleta. Towards efficient supercomputing: A quest for the right metric. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005. 9

[75] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *ACM SIGPLAN Notices*, volume 38, pages 38–48. ACM, 2003. 58

[76] W. Huang, M. Allen-Ware, J. B. Carter, E. Elnozahy, H. Hamann, T. Keller, C. Lefurgy, J. Li, K. Rajamani, and J. Rubio. TAPO: Thermal-Aware Power Optimization techniques for servers and data centers. *International Green Computing Conference and Workshops*, 0:1–8, 2011. 15, 16, 17

[77] A. Hylick, R. Sohan, A. Rice, and B. Jones. An analysis of hard drive energy consumption. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008. 13, 14

[78] M. Inc. Granola energy saving tool. *http* : *//grano.la*. 121

[79] Intel. Desktop 4th generation intel® core™processor family. *http* : *//www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol*-1-*datasheet.pdf*. 2, 38, 40

[80] Intel. Intel® 64 and ia-32 architectures software developer's manual. *http* : *//download.intel.com/products/processor/manual/*253669*.pdf*. 102

[81] Intel. Intel® core™ i7-800 and i5-700 desktop processor series. *http* : *//www.intel.fr/content/www/fr/fr/intelligent-systems/piketon/core-i*7-800-*i*5-700-*desktop-datasheet-vol*-1.*html*. 36

[82] Intel. Powertop. *https* : //01.*org/powertop/*. 38

[83] E. Intel. Speedstep® technology for the intel® pentium® m processor, 2004. 37

[84] Intel Corporation. Intel Xeon processor E5-1600/E5-2600/E5-4600 product families, May 2012. 73

[85] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 359–370, Dec 2006. 62, 79

[86] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitectur*, MICRO 39, pages 359–370. IEEE Computer Society, 2006. 69, 79, 95

[87] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop on*, pages 108–118, Oct 2003. 62, 95

[88] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 262. ACM, 2014. 58

[89] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. Vmad: an advanced dynamic program analysis and instrumentation framework. In *Compiler Construction*, pages 220–239. Springer, 2012. 68

[90] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming petascale applications with charm++ and ampi. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008. 133

[91] G. A. Kaminski and G. F. Squibb. Dual power supply fan control—thermistor input or software command from the processor, 01 2002. US 6349385B1. 16

[92] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33. IEEE Computer Society, 2005. 132

[93] B. Keeth. *DRAM circuit design: fundamental and high-speed topics*, volume 13. John Wiley & Sons, 2008. 12

[94] J. Kim, M. Ruggiero, and D. Atienza. Free cooling-aware dynamic power management for green datacenters. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 140–146. IEEE, 2012. 16

[95] J. Kim, S. Yoo, and C.-M. Kyung. Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(1):110–123, 2011. 58

[96] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134, Feb 2008. 38, 75, 79

[97] T. Kolpe. *Power Management in Multicore Processors through Clustered DVFS*. PhD thesis, UNIVERSITY OF MINNESOTA, 2010. 96

[98] W. R. Kundert. Fan speed controller, 02 1988. US 4722669A. 16

[99] O. R. N. Laboratory, october 2012. *https* : *//www.olcf.ornl.gov/titan/*. 11

[100] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8. USENIX Association, 2010. 2

[101] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *SIGPLAN Not.*, 35(11):105–116, Nov. 2000. 13

[102] Y. C. Lee and A. Y. Zomaya. On effective slack reclamation in task scheduling for energy reduction. *JIPS*, 5(4):175–186, 2009. 131

[103] A. Leite, C. Tadonki, C. Eisenbeis, and A. De Melo. A fine-grained approach for power consumption analysis and prediction. *Procedia Computer Science*, 29:2260–2271, 2014. 58

[104] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 2009. 70

[105] D. Li, D. S. Nikolopoulos, K. Cameron, B. R. De Supinski, and M. Schulz. Power-aware mpi task aggregation prediction for high-end computing systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010. 131, 132

[106] J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77–87, Feb 2006. 111

[107] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Software, IEE Proceedings-*, pages 14–23. IEEE, 2004. 132

[108] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar. Performance directed energy management for main memory and disks. *ACM SIGPLAN Notices*, 39(11):271–283, 2004. 12, 13, 14

[109] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC 2006 conference, proceedings of the ACM/IEEE*, pages 14–14. IEEE, 2006. 132

