System Profiling and Green Capabilities for Large Scale and Distributed Infrastructures

Ghislain Landry Tsafack Chetsa

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

T H È S E

en vue d'obtenir le grade de

Docteur de l'Université de Lyon, délivré par École Normale Supérieure de Lyon

Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques (ED 512)

Présentée et soutenue publiquement le 03 décembre 2013 par

Ghislain Landry TSAFACK CHETSA

System Profiling and Green Capabilities for Large Scale and Distributed Infrastructures

Directeurs de thèse :	M.	Laurent	LEFÈVRE	
	M.	Jean-Marc	PIERSON	
Après avis de :	M.	Nouredine	MELAB	
	M.	Domenico	TALIA	
Devant la commission d'examen formée de :				
	M.	Nouredine	MELAB	Rapporteur
	M.	Domenico	TALIA	Rapporteur
	M.	Thomas	LUDWIG	
	M.	Noel	DE PALMA	
	M.	Frédéric	DESPREZ	
	M.	Laurent	LEFÈVRE	Directeur
	M.	Jean-Marc	PIERSON	Directeur
	Mme.	Patricia	STOLF	

Acknowledgments

First and foremost, I would like to express my deep gratitude to my supervisors, INRIA permanent researcher Dr. Laurent Lefévre, Pr. Jean-Marc Pierson, and Dr. Patricia Stolf, for their guidance and unmeasurable support throughout my candidature. Over the past three years, I had the privilege to work on simulating topics and cutting edge issues in computer science. I was given the opportunity to meet and collaborate with extraordinary people all around the world. None of this would have been possible without the determination of my supervisors.

I am immensely grateful to my parents including my mum's co-spouse for their unconditional love and endless support in whatever I chose to do, for always watching my back wherever I am, and for all the honesty and ethic lessons they have provided me with. I am particularly proud of my mum Téclaire, who has been and will always be a great mother to me. Thanks also to my brothers (Ervice, Francis, Darlin, Anderson, Rostand, Modeste, and Luther) and sisters (Annie, Bertine, Judithe, Leonie, Mariane, and Aurelie) for their unconditional love, support, and companionship.

I would also like to thank all past members of the RESO project team, members of the Avalon project team, members of the SEPIA project team at IRIT, and the management staff of the LIP laboratory. In particular, I thank Christian Pérez, Laetitia Lecot, Evelyne Blesle, Sylvie Boyer, Marie Bozo, Catherine Desplanches, and Damien Séon for their support and help; Olivier Glück who has led me into this amazing teaching experience; Jean-Patrick Gelas who trusted me in the first place and introduced me to Laurent with whom he co-supervised my Master 1 & 2 projects.

I am grateful to Pr. Nouredine Melab and Pr. Domenico Talia, who have accepted to review my thesis, for their valuable comments on a preliminary version of this thesis. My gratitude also goes to Pr. Thomas Ludwig, Pr. Noel De Palma, and INRIA research director Fredéric Desprez who have accepted to be part of my thesis jury.

A special thank you goes to Dr. Georges Da Costa, Mr. Olivier Mornard, and Dr. Marcos De Assuncao for their valuable time and the many useful discussions. The support they have provided me with throughout this process has been exceptional. I also thank all my friends and everybody who have contributed to this research in any form.

Many thanks to all those who have honoured me with their presence on the big day. My gratitude also goes to all those who couldn't make it. I am thinking of my mum who would have like to witness that moment.

Finally, I also express gratitude to the French Institute for Research in Computer Science and Control (INRIA) and the Hemera or Grid'5000 project without which this research would have not been possible.

Abstract

Nowadays, reducing the energy consumption of large scale and distributed infrastructures has truly become a challenge for both industry and academia. This is corroborated by the many efforts aiming to reduce the energy consumption of those systems. Initiatives for reducing the energy consumption of large scale and distributed infrastructures can without loss of generality be broken into hardware and software initiatives.

Unlike their hardware counterpart, software solutions to the energy reduction problem in large scale and distributed infrastructures hardly result in real deployments. At the one hand, this can be justified by the fact that they are application oriented. At the other hand, their failure can be attributed to their complex nature which often requires vast technical knowledge behind proposed solutions and/or thorough understanding of applications at hand. This restricts their use to a limited number of experts, because users usually lack adequate skills. In addition, although subsystems including the memory are becoming more and more power hungry, current software energy reduction techniques fail to take them into account. This thesis proposes a methodology for reducing the energy consumption of large scale and distributed infrastructures. Broken into three steps known as (i) phase identification, (ii) phase characterization, and (iii) phase identification and system reconfiguration; our methodology abstracts away from any individual applications as it focuses on the infrastructure, which it analyses the runtime behaviour and takes reconfiguration decisions accordingly.

The proposed methodology is implemented and evaluated in high performance computing (HPC) clusters of varied sizes through a Multi-Resource Energy Efficient Framework (MREEF). MREEF implements the proposed energy reduction methodology so as to leave users with the choice of implementing their own system reconfiguration decisions depending on their needs. Experimental results show that our methodology reduces the energy consumption of the overall infrastructure of up to 24% with less than 7% performance degradation. By taking into account all subsystems, our experiments demonstrate that the energy reduction problem in large scale and distributed infrastructures can benefit from more than "the traditional" processor frequency scaling. Experiments in clusters of varied sizes demonstrate that MREEF and therefore our methodology can easily be extended to a large number of energy aware clusters. The extension of MREEF to virtualized environments like cloud shows that the proposed methodology goes beyond HPC systems and can be used in many other computing environments.

Résumé

De nos jours, réduire la consommation énergétique des infrastructures de calcul à grande échelle est devenu un véritable challenge aussi bien dans le monde académique qu'industriel. Ceci est justifié par les nombreux efforts visant à réduire la consommation énergétique de ceux-ci. Ces efforts peuvent, sans nuire à la généralité, être divisés en deux groupes : les approches matérielles et les approches logicielles.

Contrairement aux approches matérielles, les approches logicielles connaissent très peu de succès à cause de leur complexité. En effet, elles se focalisent sur les applications et requièrent souvent une très bonne compréhension des solutions proposées et/ou de l'application considérée. Ce fait restreint leur utilisation à un nombre limité d'experts puisqu'en général les utilisateurs n'ont pas les compétences nécessaires à leur implémentation. Aussi, les solutions actuelles en plus de leurs difficultés de déploiement ne prennent en compte que le processeur alors que les composants tels que la mémoire, le stockage et le réseau sont eux aussi de gros consommateurs d'énergie.

Cette thèse propose une méthodologie de réduction de la consommation énergétique des infrastructures de calcul à grande échelle. Elaborée en trois étapes : (i) détection de phases, (ii) caractérisation de phases détectées et (iii) identification de phases et reconfiguration du système ; elle s'abstrait de toute application en se focalisant sur l'infrastructure dont elle analyse le comportement au cours de son fonctionnement afin de prendre des décisions améliorant l'efficacité énergétique.

La méthodologie proposée est implémentée et évaluée sur des grappes de calcul à haute performance de tailles variées par le biais de MREEF (Multi-Resource Energy Efficient Framework). MREEF implémente la méthodologie de réduction énergétique de manière à permettre aux utilisateurs d'implémenter leurs propres mécanismes de reconfiguration du système en fonction des besoins. Les résultats expérimentaux montrent que la méthodologie proposée réduit la consommation énergétique de 24% pour seulement une perte de performance de moins de 7%. Ils montrent aussi que pour réduire la consommation énergétique des systèmes, on peut s'appuyer sur les sous-systèmes tels que les sous-systèmes de stockage et de communication. Nos validations montrent que notre méthodologie s'étend facilement à un grand nombre grappes de calcul sensibles à l'énergie (energy aware). L'extension de MREEF dans les environnements virtualisés tel que le cloud montre que la méthodologie proposée peut être utilisée dans beaucoup d'autres environnements de calcul.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Challenges of High Performance Computing

1.1.1 Need of raw performance

There is no single definition for High Performance Computing (HPC) or High Performance Cluster. Figure 1.1 offers an outline of HPC nowadays. At the one hand, from a user perspective HPC can be thought of as a set of services that enable new levels of innovation and insights for organisations that seek excellence in fields including Research and Development (R&D), science, engineering, among others. At the other hand, from a technological perspective, HPC is viewed as the use of clusters of servers and supercomputers, along with associated software, tools, interconnects, storage, and services involved in running an HPC environment or system[1]. Servers in an HPC environment are often called nodes.

The increasing reliance on computing by scientific endeavours, industry and government agencies (particularly the military) has made HPC mainstream in several areas including, but not limited to, climate research, disease control, homeland security, drug discovery. Organisations often rely upon HPC for enhancing their product line. For example, a bank uses HPC to analyse high volumes of digital transactions, maximising investments and protecting its client from frauds. Likewise, to bring superior products to market, a manufacturer may consider using HPC to built prototypes of its products. Similarly, organisations seeking for the creation of large, high fidelity models that yield accurate and detailed insight into the performance of their designs often rely upon HPC for performing extensive simulations. Along the same lines, architects use HPC to evaluate buildings/structures by simulating their

[1] Intersect360: http://www.intersect360.com

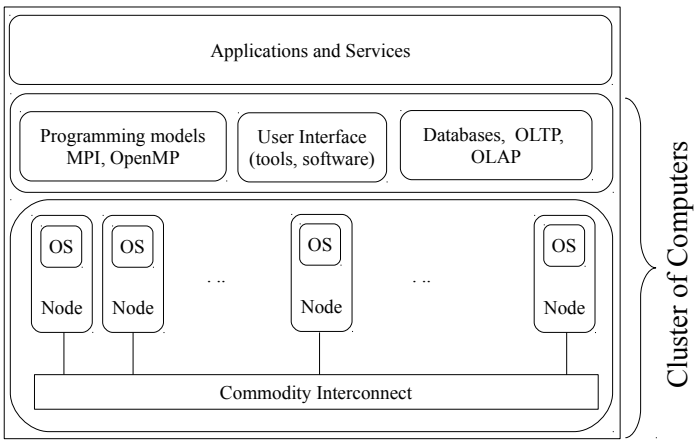


Figure 1.1: An overview of high performance computing (HPC).

prototypes in realistic scenarios. Simulating structures in multiple environments helps improve structural design to minimise damage and save lives under disasters. As the demand for processing grows, HPC will likely gain interest in businesses of all sizes, particularly for transaction processing and data warehouse.

1.1.2 Today's HPC systems

HPC inevitably owes its success to the massive computational power it is capable of achieving for solving complex problems. At the design level, to ensure that applications that run on HPC systems are reaching their maximum performance, system designers generally place a great emphasis on a handful of components. These include the processor architecture, memory subsystem, storage subsystem, communication subsystem, and the management framework. The emphasis on these components is justified by the fact that performance of the majority of HPC applications (use interchangeably with workload) relies upon them. For example, as the storage subsystem is an important factor for IO intensive applications, selecting a suitable storage subsystem for the application task can help enhance IO performance. A similar analysis can be waged by dimensioning each subsystem. In practice, although this offers reasonable performance over a wide range of applications, it often results in power dissipation/inefficiency for some workloads or specific phases of a workload. Unfortunately, addressing this at the system design stage is near to impossible; unless the designer knows all user applications that the future platform will accommodate.

Still at the design level, computer chips seem to have hit a wall, meaning that they can hardly be made any faster with current technology. Consequently, supercomputers designers just have to add more and more chips to increase computing power. However, this approach has a significant impact on energy usage. In light of what precedes, it is not surprising that, different from a decade ago where su-

percomputers were only ranked by their peak performance[2], nowadays, they are also assessed based on their energy efficiency[3]. The ranking of supercomputers by their energy efficiency places a great emphasis on their energy consumption through the number of PFlops (petaflops) they can achieve per Watt. For example, the Tianhe-2 machine, which sits on top of the performance list (Top500 list), delivers a computing power of over 33PFlops and shows an energy efficiency of 1.9GFlops/W; while CINECA which leads the green list (Green500 list) with an energy efficiency of 3.9GFlops/W delivers a computing power of less than 2PFlops.

Tremendous efforts are being undertaken by HPC operators from multiple levels to make supercomputers greener. This is evidenced by the Green500 list; its latest issue shows that the greenest supercomputers are getting greener. Their sudden improvement in energy efficiency can be attributed to the rise of graphic processors in massive cluster servers and the acquisition of low power memories. Similar efforts are being carried out in regard to the all HPC subsystems from the processor to the memory to the storage and communications subsystems. Unfortunately, at the current speed significant efforts still need to be done if today's supercomputers want to meet the 20MW constraint for exascale.

There is a common believe that a considerable share of energy consumed by HPC systems during their operations could be potentially saved if user applications were programmed differently. Put it in another way, throughout their life cycle, user applications exhibit behaviours whose understanding allows implementing power reduction schemes which can significantly reduce the amount of energy they consume at runtime. This has been proven right by different work [Kimura *et al.* 2010, Kappiah *et al.* 2005, Rountree *et al.* 2009, Lim *et al.* 2006, Choi *et al.* 2006, Ge *et al.* 2005].

Consequently, making HPC applications more energy friendly requires designing or rewriting applications with energy constraints in mind. These alternatives may not always be feasible. Although there is not any evidence, rewriting some HPC applications is so costly that most people find paying the electrical bill worth, whereas application developers usually do not pay much attention to how much energy their applications will consume. There are several reasons to this; besides the fact that they are already struggling to get their code work, current power saving schemes are platform specific. For example, let us consider the Dynamic Voltage and Frequency Scaling (DVFS) technology which allows scaling the processor's frequency according to the workload in some cases. Integrating DVFS into a program source code assumes that the developers know all the potential platforms that will run their applications which is not realistic. Although DVFS support is available in nearly all platforms today, at some point one need to select the appropriate frequency at which a specific must run. This can be very difficult to achieve at the coding stage since CPU frequency ranges are processor specific.

One could rely upon existing approaches such as those in the above references.

[2]Top500 List – June 2013: http://www.top500.org
[3]Green500 List – June 2013: http://www.green500.org

Unfortunately, they require expert knowledge and/or vast technical details behind the energy saving scheme proposed. As a result, it can be extremely difficult or near to impossible to implement in once HPC environment.

The energy consumption problem in HPC has been widely investigated over the past years. However, despite the fact that nearly all HPC subsystems are provided with energy saving capabilities, current efforts for reducing the energy consumption of HPC systems from a software perspective are directed toward the processor to the best of our knowledge. In other words, current efforts ignore all subcomponents save the processor.

With the current trend, to efficiently address the power consumption issue in HPC, things need to be approached differently than in the past. The processor has traditionally dominated supercomputers energy consumption, but the tendency is being reversed. In 2010, HPC subsystems including the memory, storage, and communications subsystems accounted for up to 55% of the total energy consumption of a typical supercomputer [Liu & Zhu 2010]. Thus, a fine-grained management of these subsystems can result in significant energy savings.

1.1.3 Problematic and objectives

High performance computing systems keep growing all around the globe increasing the power demand for operating them. Which in turn contributes to the carbon dioxide (CO_2) emission. In 2009, the International Telecommunication Union (ITU) has estimated the contribution of the Information and Communication Technology sector (excluding the broadcasting sector) to climate change at between 2% and 2.5% of total global carbon emissions [ITU 2009]. This thesis investigates means for reducing the energy consumption of large-scale and distributed infrastructures without a priori information about applications or services that share the infrastructure, while taking into consideration any energy reduction opportunities available to the most manageable components of the infrastructure. We concentrate on reducing the energy consumption without a priori knowledge of application and services while taking into consideration any energy reduction opportunities for several reasons including the following:

- Current, energy saving schemes fail to find their way into real HPC deployments because they often require thorough understanding of proposed scheme from a third party that would like to implement them on once system. In addition, they often rely on the assumption that the environment is a single task environment.
- Today's HPC applications are not optimized for saving energy; however, they are too complex to be rewritten or modified. This makes code instrumentation, which is also error prone, nearly impractical and significantly limits the scope of current energy saving schemes since they are application oriented.
- Although the current trend consists of adding more and more components to servers for increasing the computational power, energy saving schemes for

HPC have traditionally focused on the processor. As a misfortune, the power demand of servers increases as more components are added. This is illustrated by the increasing size of the memory subsystem. The same goes with storage and communication subsystems with the rise of Big Data. As mentioned earlier, these components account for more than half of the energy consumption of a typical HPC system. Moreover, in current mid-market and high-end servers, the memory subsystem already consumes more energy than the processor.

Our objectives in this thesis are to:

- Propose an energy reduction policy that concentrates on reducing the energy consumption of high performance computing systems instead of that of individual applications.
- Propose ways to address the energy consumption issue in HPC by taking advantage of all HPC subsystems. A basic requirement to this is developing power saving schemes that concentrate on reducing their energy.
- Propose an energy reduction policy that offers ways to benefit from variabilities among HPC workloads or within a specific workload while abstracting away from any individual applications.
- Propose a "user friendly" energy reduction policy for HPC systems. By user friendly, we mean an energy reduction policy that: (i) does not require any specific knowledge from a third party; (ii) takes into account the fact that real life environments are often shared by multiple applications; (iii) allows users to design and implement their own energy reduction strategies without extensive efforts.
- Provide a software framework for reducing the energy consumption of HPC systems that implements all features required to fulfil previously mentioned objectives.

1.2 Contributions

In this thesis, we are more interested in proposing a solution to the energy consumption problem in large-scale and distributed infrastructure. We introduce an automated, and scalable approach for reducing the energy consumption of HPC systems without a priori knowledge of workloads being executed. The approach takes advantage of HPC's workloads variability along with multi-configuration or reconfigurable hardware to reduce the energy consumption of the overall computing infrastructure. The main contributions of this thesis are as follows:

- We propose a methodology for reducing the energy consumption of HPC systems, it is original in the sense that it does not require any knowledge from users. Moreover, it takes into account all HPC subsystems –

from the processor to the memory to the storage and communication subsystems – and allows users to implement their own power saving schemes [Tsafack *et al.* 2012a, Tsafack *et al.* 2012c, Tsafack *et al.* 2013a].

- We propose on-line and off-line phase changes detection techniques [Tsafack *et al.* 2013b, Tsafack *et al.* 2012b]. Although allowing the detection of execution phases in specific program, our phase detection techniques innovate in the sense that they concentrate on detecting phases of execution of the system. System phase detection is similar to program phase detection, but offers the advantage that it abstract away from any individual program. As proposed techniques do not require any specific information about applications being executed, users lacking technical skills or expertise can use them for system analysis.

- Multiple workload characterisation schemes are proposed and evaluated in this thesis [Tsafack *et al.* 2013c]. Workload characterisation schemes serve the purpose of guiding system management decisions. Their particularity lies on the fact that they allow fast and accurate on-line characterisation of system phases as well as workloads.

- We propose and evaluate simple, but effective off-line and on-line techniques for identifying recurring phases in the runtime behaviour of a system [Tsafack *et al.* 2013c, Tsafack *et al.* 2013b]. Recurring phase identification basically enable reuse of reconfiguration information. To support system reconfiguration, we investigate the relevance of non conventional (not commonly used) power saving schemes including: memory size scaling, energy consumption prediction for platform selection, and core switch off/on.

- We present and demonstrate the effectiveness of a Multi-Resource Energy Efficient Framework (MREEF), an implementation of our methodology for reducing the energy consumption of high performance computing systems. MREEF is evaluated both in HPC and cloud environments. MREEF is energy oriented, so it emphasises on the use of green capabilities that we refer to as power saving schemes for system reconfiguration.

1.3 Structure of the Manuscript

The remaining of the manuscript is organised as follows: Chapter 2 reviews the state of the art on power/energy aware HPC. Our methodology for reducing the energy consumption of HPC systems is summarised in Chapter 3. Two phase detection methodologies that we refer to as *"power-based phase detection"* and *"EV-based phase detection"* methodologies are presented and evaluated in Chapter 4. Chapter 5 discusses several workloads characterisation schemes for on-line workload characterization. Chapter 6 discusses about system phase identification and power saving schemes. Chapter 7 evaluates our energy reduction methodology in HPC and cloud

environments through MREEF. Finally, Chapter 8 concludes our work in this thesis and presents future directions.

Chapter 2

Energy/Power Aware High Performance Computing: State of the Art

Contents

Over the past few years, reducing the energy consumption of High Performance Computing (HPC) systems, or making them energy efficient, has become of one of the biggest Information Telecommunication Technology (ITC) challenges. As a result, several research activities focusing on large-scale HPC have been initiated, such as understanding and/or modeling the energy consumption of HPC systems and applications to design efficient hardware and implement effective energy efficiency practices. This chapter reviews the state of the art on power/energy aware HPC. Section 2.1 discusses methodologies for modeling and predicting the energy consumption of HPC systems and applications. Section 2.2 surveys energy reduction techniques specific to HPC environments. Section 2.3 provides a comprehensive overview of phase detection techniques, whereas Section 2.4.1 discusses the need for having more sophisticated power saving schemes. Finally, Section 2.4.2 concludes the chapter.

2.1 Understanding HPC Systems' Power Consumption

Improving the energy efficiency of HPC can be viewed as a two-step process, where the first step may inevitably involve answering several questions, such as:

- How much energy does the system consume?
- What is the energy consumed by each system component?
- How can one measure the energy consumption of the overall system and its components?
- What is the ratio between active and static power consumption?

Answers to these questions can lead to valuable insights into the energy consumption of individual system components. This information can be a prerequisite for the second step of improving the energy efficiency of HPC, which consists of designing and evaluating energy aware architectures and algorithms (as discussed in Section 2.2).

From a technological point of view, an HPC system can be viewed as a combination of servers, clusters, supercomputers, and required software, tools, components, storage, and services, working in tandem to fulfil the intensive processing and storage requirements of scientific, engineering, or analytical applications[1]. Consequently, understanding how much energy an HPC system consumes boils down both to evaluating the energy consumption of massive numbers of servers and identifying how applications perform. In the scope of this thesis, an HPC system is a set of servers and application tasks they perform are referred to as HPC applications.

2.1.1 Components' power models

Wattmeters, often used to measure the energy consumption of servers, can be completely external devices inserted between the wall socket and the server plug, or be integrated into Power Distribution Units (PDU). Power modelling is a technique that gained popularity as power dissipation of the most power consuming components (*i.e.* processor and memory subsystems) within a server can be accurately estimated by simple power models. Static power models rely on the relationship between the supplied voltage and the electrical current traversing a component. They are used to describe both the static power or idle power consumption – the power consumed when there is no circuit activity – and the dynamic power of some server components. The following describes power models for processor and memory subsystems.

[1] http://www.intersect360.com

2.1.1.1 Processor

The power consumption of the processor can be broken down into static and dynamic power as described next.

(a) Static power consumption

Static consumption refers to the amount of power used due to leakage current in the absence of any switching activity. Direct Current (DC) power dissipation also known as static power can be estimated by the worse-case equivalent equation [Maede & Diffenderfer 2003]:

$$P_{static} = IV \tag{2.1}$$

where V is the supply voltage and I the direct current traversing the processor. Most modern processors have multiple cores, which means that processors consist of basically an integrated circuit to which more than one processor have been attached (in this case, a single processor is referred to as a CPU core or simply core). Consequently, the static power consumption of a multi-core processor can be expressed as the arithmetic sum of the static power of its cores; hence, Equation 2.1 can be rewritten as:

$$P_{static} = \sum_{k=1}^{n} I_k V_k = \sum_{k=1}^{n} P_k \tag{2.2}$$

where k is the number of cores, and V_k and I_k are respectively the supplied voltage and the DC current traversing the processor core k.

The power consumption of a core depends on its number of transistors; so in using Equation 2.1, the power consumption of the l^{th} transistor of the CPU core k can be estimated by Equation 2.3,

$$P_{lk} = I_{lk} V_{lk} \tag{2.3}$$

where I_{lk} and V_{lk} are the leakage current and power voltage supplied to the l^{th} transistor of the CPU core k. Substituting Equation 2.3 into Equation 2.2 yields Equation 2.4.

$$P_{static} = \sum_{k=1}^{n} \sum_{l=1}^{m} I_{lk} V_{lk} \tag{2.4}$$

(b) Dynamic power consumption

Dynamic power, the only mode of power dissipation in CMOS circuitry, represents a considerable share of the total power consumed by CMOS based processors. It is described by Equation 2.5 where C is the capacitance of switching nodes, V is the supply voltage, and f is the effective operating frequency (frequency times activity factor) [Weste & Eshraghian 1985].

$$P_{dyn} = CV^2 f \tag{2.5}$$

As for the static power consumption, the dynamic power consumption of an N core processor is described by Equation 2.6 where N is the number of CPU-cores; the remaining parameters are the same as in Equation 2.5.

$$P_{dyn} = \sum_{k=1}^{N} CV^2 f_k \tag{2.6}$$

However, Basmadjian *et al.* [Basmadjian & De Meer 2012] show that the power consumption of an n-core processor ($n > 1$) is not the exact arithmetic sum of the power consumption of all its n CPU cores as suggested by Equation 2.6. Consequently, to model the power consumption of a multi-core processor, they decompose it into three component levels, which they refer to as (i) chip, (ii) die, and (iii) core levels respectively, and model the power consumption of each. Their findings also reveals that Equation 2.6 overestimates the processor's power consumption quite often. They handle this by proposing a power consumption model that takes into account both resource-sharing and energy-saving mechanisms.

2.1.1.2 Memory

The power consumption of memory can also be broken down to total static power and total dynamic power.

(a) Static power consumption

The static power consumption of the memory is described by Equation 2.1. Back in 1919 it was shown that there is a linear relationship between the supplied voltage V and the DC current I when a device operates between 0 volt and 2 volts [Bijl 1919]. Since standard next-generation Dual Data Rate 3 (DDR3) memory technology and Dual Data Rate 2 (DDR2) technology operate within the above voltage range, the current traversing a memory is proportional to the supplied voltage as shown by Equation 2.7 where the constant c equals 0.00043 and 0.00013 for DDR3 and DDR2 respectively.

$$I = cV \tag{2.7}$$

The static power of a memory module of size s operating at frequency f can be described by Equation 2.8.

$$P(f, s) = cV^2 \tag{2.8}$$

To reflect the influence of the size and frequency of the memory module the static power consumption of a memory module can also be described by Equation 2.9

$$P = cfsV^2 \tag{2.9}$$

The memory or RAM of a typical server often comprises several memory modules; the static power of a memory with N modules is straightforward to determine. Equation 2.10 where f_k, s_k, and V_k are respectively the operating frequency, the

size, and the supply voltage of the k^{th} memory module, describes the static power of RAM composed of N memory modules.

$$P_{static} = \sum_{k=1}^{N} c f_k s_k V_k^2 \tag{2.10}$$

(b) Dynamic power consumption

The dynamic power of the memory subsystem mainly results from access (there is only 1 active operating rank per channel regardless of the number of modules or module ranks in the system[2]). The main memory is asynchronous in operation (modern servers requiring large amounts of memory use Dynamic Read Access Memory for main memory); thus, its power (actually the DRAM array power) is not dependent on memory frequency only, but on access count. As the DRAM array draws a constant amount of power regardless of the type of operation issued (write, read, precharge) the dynamic power of the memory can be described by Equation 2.11 where $\gamma \in [0, 1]$ is the probability that a memory access is performed (i.e., either the read, write or precharge command is active). β equals $7W$, $17W$, and $10W$ for unbuffered DDR2, fully buffered DDR2, and unbuffered DDR3 memory modules respectively.

γ reflects the utilisation of the memory and is expressed as the ratio of the used memory to the total memory.

$$P'_{dyn} = \gamma\beta \tag{2.11}$$

2.1.2 HPC applications' power: modelling and prediction approaches

Although modelling the power usage of individual applications has been widely investigated in mobile computing environments, it has not received extensive attention in HPC systems. This can be extremely difficult; especially, knowing that HPC applications are likely to spread over multiple nodes. However, it creates opportunities for energy based scheduling and power optimisation techniques [Bhattacharjee & Martonosi 2009, Merkel & Bellosa 2006, Singh *et al.* 2009]. Moreover, hardware for direct power measurement is largely nonexistent. In this section, we review power modelling and prediction techniques.

Power models have in common the fact that they monitor system components (in particular the processor and the memory) during the program's execution via hardware performance/monitoring counters (PCM) and correlate them with the power consumed by the system when running the program to derive its power model. Embedded hardware events counters of modern microprocessors, or simple hardware performance/monitoring counters are on-chip integrated facilities for counting events, so reading them can be done without any additional overhead.

[2] http://www.rampedia.com

Analytic processor power model based on performance counters are presented in [Isci & Martonosi 2003b, Joseph & Martonosi 2001]. The proposed models are highly accurate but only model the power consumption of the processor. Authors of [Kadayif *et al.* 2001] propose a model, which they claim to be 2.4% as compared to circuit level simulation, for estimating the energy consumption of the UltraSPARC memory hierarchy [SUN 1995]. They estimate the UltraSPARC CPU memory energy consumption considering PMCs providing the following information: Data cache read hits, Data cache read references, Data cache write hits, Data cache write references, Instructions cache references, Extended cache misses with write-backs. Energy consumption of the high-performance processor AMD Phenom is estimated in order to guide power aware policies in [Singh *et al.* 2009]. Authors use a set of micro-benchmarks that stress specific components of the processors architecture being modelled. They next categorise AMD Phenom PMCs into four buckets – FP Units, Memory, Stalls, and Instruction Retired – and consider performance events which express best their power consumption. These performance counters include L2_cache_miss:all, Retired_uops, Retired_mmx_and_ft_instruction:all, and Dispatch_stalls. These models often require knowledge of hardware component implementation.

The assumption of a linear relationship between the processor's power consumption and several hardware monitoring counters (instruction retired and translation look-aside buffer misses) has motivated the design of "black-box" microprocessor power models. They get their name from the fact that they do not require any knowledge of hardware component implementation in contrast to above power models. Authors of [Contreras 2005] present a first order linear model that uses hardware monitoring counters to estimate the run-time energy consumption on the Intel PXA255 [INTEL 2003] processor. They show that their model exhibits an average estimation error of 4%. Table 2.1, where the first column provides CPU related PMCs and the second provides PMCs related to the memory, offers an outline of performance monitoring counters that authors used in their power model. A scheme to associate energy usage pattern with every process for the purpose of thermal management in proposed in [Bellosa 2000]. The author correlates hardware monitoring counters to energy determinate the energy pattern of a thread. He then uses that energy information for energy-aware thread scheduling. One of the ideas behind this kind of scheduling is that by respecting the cache affinity of individual threads, and improving the cache reuse of individual threads that use share memory segment bus transactions and CPU stall cycles due to cache misses can be avoided. These models are simple and low-overhead; however, they do not model the whole system power consumption. In addition, they do not take into account all subsystem and may have portability related issues.

At the system level, [Economou *et al.* 2006] uses system components activity metrics such as CPU load and IO activity, and hardware monitoring counters to model the power consumption of a blade and an Itanium server. The proposed model captures power characteristics of system components by correlating hardware monitoring counters with power utilisation during the calibration phase. It

Table 2.1: Performance events selected to estimate CPU and memory power consumption for Intel PXA255 processor.

CPU performance events	Memory performance events
Instructions executed	Instruction Fetch Misses
Data dependencies	Data Dependencies
Instruction Cache Misses	
TLB Misses	

next uses parameters of the derived model for power consumption prediction based on the same PMCs. More recently, other researchers [Costa & Hlavacs 2010] have presented a methodology of measurement of the energy consumption of a single process application running on a standard PC. They defined a set of per process and system-wide variables to demonstrate their accuracy in measuring the energy consumption of a given process using multivariate regression. Authors of [Spiliopoulos *et al.* 2012] propose a profile based measurement infrastructure for measuring the power consumption of a program. The program goes through a first execution during which data required for performance and power prediction are collected. Still at the system level, some researchers attempt to take thermal issues into account when designing system-wide energy consumption models. For example, authors of [Lewis *et al.* 2012] propose a system-wide energy consumption models for server blades. Their energy model, which makes use of PCMs along with system ambient temperature, proposes a linear regression model that relates system energy input to subsystems' energy consumption.

To summarise, work listed above demonstrates that performance counters can accurately estimate or predict the power consumption of a program. The research also suggests that the accuracy of a power/energy model depends on the workload at hand. Put simply, a power model designed for estimating the power consumption of a compute-bound workload may not fit well a memory-bound workload. This is obvious for communication intensive workloads.

2.2 Energy Reduction in HPC Systems

In the past few years, HPC systems have witnessed the emergence of energy consumption reduction techniques from the hardware level to the software level. This section reviews power reduction techniques used in HPC environments.

2.2.1 Energy efficient hardware design

At the hardware level, architects and equipment vendors are bringing to market multi-configuration HPC subsystems – including processor, communications, memory and storage – that can be dynamically reconfigured to reduce the energy consumption of the overall HPC infrastructure while maintaining reasonable perfor-

mance. For example, the majority of modern processors is provided with Dynamic Voltage and Frequency Scaling (DVFS) technology, which allows “on the fly" adjustment of the processor's frequency and voltage either to conserve power or to reduce the amount of heat generated by the chip. Another emerging technology is the Low Power Idle (LPI) for Network Interconnection Cards (NICs).

The power consumption of a CMOS integrated circuit (such as a modern processor) can be described by Equation 2.12 where C is the capacitance of the feature gate, f is the operating frequency and V is the supply voltage. As the supply voltage is determined by the frequency at which the circuit is clocked, it can be reduced as the frequency decreases. Consequently, DVFS can significantly reduce the power consumption of a CMOS integrated circuit because its dynamic power consumption is proportional to the square of the supply voltage as shown in Equation 2.12.

$$P = CfV^2 + P_{static} \tag{2.12}$$

Alternatively, some architects address this issue by using the most efficient components for their equipments. This is illustrated by chip manufacturers who are bringing low voltage Dynamic Random Access Memory (DRAM) to market. Market leaders include Kingston “LoVo" (low voltage) HyperX DDR3 (DDR3 stands for dual data rate 3, similarly DDR2 stands for dual data rate 2), Micron's low voltage Aspen Memory, and Samsung Green DDR3. As opposed to standard next-generation DDR3 memory technology which operates at 1.5 volts and 1.8 volts for DDR2 memory, low power memory chips operate either at 1.25 volts or 1.35 volts. However, the impact of low power memory may not be noticeable because a large amount of servers around the globe uses chip memories which operate at 1.5 volts or higher.

Similarly, next-generation Solid State Drives (SSD) consume less power than traditional hard drives. For example, Intel's next-generation SSD DC 3700 Series reduces active power consumption to 6 watts and idle power to 650 milliwatts[3], which lowers power and cooling costs. While Samsung's Green SSDs drive approximately consumes on average 60% less power than traditional hard disk drives[4].

Power saving mechanisms are nonexistent in most of today's network interconnects. However, efforts are being undertaken to make interconnection networks greener. IEEE's task force on energy efficient Ethernet (IEEE 802.3az) examines power saving techniques such as dynamic link-speed reduction, and Low-power-idle (deep sleep states). In high bandwidth networks (100 Mbits/s and up), speed data links energy is used to keep the physical layer on all the time. That energy could be saved if they could be put in deep sleep when there is no data being transmitted [Merritt 2013]. Upon receiving the Low Power Idle signal the transmit chip in the system can be turned off when there is no data to sent and back on when needed. Stage-changes can take up to 10 microseconds, so HPC applications whose performance depends on the network are likely to suffer from performance degradation.

[3]Intel, http://newsroom.intel.com
[4]Samsung Green SSD, http://www.samsung.com

However, software mechanisms can first be used for profiling the behaviour of the application.

2.2.2 Software solutions to the energy consumption issue in HPC

Software initiatives for reducing the power consumption of HPC systems take advantage of the variability of HPC workloads (in terms of resource demand) to reduce the overall system's power consumption. The power consumption is basically reduced by dynamically reconfiguring the processor via DVFS when executing some workloads or specific phases of a workload. Speaking of phases, a program phase or simply a phase is a period of execution of a program throughout which the program is relatively stable w.r.t a given metric. Initiatives for reducing the energy consumption in HPC environments can roughly be divided into off-line and on-line approaches. They are alike since they both attempt to scale the processor's frequency down/up according to a program's phases. However, on-line approaches lack detailed knowledge of the program phases.

2.2.2.1 Off-line approaches

Off-line approaches necessitate human intervention and involve several steps including source code instrumentation for performance profiling; execution with profiling; determination of the appropriate processor frequency for each phase or region of execution of the program throughout which the program is relatively stable with respect to specific metrics; and source code instrumentation for inserting dynamic voltage and frequency scaling instructions. In [Freeh & Lowenthal 2005] the authors exploit MPI standard profiling interface (PMPI) to time Message Passing Interface (MPI) to insert DVFS scheduling calls based on duration, while other researches profile MPI communications [Cameron *et al.* 2005].

Similarly, authors of [Kimura *et al.* 2010] instrument the program source code to insert DVFS directives according to the program's behaviour. They divide the program into regions or phases and execute phases with low computational requirements at lower CPU frequencies. A DVFS control algorithm for sequential codes is presented in [Hsu & Kremer 2003], where the authors use compiler instrumentation to profile the program.

To reduce the energy consumption of HPC systems, some work suggested using inter-node imbalance analysis [Kappiah *et al.* 2005, Rountree *et al.* 2009]. In [Rountree *et al.* 2009] node imbalance is used to reduce the overall energy consumption of a parallel application. Authors track successive MPI communication calls to divide the application into tasks composed of a communication portion and a computation portion. A slack occurs when a processor is waiting for data during the execution of a task. This allows slowing the processor down with almost no impact on the overall execution time of the application. Consequently, authors developed *Adagio* [Rountree *et al.* 2009] which tracks task execution slacks and computes the appropriate frequency at which it should run. Although the first instance of a task

is always run at the highest frequency, further instances of the same task are executed at the frequency that was computed after it is first seen. [Kappiah *et al.* 2005] propose a tool called Jitter. Jitter detects slack moments in performance to performance inter-node imbalance and next uses DVFS to adjust the CPU frequency so that the processor does not have to wait for the completion of any task. In MPI programs, load imbalance often refers to situations where the completion of collective operations is delayed by a slower process or processes.

2.2.2.2 On-line approaches

The way a program's execution changes often falls into repeating behaviours also knows as Phases. As on-line methodologies for reducing energy consumption in HPC systems lack detailed information about programs' phases they usually take advantage of those repeating behaviours. Consequently, the effectiveness of such power saving schemes depends on the accuracy of the program phase changes detection mechanism. We provide a comprehensive overview of program phase detection techniques in Section 2.3.

In [Choi *et al.* 2006, Isci *et al.* 2006], authors use on-line techniques to detect program execution phases or simply program phases, characterise them and set the appropriate CPU frequency accordingly. They rely upon hardware monitoring counters to compute run-time statistics – including cache hit/miss ratio, memory access counts, and retired instructions counts – that they use for program phase changes detection and characterisation. However, policies developed in [Choi *et al.* 2006, Isci *et al.* 2006] tend to be designed for single task environment.

[Lim *et al.* 2006] looked at on-line recognition of communication phases in MPI applications. Authors apply CPU DVFS to save energy once a communication phase has been reached. Their CPU DVFS accomplishes this saving by intercepting and recording the sequence of MPI calls during program execution. During this time it considers a segment of program code to be reducible if there are high concentrated MPI calls or if an MPI call is long enough. The CPU is then set to run at the appropriate frequency when the reducible region is recognised again. Authors of [Ge *et al.* 2007] proposed a system-wide and application-independent DVFS scheduler which scales down the processor's frequency when slower processor frequency does not have a significant impact on performance. They assume the program's run-time is a succession of time intervals or phases, gather PMCs for each phase and use past history to select the appropriate CPU frequency for the next interval.

More recently, a slack time based model for reducing the energy consumption was presented in [Spiliopoulos *et al.* 2011]. Authors developed analytical DVFS models with attempt to reduce slacks in the program's execution, and therefore lower the power consumption. Relying on an analytical DVFS model, authors propose a run-time framework which breaks the program into fixed length interval, uses hardware monitoring counters for characterising individual program interval and optimises them for power reduction accordingly.

2.3 Focus on Program Phase Detection for Energy Efficient HPC

A phase change can be thought of as a sudden change in the program's behaviour. As presented in the previous section, on-line methodologies for reducing the energy consumption of high performance computing systems generally go through program phase changes detection.There is a large body of work dealing with program phase changes detection in the literature. The most popular approaches are based on *basic bloc vector*, *working set signature*, and *conditional branch counter*.

Authors of [Sherwood *et al.* 2003, Sherwood *et al.* 2001] and [Ratanaworabhan & Burtscher 2008] use Basic Bloc Vectors (BBVs) to detect program phase changes. A basic bloc vector is a list of all blocs entered during program execution, and a count of how many times each basic bloc was run. They keep track of basic bloc vectors at fixed interval and then use a similarity threshold to decide whether a phase change has occurred or not. As similarity criterion, they use the Manhattan distance between consecutive basic bloc vectors. The similarity actually tells how close BBVs are to each other. Entire BBVs cannot be stored in hardware, to overcome that limitation, authors suggested to approximate them by hashing into an accumulator table containing a few larger counters.

Phase changes detection using conditional branch counters is presented in [Balasubramonian *et al.* 2000]. Authors keep track of conditional branches executed over a fixed execution interval, and detect a phase change when the difference in branch counts between consecutive intervals exceeds a threshold which varies throughout the program's execution. As long as a program phase is a program execution period throughout which specific metrics are relatively stable, there can be many ways of detecting program phase changes. Authors of [Huang *et al.* 2003] propose to use subroutines as a program phase granularity. They rely upon hardware call stack for identifying major program subroutines and detect a program phase change by comparing the program's behaviour across different subroutines. Typically, they track the time spent in each subroutine and detect a major phase when the time spent in a subroutine is greater than a preset (fixed) threshold.

In [Dhodapkar & Smith 2002b] authors use program instruction working set to detect phase changes. They define a program phase as a set of instructions touched in a fixed interval of time and refer to that as an instruction working set. Similarly to BBVs, complete working sets can be too large to efficiently represent and compare in hardware. Authors handle this by using a loosy-compressed representation of working sets called working set signature [Dhodapkar & Smith 2002a, Dhodapkar & Smith 2002b]. Instruction working sets are compared for phase changes detection. To accomplish this, authors use the relative signature distance between consecutive working set intervals to detect phase changes when that relative signature distance exceeds a predefined threshold.

Other researchers use methods from signal processing for program phase detection. In [Casas *et al.* 2007], signal processing techniques are used to automatically

detect periodic phases in MPI programs. The approach works by analysing the correlation of message passing activity in the application. The phase detection approach proposed in [Casas *et al.* 2007] and that in [Fürlinger & Moore 2008] are very similar; however, they differ in that the latter identifies iterative phases in the application by directly analysing the control flow graph of the application. In [Wimmer *et al.* 2009] trace compilation is used for detecting program phase changes. The program's execution is a collection of trace trees. Speaking of trace trees, a trace tree is a collection of frequently executed code paths through a code regions. Assuming that the program execution remains within a trace tree during a stable phase, a phase change occurs when there is a sudden increase in side exits from the trace tree.

ScarPhase an execution history-base on-line library for detecting and classifying phases in serial and parallel applications was recently proposed in [Sembrant *et al.* 2012, Sembrant *et al.* 2011]. The library divides the program's execution into non-overlapping windows and samples conditional branches during the execution of each window using performance monitoring counters. The address of each branch instruction is next hashed into a conditional branch vector whose entries show how many times corresponding branches were sampled during the window. Program phases are next determined by clustering branch conditional vectors so that similar vectors belong to the same phase.

A power oriented phase detection mechanism has also been investigated. In [Isci & Martonosi 2003a] authors employed run-time power measurements and power estimated with performance counters to identify execution phases of a program. The methodology lies on the assumption that changes in the program's behaviour are also reflected in its power consumption behaviour.

2.4 Conclusions and Discussion

2.4.1 Discussion: need to address the energy consumption issue in HPC environments differently

As we have seen in the previous sections, the problem of energy consumption in high performance computing has been widely investigated. However, software solutions to that problem are less successful than their hardware counterpart. Current power reduction techniques as presented herein show that they are effective in the sense that they permit to reduce application's energy consumption without significant performance degradation. Unfortunately, to use these techniques, one would have to be an expert because of the complexity of their nature based on many different applications. Although intercepting MPI calls may be transparent, there is still the need to know what the application is doing in between those calls to set the appropriate frequency for example.

Power/energy reduction techniques presented above all have in common the fact that they attempt to reduce the energy consumption of the infrastructure from the application perspective, i.e., the focus is put on the application instead of the

infrastructure itself. This is a limiting factor for several reasons including the fact that one would have to possess wast technical details behind the energy reduction scheme proposed. Moreover, the HPC infrastructure operator may not be authorised to look into users' workloads for specific reasons. In summary, more accessible (easy to use), easy to scale, and automated energy saving schemes need to be designed. In other words, energy saving schemes cannot succeed in real-life environments, unless they can be implemented without extensive efforts.

The processor has long been considered the most power hungry hardware among HPC subsystems, recent statistics place the memory subsystem at the top of the list. According to Samsung, considering the 8 hours active and 16 hours idle status in server, the memory subsystem is responsible for 15% of the overall energy consumption of a 16GB, 60nm 1GB dual data rate 2 (DDR2) based server. The share of energy consumed by the memory is even more for 32GB and 48GB servers using the same process technology, accounts for 21% and 26% of their total energy consumption respectively. In the last case, the energy consumption of the memory exceeds that of the processor (20%). Consequently, power saving schemes must take the memory subsystem more seriously. In other words, energy saving schemes must provide means for reducing the power consumption of the memory. The same goes for other HPC subsystems including the storage and communication subsystems. In a few words, it must offer users the opportunity to design system specific power saving schemes i.e., power saving schemes of their own.

2.4.2 Conclusions

This chapter discussed approaches for making High Performance Computing (HPC) systems more energy efficient. Existing solutions are presented along with supporting mechanisms. Without lost of generality, these techniques include power and program analysis techniques.

Unlike hardware solutions, software approaches are often too complex and limited to the processors subsystem although other subsystems – memory, storage, and network interconnects – hold a considerable share in the overall power consumption of a typical supercomputer. By using the aforementioned steps to improve the efficiency of HPC systems the last step should emphasise on the design of user friendly solutions.

Chapter 3

A Blind Methodology for Improving Computing Infrastructures' Energy Performance

This introductory chapter summarizes our methodology for improving computing infrastructures' energy performance. We think of improving energy performance of a computing infrastructure as reducing its energy consumption without significant performance degradation. The term "significant performance degradation" being a relative term, it may be interpreted differently; however, a performance degradation of up to 10% is often acceptable. Note, unless expressly stated otherwise, we assume that (i) a high performance computing (HPC) system is a set of computing and storage nodes excluding network equipment such as routers and switches because of their nearly flat power consumption; (ii) whereas the term "system" designates a single node of the HPC system.

A typical HPC system throughout its life cycle exhibits several behaviours – in terms of utilization of available resources (processor, memory, storage, and communication subsystems) – reflecting phases of execution of a specific workload or workloads. Some of those phases or workloads are often similar in comparison with other workloads or regions of execution of a specific workload.

We saw in Chapter 2 that HPC equipment vendors are bringing multi-configuration or reconfigurable hardware to market in response to the energy reduction challenge in those environments. Relying upon those multi-configuration hardware, we propose a "blind" and general purpose methodology that leverages HPC workloads variability to reduce the energy consumption of the overall infrastructure [Tsafack *et al.* 2013c]. Our methodology breaks into multiple steps including: (i) *phase detection*, (ii) *phase characterization*, and *phase identification and reuse of configuration information*. It is labeled as blind because it does not require any information about workloads. Roughly speaking, users do not need any a priori information about workloads being executed.

Figure 3.1 offers an outline of the whole methodology on a system which successively runs five different workloads. For this specific case, each workload is detected as a phase or behaviour the system went through. As it can be seen, upon receiving a new execution vector or simply EV (the concept of execution vector is explained later on in Chapter 4), all the three steps are performed if necessary. Especially,

when the newest EV suggests a change in the behaviour of the system, the just completed phase is characterized; in any case, the newest EV is assigned to one of the classes represented by existing phases and reconfiguration decisions taken accordingly.