[110] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012. 147

[111] W. Liu, W. Du, J. Chen, W. Wang, and G. Zeng. Adaptive energy-efficient scheduling algorithm for parallel tasks on homogeneous clusters. *Journal of Network and Computer Applications*, 41:101–113, 2014. 131, 132

[112] Y. Liu, R. P. Dick, L. Shang, and H. Yang. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1526–1531, San Jose, CA, USA, 2007. EDA Consortium. 15, 16, 19, 23

[113] K. Ma, X. Li, M. Chen, and X. Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 449–460, June 2011. 79

[114] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 14–25, June 2003. 79

[115] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *ACM SIGARCH Computer Architecture News*, 31(2):14–27, 2003. 79

[116] K. Malkowski, P. Raghavan, M. Kandemir, and M. J. Irwin. Phase-aware adaptive hardware selection for power-efficient scientific computations. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 403–406. IEEE, 2007. 58

[117] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 37–48. IEEE Computer Society, 2012. 11, 13

[118] A. J. Martin, M. Nyström, and P. I. Pénzes. Et2: A metric for time and energy efficiency of computation. In *Power aware computing*, pages 293–315. Springer, 2002. 9

[119] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmer's view. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010. 40, 133

[120] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of cpu frequency transition latency. *Computer Science-Research and Development*, pages 1–9, 2013. 73, 74

[121] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979. 158

[122] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261–270, Sept 2009. 41

[123] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz. The implementation of a 2-core, multi-threaded itanium family processor. *Solid-State Circuits, IEEE Journal of*, 41(1):197–209, Jan 2006. 16

[124] J. Noudohouenou and W. Jalby. Using static analysis data for performance modeling and prediction. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 933–942. IEEE, 2014. 41

[125] J. Noudohouenou, V. Palomares, W. Jalby, D. C. Wong, D. J. Kuck, and J. C. Beyler. Simsys: A performance simulation framework. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM. 41, 119

[126] P. Nowoczynski, M. Vildibill, J. Cope, and P. Uppu. Minimizing micro-interruptions in high-performance computing, Nov. 13 2014. US Patent 20,140,337,557. 14

[127] OpenMP Architecture Review Board. OpenMP application program interface version 3.1, july 2011. *http : //www.openmp.org/mp-documents/OpenMP*3.1.*pdf*. 133

[128] E. Pakbaznia and M. Pedram. Minimizing data center cooling and server power costs. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 145–150. ACM, 2009. 15

[129] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006. 19, 106

[130] H.-J. Park and J. Wallace. Fan speed control of silicon based devices in low power mode to reduce platform power, 03 2012. US 8145926B2. 16

[131] F. Petrini, J. Moreira, J. Nieplocha, M. Seager, C. Stunkel, G. Thorson, P. Terry, and S. Varadarajan. What are the future trends in high-performance inter.connects for parallel computers? [panel 1]. In *High Performance Interconnects, 2004. Proceedings. 12th Annual IEEE Symposium on*, pages 3–3, Aug 2004. 1

[132] J.-M. Pierson and H. Casanova. On the utility of dvfs for power-aware job placement in clusters. In *Euro-Par 2011 Parallel Processing*, pages 255–266. Springer, 2011. 131, 132

[133] C. Piguet, C. Schuster, and J. L. Nagel. Optimizing architecture activity and logic depth for static and dynamic power reduction. In *Circuits and Systems, 2004. NEWCAS 2004. The 2nd Annual IEEE Northeast Workshop on*, pages 41–44, June 2004. 111, 112

[134] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 369–379, New York, NY, USA, 2014. ACM. 14

[135] B. Putigny, B. Goglin, and D. Barthou. Performance modeling for power consumption reduction on scc. In *4th Many-core Applications Research Community (MARC) Symposium*, page 21. 131

[136] B. Putigny12, B. Goglin12, and D. Barthou. Performance modeling for power consumption reduction on scc. 40

[137] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition, 2013. 133

[138] R. Rao and S. Vrudhula. Performance optimal processor throttling under thermal constraints. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 257–266, New York, NY, USA, 2007. ACM. 15

[139] R. Rao and S. Vrudhula. Performance optimal processor throttling under thermal constraints. In *Proceedings of the 2007 International Conference on*

Compilers, Architecture, and Synthesis for Embedded Systems, CASES '07, pages 257–266, New York, NY, USA, 2007. ACM. 23

[140] F. Real, M. Trumm, V. Vallet, B. Schimmelpfennig, M. Masella, and J.-P. Flament. Quantum Chemical and Molecular Dynamics Study of the Coordination of Th(IV) in Aqueous Solvent. *J. Phys. Chem. B*, 114(48):15913–15924, 2010. 121