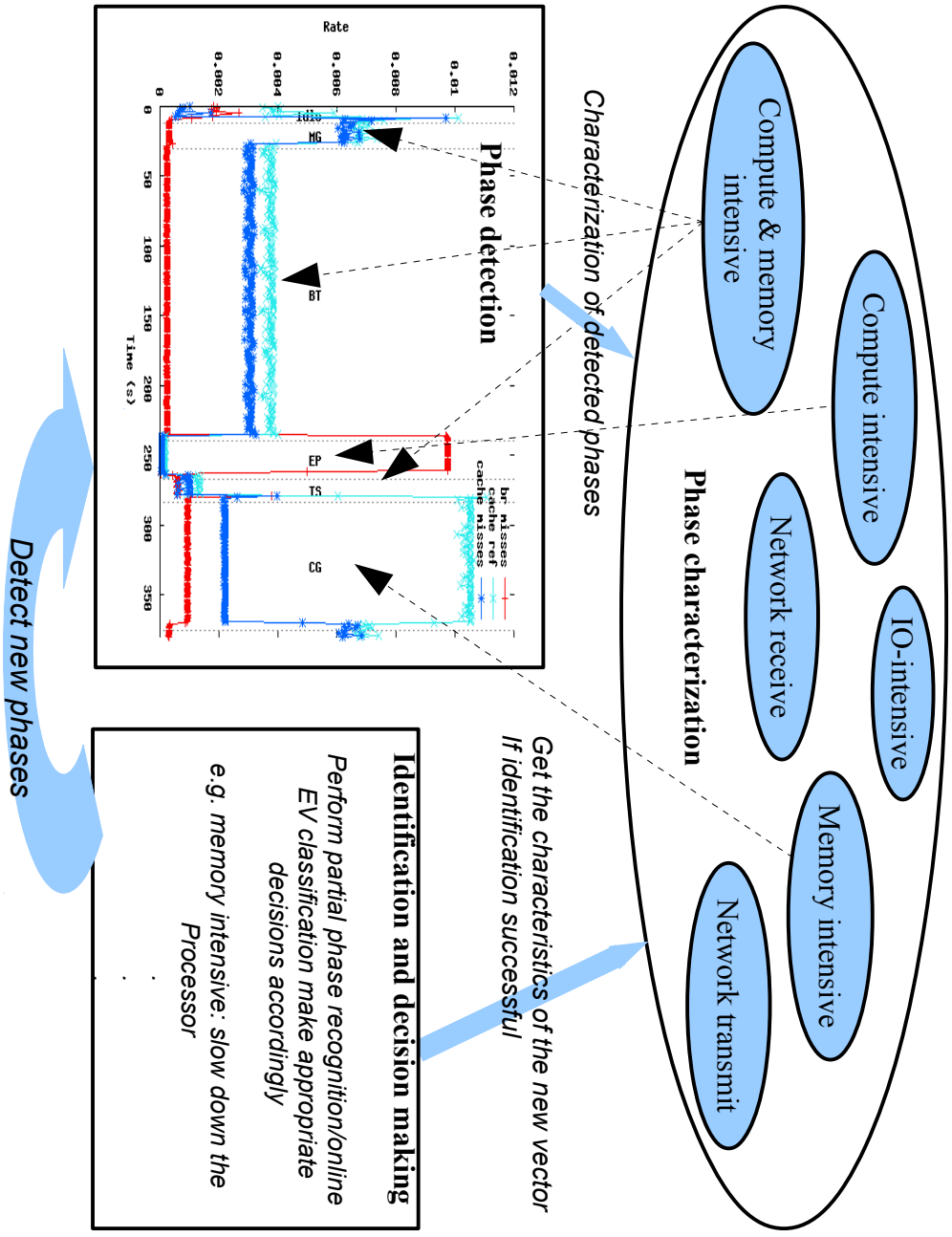


Figure 3.1: A summary of the methodology on a system which successively runs five different workloads.

Step 1: phase detection

The initial step, which we refer to as phase detection, is the process through which program/system phase changes are detected. A program phase is defined with respect to the stability of a specific metric or metrics. This stability is also translated in the program's performance, meaning that performance of the program is relatively stable throughout a phase of execution of the program as well. Phase detection techniques fall into on-line and off-line techniques.

As a rule, phase detection mechanisms attempt to detect program phase changes. This often requires thorough understanding of the program at hand. To abstract away from any particular program, we suggest detecting phase changes at the system level [Tsafack *et al.* 2013b]. The rational behind detecting phases of execution of the system or simply system phases is that changes in the programs are also reflected in the behaviour of the system through resource utilization. For example, when a program changes from a compute intensive/bound (we use the terms intensive or bound interchangeably) phase to a communication phase, this also results in changes in the utilization patterns of processor and communication subsystems. Further details in regard to our phase detection methodology are provided in Chapter 4

Step 2: phase characterization

In the presence of dynamically reconfigurable hardware, initiating system reconfiguration at the right time is as important as selecting hardware or software eligible for reconfiguration. Current, only the processor hardware is often reconfigured for energy savings purposes in HPC. As we brought up earlier, a system goes through different phases or behaviours throughout its life cycle, so initiating system reconfiguration at the boundary of a phase seems natural; however, reconfiguring "non eligible for reconfiguration" hardware can result in significant performance degradation.

Therefore, our phase characterization process, which is described in Chapter 5, aims to determine the type of reconfiguration decisions that are acceptable for a given workload or any specific phase of a workload.

Step 3: phase identification and system reconfiguration/adaptation

Phase identification is the ability to identify recurring phases, or more generally to identify phases with each other. It is a desirable property for phase detection techniques, since it can be used in tuning algorithms to reuse previously found optimal configurations for recurring phases.

Phase identification is often used in conjunction with phase prediction. The idea behind this is that by predicting the upcoming phase, it is possible (when the predicted phase is successfully identified with an existing phase) to set up an

optimal system configuration (adequate processor speed, memory size, storage space, and network bandwidth) for that phase before it gets started.

Reconfiguration decisions involve dynamically tuning available resources or subsystems to workloads' requirements, and often depend on the target objective (details are provided in Chapter 6).

Chapter 4

A Phase Detection Approach for HPC Systems' Analysis

Contents

4.1 Introduction

A large array of today's High Performance Computing (HPC) applications exhibits recurring behaviours or phases during their execution. Accurate detection of application or program phases allows for reconfiguring a system for better performance and for exploring energy trade-offs. As energy consumption becomes a limiting factor in the operation of HPC systems, detecting program phases can help devise schemes to reduce the energy consumption of HPC environments by enabling dynamic reconfiguration of available resources (e.g. processor, memory, storage, network interconnects) for specific phases of a workload or workloads. This can reduce the overall energy consumption while maintaining reasonable performance [Lim *et al.* 2006, Kimura *et al.* 2010, Balasubramonian *et al.* 2000, Huang *et al.* 2003].

Detecting phases often requires extensive knowledge of an application and/or the system architecture, or that the application be compiled with special instrumentation libraries. HPC users, however, rarely posses adequate knowledge for performing such tasks. HPC applications are generally complex, have been developed throughout several years and are often built exploring expert knowledge from multiple domains. Furthermore, HPC infrastructure commonly accommodates multiple workloads that are executed concurrently (i.e. the infrastructure is shared by multiple applications), in which case detecting phases of individual executions is nearly impossible. However, from a system stand point applications can be treated as a single workload when considering they might have uniform resource utilisation.

Application phases also result in different system behaviours or phases (e.g. system resource utilisation depends on how applications consume resources), which can be used as an alternative means of program phase detection for users lacking expertise or who are not aware of all intricacies of their applications. Similar to a program phase, a system phase is an execution region where the system behaviour is stable in comparison to other execution regions. More formally, a system phase is a continuous execution interval wherein measured system metrics are relatively stable. Detection of system-phase changes (hereafter also called system phase detection) exploits the fact that program phase changes are also reflected in different behaviours the system undergoes during its lifecycle. System phase detection abstracts away individual applications and reduces the burden of discovering the phases of individual applications.

This chapter describes the concept of system-phase detection and details two system-phase detection methodologies which we refer to as "*power-based phase detection*" and "*Execution Vector (EV) based phase detection*". The power-based methodology is off-line and detects phases using the system's power consumption footprint; whereas the EV-based approach is on-line and relies upon system's resource utilisation and explores concept of EV to detect phases.

The system-phase changes detection methodologies introduced in this chapter use similar principles to those employed in detecting program phases. Program phase detection methods are generally interval-based. In other words, during fixed length intervals (also known as sampling intervals), specific metrics are measured; values of those metrics between consecutive intervals are compared afterwards to determine whether a phase change has occurred. This means that phase detection methods detect changes in program behaviour that are assumed to result from phase changes.

This chapter is organised as follows. Section 4.2 details the power-based system phase detection approach and shows how it can be used for modelling an HPC system. Section 4.3 presents the EV-based phase detection approach and evaluates it using synthetic benchmarks. A case study where phases of a real-life application are detected using the EV-based phase detection methodology is presented in Section 4.4. Finally, Section 4.5 concludes the chapter.

4.2 Power-based Phase Detection

The power-based phase detection methodology which is also referred to as DNA-like system modelling attempts to describe a system (basically a single node of the HPC system) as a state graph whose final and initial states are configurations wherein it is idle. As idle goes, an idle system is defined as a system on which no user application is running. A transition between two states s_n and s_m of the graph is weighted by conditional probability that the system goes from s_n to s_m. Each state of the graph describes the system's behaviour over a fixed length time interval, and the sequence of successive states which the system undergoes throughout its life cycle is referred to as its "DNA-like" structure. Since each state of the graph describes a behaviour of the system over a given period of time, the DNA-like structure of a system can be thought of as the succession of behaviours through which it went over time. To remain in line with what precedes, we define the terms "letter" and "system description alphabet" respectively as follows: a letter or phase is defined as a behaviour in the DNA-like structure of a system; whereas the system description alphabet designates the set of possible behaviours.

Given the above, the run-time behaviour of a system can be described by a sequence of the form $L_i \dots X_j \dots L_k$ where the L_i are elements of the system description alphabet (details regarding their construction are presented in Section 4.2.1). Not all behaviours may be known, i.e., some states could not appear in the system description alphabet; the X_j notation is used for representing such states.

4.2.1 Phase tracking or letter modelling

Performance Monitoring Counters (PMC) have successfully been used for modelling the energy/power consumption of a wide range of applications (details can be found in Chapter 2). However, Chapter 2 also highlights that PMCs used in a model are strongly related to the type of application at hand. For example, PMCs used to model the power consumption of a CPU-intensive application often differ from those that are used to model the power consumption of a memory intensive application. Some power models may use more than PMCs for predicting or estimating the power consumption of an application, so without loss of generality, we will use the term "sensor" to designate either a PMC or any other system metric (network bytes sent/received and disk read/write counts for example) that can be used in a power model.

We assume that if a finite set of sensors is used to model the power consumption of a specific category of workloads, then a change in the set of sensors used for modelling the power consumption may suggest a change in the behaviour of the system (change of the type of workload or in the behaviour of the workload being executed). Based on this assumption, we propose an algorithm (Algorithm 1) for partitioning the run-time behaviour of a system into phases, so that the power consumption of the system over a phase is estimated using a unique set of sensors. In other words, a system-phase change occurs when there is a change in the set of

sensors relevant to power estimation.

Data: A: a set of units, where a unit is composed of values of sensors collected at a given time; units are sampled on a per second basis. Note that they are arranged in their order of occurrence in time.
Result: $P = \{t_i\}$ where t_i are points in time at which changes in the behaviour of the system were detected.
Initialization: remove k successive (starting from the first unit) units from A and put them in S; let us denote by t_S the time stamp of the last unit in S; k is chosen such that $k > p + 1$, where p is the number of sensors.
$P \leftarrow P \cup \{t'\}$, where t' is the point in time at which the first unit of S was sampled
Compute the set R_0 of sensors relevant to power consumption estimation using the dataset composed of units in S
while *units available in* A **do**
    Remove k more successive units from A starting at $t_S + 1$ and add them to those contained in S
    $t_S \leftarrow t_S + k$ (S upper bound is updated to $t_S + k$)
    Compute the set R_t of relevant sensors from S
    if $R_{t-1} \neq R_t$ **then**
        Find the point in time $j \in [t_S - k, t_S]$ such that the set of relevant sensors R computed from the set whose last unit was sampled at time j is the same as R_{t-1}
        Remove all units whose time stamp if less or equal to j from S
        Go to **Initialization**

Algorithm 1: Power-based system phase detection algorithm.

Finding out which sensors are relevant to power estimation of a given workload is not always straightforward. To accomplish this, we conduct multi-linear regression where we retain coefficients α_i and sensors C_i exhibiting a 5% (or higher) level of statistical significance to power consumption estimation given the power model described by Equation 4.1, where α_i and C_i are model coefficients and sensors respectively. For the sake of simplicity, the number of sensors relevant to power consumption estimation is limited to 4 (i.e. basically a letter or phase is described by 4 sensors).

$$Power \sim \sum_{k=1}^{n} \alpha_i C_i \tag{4.1}$$

Although phases or letters are represented using only four sensors, comparing them can be very costly if they are left that way. To handle that, once a phase is defined, we use the following formalism for its encoding: let us assign each sensor to a four-bit aggregation or half-byte. Our quadruplet is therefore of the form (b_1, b_2, b_3, b_4) where each b_i is half byte. Now deleting commas in between the b_i gives

a sixteen-bit aggregation, which converted into decimal is an unsigned integer. The unsigned integer obtained from the above transformation serves as the representation of a letter or phase.

4.2.2 Example

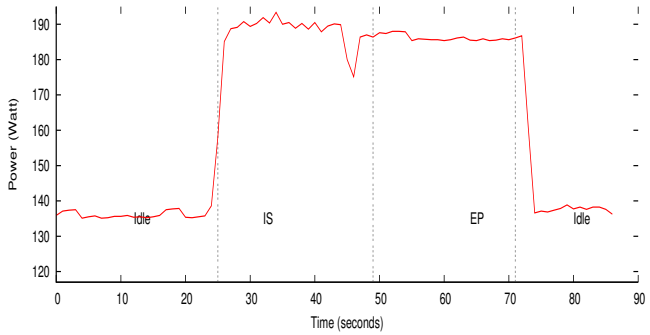
This example illustrates our power-based phase detection methodology on a system which successively runs the following from NAS Parallel Benchmark (NPB) suite [Bailey *et al.* 1991]: Integer Sort (IS) and Embarrassingly Parallel (EP). These applications are opposite from their computational stand point in the sense that EP is mainly computing while IS is not. Data collected during their execution serve as input to Algorithm 1. Figure 4.1 where doted vertical lines respectively indicate the beginning and the end of detected phases (Figure 4.1(a)), and the actual beginning and finishing time of each workload (Figure 4.1(b)); offers a graphical representation of Algorithm 1 on a system that successively runs the benchmarks we just mentioned. Figure 4.1 shows that it is possible to detect phases a system went through by exploiting its power consumption. We can also observe in Figure 4.1 that points in time at which phase changes are detected are slightly shifted from actual start times of the programs. This is normal because when a new program starts, the system needs some adaptation time prior to reflecting its actual power consumption.

4.3 EV-based System Phase Detection Mechanism

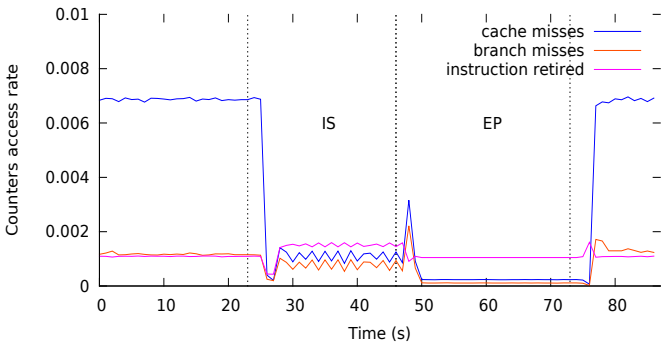
Despite its effectiveness, the power-based phase detection approach may not perform well when the device recording the power consumption lacks accuracy (the power consumption is nearly constant). This is because the linear regression often yields less interesting results when the predicted parameter, the power consumption in our case, is constant.

Figure 4.1(b), where the y-axis represents the access rate of PMCs or more generally sensors and the x-axis the execution time-line, suggests that changes in the system's behaviour are also reflected in the access pattern of sensors. As sensors go, the access rate of a sensor is the ratio of its raw value to the number of CPU cycles. Using sensor's access rates instead of their raw values prevents different dimensions from compensating one another.

We introduce here the concept of "execution vector" (EV), which seems adequate for system phase detection because of its similarity to power vectors in [Isci *et al.* 2006]. An execution vector is defined as a column vector in a 9-dimensional space (each EV has nine entries) and whose entries are access rates of system's metrics, including performance monitoring counters, network bytes sent/received, and disk read and write counts. Performance monitoring counters provide insight into the processor and memory activities, while network bytes sent/received, and disk read and write provide information about network and disk activities respectively.



(a) Graphical representation of the output of Algorithm 1 on a system that successively runs two benchmarks.



(b) PMCs access pattern throughout the execution of the two benchmarks.

Figure 4.1: Graphical representation of the output of Algorithm 1 along with PMCs access pattern.

Table 4.1: List of sensors describing an execution vector (we will use more human friendly names to refer to those sensors).

PERF_COUNT_HW_INSTRUCTIONS	
PERF_COUNT_HW_CACHE_MISSES	PERF_COUNT_HW_CACHE_REFERENCES
PERF_COUNT_HW_BRANCH_INSTRUCTIONS	PERF_COUNT_HW_BRANCH_MISSES
netSENTbyte	netRCVbyte
Write IO	Read IO

To avoid redundancy, only general purpose PMCs are considered as sensors, hence providing the following information: the number of retired instructions (can be thought of as the number of instructions that are actually executed and completed by the processor), last level cache references and misses, and branch instructions and misses. Table 4.1 offers an outline of sensors describing an execution vector (for further information about performance counters refer to the documentation available in Linux kernel source[1]).

4.3.1 EV-based phase changes detection algorithm

The EV-based phase detection approach works with the concept of execution vector we have just introduced. Unlike most phase detection mechanisms, the EV-based phase detection uses variable size intervals (i.e., all the phases do not necessary have the same length); however, EVs are sampled on a per second basis and the unweighted sliding-average smooth is applied to remove short-term fluctuations.

A system phase is an interval during which the system behaviour must be stable according to a given metric (or metrics). Let us consider a system that runs four workloads – including Lower-Upper symmetric Gauss-Seidel (LU), EP, Conjugate Gradient (CG), and IS from NPB benchmark suite – separated by random length idle periods. Figure 4.2 where the diagonal line from the upper left corner to the lower right represents the execution time-line offers a graphical representation of the matrix of distance (Manhattan) between execution vectors collected on the system we just described. At coordinate i, j in the upper matrix of Figure 4.2 the colour represents on a greyscale the distance between the EV_i and EV_j (EVs sampled at time i and j respectively). Hence, the colour at the point i, j tends to black when the distance between EV_i and EV_j tends to zero; conversely, the colour at the point i, j tends to white as the distance between EV_i and EV_j increases. Along the diagonal line, we can easily observe 7 triangular blocks representing either workload or idle periods. Actually, it reflects the order workloads are executed; that is, LU, idle, EP, idle, CG, idle, and IS. More interestingly, Figure 4.2, shows that distances between EVs within the same block tend to 0 (they are closer), while in between blocks the distance between EVs tends to 1 (white on a greyscale).

Based on the above observation, we define a similarity or resemblance criterion between EVs as the Manhattan distance between them. The Manhattan distance, which suits the case, is the distance between two points in an n-dimensional space if a grid-like path is followed and offers the advantage that it does not depend on

[1] https://www.kernel.org

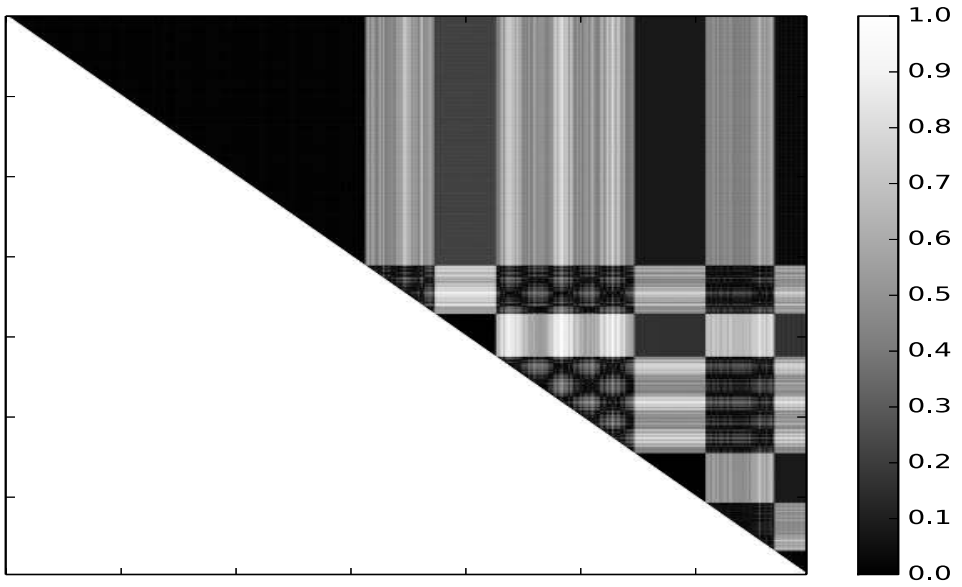


Figure 4.2: Graphical representation on a greyscale of the matrix of distance (Manhattan) between execution vectors; where the diagonal represents the execution time line. The darker a point of coordinate i and j is, the closer are EV_i and EV_j.

the translation of the coordinate system with respect to a coordinate axis, i.e., it weights more heavily differences in each dimension.

A phase change occurs when the Manhattan distance between consecutive EVs exceeds a preset (fixed) threshold. The threshold is fixed in the sense that it is always the same percentage – that percentage is referred to as the *detection threshold* – of the maximum distance between consecutive EVs. In other words, if the detection threshold is $X\%$, then the threshold is $X\%$ of the maximum distance between consecutive EVs. However, maximum distance between consecutive EVs is zeroed when a phase change is detected. Hence, the threshold varies throughout the system's lifecycle. In addition, the maximum existing distance between consecutive EVs is continuously updated until a phase change is detected, when it is then zeroed. Doing so allows detecting phase changes when moving from a phase where distances between consecutive EVs are big to a phase where they are not and vice-versa. Algorithm 2, which we refer to as EV-based Phase Detection Algorithm (EVPDA), offers an outline of the EV-based phase detection methodology. To summarise, for each newly sampled EV, the EVPDA computes the Manhattan distance between that vector and the previously (along the execution timeline) sampled EV and detects a phase change accordingly.

4.3.2 Illustrative scenarios and analysis

This section investigates the effectiveness of the EV-based phase detection methodology by detecting phase changes of a two-node cluster system running synthetic benchmarks. Synthetic benchmarks composed of several benchmarks – including

```
Initialization: max_distance = 0 ; phase_start = False
// threshold is a fixed percentage of the maximum existing
distance max_distance
while True do
    Compute EV_t: basically, sample a new execution vector
    Compute dist: the distance between EV_t and EV_{t-1}
    //update the maximum existing distance max_distance
    if max_distance ≤ dist then
        max_distance ← dist
    end
    if dist ≤ max_distance * threshold and phase_start is True then
        Start a new phase
        phase_start = False
    end
    else
        if dist > max_distance * threshold and phase_start is False then
            phase_start = True
            //reinitialize the maximum existing distance
            max_distance = 0
        end
    end
    t ← t + 1
end
```

Algorithm 2: EVPDA (EV-based Phase Detection Algorithm): an on-line algorithm for system phase changes detection.

Multi-Grid (MG), Block Tri-diagonal solve (BT), Embarrassingly Parallel (EP), Integer Sort (IS), and Conjugate Gradient (CG) from NPB-3.2 benchmark suite [Bailey *et al.* 1991] – only differ in that fixed length idle periods are inserted in between workloads in one of them.

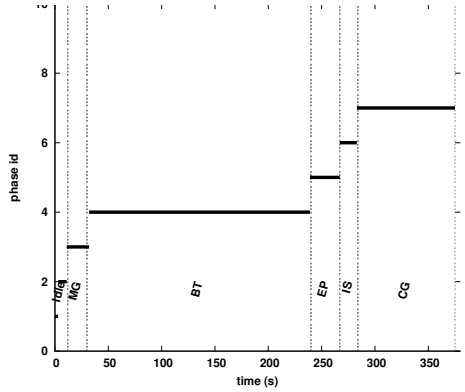
Benchmarks composing the synthetic benchmark have unique execution patterns; consequently, using them guarantees that the system will go through different behaviours. More importantly, assuming that the execution of a workload in the synthetic benchmark corresponds to an execution phase of the synthetic benchmark, we know in advance when phase changes occur. So doing allows us to tell how close to reality is the EV-based phase detection methodology. In the remaining of this chapter, an empirical evidence based *detection threshold* of 15% is used; thus, the threshold is 15% of the maximum existing distance between consecutive EVs.

We further consider two scenarios. In the first scenario, a synthetic benchmark referred to as *bench_1* is used. *bench_1* successively runs benchmarks listed above (from left to right starting with MG). The second scenario involves *bench_2*, which runs the same list of benchmarks as *bench_1* (in the same order). *bench_1* and *bench_2* are alike except that 30 second idle periods are inserted in between benchmarks in *bench_2*. Inserting idle periods in between workloads (*bench_2*) coerces the system to effectively go through different behaviours, while *bench_1* presents a more complex scenario where successive behaviours might not differ.

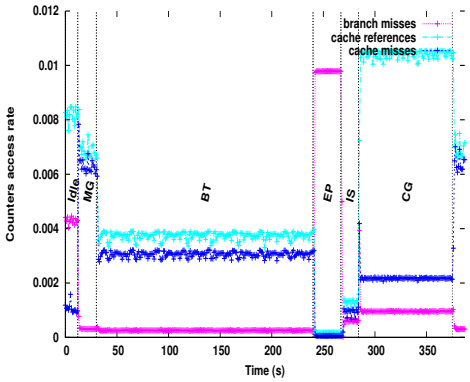
Figure 4.3(a) where dashed vertical lines indicate the beginning time and the end time of workloads in the synthetic benchmark, offers a graphical representation of the output of EVPDA when the system was running *bench_1*. The left end of horizontal solid lines indicates the point, in the execution time-line, at which phase changes are detected and their length indicate the duration or length of the corresponding phases. Note, the x-axis represents the execution time-line, while the y-axis represents IDs associated to detected phases (IDs are non zero integers ordered by their appearance order).

It can be seen from Figure 4.3(a) that all expected phase changes are successfully detected. Figure 4.3(c) which shows the variation of the distance between consecutive EVs along the execution time-line (x-axis) indicates that micro phases could have been detected when running BT if the threshold would have been different. This is easily achievable depending on the granularity at which one wants to detect phase changes. Indeed, the detection mechanism can use a tighter threshold to detect these regions. As for *bench_1*, Figure 4.4(a) and Figure 4.4(c) offer a graphical representation of the output of Algorithm 2 when the system was running *bench_2*. One can easily notice that the EV-based phase changes detection methodology is capable of differentiating periods wherein the system is loaded from those in which it is not (idle periods). Figure 4.3(b) and Figure 4.4(b) where the x-axis represents the access rate to sensors and the y-axis the execution time-line corroborates our phase-change detection. Indeed, it can be seen that phase changes results in different access pattern of sensors. Note, not all sensors are plotted for the sake of clarity. Figure 4.3(d) and Figure 4.4(d) offer a graphical representation of phases detected on the second node which is referred to as the slave node as opposite to the master

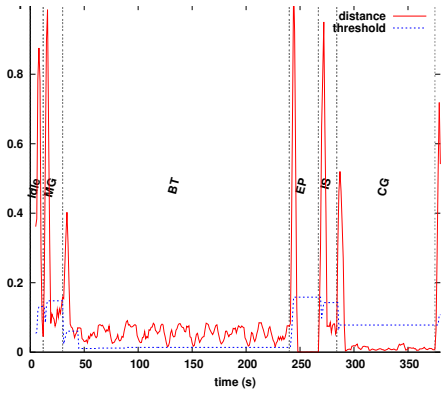
node.



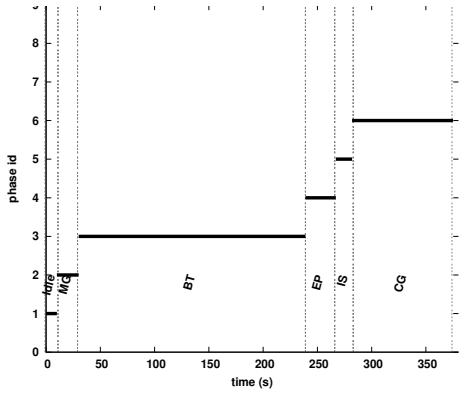
(a) Graphical view of system phase distributions (master node).



(b) Cache reference and miss rates along with branch miss rate (master node).

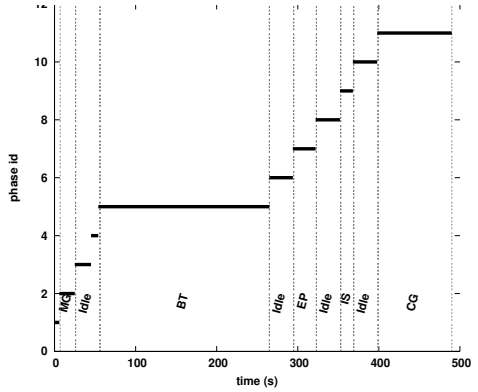


(c) Variations of the distance between EVs and the detection threshold (master node).

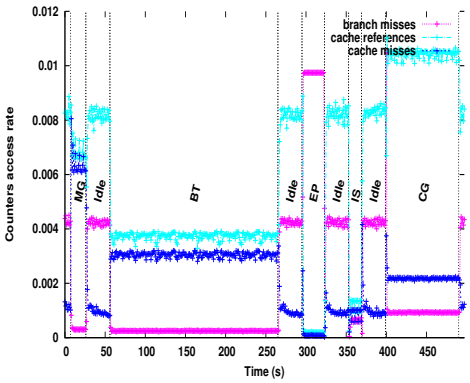


(d) Graphical view of system phase distributions (slave node).

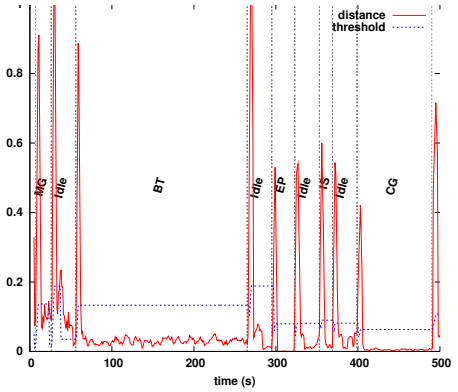
Figure 4.3: Phase changes detection using EVPDA when running *bench_1*; the threshold is 15% of the maximum existing distance between consecutive EVs.



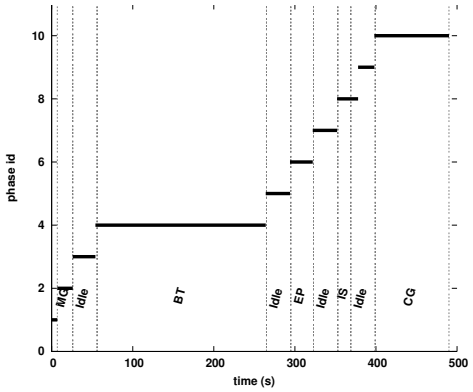
(a) Graphical view of system phase distributions (master node).



(b) Cache reference and miss rates along with branch miss rate (master node).



(c) Variations of the distance between EVs and the detection threshold (master node).



(d) Graphical view of system phase distributions (slave node).

Figure 4.4: Phase changes detection using EVPDA when running *bench_2*; the threshold is 15% of the maximum existing distance between consecutive EVs.

4.3.3 Evaluation of the EV-based phase detection algorithm: false positives, sensitivity, and mean detection time

Phase detection generally serves as starting point for power/performance optimisation algorithms [Kimura *et al.* 2010, Lim *et al.* 2006, Balasubramonian *et al.* 2000] and simulation (see Chapter 2 for further details). The simulation time of a program can significantly decrease given an effective identification of sections of code whose performance is representative of that program [Sherwood *et al.* 2003, Sherwood *et al.* 2001]. Consequently, it is essential that the phase detection mechanism detects phases that actually result in significant change in the program's or system's behaviour.

To evaluate the EV-based phase detection mechanism, we consider three metrics: (i) *sensitivity*, (ii) *number of false positive*, and (iii) *mean time to detection*. It is difficult to tell how significant a change in the system's behaviour has to be in order to be considered as a significant change. For the evaluation, we assume without loss of generality that a significant phase change in the behaviour of the system boils down to a change of workload. *bench_1* and *bench_2* are executed five times each to compute above listed evaluation metrics.

The sensitivity is defined as the ability of the phase detection mechanism to detect a change that results in significant change in the system's behaviour (or performance knowing that performance is relatively stable during a phase). For example, let us assume that the system has 100 significant behaviour changes. If the phase detection mechanism indicates 87 of these 100 behaviour changes, then its sensitivity is 87%. Similarly, if the detection mechanism detects all of the 100 significant behaviour changes, then it is said to be 100% sensitive. Note that the sensitivity will still be 100% if the phase detection mechanism indicates some other phase changes in addition to those expected.

Seeking a good sensitivity often leads to false positives. We define the number of false positives as the number of points in time where the system shows no significant behaviour change, but the phase detection mechanism indicates a phase change. The third and last evaluation metric which we refer to as the mean detection time is the average time that the phase detection mechanism takes to notice a significant phase change (only expected phase changes are taken into account). In other words, the mean detection time will be the average time the phase detection mechanism takes to notice a change of workload.

Figure 4.5 – where the steps of the drawn step function indicate detected phases and vertical lines delimit workloads – provides a graphical representation of the output of the EV-based phase detection algorithm (EVPDA - Algorithm 2) for five successive executions of *bench_2*. We can observe that the sensitivity of the detection mechanism for that workload is 100%. There is a handful of false positives; however, that is understandable since they occur during idle periods. Despite the assumption that the system has a stable behaviour during idle periods, there might be some system related tasks that are executed during those periods. And whose execution can potentially change the behaviour of the system throughout an idle

Table 4.2: Performance summary of our phase detection algorithm considering the two synthetic benchmarks.

Benchmark	sensitivity	false positives	mean detection time
Bench_1	100%	0	0.15 seconds
Bench_2	100%	4	0.5 seconds

period. Still from Figure 4.5, we can observe that recurring workloads approximately have the same length (duration) according to the phase detection mechanism.

Table 4.2 offers a summary of evaluation statistics for synthetic benchmarks *bench_1* and *bench_2*. We can observe that the mean detection time is less than one second (false positives are not taken into account). The mean detection time suggests that the latency of detecting a phase is on average less than one second. This comes from the fact that the phase detection software is contained within an independent thread which runs as any other application. The phase detection software nearly has no overhead since it boils down to reading a few sensors and computing the Manhattan distance between vectors in a 9-dimensional space (each execution vector has nine entries).

Modern processors are provided with on-chip facilities for counting events [Intel 1996, Welbon *et al.* 1994, MIPS 1996]. These facilities enable very fast access to all necessary register and allow reading and setting performance counters without any additional overhead.

4.3.4 Phase representation and selection of simulation points

Depending on its length, a phase can be too costly (storage space and computation time) to efficiently represent and compare in hardware. Consequently, each detected phase is summarised with three pieces of information: a representative vector, a reference vector, and the average distance from all vectors in a phase to the reference vector. That average distance can be used for classifying new execution vectors in existing phases. The reference vector of a phase is defined as the closest vector to the centroid of the group of EVs belonging to that phase and is used for phase identification. Finally, the representative vector of a phase is the EV resulting from the component-wise arithmetic average of all EVs belonging to the corresponding phase.

The latter (representative vector) can be used in conjunction with selected simulation points to reconstruct the original traces for simulation. We do not give much space to trace reconstruction because we are more interested in real systems; however, traces can easily be reconstructed by applying linear regression techniques provided an abstract model for the data is defined. As a simulation point goes, we consider as simulation point for a phase the start point (the point along the execution time-line at which the first EV occurs) of that phase. The literature suggests selecting simulation points earlier in the execution time-line in order to reduce the time to fast forward (executing the program without performing any cycle accurate simu-

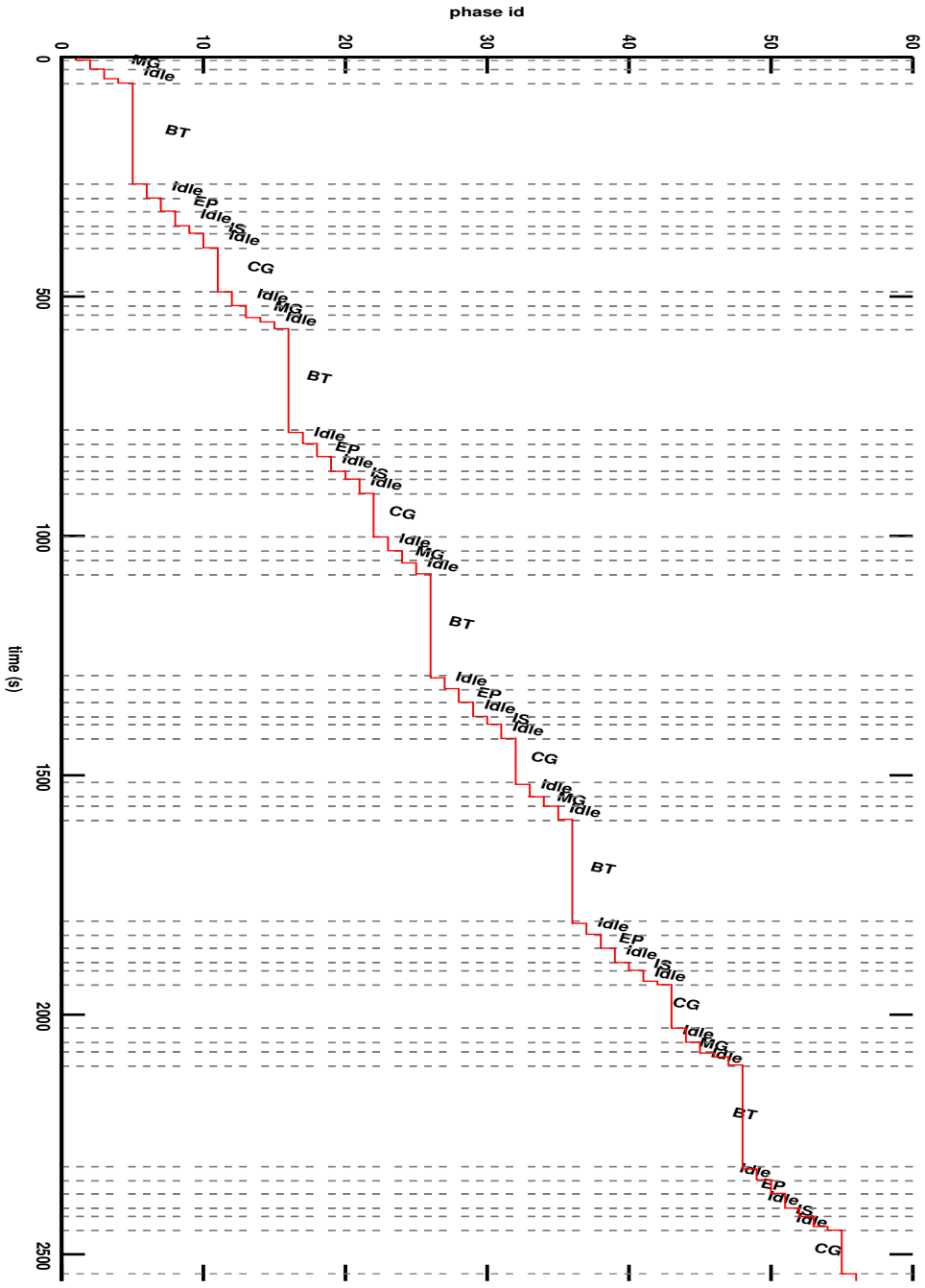


Figure 4.5: Graphical view of phase detected when successively running *bench_2* five times. Steps of the drawn step function indicate detected phases. The detection threshold is 15% of the maximum existing distance between consecutive EVs (master node).

lation) the selected simulation points [Sherwood *et al.* 2002, Perelman *et al.* 2003].

4.4 Case Study: The Advance Research Weather Research Forecasting (WRF-ARW) model

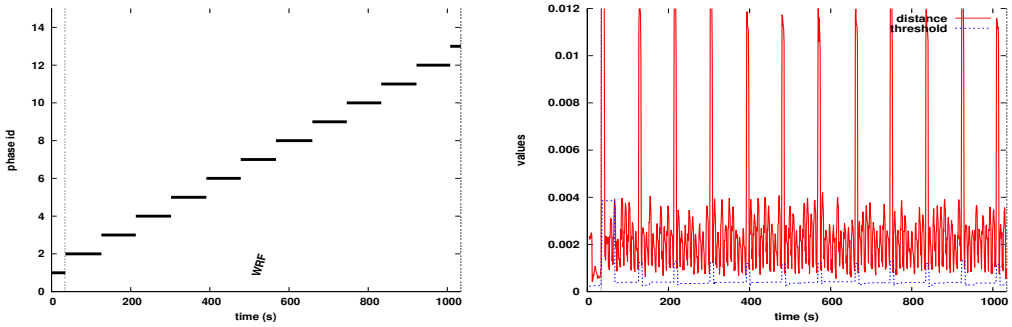
Previous sections demonstrate the effectiveness of the EV-based phase detection mechanism using synthetic benchmarks. In this section, we investigate its effectiveness using a real life workload that is representative of HPC applications: the Advance Research Weather Research and Forecasting (WRF-ARW) model [Skamarock *et al.* 2005]. WRF-ARW is a fully compressible conservative-form non-hydrostatic atmospheric model. It uses an explicit time-splitting integration technique to efficiently integrate the Euler equation.

4.4.1 Phase analysis and detection results

Figure 4.6 offers a graphical representation of system phases detected using EVPDA when running WRF-ARW. Figure 4.6(c) where the x-axis represents the execution time-line and the y-axis the access rate of a few sensors, shows without loss of generality WRF-ARW's resources utilisation pattern. In using the assumption that phase changes methods detect changes in program behaviour that result from phase changes, Figure 4.6(a) indicates that system phases detected by the EV-based phase detection algorithm actually correspond to phase changes in the runtime behaviour of WRF-ARW. Note in passing that in Figure 4.6(a) – where the y-axis represents ids of phases and the x-axis the execution timeline – dashed vertical lines indicate the beginning time and finishing time of the program, and the left end of horizontal solid lines indicates the point at which phase changes are detected. Variation of distances between consecutive EVs along the execution time-line is depicted in Figure 4.6(b).

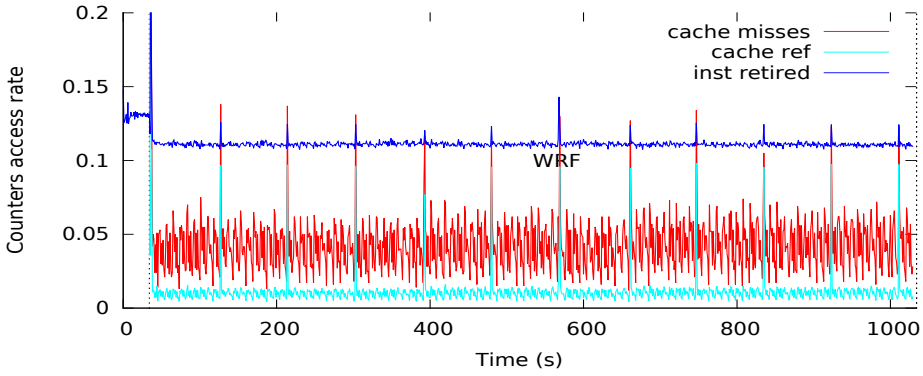
The corresponding distance matrix (Figure 4.7) corroborates the results of the EVPDA. It can be seen that along the execution time-line the colour at the points at which phase changes are detected tends to white, which is interpreted as a significant change in the behaviour of the system.

Overall, it can be seen that EVPDA performs as well with "home made" synthetic benchmarks as with a real life workload. The sensitivity is 100% and for this specific case there is no false positive. The mean detection time is still less than one second. However, phase changes detection may be influenced by the selected detection threshold. Figure 4.8 offers a graphical representation of the output of the phase detection algorithm using a 10% detection threshold. Note that the number of detected phases has slightly increased from 12 (Figure 4.6(a)) to 13 (Figure 4.8). The next section analyses the impact of the detection threshold on phase changes detection.



(a) Graphical representation of system phase distributions.

(b) Distance between EVs and variation of the detection threshold.



(c) Cache reference and miss rates along with branch miss rate.

Figure 4.6: Phase changes detection using the EVPDA (Algorithm 2) when running WRF-ARW; the detection threshold is 15% of the maximum existing distance between consecutive EVs.

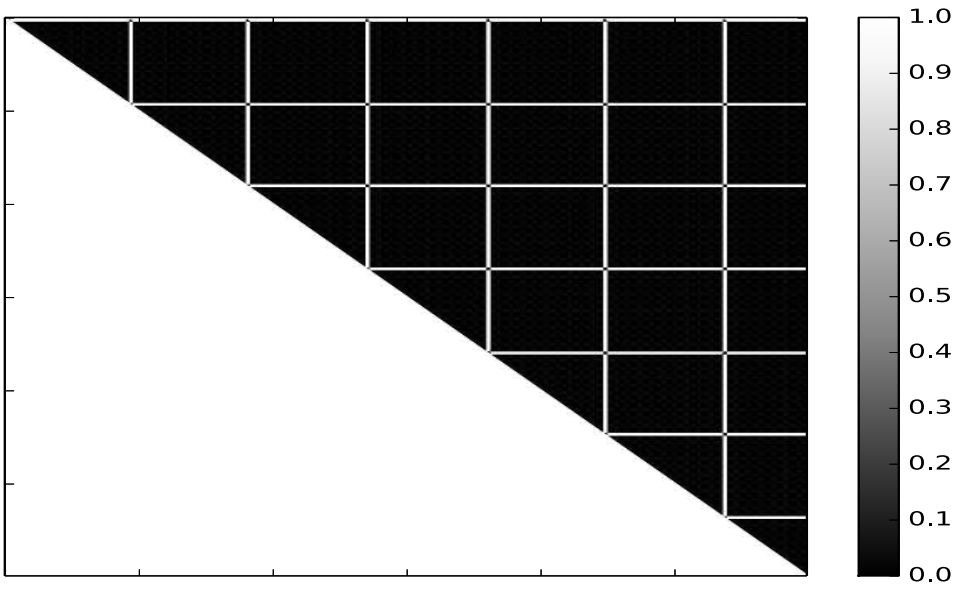


Figure 4.7: Matrix of distance between EVs for WRF-ARW (the matrix corresponds to half of the execution of the program).

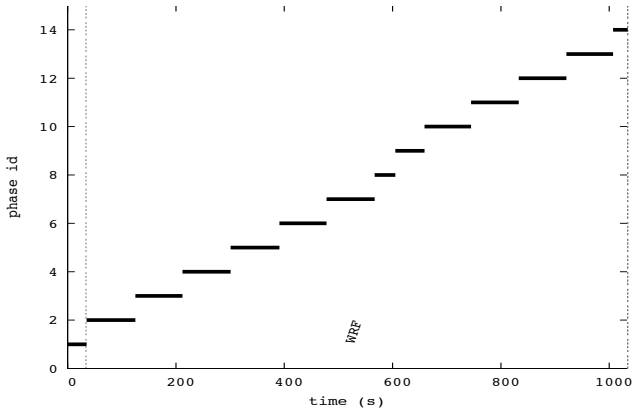


Figure 4.8: Phase changes detection using the EVPDA when running WRF-ARW; the detection threshold is 10% of the maximum existing distance between consecutive EVs.

4.4.2 Influence of the detection threshold

The selection of the detection threshold basically depends on the granularity at which one wants phases to be detected. Table 4.3 shows how the number of phases detected varies with respect to the detection threshold when the system is running WRF-ARW. The number of phases detected is one when the whole application is considered as a single phase. Table 4.3 suggests that when the detection threshold is too low, nearly no execution vector is similar to another. Likewise, when the detection threshold is too high, all execution vectors are similar to each other. There is no linear relationship between the detection threshold and the number of detected phases. That number mainly depends on the distances between consecutive execution vectors.

Table 4.3: Variation of the number of phases detected with respect to the detection threshold.

Detection threshold (%)	1	5	10	15	20	30	35	40	50
Number of phases detected	1	2	13	12	27	52	1	1	1

4.5 Conclusions

As shown in this chapter, system-phase detection is more user-friendly compared to detecting application phases. We proposed two methodologies to detect system phases. The power-based phase detection, which is off-line, uses the linear relationship between sensors and the power consumption to detect phase changes. It fails, however, to detect system phases when the power measurements lack accuracy and hence cannot be used for on-line system optimization.

The EV-based phase detection methodology leverages the fact program phase changes affect system behaviour by using resources; this fact is used to detect phases of the system instead of those of individual applications. System resource utilisation related information encapsulated in the concept of execution vectors is transmitted to the detection mechanism which computes their similarities for phase change detection. Its effectiveness is shown on scenarios using synthetic benchmarks and a real life application. Proposed evaluation metrics (number of false positive, sensitivity, mean detection time) reveal the ability of our phase detection mechanism to capture program phases including idle periods at the system level. A methodology for representing fixed runtime execution periods of the system with a small set of execution vectors and selecting simulation points is discussed.

As our phase detection methodologies do not require any information about applications being executed (user friendly) it can be used to address a wide range of problems, including the energy consumption problem by users lacking good understanding of their applications. We presented results for a two node cluster, but the methodology can be adapted to HPC systems as it applies to individual systems or nodes of the HPC system.

As we mentioned earlier, phase change detection allows system optimisation for energy saving purposes. We also brought up the fact that their complexity often limits their scope. In this chapter, we have introduced an easy to use and user friendly (in the sense that it does not require any specific knowledge from users) phase detection mechanism. We will show in the remaining chapters of this thesis how it can efficiently be used for reducing the energy consumption of HPC systems without any knowledge on the workloads being executed.

Chapter 5

System Phase Characterization

Contents

5.1 Introduction

Our three-step methodology (or roadmap) for improving the energy efficiency of High Performance Computing (HPC) systems comprises phase characterisation followed by phase identification and system reconfiguration. Phase characterisation serves as a pre-requisite for reusing optimal configuration information for recurring phases or workloads since it implicitly suggests reconfiguration decisions given a specific class or type of workload. As characterisation can use techniques similar to those used for phase-change detection, our characterisation methodology works with the concept of Execution Vectors (EVs).

Although gathering EVs throughout a phase is quite straightforward, extracting relevant information from the data can be difficult. Our phase characterisation process aims to provide insights into the computational behaviour of the system throughout a phase. It aims to group phases into labelled classes so that similar phases according to system resource utilisation appear under the same label. We consider processor, memory, disk and network interconnects as HPC resources or subsystems. A label has the particularity that it implicitly dictates the kind of reconfiguration decisions that are acceptable for the class of workload to which it refers. This is needed to prevent some reconfiguration decisions from hindering system's and workload's performances.

From a resource utilization point of view, HPC workloads commonly fall into the following categories: compute intensive, memory intensive, IO intensive, communication intensive, and any combinations of these. There is no uniform definition of those groups of workloads; nevertheless, the meaning we associate to each group can be found in Table 5.1.