[141] N. B. Rizvandi, J. Taheri, A. Y. Zomaya, and Y. C. Lee. Linear combinations of dvfs-enabled processor frequencies to modify the energy-aware scheduling algorithms. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 388–397. IEEE, 2010. 131

[142] G. Roeck. Experimental linux driver for NCT6775 chips., 2012. *https : //github.com/groeck/nct*6775. 29

[143] B. Rountree, D. Lowenthal, S. Funk, V. W. Freeh, B. De Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–9, Nov 2007. 95, 133, 149, 151, 160, 164

[144] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 49. ACM, 2007. 81, 107

[145] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM. 95, 132

[146] O. Sarood, A. Gupta, and L. V. Kalé. Cloud friendly load balancing for hpc applications: Preliminary work. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 200–205. IEEE, 2012. 15

[147] O. Sarood and L. V. Kale. A 'cool' load balancer for parallel applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 21:1–21:11, New York, NY, USA, 2011. ACM. 15

[148] O. Sarood, E. Meneses, and L. V. Kale. A 'cool'way of improving the reliability of hpc machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12. IEEE, 2013. 15

[149] Schlumberger. Reverse time migration, 2014. 72

[150] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002. 147

[151] R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced cpu clock speeds: an analysis of current x86 64 processors. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, pages 9–9. USENIX Association, 2012. 109

[152] Seagate. Seagate solid state hard drives, 2014. *http* : *//www.seagate.com/www-content/product-content/ssd-fam/*1200-*ssd/en-us/docs/*1200-*ssd-ds*1781-4-1310*us.pdf*. 57

[153] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 356–367, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 109

[154] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *SIGARCH Comput. Archit. News*, 31(2):336–349, May 2003. 69

[155] D. Shin, J. Kim, N. Chang, J. Choi, S. W. Chung, and E.-Y. Chung. Energy-optimal dynamic thermal management for green computing. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 652–657, New York, NY, USA, 2009. ACM. 15

[156] N. R. Software. Numerical recipes. 58

[157] SPEC. Standard performance evaluation corporation, 2006. SPEC CPU benchmark suite. 57, 58, 90, 101

[158] V. Spiliopoulos, A. Sembrant, and S. Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 241–250. IEEE, 2012. 95

[159] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 276–, Washington, DC, USA, 2004. IEEE Computer Society. 29

[160] M. Talbar, F.and Bordet and N. Petit. Codelet tunning infrastructure, 2014. *https* : *//code.google.com/p/codelet-tuning-infrastructure/*. 62

[161] top500, june 2015. *http* : *//www.top*500.*org/list/*2015/06/. 11, 131

[162] G. L. Tsafack Chetsa, G. Da Costa, L. Lefevre, J.-M. Pierson, O. Ariel, and B. Robert. Energy aware approach for hpc systems. In Emmanuel Jeannot and Julius Zilinskas, editor, *High-Performance Computing on Complex Environments*. John Wiley & Sons, June 2014. *https* : *//hal.inria.fr/hal*-00925310. 20, 35

[163] P. K. Uppu, J. M. Cope, P. Nowoczynski, and M. Piszczek. Method and system for data transfer between compute clusters and file system, Nov. 27 2014. US Patent 20,140,351,300. 14, 147

[164] C. Valensi. *A generic approach to the definition of low-level components for multi-architecture binary analysis*. PhD thesis, Versailles-St Quentin en Yvelines, 2014. 175

[165] L. Wang, G. von Laszewski, J. Dayal, and F. Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In

Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, pages 368–377, May 2010. 20, 35, 111

[166] Z. Wang, C. Bash, N. Tolia, M. Marwah, X. Zhu, and P. Ranganathan. Optimal fan speed control for thermal management of servers. *Proceedings of the ASME Conference*, 2009:709–719, 2009. 15, 17, 18, 34

[167] A. Weissel and F. Bellosa. Self-learning hard disk power management for mobile devices. In *Proceedings of the Second International Workshop on Software Support for Portable Storage (IWSSPS 2006)(Seoul, Korea*, pages 33–40, 2006. 14

[168] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282. IEEE Computer Society, 2005. 58

[169] T. Yuki and S. Rajopadhye. Folklore confirmed: Compiling for speed= compiling for energy. In *Languages and Compilers for Parallel Computing*, pages 169–184. Springer, 2014. 51