We further divide communication intensive workloads into network transmit and network receive. The rationale behind dividing communication intensive workloads into network receive and transmit is that on most systems, receiving packets requires more processing than sending; thus, they can be treated differently. We define six types of labels inline with HPC workloads and according to system's resource utilization. These labels are: *"compute-intensive", "memory-intensive", "mixed", "IO-intensive", "network-transmit", and "network-received"*. They are self explanatory with the exception of "mixed". Workloads/phases labelled as mixed are both memory and compute intensive as they alternate between compute intensive and memory intensive behaviours. At one hand, they do not spend enough time being compute intensive to be labelled as compute-intensive and at another hand, they do not spend enough time being memory intensive to be considered as memory-intensive either. As labels dictate reconfiguration decisions, they also reflect the predominant behaviour of a phase and can be thought of as "basic" workloads classes or categories. Hence, a workload or phase can potentially combine two or more labels. However, for the sake of simplicity, the characterisation process makes compute-intensive, memory-intensive, and mixed labels mutually exclusive.

The effectiveness of system reconfiguration decisions widely depends on the accuracy of the phase characterization mechanism. Consequently, it is important that the characterisation process be carefully performed, for misleading decisions can result in significant performance degradation.

This chapter discusses our workload/system phase characterisation schemes and is organised as follows: Section 5.2 presents the Last Level Cache References per Instruction Ratio based (LLCRIR-based) phase characterisation, which suggests a scheme to classify workloads according to their cache sensitivity. Two system phase characterisation schemes using statistical techniques are presented in Section 5.3. A comparative analysis of two phase characterisation algorithms, which rely upon above mentioned characterisation schemes, is presented in Section 5.4. Finally, Section 5.5 presents concluding remarks.

5.2 LLCRIR-based Workload Characterization

Although there is no standard definition of cache sensitivity, here it refers to last-level cache references per instruction ratio. This section proposes a scheme to classify workloads according to their cache sensitivity.

Table 5.1: HPC workloads categories and their description.

Workload category	Description
Compute intensive	applies to any computer application that demands a lot of computation; their performance are often limited by the processor's speed.
memory intensive	used to refers to applications that require more shared memory than what is available on standard computers.
mixed	refers to any computer application that shares the characteristics of memory intensive and compute intensive. applications
communication intensive/ network receive	refers to any computer application that has relatively high network requirements for receiving large volumes of data from the network
communication intensive/ network transmit	refers to any computer application that has relatively high network requirements for sending large volumes of data over the network
IO intensive	refers to applications that read and/or write a large amount of data; performance of such applications depends on the speed of the peripheral device.

5.2.1 Workload characteristics and cache sensitivity

The amount of memory available on modern HPC servers has considerably increased over the past years. To find out whether an application can fully benefit from the memory available on the system (very useful when planning capacity for new systems), it is necessary to know its cache usage characteristics. To accomplish this, we execute benchmarks representative of HPC workloads and collect data that we use to compute statistics in regard to the last level cache utilization. The statistic of concern is the LLCRIR. Benchmarks we just mentioned – including Lower-Upper Guass-Seidel solver (LU), Block Tri-diagonal solve (BT), Conjugate Gradient (CG), Embarrassingly Parallel (EP), Integer Sort (IS), Unstructured Adaptive mesh (UA), Scalar Penta-diagonal solver (SP), and Multi-Grid (MG) from NAS Parallel benchmark suite [Bailey *et al.* 1991] – exercise different aspects of the system.

These benchmarks serve as reference workloads and are used to expose characteristics that workloads may have in common. In this particular case, we want to investigate their use of the last-level cache knowing that except for CG and EP, which are either extremely memory intensive (CG) or extremely compute intensive (EP); the aforementioned benchmarks are somewhere in between memory intensive and compute intensive. Table 5.2 where "standard dev." is the relative standard deviation of the mean LLCRIR (mean) offers an outline of statistics in regard to last level cache references per instruction ratio of workloads listed above. Note, we use the OpenMP version of the benchmarks on an Intel Quad-core Xeon E5506

Table 5.2: Per program average LLC references per instruction ratio (average over 15 runs of each program). The first column lists programs names and the first row represents class problem sets.

	A		B		C	
	mean	**standard dev.**	**mean**	**standard dev.**	**mean**	**standard dev.**
CG	1.30×10^{-2}	1.05×10^{-2}	7.79×10^{-2}	2.59×10^{-2}	8.93×10^{-2}	2.09×10^{-2}
IS	1.24×10^{-2}	1.14×10^{-2}	1.58×10^{-3}	1.43×10^{-3}	1.35×10^{-1}	9.02×10^{-2}
EP	$\mathbf{4.69 \times 10^{-4}}$	$\mathbf{9.27 \times 10^{-4}}$	$\mathbf{2.13 \times 10^{-4}}$	$\mathbf{4.26 \times 10^{-4}}$	$\mathbf{1.70 \times 10^{-4}}$	$\mathbf{2.82 \times 10^{-4}}$
BT	1.44×10^{-3}	3.45×10^{-4}	1.51×10^{-3}	2.72×10^{-4}	2.76×10^{-3}	5.24×10^{-4}
FT	4.77×10^{-3}	7.34×10^{-4}	9.63×10^{-3}	2.96×10^{-3}	-	-
MG	4.01×10^{-3}	1.04×10^{-3}	3.30×10^{-3}	8.51×10^{-4}	-	-
SP	3.78×10^{-3}	1.34×10^{-4}	4.03×10^{-3}	1.08×10^{-4}	4.42×10^{-3}	4.10×10^{-4}
UA	3.33×10^{-3}	1.16×10^{-4}	3.38×10^{-3}	1.25×10^{-4}	3.30×10^{-3}	5.58×10^{-4}

Table 5.3: Order of magnitude of LLC references per instruction ratio and associated labels.

Workload label	order of magnitude of LLCRIR
Compute intensive	$\leq 10^{-4}$
memory bound	$\geq 10^{-2}$
mixed (both memory compute intensive)	10^{-3}

CPU with 12GB of RAM (Random Access Memory), and a last-level cache size of 4MB. In Table 5.2, block letters A,B, and C in the headline refer to problem classes; each class englobes a problem size and its parameters. For classes A, B, and C the problem size increases going from one class to the next.

At a glance, Table 5.2 indicates that we can roughly group those workloads by the order of magnitude of their average LLCRIR regardless of the problem set (the order of magnitude of LLCRIR is the power of ten of the number that describes it). However, using that grouping, IS kernel does not always fall in the same group. This can be attributed to its irregular memory access patterns.

We group our workloads in three categories or classes according to the order of magnitude of their average LLCRIR. The first class, which we label as *compute-intensive*, includes EP. The *memory-intensive* class includes CG and IS. Finally, the *mixed* class is composed of FT, MG, SP, and UA. Table 5.3 summarises the relationship between workloads classes and cache sensitivity of workloads belonging to the corresponding classes. Labels assigned to a workload also reflect our knowledge of that workload.

The above analysis indicates that our cache sensitivity metric can successfully classify memory intensive, compute intensive workloads and mixed workloads under the appropriate label regardless of the program input (problem set).

Table 5.4: Per program average LLCRIR (average over 15 runs of each program) at different processor's frequency (class B problem set).

	1596 MHz (class B problem)		1862 MHz (class B problem)		2128 MHz (class B problem)	
	mean	**standard dev.**	**mean**	**standard dev.**	**mean**	**standard dev.**
CG	7.76×10^{-2}	2.48×10^{-2}	7.24×10^{-2}	2.77×10^{-2}	7.79×10^{-2}	2.59×10^{-2}
IS	1.52×10^{-3}	1.57×10^{-3}	1.52×10^{-3}	1.42×10^{-3}	1.58×10^{-3}	1.43×10^{-3}
EP	2.13×10^{-4}	4.70×10^{-4}	2.08×10^{-4}	4.75×10^{-4}	2.13×10^{-4}	4.26×10^{-4}
BT	1.41×10^{-3}	2.96×10^{-3}	1.44×10^{-3}	2.32×10^{-4}	1.51×10^{-3}	2.72×10^{-4}
FT	8.27×10^{-3}	2.63×10^{-3}	8.95×10^{-3}	2.73×10^{-3}	9.63×10^{-3}	2.96×10^{-3}
MG	2.30×10^{-3}	1.04×10^{-3}	2.82×10^{-3}	9.47×10^{-3}	3.30×10^{-3}	8.51×10^{-4}
SP	3.45×10^{-3}	2.11×10^{-4}	3.76×10^{-3}	2.23×10^{-4}	4.03×10^{-3}	1.08×10^{-4}
UA	2.73×10^{-3}	1.86×10^{-4}	3.06×10^{-3}	1.40×10^{-4}	3.38×10^{-3}	1.25×10^{-4}

5.2.2 Impact of input and system parameters of LLCRIR-based workload characterization

One of the most famous power saving schemes often referred to as Dynamic Voltage and Frequency Scaling (DVFS) consists of scaling the CPU frequency down/up according to workload requirements. Hence, one may be interested to know whether our workload characterization scheme does not suffer from CPU frequency changes; whether the characterization scheme guarantees that the class of a workload remains unchanged independently of the processor's frequency.

We showed in the previous section that our characterization does not change with the program's input (problem sets). To find out whether a label associated to a workload persists when the processor's frequency changes, we consider the Class B problem set of the benchmarks described in Section 5.2.1. Table 5.4 offers an outline of statistics regarding the LLCRIR for each of our benchmarks at different CPU frequencies. IS kernel is the only workload that does not remain in the class into which it was originally assigned (memory intensive).

We have just shown that the LLCRIR metric is a useful to determine whether a workload is either memory intensive, compute intensive or both memory and compute intensive regardless of programs' inputs and the processor's speed. Unfortunately, it does not tell how to determine whether a workload is either IO intensive or communication intensive. This being an on-line power oriented workload characterization, a detailed workload characterization might be too costly. For characterising IO intensive workloads, we use the percentage of CPU time during which IO requests were issued to any storage devices (bandwidth utilization for the device) as the IO sensitivity metric. That percentage increases as the load on the disk increases; typically, a value close to 100% indicates that the disk is fully loaded. Unless expressly mentioned, we assume that a workload is IO intensive when its disk utilization exceeds 50% (CPU time during which IO requests are issued). Although this allows determining whether a workload is IO intensive, it also brings an additional overhead; the overhead related to reading and processing storage statistics, which can be costly for an on-line characterization algorithm.

We do not characterize network or communication intensive workloads using the

LLCRIR approach, because they may seem to be mixed workloads while they are not; alternatively, we proceed by discrimination meaning that, if a workload does not fall into a known and characterized class, then it is probably network intensive. The LLCRIR-based system phase characterization excludes periods wherein the system is idle. However, idle periods must be properly addressed, since our methodology focuses on the system which is likely to experience idle periods from time to time throughout its lifecycle. Furthermore, some workloads may combine multiple behaviours (compute intensive and network intensive for example), which is difficult to identify using the LLCRIR-based workload characterization.

5.3 Statistical-based Phase Characterization

To overcome the limitations of the LLCRIR-based phase characterization methodology, this section introduces two phase characterization schemes. They all attempt to extract useful information from the data using statistical analysis techniques. More precisely, they use Principal Component Analysis (PCA) [Hastie *et al.* 2001]; however, they differ in the way PCA results are interpreted. PCA is mainly concerned with identifying correlations in the data. It is used for two objectives: (i) removing redundant variables in a dataset while retaining the variability in the data (can also be thought of as reducing the number of variables) and (ii) identifying patterns in the data and classifying them according to how much of the information stored in the data, they account for. In a dataset comprised of numerous dimensions (variables), it is likely that subsets of variables are highly correlated with each other. Highly correlated variables are in fact redundant because of the linear relationship that exists between them. In summary, PCA permits to identify the principal directions in which the data vary.

5.3.1 A low overhead phase characterization approach

Our low overhead phase characterization approach which we refer to as "sensors-based workload characterization" is very simple; it exploits the first objective of PCA. The main purpose is to find out what the predominant behaviour of a phase is by interpreting data collected throughout that phase. To this end, we apply PCA to the dataset made up with EVs sampled during the execution of the phase. We next select five variables (sensors in this specific case) among those contributing the least to the first Principal Component (PC) of PCA. Variables contributing the least to the first PC of PCA do not shape much of the information contained in the data. As a consequence, one can without loss of generality assume that information in regard to what the system did not do during a phase is captured by sensors contributing the least to the first PC of PCA.

Sensors selected from PCA serve as phase characteristic and are used to assign labels to phases following rules given in Table 5.5. Rules of Table 5.5 are based on observation; however, beneath is an attempt to explain a few.

Table 5.5: Rules for assigning labels to phases given sensors selected from PCA.

Sensors selected from PCA for phase characterisation	**Label associated**
cache_ref & cache_misses & branch_misses or branch_ins	compute-intensive
no IO related sensor	communication IO intensive
branch_misses & hardware_ins or branch_ins	mixed
hardware_ins & cache_ref or cache_misses	memory-intensive

Let us comment the first row of Table 5.5. Workloads/applications with frequent cache references and misses are likely to be memory intensive. The fact that sensors referring to cache misses and references (cache_ref and cache_misses) are selected from PCA indicates that the workload is not memory intensive. If in addition we have branch misses or branch instructions selected from PCA then we assume the workload to be compute intensive. For the second row, sensors selected from PCA exclude any IO related sensor; we can therefore assume that the system was running either a communication or an IO intensive workload. Similar analysis based on empiric evidences can explain the remaining rules of Table 5.5.

5.3.2 System phase characterization using principal component analysis (PCA)

The sensor-based workload characterization as presented in the previous section is very simple to compute, but it is not very accurate because it depends on the way variables are interpreted. Moreover, similarly to the LLCRIR-based phase characterization it does not allow characterizing idle periods and performs poorly for IO and communication intensive workloads. In this section, we present a third phase characterization scheme that we refer to as PCA-based phase characterization. It is similar to the sensor-based approach in the sense that they both use PCA.

The main objective of our PCA-based workload characterization is to discover patterns shared by workloads of the same category. It aims to extract characteristics shared by workloads having the same predominant behaviour. To accomplish this, we define a set of "reference" workloads – composed of compute intensive, memory intensive, IO intensive, and communication intensive workloads – in line with the general trend of HPC workloads. We next apply PCA to individual datasets made up with execution vectors collected during the execution of each workload category (CPU intensive, memory intensive, IO intensive, communication intensive) to find out how variables (sensors to be specific) correlate with principal components (PCs) of PCA. The correlation of a variable with a PC is defined as the value of that PC in the projection onto the plane of PCs. Figure 5.1 offers a graphical representation of PCA results for CG, FT, MG, SP, and BT benchmarks. Figure 5.1 reveals that excepting CG with is exclusively memory intensive, the first principal component of PCA (PC 1) opposes sensors related to the processor (hardware instructions,

branch instructions, and branch misses) to those related to the memory (cache misses and cache references) for all listed workloads which appear to be mixed workloads. Hence, we can postulate that for mixed workloads, sensors related to the memory are negatively correlated to PC 1 while those related to the processor are positively correlated to PC 1. The correlation of sensors related to network and IO activities is nearly zero because they have very low network and IO activities. This also allows us to postulate that for workloads that are neither communication nor IO intensive, the correlation of sensors related to IO and network activities with PC 1 and PC 2 is insignificant.

For a memory intensive workload such as CG (Figure 5.2), we can easily notice that sensors related to the memory (cache references and cache misses) are symmetric with respect to the origin of the two-dimensional plane generated by PC 1 and PC 2. Similar observations in regard to IO related sensors (wIO and rIO) can be made from Figure 5.3(b) which offers a graphical representation of PCA results applied to the dataset made up of EVs collected when running IOzone. IOzone is a file system benchmark tool[1].

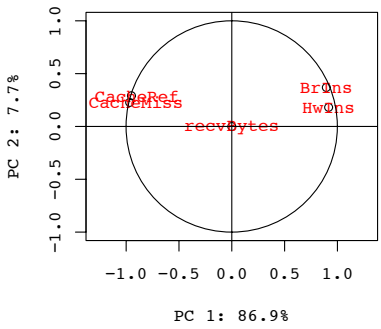
Figure 5.3 corroborates our assumption that for workloads which are neither IO intensive nor network intensive, the correlation of IO and network related sensors with PC 1 and the second principal component of PCA (PC 2) is insignificant. Figure 5.3(a) offers a graphical representation of the result of PCA applied to the dataset made up of execution vectors collected when running a network intensive program: Netperf[2]. It can be seen that unlike graphics of Figure 5.1 the correlation of network related sensors (sentBytes and recvBytes) with either PC 1 and PC 2 is no longer negligible.

We also applied PCA to datasets made up of EVs collected when the system was running extremely compute intensive workloads such as embarrassingly parallel (EP) from NAS parallel Benchmark and cpubrun, a CPU stress tester available to Linux platforms. However, the corresponding graphics are not plotted, as for those workloads, PCA only produces one principle component.

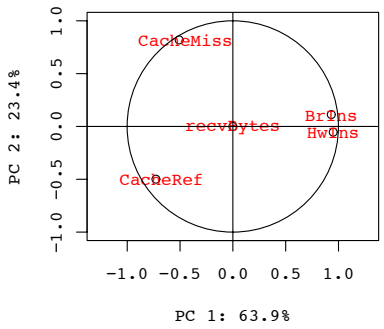
As discussed earlier, the main issue with LLCRIR-based and sensor-based phase characterization schemes is that they do not permit us to characterize periods wherein the system is not running any user applications (i.e., idle periods). To characterize idle periods, we consider two datasets made of EVs collected at two different points in time when the system was idle. We refer to those datasets as "idle_1" and "idle_2" respectively. Figure 5.4 offers a graphical representation of the result of PCA applied to "idle_1" and "idle_2". It can be seen that correlations of sensors with either PC 1 or PC 2 are nearly the same in both cases except that Figure 5.4(a) seems to be the translation of Figure 5.4(b) with respect to PC 1 and vice versa. We can also observe that CPU related sensors are highly correlated with PC 1 (negative correlation) and have very low correlations with PC 2.

Overall, for each category of workloads, variables have a specific patterns with

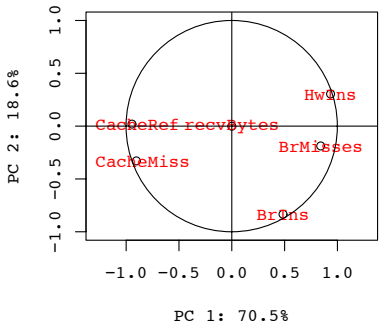
[1] http://www.iozone.org/
[2] Netperf, http://www.netperf.org



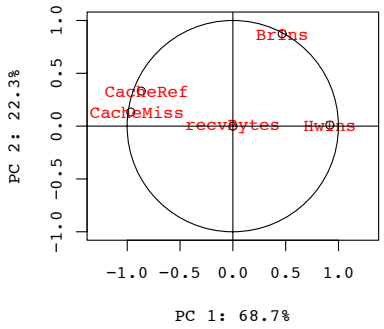
(a) SP, PC 1 and PC 2 explain 86.9% and 7.7% of the variability respectively.



(b) FT, PC 1 and PC 2 explain 63.9% and 23.4% of the variability respectively.

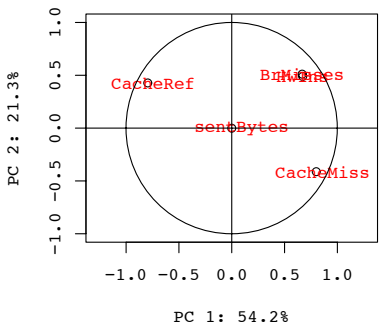


(c) BT, PC 1 and PC 2 explain 70.5% and 18.6% of the variability respectively.



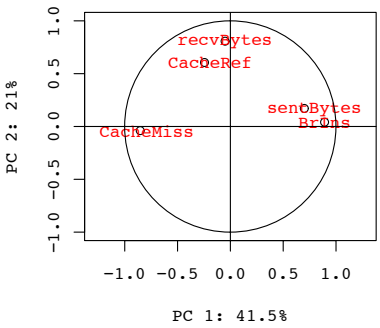
(d) MG, PC 1 and PC 2 explain 68.7% and 22.3% of the variability respectively.

Figure 5.1: Principal component analysis (PCA) of Benchmarks from NPB benchmark suite. Variables are projected on the plane of the first two principal components. Variables may not be the same in all cases because, we remove correlated variables before applying PCA.

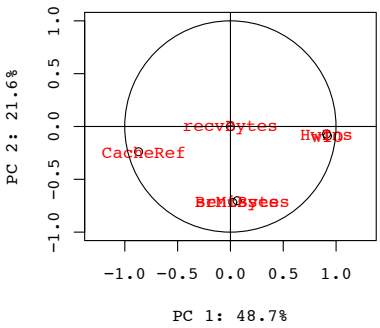


(a) CG, PC 1 and PC 2 explain 54.2% and 21.3% of the variability respectively.

Figure 5.2: Principal component analysis (PCA) of CG benchmark Benchmarks from NPB benchmark suite. Variables are projected on the plane of the first two principal components. Variables may not be the same in all cases because we remove correlated variables before applying PCA.

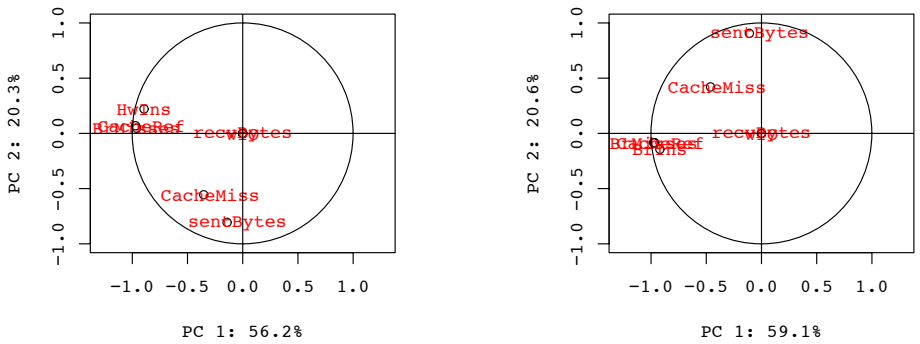


(a) Netperf (send and recieve), PC 1 and PC 2 explain 41.5% and 21% of the variability respectively.



(b) IOzone read-write test, PC 1 and PC 2 explain 48.7% and 21.6% of the variability respectively.

Figure 5.3: Principal component analysis (PCA) of network 5.3(a) and IO intensive 5.3(b) workloads. Variables are projected on the plane of the first two principal components. Variables may not be the same in all cases because, we remove correlated variables before applying PCA.



(a) idle_1, PC 1 and PC 2 explain 56.2% and 20.3% of the variability respectively.

(b) idle_2, PC 1 and PC 2 explain 59.1% and 20.6% of the variability respectively.

Figure 5.4: Principal component analysis (PCA) applied to data collected when the system was idle. Variables are projected on the plane of the first two principal components. Variables may not be the same in all cases because, we remove correlated variables before applying PCA. The two datasets are made up of EVs collected at two different points in time when the system was idle.

respect to the first principal component of PCA (PC 1) and the second principal (PC 2). Now we are left with writing specific code segments describing each class of workloads for on-line use. For memory intensive workloads for example, it boils down to expressing the fact that sensors related to the memory are symmetric with respect to the origin of the PC1-PC2 plane.

5.4 Phase Characterization Algorithms: a Comparative Analysis

As we mentioned earlier, the effectiveness of system reconfiguration based on reuse of known optimal configurations is closely constrained by the characterization process. In other words, the characterization process must be accurate enough to guarantee either performance and/or energy performance improvement depending on the target objective. To mitigate misleading reconfiguration decisions we devise and compare two phase characterization algorithms using characterization schemes described above.

Our first characterization algorithm is a majority-rule-based algorithm. It is majority-rule-based because to a given phase it applies the three phase characterization schemes we just presented (Section 5.2.1, Section 5.3.1, and Section 5.3.2) and uses the majority rule to pick-out the appropriate label. When the rule of majority cannot apply (i.e., labels returned by all three characterization schemes are all different). The result of the characterization scheme deemed more accu-

rate is selected. For example, the LLCRIR-based phase characterization has very high accuracy predicting compute intensive workloads; thus, if the majority-rule fails and the LLCRIR scheme says it is compute intensive then the selected label for the phase will be compute-intensive. Similarly, as the PCA-based approach is the most accurate for memory intensive, IO intensive, and communication intensive workloads, when the rule of majority cannot apply its result is considered the most relevant. We also use the PCA-based result when the rule of majority fails and the PCA-based approach suggests that the workload is mixed.

In the same line, the second phase characterization algorithm we propose consists of using the LLCRIR-based phase characterization scheme when the PCA-based schemes fails to return a label (it is referred to as "PCA then LLCRIR" in Table 5.6). This may happen in some cases, the PCA-based scheme assigns a workload/phase to the appropriate class (by selecting the label representing that class of workloads) using known patterns. Therefore, when the pattern of the new phase is unknown, it is unable to assign that phase to a given class. The LLCRIR-based scheme takes over at this point as it is guaranteed to always return a label.

To see how close to reality our characterization algorithms are, we executed a set of workloads that we characterize by assigning each to a label using the two algorithms described earlier. Table 5.6 – where "IDLE" is assimilated to a program within which only operating system (OS) related tasks are executed – offers an outline of the characterization results using individual schemes and when they are combined (majority-rule-based and PCA then LLCRIR). Note, when labels returned by the three characterization schemes are all different, the PCA-based label is selected; however, if the PCA-based failed to return a label the one returned the by LLCRIR-based scheme is used. Programs under consideration were executed from top to bottom as they appear in Table 5.6. SCP or secure copy is a remote file copy program which copies files between hosts on a network. Excepting SCP and IDLE, the remaining programs are from NAS Parallel Benchmark suite [Bailey *et al.* 1991].

We can observe that our two algorithms in most cases assign the same label to a given workload, but they have relatively poor performance when the workload at hand is the IDLE program. This can be easily addressed by filtering out idle periods; for example, using the system's load provided an acceptable definition of the concept of "idle system" in regard to the system's load.

5.5 Conclusions

In this chapter, we proposed three system phase characterization schemes and showed how they can be used in a real life environments. The proposed characterization schemes mainly intend to facilitate reuse of configuration information for recurring phases. To accomplish this, they assign system phases to classes represented by labels which implicitly tell the kind of reconfiguration decisions acceptable for a specific class of workloads depending on the target objective. We defined six basic classes of workloads (in line with high performance computing workloads)

Table 5.6: Sample characterization results with varied workloads. Program names are written in block letters

Program names	Characterization schemes			Algorithms	
	LLCRIR-based	**PCA-based**	**Sensor-based**	*Majority-rule*	*PCA then LLCRIR*
IDLE	mem	idle	mem	mem	idle
FT	mixt	mixt	mem	mixt	mixt
SCP	mixt	IO netRecv netTransmit mem	mem	IO netRecv netTransmit mem	mem netRecv IO netTransmit
BT	mixt	mixt	mixt	mixt	mixt
IDLE	mem	netRecv netTransmit	mem mem	mem mem	netRecv netTransmit
CG	mem	mem	mem	mem	mem
IDLE	mem	idle	mem	mem	idle
EP	cpu	cpu	cpu	cpu	cpu
IDLE	mem	idle	mem	mem	idle
UA	mixt	mixt	mixt	mixt	mixt
IDLE	mem	idle	mem	mem	idle
MG	mixt	–	mixt	mixt	mixt
IDLE	mem	netRecv mem	mem	mem	netRecv mem
SP	mixt	cpu	mixt	mixt	cpu
IDLE	mem	idle	mem	mem	idle
FT	mixt	mixt	mixt	mixt	mixt
SCP	mixt	cpu	mem	cpu	cpu
BT	mixt	mixt	mixt	mixt	mixt
IDLE	mem	netRecv mem	mem	mem	netRecv mem
CG	mem	mem	cpu	mem	cpu
IDLE	mem	netRecv	mem	mem	netRecv
EP	cpu	cpu	cpu	cpu	cpu
IDLE	mem	idle	mem	mem	idle
UA	mixt	mixt	mixt	mixt	mixt
IDLE	mem	netRecv mem	mem	mem	netRecv mem
MG	mixt	netRecv mixt netTransmit	mixt	mixt	netRecv mixt netTransmit
IDLE	mem	netRecv mem	mem	mem	netRcev mem
SP	mixt	cpu	mixt	mixt	cpu
EP	mixt	cpu	cpu	cpu	cpu
SCP	mixt	IO mem netTransmit	mixt mixt	mixt mixt	IO mem netTransmit
BT	mixt	mixt	mixt	mixt	mixt
IDLE	mem	idle	mem	mem	idle
CG	mem	mem	mem	mem	mem
EP	cpu	cpu	cpu	cpu	cpu
IDLE	mem	idle	mem	mem	idle
UA	mixt	mixt	mixt	mixt	mixt
MG	mixt	–	mixt	mixt	mixt
SP	mixt	–	mixt	mixt	mixt
IDLE	mem	idle	mem	mem	idle

Meaning of characterization labels:

cpu	compute intensive
mem	memory intensive
mixt	both memory and compute intensive
netRecv	receive intensive (communication)
netTransmit	transmit intensive (communication)
idle	idle system (no user application running)
–	no label was returned by the corresponding characterization scheme

according to system resource utilization and provide simple, but accurate characterization algorithms to identify them at run-time.

Although some phase characterization schemes may seem very complex for online use, they are very fast and have nearly no impact on system performance. The amount of time needed for computing our three phase characterization schemes is generally in the order of a second because phases are of relatively short durations (from a few seconds to a few minutes or hours). In addition, phase characterization is only performed when the process of identifying the phase or workload at hand with an existing phase fails. In summary, the time it takes to perform phase characterization is negligible.

Chapter 6

Phase Identification and Power Saving Schemes

Contents

6.1 Introduction

Chapter 4 and Chapter 5 presented the two first stages of our methodology for improving the energy efficiency performance of High Performance Computing (HPC) systems. This chapter presents the third and last step which we refer to as phase identification and system reconfiguration (used interchangeably with the term adaptation hereinafter). Phase identification and system reconfiguration is the stage where phase detection and characterization are exploited; it is a desirable property of phase detection mechanisms since it enables reuse of known optimal configuration information for recurring phases.

To exploit phase detection and characterization, it is essential that recurring phases be identified. However, phase identification techniques assume that phases being identified are already finished, which is not useful for an on-line system. This is justified by the fact that system reconfiguration decisions triggered when the appropriate phase finishes might not lead to the expected result. The literature suggests predicting the upcoming or next phase along the execution timeline [Sherwood *et al.* 2003, Ge *et al.* 2007, Spiliopoulos *et al.* 2011] (further details can be found in Chapter 2). Unfortunately, predicting the upcoming phase is a non trivial task when there is no information about workloads being executed.

System reconfiguration decisions, depending on the target objective, can be either performance oriented (the focus is put on reducing the execution time of the application running) or energy performance oriented. The last case focuses on reducing the energy consumption of the program being executed or that of the overall infrastructure without necessarily reducing the execution time of applications being executed. In this thesis, we are interested in aspects of energy performance improvement; thus, system reconfiguration decisions which we also refer to as "*power saving schemes*" or "*green capabilities*" are energy performance oriented.

This chapter discusses about our phase identification techniques and power saving schemes; it is organised as follows: Section 6.2 presents two complementary phase identification and prediction methodologies. Section 6.3 introduces and analyses potential power saving schemes for HPC systems. Finally, concluding remarks are provided in Section 6.4.

6.2 Recurring Phase Identification and Prediction

Phase identification is often used in conjunction with phase prediction; however, as we already discussed, it is difficult to predict the behaviour of the upcoming phase in scenarios wherein the system can potentially run multiple applications (recall that the term application is used interchangeably with the term workload) about which there is no information. This section presents two alternatives to phase prediction: *partial phase recognition* and *Execution Vectors (EVs) classification*. They are tightly related to the phase detection mechanism; in other words, they work with the concept of execution vector.

In both cases, when phase identification is successful, rules defined in Table 6.1 are used to determine adequate power saving schemes referring to labels assigned to the known phase (the phase that the new phase is identified with). For example, in a compute intensive phase (compute-intensive label in Table 6.1), one can consider switching off memory banks, putting disks into sleep mode, and putting Network Interconnects (NICs) into the Low Power Idle (LPI) mode. The previous example assumes that the compute-intensive label is the only label associated to the corresponding phase. When there are several labels associated to a given phase, the resulting system reconfiguration decisions attempt to offer reasonable performance. To illustrate, in the presence of memory-intensive and communication-intensive labels for example, the system will be configured so as to guarantee "good" performance to both memory intensive and communication intensive workloads.

6.2.1 Partial phase recognition

Partial phase recognition suggests identifying an ongoing phase by comparing well defined portions of two phases then extrapolating the result to the remaining part. In a more formal way, partial phase recognition is the process of identifying an ongoing phase (the phase has started, but is not finished yet) $P_{ongoing}$ with a known

Table 6.1: Phase labels and associated power saving schemes.

Phase label	Possible reconfiguration decisions
compute-intensive	switch off unused memory banks; send disks to sleep; scale the processor frequency up; put NICs into LPI mode.
memory-intensive	scale the processor frequency down; decrease disks; or send them to sleep; switch on memory banks.
mixed	switch on memory banks; scale the processor frequency up; send disks to sleep; put NICs into LPI mode.
communication intensive	switch off memory banks; scale the processor frequency down; switch on disks.
IO-intensive	switch on memory banks; scale the processor frequency down; increase disks (if needed).

phase P_j by only considering the already executed part of $P_{ongoing}$. The just mentioned already executed part of $P_{ongoing}$, expressed as a percentage of the length (duration) of P_j is referred to as the "*recognition threshold*" and denoted as RT. Thus, with a RT% recognition threshold and assuming that the reference vector of P_j is EV_{P_j} and that its length is l_j, an ongoing phase $P_{ongoing}$ is identified with P_j if and only if: the Manhattan distance between EV_{P_j} and each EV pertaining to the group made up of EVs collected when running the already executed part of $P_{ongoing}$ (corresponding in length to RT% of l_j) is within a threshold called ST. Note that the detection threshold ST is the same that was introduced in Chapter 4 for phase detection. In fact, two EVs are within the threshold when they are similar; in other words, the Manhattan distance between them is smaller than the threshold. Partial phase recognition is summarised in the pseudo algorithm bellow, where Manhattan(v,EV_{P_j}) is the Manhattan distance between v and EV_{P_j}

1 Let P_j be a completed phase, EV_{P_j} its reference vector, and l_j its duration

2 $P_{ongoing}$ whose initial EV is timestamped t is partially recognized/identified as P_j

$$\iff \forall\ v, \in \{\ v_i, \text{EVs sampled from } t \text{ to } t + RT \times l_j\}$$
$$\text{Manhattan}(v,EV_{P_j}) \leq ST$$

With respect to reusing known configuration information, partial phase recognition can be very effective. In fact, when partial phase recognition is successful, the reconfiguration decision is made and lasts for the remaining part of the ongoing phase. In other words, if adequate decisions are made (the decision making process relies upon the phase characterization mechanism), the system can experience significant performance improvements; otherwise (reconfiguration decisions did not lead to the expected results), significant performance degradation. To mitigate the

impact of misleading reconfiguration decisions, the next section introduces a new phase identification and prediction methodology.

6.2.2 Execution vectors' classification

The execution vector classification and prediction approach explicitly attempts to match each newly sampled EV with known phases. When the attempt to match an EV sampled at time t to an existing phase succeeds, system reconfiguration decisions or power saving schemes are triggered to reconfigure the system for the next second (the matching process is discussed in the next paragraph) accordingly. Roughly speaking, when the EV sampled at time t matches to an existing phase P, the configuration of the system at time $t + 1$ is the configuration that was found optimal for P. In summary, instead of predicting the next phase, we determine the behaviour of the next EV along the execution time line using a principle widely exploited by caching algorithms. The idea behind can be expressed as follows: if the system is running a task labelled as $label_1$ at time t, then it is likely to be running a task with the same label a time $t + 1$.

The matching process is as simple as comparing a candidate pattern (EV just sampled) to known patterns i.e., representative vectors of known phases. To accomplish this, we define the error resulting from matching an EV to the representative vector of an existing phase as a vector of component-wise absolute difference between them (each element in one vector is subtracted to its counterpart from the other). In a formal manner, given two vectors $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$ in an n-dimensional space, the error resulting from matching them is given by a new vector $W(abs(y_1 - x_1), \ldots, abs(y_n - x_n))$. Note, the Manhattan distance between X and Y is obtained by summing elements of W.

Entries of W which we refer to as "component-wise" errors, show how each component in the vector X differs from its counterpart in Y, while the Manhattan distance between X and Y shows how X differs from Y. In using the component-wise error between two vectors, we assume that an EV matches to a representative vector of a phase when each entry of the vector made up of their component-wise absolute difference is not greater than a threshold th. In mathematical literature, X matches to Y if $w_i \leq th, \forall w_i \in \{abs(y_1 - x_1), \ldots, abs(y_n - x_n)\}$. Empirical analyses showed that a threshold of 0.1 ($th = 0.1$) is effective in our case.

6.2.3 Off-line phase identification

Phase identification techniques we presented so far are designed for on-line use; however, off-line phase identification is often needed when performing program simulation (not simulating recurring phases can save a lot of time). It is off-line because phase being compared have both competed. We systematically perform off-line phase identification for each newly detected phase in order to avoid unnecessary phase characterization. Off-line phase identification is a way easier than its on-line counterpart, for it uses traces of execution of the application. Put simply, in

an off-line context, all phases are finished, which makes phase identification a lot easier.

Remember from Chapter 4 that we represent a phase with a reference vector along with some other information. Now, let us look at Figure 4.7, from Chapter 4, which for the sake of clarity is replicated here as Figure 6.1. It is obvious that for comparing phases only a single point in the similarity matrix is needed. That point of coordinates, let us say, (i, j) is the point representing on a gray-scale the Manhattan distance between EVs sampled at times i and j respectively; i and j only have to be the timestamps associated to reference vectors of the phases we want to compare. More importantly, those vectors can be compared in the same way as all other vectors, i.e., by using a fixed percentage on the maximum distance between reference EV.

Consequently, we state that two phases P_1 and P_2 are identified with each other if the Manhattan distance between them (technically their reference vectors) does not exceed a specific percentage of the maximum distance between all reference execution vectors. In light of what precedes, we suggest that the percentage of the maximum distance between reference EVs be the same as the detection threshold used for phase detection.

To illustrate our off-line phase identification approach, we conduct some experiments using synthetic benchmarks (*bench_ 1* and *bench_ 2*) introduced in Chapter 4. In a few words, *bench_ 1* successively runs workloads from NAS Parallel Benchmark; *bench_ 2* runs the same set of workloads, but inserts fixed length idle periods in between them. We execute each of these benchmarks five times and attempt to identify recurring phases among those that are detected. Figure 6.2 – where doted vertical lines indicate the start and end times of the programs, and horizontal solid lines detected phases – offers a graphical representation of the result of our phase identification methodology. Note that phases that are identified with each other are on the same virtual horizontal line. It can be seen that with this simplistic approach, we loosely capture recurring phases. More interestingly, idle phases are not identified with non idle phases (Figure 6.2(b)).

6.3 Power Saving Schemes

Section 6.2 discusses scenarios that permit the trigger of power saving schemes. In this Section, we will review in details "non conventional" (not commonly used) power saving schemes.

There is no uniform definition of a power saving scheme, but we use it to refer to any action destined to reduce the power/energy consumption of a system or an HPC system without significant performance degradations (recall that up to 10% performance degradation is often acceptable). Power saving schemes range from management practices including system reconfiguration to "good" practices such as executing programs on less power/energy hungry platforms. We do not emphasise on proven power saving schemes such as Dynamic Voltage and Frequency

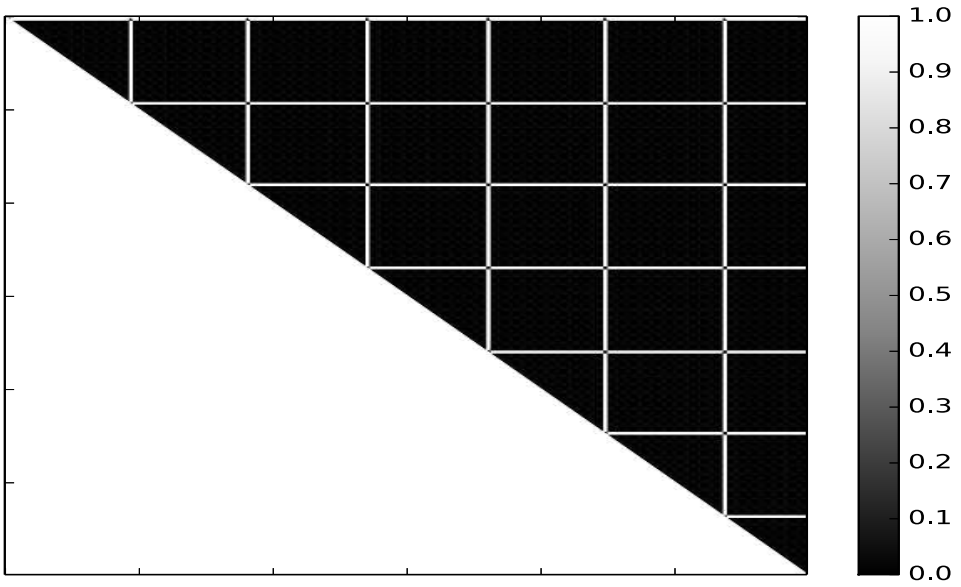
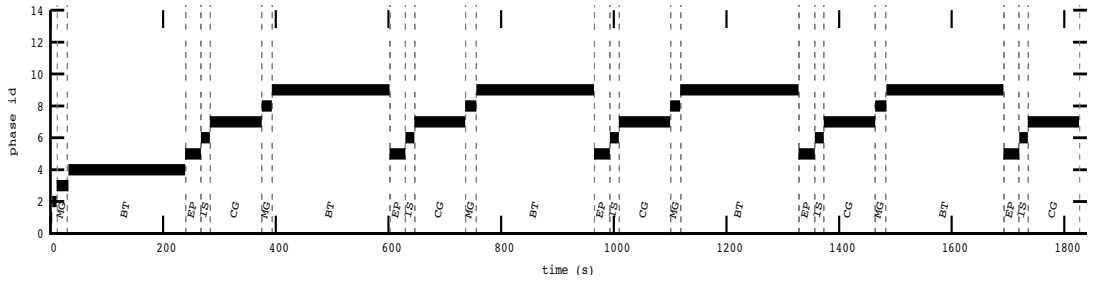
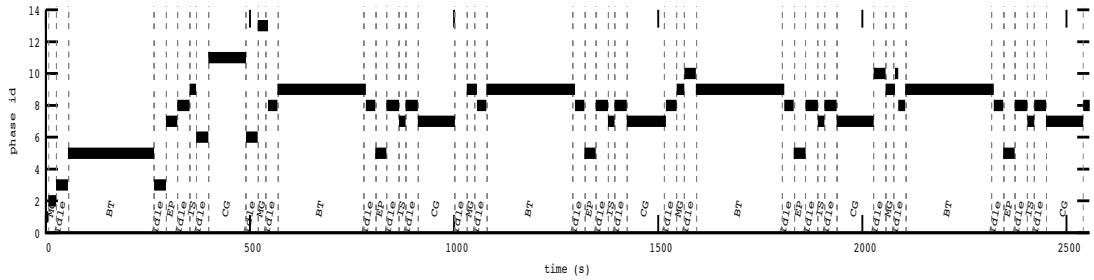


Figure 6.1: Matrix of distance between EVs for WRF-ARW (the matrix corresponds to half of the execution of the program).



(a) Graphical view of system phase distributions resulting from five successive executions of *bench_ 1*.



(b) Graphical view of system phase distributions resulting from five successive executions of *bench_ 2*.

Figure 6.2: Phase identification illustrated with five successive run of each benchmark; the detection threshold is 15% . Phases are compared using their reference vectors.

Scaling (DVFS), and LPI mode for network interconnects as they where introduced in Chapter 2. For the same reasons, we will not discuss about disk spin-down, which consists of switching the disk between operating modes.

6.3.1 Platform selection via cross platform energy prediction

6.3.1.1 Methodology

Users often have more that one candidate platform for running their jobs, in which case choosing the least energy consuming platform can be beneficial both for them and the platform provider. However, it is difficult to know in advance, how much energy an application will consume on a given platform. We suggest a methodology to gather that piece of information. In other words, given an application we attempt to estimate/predict its energy consumption on a target platform.

To achieve this, we implicitly use two datasets: one from the reference platform – a platform on which the application is well known – and the other from the target platform. The target platform is the platform on which we want to estimate the power consumption of our application. The dataset from the reference platform is provided by its DNA-like structure (see Chapter 4 for details). For simplicity, we assume that the application we are interested in estimating its power consumption is the sole program running on the platform. A reference platform will always exists, because each application is written and tested on at least one reference platform. It is the platform on which the DNA-like structure of that application was built. Note, the DNA-like structure of a program is a succession of behaviours through which the system went at the time that program was running. It provides information including the power consumption of the corresponding application.

With the DNA-like structure of an application, one can easily identify applications having similar computational requirements. We accomplish this by comparing the DNA-like structure of the application of interest to known DNA-like structures. A match is found when a given percentage of the application whose energy consumption is being predicted is identified with the same percentage of a known application.

Let us denote by E_{tar} the energy consumption of the part of the application whose energy is being estimated and by E_{ref} the energy consumption of the corresponding part of the application whose DNA-like structure matches which the application at hand. For example, considering an application that lasts 60 minutes on its reference platform, let us assume that E_{tar} represents the energy consumed by the same application on the target platform after 10 minutes; therefore, E_{ref} represents the energy consumed by the application on the reference platform during the first 10 minutes of its execution. We introduce the concept of relative energy consumption described by Equation 6.1 and denoted as $E_{relative}$; it basically captures the variation in energy consumption between the target and the reference platforms.

$$E_{relative} = \frac{E_{tar}}{E_{ref}} \tag{6.1}$$

Using the relative energy consumption between the two platforms, the estimated

energy consumption of the application on the target platform can be expressed by Equation 6.2 where $\int_0^{X\%} P(t)_{i,tar}\,dt$ is the energy consumed by the application on the target platform before a match is found with a known DNA-like structure. Either measured or estimated, $P(t)_{i,tar}$ is the instantaneous power usage of the application on the target platform. Likewise, $P'(t)_{j,ref}$ is the instantaneous power usage of the application on the reference platform and can be obtained from its DNA-like structure.

$$E_{est} = \int_0^{X\%} P(t)_{i,tar}\,dt + E_{relative} * \int_{X\%}^{end} P'(t)_{j,ref}\,dt \tag{6.2}$$

The power consumption might unexpectedly change after the X% threshold. If that change does not result in a change in the set of sensors used to estimate the power consumption as defined in Chapter 4, then we assume it is the same application; otherwise, we seek for another match.

The estimation/prediction accuracy described by Equation 6.3 is defined as the ratio between the estimated and measured energy consumption on the target platform.

$$Accuracy = \frac{E_{est\_tar}}{E_{tar}} \tag{6.3}$$

Comparing two DNA-like structures boils down to comparing two strings. Therefore, the overhead associated with matching the DNA-like structure of a running application with previously seen and known applications is proportional to the number of already known applications (could be classes of applications instead) times the size of the DNA-like structure of the application at hand.

To simplify, we assume that our profile database (list of known applications) contains a unique DNA-like structure, that of the application whose energy consumption is being estimated on the target platform. We also assume that applications follow a very simple execution pattern (it is actually the case for the majority of scientific applications) which starts with an initialisation phase and finishes with a finalisation phase. Between the initialisation and the finalisation phases there are some iterative computation followed by optional communications or IO activities. Finally, from the assumption that the power consumption of the application in each iteration is approximately the same, Equation 6.2 can be simplified to Equation 6.4 where E_{init} is the energy consumed by the application on the target platform during the initialisation phase, $E_{ref-init}$ is the energy consumed by the application on the reference platform from the end of the initialisation phase to the completion of the whole application, and $E_{ref-exe}$ is the measured energy consumption (resulting from its whole execution) of the application on the reference platform.

$$E_{est} = E_{init} + E_{relative} * (E_{ref-exe} - E_{ref-init}) \tag{6.4}$$

6.3.1.2 Illustrative example

This example serves to illustrate the power saving scheme we just introduced. We attempt to predict the power consumption of two applications: (i) a synthetic benchmark *workload_1* which iteratively computes the inverse of a 10×10 matrix and copies a large file from a remote repository; and (ii) GeneHunter [Conant *et al.* 2002] a real life program for linkage analyses.

We further consider three scenarios: (a) the first scenario estimates the energy consumption of *workload_1* on an Intel Xeon node running at 2.13GHz. And uses the same node running at 1.6GHz as the reference platform. (b) The second scenario still estimates the energy consumption of *workload_1*, but uses a Dell Power Edge server and a Sun fire V20z as reference and target platforms respectively. (c) In the third scenario, we attempt to estimate the energy consumption of GeneHunter. In the last scenario, we consider as reference platform an Intel Xeon E5506 with 8 cores and 12GB of RAM (Random Access Memory) and as target platform an Intel Xeon X3440 with 4 cores and 16GB of RAM.

In each scenario, an empiric partial recognition threshold of 20% is used. This means that a match with an existing DNA-like structure DS is found if the already executed part of the workload whose energy is being estimated matches with 20% of DS. For example, assuming that DS lasted 60s, a match will be found if the already executed part of the workload matches with the DNA-like structure describing the first 12s of DS. We compute for each scenario the estimated energy consumption using Equation 6.4. Figure 6.3 – where the y-axis represents the accuracy of the prediction and the x-axis corresponding scenarios – offers an outline of the accuracy of our energy prediction methodology. We can observe that the accuracy, which is computed using Equation 6.3 is relatively high. Note in passing that the statistics shown in Figure 6.3 are based on average energy consumption. Although the energy consumption is overestimated with respect to that baseline (average energy consumption), we believe it is acceptable to do so since the peak energy consumption is typically higher than the average. For GeneHunter, the accuracy approaches 1.2. The program uses a hidden Markov model (HMM) to calculate identity by descent (IBD) sharing probabilities, so the computation requirement of all iterations might not be the same, which diverges from our assumption that all iterations are nearly the same. This could possibly explain its overestimated energy consumption.

6.3.2 Memory size scaling

This section investigates the relevance of memory size scaling (MSS), which consists of adapting the size of the memory to workload's demands in a CPU frequency scaling like fashion. To this end, we execute a set of "reference" workloads – including Lower-Upper Gauss-Seidel solver (LU), Block Tri-diagonal solver (BT), Conjugate Gradient (CG), Embarrassingly Parallel (EP), Integer Sort (IS), and Unstructured Adaptive mesh (UA) – from NAS Parallel benchmark suite [Bailey *et al.* 1991] while statically limiting the size of the main memory.

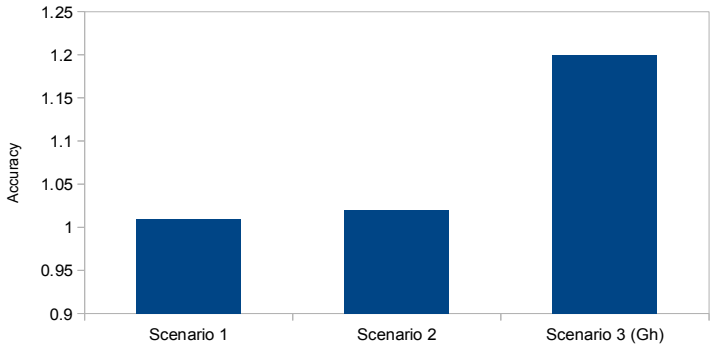


Figure 6.3: Per scenario energy prediction accuracy.

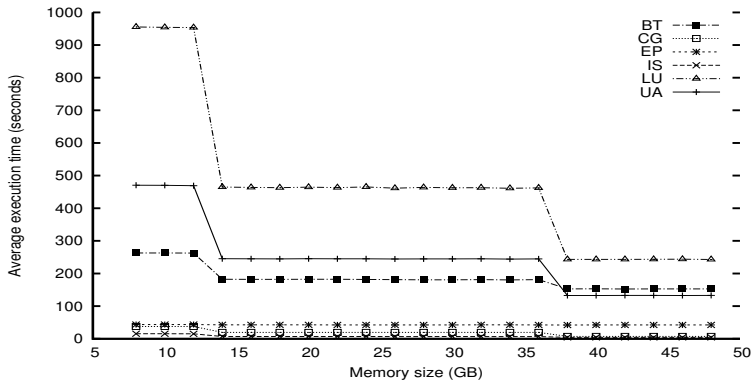


Figure 6.4: Average execution time of NPB-3.3 benchmarks with respect to the memory size.

Figure 6.4 offers an outline of the average execution time of each workload w.r.t the size of the memory on a 48GB server system. Based on the graphic of Figure 6.4, we introduce the concept of "optimal memory (RAM) size" which we define as the memory size from which the execution time of the workload remains nearly constant. The optimal memory or RAM size being workload dependent, Table 6.2 offers an outline of the optimal RAM size for workloads involved in our analysis. For example, EP's optimal RAM size is 8GB which means that EP's execution time is nearly constant as long as the memory size is greater than 8GB. Figure 6.5 outlines the difference between the power consumed by each workload at its optimal point and the power consumed when the 48GB of memory available are used. It suggests that depending on the workload, one could possibly reduce the peak power consumption of our server of up to 10%, if it was possible to scale the memory's size down when needed. Still on Figure 6.5, on top of each point is displayed the corresponding percentage of the peak power consumption (278 Watts) of our 48GB server.

We believe memory size scaling has great potentials for reducing the energy consumption of high performance computing infrastructure. The latent message behind Figure 6.4 and Figure 6.5 is that: "the more memory you have the more power you are likely to waste". To illustrate our point, we conducted a survey of

Table 6.2: Optimal RAM size per workload.

EP	CG	IS	BT	LU	UA
8 GB	14 GB		36 GB		

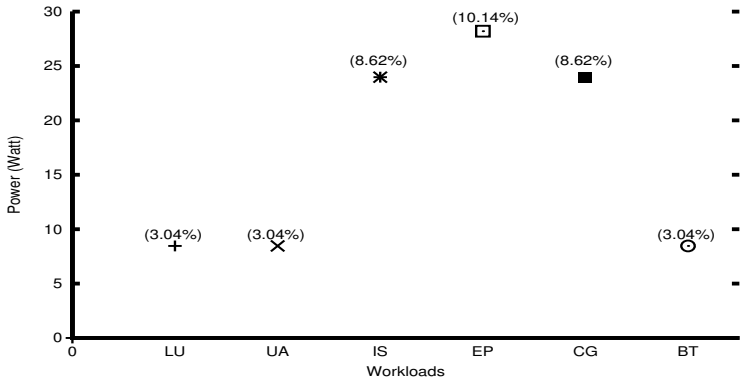


Figure 6.5: Difference in power usage between the optimal memory size and the total memory.

users of the Reims site of Grid'5000 [Bolze *et al.* 2006] to determine the average amount of memory they usually need for their experiments. Grid'5000 is a French large-scale experimental platform distributed over 10 sites (refer to Chapter 7 for further details) while the Reims site hosts a cluster of 44 HP Proliant DL165 G7 nodes, each provided with 48GB of memory. As shown in Figure 6.6 nearly 60% of users out of 35 users who responded to the survey need in average 6GB of RAM per node, while less that 10% need 48GB of RAM. In our opinion, memory size scaling is probably an effective way of addressing the memory's power consumption issue, because HPC systems' designers just don't know in advance how much memory users' applications will require.

6.3.3 CPU cores switch on/off

Modern processors chips are often provided with several CPU cores. Following the logic that the more CPU cores you have the more energy you consume, this section investigates whether core switching, which consists of dynamically switching off/on some CPU cores when executing specific workloads, can be of any help in regard to the energy reduction problem in HPC systems. To accomplish this, we consider four workloads that we successively run while varying the number of available CPU cores. These workloads include: Lower-Upper Gauss-Seidel solver (LU), Conjugate Gradient (CG), Embarrassingly Parallel (EP) from NAS Parallel Benchmarks suite, and Netperf[1] a network performance benchmark. Netperf is configured to send 10GB of data over the network. The host system is an Intel Xeon E5506 with 8 cores and 12GB of RAM. Note, each benchmark has the same workload regardless

[1]Netperf, http://www.netperf.org/netperf/

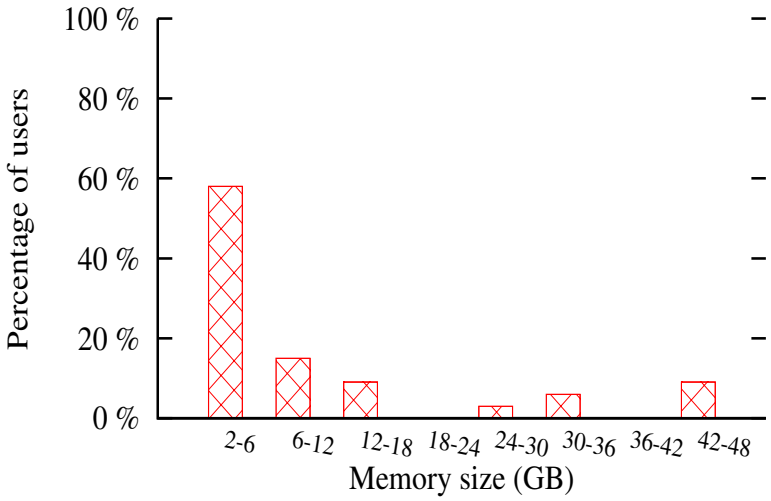
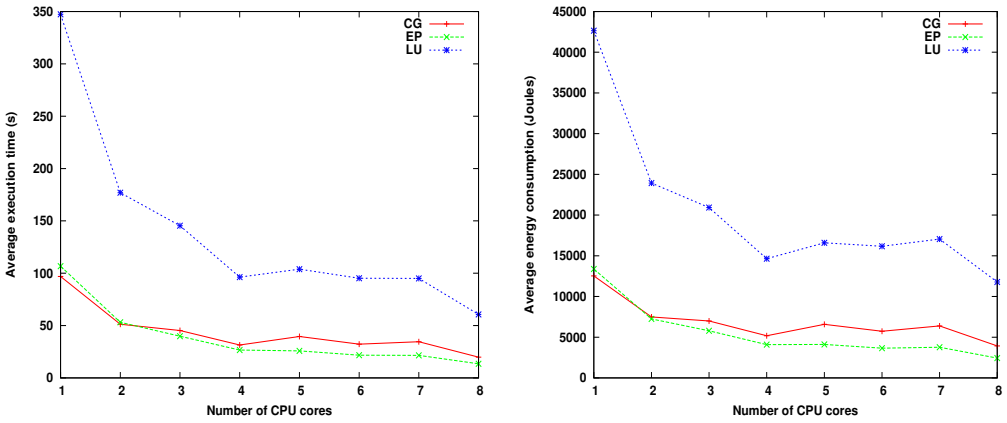


Figure 6.6: Memory requirement per node on the Reims site of Grid5000 for users experiments.

of the number of CPU cores available.

Figure 6.7 where the y-axis represents the average execution time (Figure 6.7(a)) or the average energy consumption (Figure 6.7(b)) offers an outline of the variations of both the energy consumption and the execution time of workloads we just mentioned with respect of the number of CPU cores available. Although performance seems relatively stable when the number of CPU cores ranges from 4 to 7, the main conclusion we can draw is that CPU core switch off/on might not be a viable option for certain types of workloads; more precisely, compute intensive, memory intensive and mixed workloads. For example, LU experiences a 58% performance degradation when executed on 7 CPU cores instead of 8 CPU cores.

Contrary to LU, EP, and CG, CPU core switch on/off is worth when running network intensive workloads such as Netperf. Considering as baseline configuration the configuration within which there is only one CPU core active, Figure 6.8 compares the baseline system configuration with system configurations within which the number of active CPU cores ranges from 2 to 8. The difference in energy consumption between the baseline and the system configuration wherein there are 2 active CPU cores is insignificant; however, the difference in energy consumption between the same baseline and the system configuration wherein 8 CPU cores are active is nearly 8% of the energy consumed by the baseline configuration. More interestingly, we can notice that regardless of the number of active CPU cores, the execution time of the program is the same (this is the reason why the execution time histogram is non-existent in Figure 6.8).



(a) Impact of the number of CPU cores available on workloads' performance.

(b) Impact of the number of CPU cores available on workloads' energy consumption.

Figure 6.7: Impact of CPU cores switching of workloads' execution time and energy consumption; figures are averaged over 20 executions of each workload in each system configuration (1 CPU core, 2 CPU cores, ...).

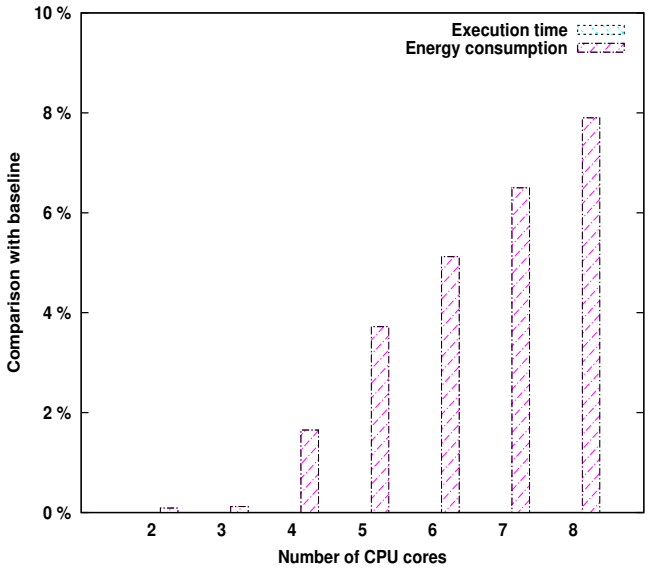


Figure 6.8: Comparison of baseline configuration (only 1 active CPU core) to configurations wherein the number of active CPU cores ranges from 2 to 8; figures are averaged over 20 executions of Netperf in each system configuration (1 CPU core, 2 CPU cores, ...).

6.4 Conclusions

In this chapter, we presented different techniques for recurring phase or workload identification and showed how they can exploit phase detection and characterization to suggest system reconfiguration decisions. These techniques are namely partial phase recognition, execution vectors' classification, and off-line phase identification. Partial phase recognition and execution vectors' classification are used as alternative to phase or workload prediction for guiding system reconfiguration. Although labelled "off-line" the off-line phase identification mainly serves a single purpose in our on-line methodology. That purpose is preventing unnecessary phase or workload characterization and redundancies among known phases. By identifying a newly detected phase with an existing, it is no longer necessary to characterize it; nevertheless, off-line phase identification can also be used for reducing the program simulation time by identifying section of codes whose performance is critical to the entire program. Phase identification techniques as presented in this chapter can serve as a basic point for different types of system optimisations; however, we focused on system reconfiguration decisions having as target objective the reduction of the overall system's energy consumption. We have also introduced non trivial power saving schemes – including platform selection, memory size scaling, and core switch off/on – and studied their feasibility and effectiveness in real life systems. Our static analyses revealed that core switch on/off might not be adequate for our processor class. Nevertheless, we expect future processors to handle it more efficiently.

Memory size scaling has great potentials for reducing the energy consumption of today's High-End and Mid-Market servers. Unfortunately, that feature is not yet available to today's computing systems, but we believe that kernel developers and/or organisations dedicated to performance tuning who have been working on its implementation into future Operating Systems (OSs) (a few patches for Linux OS are under evaluation) will soon come up with a memory size scaling enabled OS or at least with an OS provided with a similar feature.

Platform energy prediction can potentially be used in today's systems, but requires that the program whose power usage is being estimated to be the only application running on the system, which slightly departs from our promise to provide a methodology that focuses on the system and can fully handle the fact that a system can be shared by multiple applications. However, it could be used to provide feedback as whether a power saving scheme will be effective.

We are now left to explore DVFS, NIC speed scaling, and disk spin-down. DVFS is very well supported by processors, so we will use it as our main power saving scheme. Network interconnect speed scaling is poorly supported by some NICs, but it can still be used and we will show that one can benefit from it. Nowadays, nearly all hard disk drives have multiple operating modes, in particular the "sleep" mode which significantly reduces its power consumption.

Chapter 7

Framework-based Implementation and Experimentation: Analysis and Discussion

Contents

7.1 Introduction

Typically, before an energy/power reduction methodology can be implemented in a production environment, its gains must be verified against a large number of applications. As energy reduction and performance (in terms of reduction of applications' execution time) may be conflicting goals, performance degradation resulting from energy reduction decisions must be assessed. Thus far we focused on proposing techniques that allow automated analysis of High Performance Computing (HPC) systems. This chapter investigates the effectiveness of our energy reduction methodology on real systems considering both real life HPC applications and benchmarks that are representative of HPC workloads. We present and evaluate Multi-Resource Energy Efficient Framework (MREEF), an implementation of our methodology for reducing the energy consumption of HPC systems. It is multi-resource in the sense that it can be used to efficiently address the energy consumption issue of all HPC subsystems (or resources) ranging from the processor to memory to network communications and storage subsystems. The diagram in Figure 7.1 offers a graphical

representation of the way components of our energy reduction methodology are arranged in MREEF. In a nutshell, when a phase change is detected, off-line phase identification is performed to determine whether the newly detected phase should be characterized and/or stored. Meanwhile, regardless of whether a phase change is detected, on-line phase identification is performed and system reconfiguration decisions taken accordingly.

The remaining of this chapter is organised as follows: details in regard to our implementation of the energy reduction methodology are discussed in Section 7.2. Section 7.3 describes our experimentation methodology and evaluation protocols. Section 7.4 presents and discusses experimental results. Section 7.5 investigates the relevance of our MREEF in Cloud based HPC environments. Finally, Section 7.6 concludes the chapter and provides recommendations.

7.2 Implementation

This section highlights the most important implementation aspects of the MREEF, which implements our methodology for reducing the energy consumption of HPC systems through two components: the "coordinator" and the "reconfiguration decisions enforcer" which both reside on each node of the HPC system and act in a client server fashion. The coordinator performs system profiling related tasks including phase detection, phase characterization and phase identification. It then notifies the reconfiguration decision enforcer to reconfigure the system when the phase identification process is successful. However, if the reconfiguration decision is to configure the system for a given type of workloads, there is no need of going through if the system is already configured to accommodate such workloads. The reconfiguration decisions enforcer captures resource utilization metrics by reading sensors and implements system reconfiguration decisions. Figure 7.2 summarises the interactions between the coordinator and the reconfiguration decisions enforcer. Decisions are local to each system, but the coordinator, which is also local to each node, is capable of acting in a centralised manner if needed. To illustrate the last point, our first experiment (Section 7.4.1.1) presents and evaluates a centralised version of MREEF.

In Chapter 6 we have introduced and demonstrated the usefulness of several power saving schemes (system reconfiguration decisions). Unfortunately, most of those power saving schemes are either not well supported by today's systems (interconnect speed scaling for example) or not fully implemented (memory size scaling) in current systems. Consequently, unless expressly mentioned, all power saving schemes (system reconfiguration decisions aiming to reduce the energy consumption of the system) we apply are oriented toward the processor through the use of DVFS.

Using two different components for profiling the system (phase detection, phase characterization, and phase identification) and implementing power saving schemes makes our implementation very flexible enough to reflect the proposed methodology. Users who are interested in integrating any power saving scheme of their choosing only have to concentrate on that. More importantly, those power saving schemes can

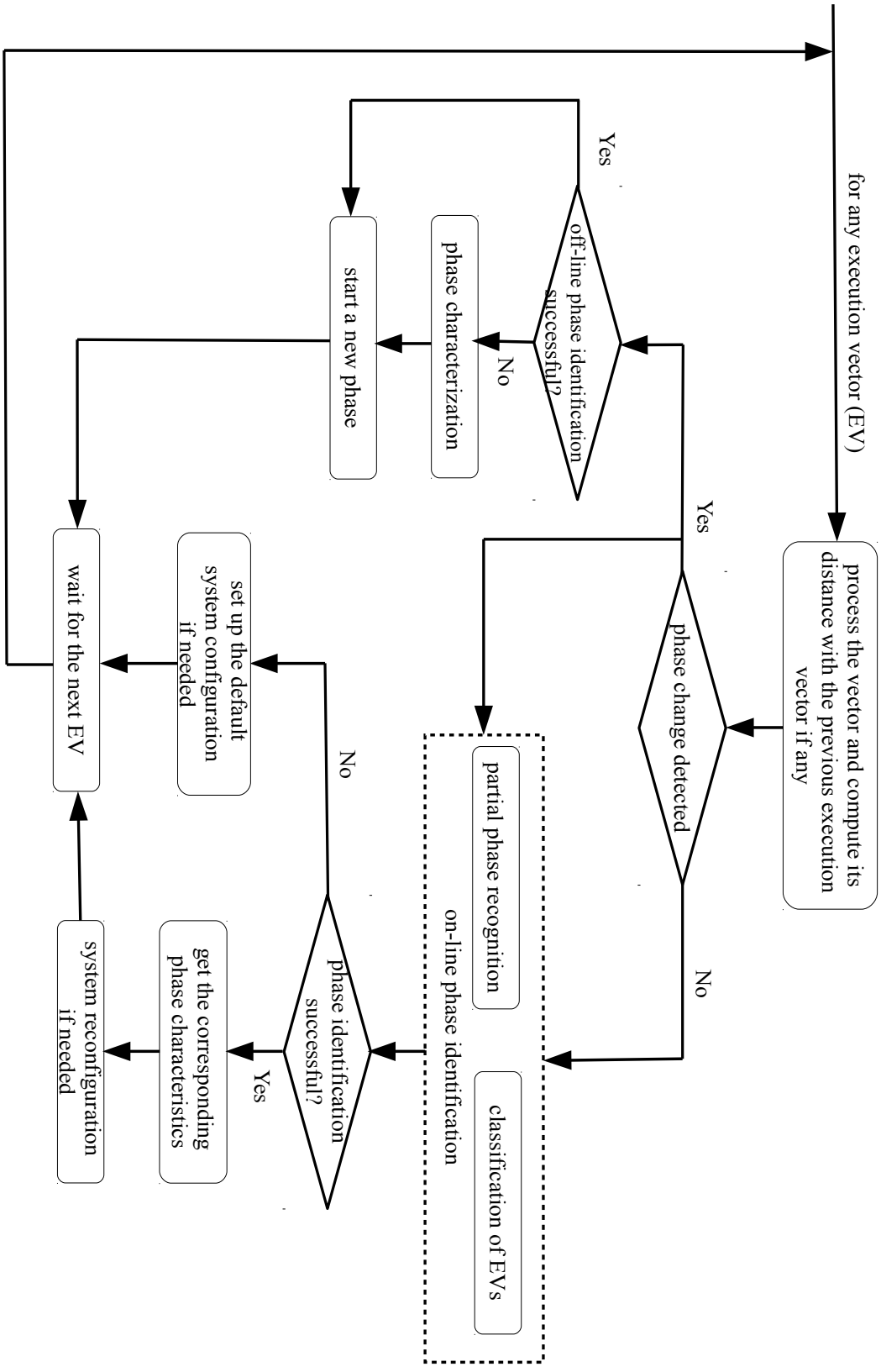


Figure 7.1: Overview of the way in which components of the roadmap are arranged in MREEF.

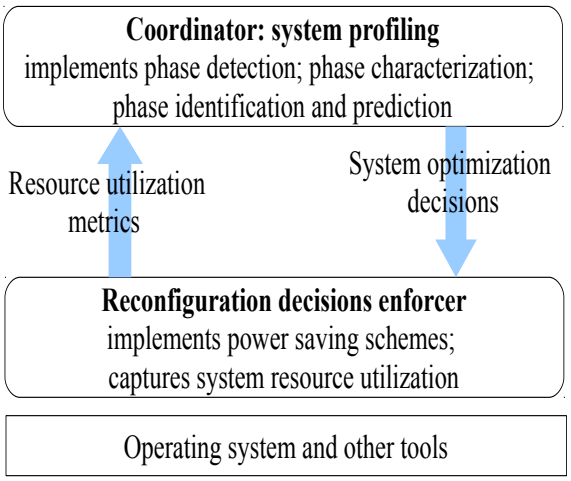


Figure 7.2: MREEF architecture overview.

be directed toward the processor, memory, storage and communications subsystems.

7.3 Experimental Setup and Methodology

We have conducted several experiments with various configurations of MREEF under multiple settings to demonstrate the practicality of our methodology for reducing the energy consumption of HPC systems. In this section, we present our evaluation platform along with experimental protocols and tools.

7.3.1 Evaluation platform description

7.3.1.1 Hardware

Our reference experimental testbed is the French large-scale platform called Grid'5000 [Bolze *et al.* 2006]. Grid'5000, as depicted by Figure 7.3, is a distributed experimental platform funded by the French Ministry of Research, regional councils, the French National Center of Scientific Research[1] (CNRS), the French National Institute for Research in Computer Science and Control[2] (INRIA), and several universities. Grid'5000 is a nation-wide infrastructure distributed across 10 sites, containing 30 clusters with nearly 7000 CPU cores, interconnected through 10Gbps links supported by the Renater Research and Educational Network[3].

Grid'5000 offers an extremely flexible experimental platform as users are free to deploy their own computing environments (from selecting their own operating system to defining CPU cores aggregation: clusters systems against cloud environments) along with associated tools. Grid'5000 implements a general purpose Application Programming Interface (API) that allows users to gather information

[1] http://www.cnrs.fr
[2] http://www.inria.fr
[3] http://www.renater.fr

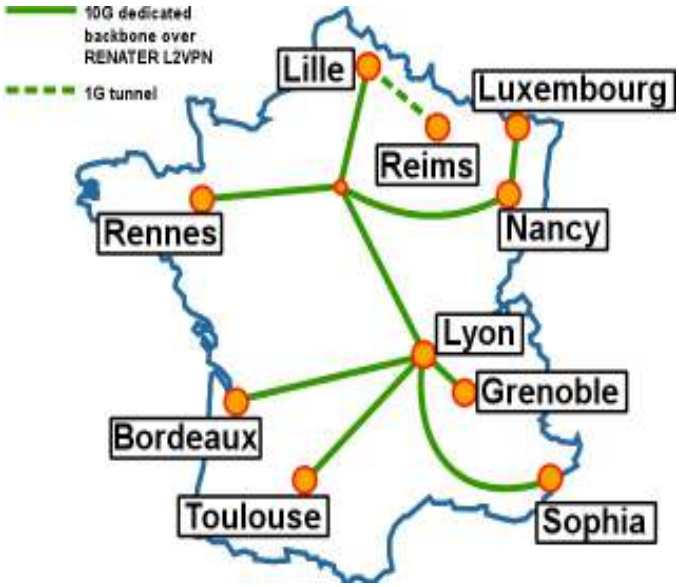


Figure 7.3: The Grid'5000 infrastructure. Each site is labeled by the name of the city in which it is deployed.

– including processor, network, storage, and memory usage – from nodes of the platform. More interestingly, Grid'5000 also provides a power monitoring API that permits to retrieve on a user-defined granularity the power consumption logs for specific nodes [Dias de Assuncao *et al.* 2010]. However, this feature is not available for all nodes of the Grid'5000 platform as some nodes are not provided with power monitoring units or wattmeters. Since in any research aiming to improve energy efficiency, the ability to measure power consumption of compute devices is a prerequisite, the size of our experimental platform is limited by the number of nodes whose power usage can be monitored. Therefore, depending on the number of monitored nodes, in our experiments we use HPC clusters of 15 (60 CPU cores) to 34 (136 CPU cores) Intel Xeon X3440 servers (the exact figure will always be specified). Each Intel Xeon server of our experimental support is provided with 4 cores, 16GB of RAM and has DVFS enabled. Available frequency steps for each core are: 2.53 GHz, 2.40 GHz, 2.27 GHz, 2.13 GHz, 2.00 GHz, 1.87 GHz, 1.73 GHz, 1.60 GHz, 1.47 GHz, 1.33 GHz and 1.20 GHz.

Grid'5000 is not a full production environment, but its flexibility, high availability, and the fact that we can easily reproduce experiments with identical settings justify its use in our research. Furthermore, issues addressed on an experimental platform such as Grid'5000 are equally relevant on any production environment.

7.3.1.2 Software and tools

As stated earlier, we use clusters of varied sizes with nodes operating under Linux kernel 2.6.35. The Linux kernel 2.6.35 is a production kernel intended for everyday use. On each system or node, we use *perf_ events* API for accessing sensors related

to Performance Monitoring Counters (PMCs), and retrieve disk and network usage information from the Linux's */proc/stat* file.

Perf_ events interface offers the advantage that it is included in the Linux kernel. It provides a common subset of useful events that are available on modern processors and allows for querying by any application. Currently, the documentation on *perf_ events* is poor, but further information can be found in [Weaver 2013]. Python interface to R statistical software [R Core Team 2013] is used for performing Principal Component Analysis (PCA) when needed.

Throughout the experiments, we used subsets of benchmarks representative of HPC applications as well as real life HPC applications. The benchmark set is composed of Lower-Upper Gauss-Seidel solver (LU), Block Tri-diagonal solve (BT), Conjugate Gradient (CG), Embarrassingly Parallel (EP), Integer Sort (IS), and Unstructured Adaptive mesh (UA) from NAS Parallel Benchmark suite [Bailey *et al.* 1991]. The set of real life applications includes Molecular Dynamic Simulation (MDS) [Binder *et al.* 2004], the Advanced Research Weather Research and Forecasting (WRF-ARW) model [Skamarock *et al.* 2005], the Parallel Ocean Program (POP) X1 benchmark [4], and GeneHunter [Conant *et al.* 2002]. A short description of these applications is as follows:

- Molecular Dynamics solves numerical Newton's equations of motion for the interaction of the many particles system.

- WRF-ARW is a fully compressible conservative-form non-hydrostatic atmospheric model. It uses an explicit time-splitting integration technique to efficiently integrate the Euler equation.

- POP is an ocean circulation model, the model solves the three-dimensional primitive equations for fluid motions on the sphere under hydrostatic and Boussinesq approximations.

- GeneHunter is a program for linkage analysis. It provides a wide range of analyses for performing linkage and disequilibrium analyses. The backbone of the system is the very rapid extraction of complete multipoint inheritance information from pedigrees of moderate size.

HPC applications generally follow either the OpenMP (Open Multi-Processing) or the Message Passing Interface (MPI) model of parallel programming. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and MPI. We use the OpenMP version of the NAS parallel benchmarks and the MPI version of the real life applications (MDS, WRF-ARW, POP X1, and GeneHunter).

[4] The Parallel Ocean Program, http://climate.lanl.gov/Models/POP/index.shtml

7.3.2 Experimental protocol and tools

Throughout this chapter, we compare MREEF our implementation of the methodology for reducing the energy consumption of HPC systems with the well known *on-demand* and *performance* governors available to Linux platforms. As its name indicates the *performance* governor will always ensure that the system operates at the highest processor's frequency available. The *on-demand* governor commits to offer maximum performance whenever necessary and lowers the processor's operating frequency to reduce the energy consumption of the system whenever possible. This mechanism is made possible through the Dynamic Voltage and Frequency Scaling (DVFS) technology – as shown in Chapter 2, by reducing the processor voltage and frequency one can significantly reduce its power usage. Note that throughout the experiments the on-demand governor is left to its default configuration, which relies on its "up threshold" parameter to make a decision whether or not it should increase the frequency (the default value of that parameter is '80').

Unlike energy reduction methodologies presented in Chapter 2, Linux's *on-demand* governor is system oriented and does not require any knowledge from applications being executed on the system. In addition, it takes into account the fact that multiple applications can share the infrastructure. This makes Linux's *on-demand* governor the closest power reduction policy to our energy reduction methodology. However, the difference is significant, our management policy does not stop with addressing the processor's energy consumption, but goes further to take into account other HPC subsystems or resources from memory to storage to communications subsystems.

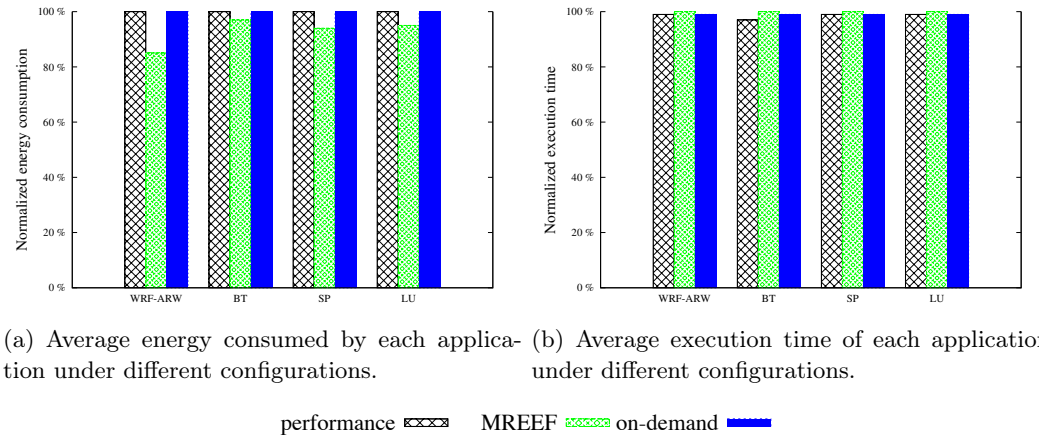
Linux operating system also provides a management policy dedicated to reduce the system's power usage: the *powersave* governor. One may wonder why we do not compare our methodology with Linux's *powersave* governor, but the reason is simple. In the power save mode, the system runs at the lowest CPU frequency available, which of course reduces its instant power usage. However, the pitfall is that in most cases (e.g., when the workload being executed is either compute intensive, memory intensive or mixed) workloads being executed will last much more longer, which in turn increases their energy consumption, which is expressed as the product of power and execution time.

7.4 Results Analysis and Discussion

7.4.1 MREEF: partial phase recognition related results

7.4.1.1 Centralized coordinator

This section evaluates a centralised version of our implementation of the methodology for reducing the energy consumption of an HPC systems. It is labelled as centralised because a single node, which in following the model presented earlier is referred to as the coordinator, is responsible for systems profiling related tasks. In summary, the coordinator performs phase detection, phase characterization, and



(a) Average energy consumed by each application under different configurations.

(b) Average execution time of each application under different configurations.

Figure 7.4: Phase tracking and partial recognition guided processor adaptation results. They are averaged over 20 executions of each workload under each system configuration; they are normalized with respect to baseline execution “on-demand".

phase identification for all nodes. Management activities on other nodes boil down to sending relevant data to the coordinator and implementing system reconfiguration decisions. It is worth mentioning that the coordinator is not a dedicated node, so it also participates in the execution of applications that the platform accommodates.

The centralised version of MREEF is evaluated on a 15 node cluster system (60 cores in total) on Grid'5000. We consider three management policies of the cluster of interest. These policies are: “on-demand”, “performance”, and “MREEF”. Under on-demand and performance policies, Linux's on-demand and performance governors are respectively enabled on each and every node of the cluster. The MREEF policy corresponds to the configuration wherein MREEF is used.

During this initial experiment, MREEF uses the following techniques among those described earlier herein: the EV-based phase detection mechanism for system phase changes detection, the sensor-based phase characterization technique, and partial phase recognition. Partial phase recognition is used as an alternative means of phase prediction for on-the-fly system reconfiguration. Since all system reconfiguration decisions are directed toward the processor through DVFS, we define three operating levels for the processor according to classes/labels associated with workloads: “high ” for compute intensive workloads, “medium” for memory intensive workloads, and “low” which involves both IO intensive and communication intensive workloads. Note that at the time we only had three workloads categories defined. The “high” operating level is associated with the highest processor's frequency available, i.e., 2.53 GHz on our platform. As opposite to the high operating level, the low operating level operates at the lowest available processor's frequency (1.20 GHz on our platform). The medium operating level on our platform is set to 2.00 GHz.

Figure 7.4(a) depicts the normalised average energy consumption of the overall cluster for each application under the three cluster's configurations. Whereas Fig-

ure 7.4(b) shows their execution time respectively. These results are normalised with respect to the baseline execution (on-demand) and averaged over several executions of each workload under each system configuration. Figure 7.4 indicates that our management framework MREEF consumes in average 15% less energy than "performance" and "on-demand" while offering nearly the same performance. For LU, BT, and SP the average energy gain ranges from 3% to 6%; however, the maximum amount of possible energy savings depends on the workload at hand and was 19% for WRF-ARW.

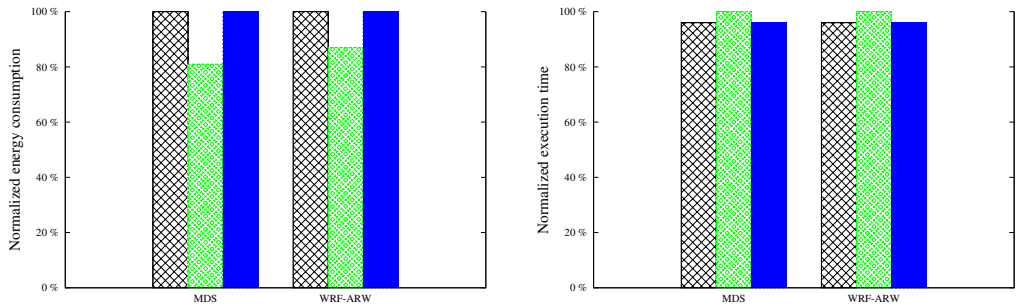
From Figure 7.4(b), we see a performance loss of nearly 3% for LU and less for the others. Performance loss with benchmarks comes from the fact that some phases were wrongly identified as being memory intensive. However, these results are similar to those observed in work using a methodology different from ours [Lively *et al.* 2011]. In addition, these applications do not offer much opportunities for saving energy without degrading performance. Contrarily, the numerical weather forecast model (WRF-ARW) has load imbalance which can help reduce its energy consumption without a significant impact on its performance (in terms of execution time) [Chen *et al.* 2005, Kimura *et al.* 2006].

Above results demonstrate the effectiveness of our methodology for reducing the energy consumption of HPC systems through a centralised version of MREEF. Our approach outperforms a Linux's governor because the Linux's "on-demand" governor will not scale the processor frequency down unless the system's load decreases below a specific threshold. The problem at this point is that the processor's load is in general very high even when running memory intensive codes that do not necessarily require the system's full computational power. In this particular scenario, network and disk intensive phases are too short (from milliseconds to a few seconds) and are often considered as boundaries of memory and compute intensive phases. Hence, the energy reduction mainly comes from scaling the processor's down in phases which according to the phase characterization and partial phase recognition mechanisms are memory intensive.

The overhead resulting from our management is counted both in the program's execution time and energy consumption. From Figure 7.4, we can claim that the overhead of our management policy is almost insignificant. However, a centralised coordinator may bring a network overhead because of messages that are exchanged between the coordinator and reconfiguration decisions enforcers which reside on nodes of the platform.

7.4.1.2 Decentralized coordinator

Section 7.4.1.1 evaluated a centralised implementation of our methodology for reducing the energy consumption of HPC systems. This section evaluates the performance of a decentralised implementation, where each and every node profiles itself in addition to implementing system reconfiguration decisions. For the evaluation, we set up a twenty five node cluster on the Grid'5000 [Bolze *et al.* 2006] French large scale experimental platform. The settings are the same as in the previous experiment,



(a) Average energy consumed by each application under different configurations: decentralized management.

(b) Average execution time of each application under different configurations: decentralized.

performance MREEF on-demand

Figure 7.5: Phase tracking and partial recognition guided processor adaptation results for the decentralized version of MREEF. They are averaged over 20 executions of each workload under each system configuration; they are normalized with respect to baseline execution “on-demand".

i.e., we use the three processor operating levels defined in Section 7.4.1.1 and consider three basic system configurations: “on-demand”, “performance” and “MREEF”. The meaning associated to these system configurations is also the one given in Section 7.4.1.1. However, as test applications, we use MDS and WRF-ARW model. In this section, we do not use NAS Parallel benchmarks because they have low network and disk activities in comparison with MDS and WRF. We further consider two levels of system adaptation:

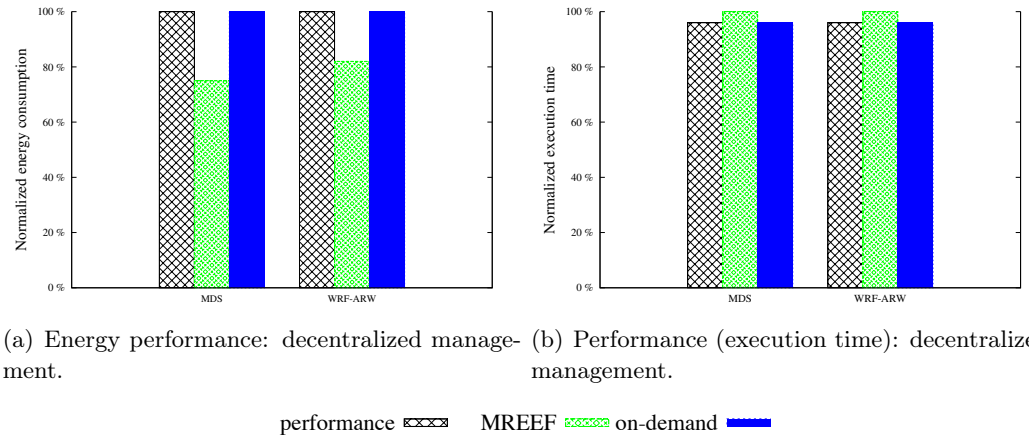
- system adaptation level one: In system adaptation level one, only processor related optimizations are performed, i.e., all system reconfigurations are directed toward the processor through the use of DVFS.
- system adaptation level two: It embraces level one, and additionally considers optimizing the interconnect and the disk.

(a) System adaptation level one: processor’s only optimization

With processor’s only optimization, the processor’s frequency is adjusted to either “high”, “medium”, or “low” according to the system’s workload when partial phase recognition is successful. Figure 7.5 offers an outline of the results in terms of energy consumption (Figure 7.5(a)) and execution time (Figure 7.5(b)). Those graphics indicate that MREEF saves up to 19% of the total energy consumption with less than 4% performance degradation.

(b) System adaptation level two: processor, disk and network optimization

In the level two system adaptation, we attempt to send disks to sleep mode and reduce the interconnect speed when they are suspected not to be in use. Nodes are



(a) Energy performance: decentralized management.

(b) Performance (execution time): decentralized management.

Figure 7.6: Phase tracking and partial recognition guided processor, disk, and network adaptation results for a decentralized version of MREEF. They are averaged over 20 executions of each workload under each system configuration; they are normalized with respect to baseline execution “on-demand".

interconnected via 1Gbps links supporting the following modes: 10baseT Half/Full duplex, 100baseT Half/Full, and 1000baseT/Full. During these experiments the link speed is scaled down to 10baseT Full (10Mbps) when the network interconnect is suspected not to be in use. It is worth mentioning that 1Gbps links consume a few watts more than 100Mbps [Gunaratne *et al.* 2005]. Also, the operating system (OS) is tuned to prevent unnecessary disk accesses regardless of the system configuration (on-demand, performance, MREEF).

Figure 7.6 offers an outline of performance (energy consumption and execution time) under different system configurations. These graphs suggest that considering the processor along with the disk and network interconnect improves energy performance of 24% with nearly the same performance degradation as with processor's only adaptation. In other words, reconfiguring the disk and the network interconnect also contributes in reducing the energy consumption of the system.

Figure 7.4(b), Figure 7.5(b), and Figure 7.6(b) show that “on-demand” and “performance” governors nearly achieve the same performance. This can be attributed to the fact that Linux's on-demand governor does not lower the processor's frequency unless the system's load has decreased below a certain threshold. Figure 7.7 – where the y-axis is the load percentage and the x-axis the execution time-line – offers a graphical representation of load traces for one node of our system participating in the computation of WRF-ARW under the on-demand system configuration. The graphic clearly indicates that the processor's load remains above 85% in which case the on-demand governor have the same behaviour as the performance governor. A similar observation applies to all our workloads; consequently, in the following we only consider two system configurations: the on-demand configuration and the managed configuration. Also, since network speed scaling is not supported by all nodes

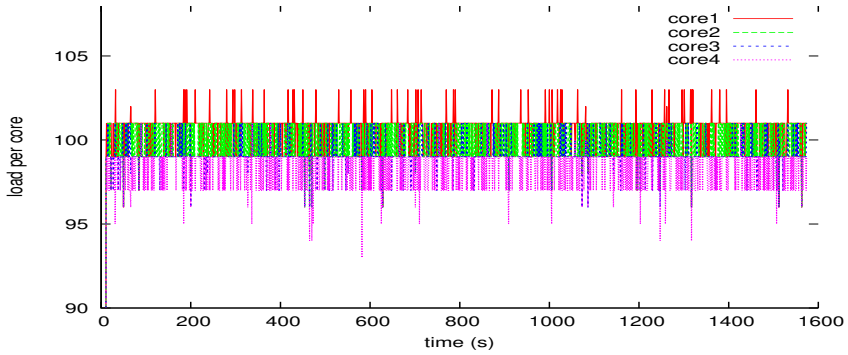


Figure 7.7: Load traces for one a node participating in the computation of WRF-ARW under the on-demand configuration.

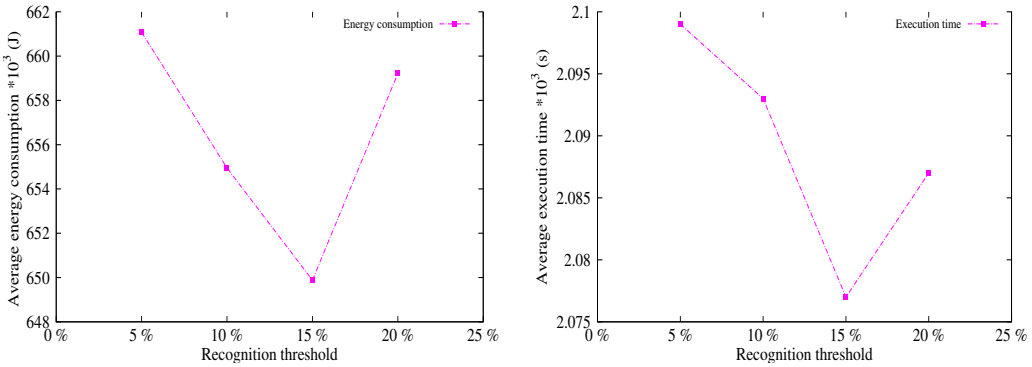
it limits the number of nodes that we can use for the experiments. Moreover, as disk optimisation sometimes require tuning the system – which slightly departs from our promise to provide a user friendly and generic methodology for improving energy performance – we will be focusing on the processor subsystem. In the following, unless expressly stated otherwise, we use the decentralised implementation of our methodology for reducing the energy consumption of HPC systems.

7.4.1.3 Recognition threshold selection and energy performance

Thus far, we have been using a 10% recognition threshold, in this section, we investigate how that parameter influences the system's power usage using WRF-ARW as test application. Recall that an X% recognition threshold means that an ongoing phase is recognised as a known phase if the already executed part of the ongoing phase matches with X% (of the duration) of a known phase. In this section, we investigate the influence of the recognition threshold on applications' performance and energy performance. In order to achieve this aim, we execute WRF-ARW on a 25 node cluster multiple times while varying the partial recognition threshold.

Figure 7.8, where the x-axis represents the recognition threshold and the y-axis either the average energy consumption (Figure 7.8(a)) or the average execution time (Figure 7.8(b)) summarises the impact of the partial recognition threshold on both the execution time and the energy consumption of WRF-ARW. According to Figure 7.8, a partial phase recognition threshold of 15% is more effective both in terms of energy consumption and execution time for WRF-ARW.

Overall, the partial recognition threshold depends on the target objective; it may also depend on the application at hand. The partial recognition threshold has and impact on adequate reconfiguration as well as inadequate reconfiguration decisions. In fact, if adequate decisions are made earlier enough, one can save more energy; inversely, making a wrong decision earlier in the execution of a phase can result in significant energy waste and/or performance degradation.



(a) Energy performance.

(b) Performance (execution time).

Figure 7.8: Influence of the partial phase recognition threshold on WRF-ARW's performance (execution time and energy consumption).

7.4.1.4 Performance analysis on a shared platform

Our approach being system oriented, it can equally be used on shared infrastructures; that capability is investigated in this section. To accomplish this, we consider two test applications, which simultaneously share a 24 node cluster system, the numerical whether research and forecasting model WRF-ARW [Skamarock *et al.* 2005] and Block Tri-diagonal solver (BT) [Bailey *et al.* 1991] from NAS parallel benchmark. The number of nodes is limited to 24 because we want them to equally share the platform. Hence, each application runs on 12 nodes and spans 48 processes. As for previous experiments, we execute those applications multiple times and compute their average power consumption and execution time. Figure 7.9, where the x-axis represents the percentage of energy consumption or execution time compares our management policy where MREEF is used with the baseline on-demand configuration. In Figure 7.9, a positive value along the y-axis indicates in proportion the increase in energy consumption or program execution time resulting from the use of MREEF. Conversely, a negative value along the y-axis indicates in proportion the reduction in energy consumption or program execution time resulting from the use of MREEF. The same goes for Figure 7.10, and Figure 7.11.

Although BT shows relatively bad performance (for the same reasons that we discussed earlier herein) in terms of execution time, our management policy still offers good results.

7.4.1.5 Overhead analysis

Reusing known phases often require storing optimal configurations along with an instance of those recurring phases for future utilisation. Consequently; one may be interested in knowing how much disk space is used for that purpose. Basically, disk space required for storing known phases is proportional to their number i.e., if there is a thousand of known phases, one will probably store all of them. Using

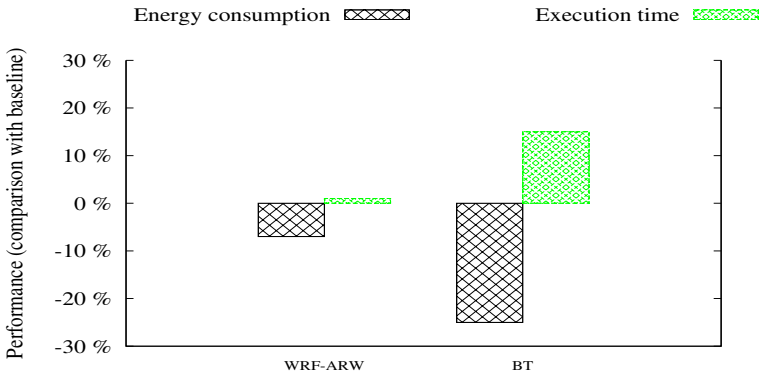


Figure 7.9: Comparison of our management policy in terms of programs' energy consumption and execution time with baseline on-demand system configuration.

HPC workloads, we have defined earlier herein six categories/classes of workloads *(compute-intensive, memory-intensive, mixed, IO-intensive, network-transmit, and network-receive)* into which we attempt to classify all the phases we detect on our system. Ideally, we will have at most 6 phases stored because we do not keep idle phases. Note that by storing a phase we mean storing its reference and representative execution vectors along with the average distance of all EVs to the reference vector of the phase and its length (duration) for partial phase recognition if needed.

In practice, there can be more than six phases stored; this is explained by the fact that there is no exact match between two phases. We use the off-line phase identification scheme (Section 6.2.3) to limit the number of stored phases. Throughout our experiments, which last on average 72 hours, (this holds for all the experiments we present in this chapter), we store at most 10 phases per node, which is insignificant. Consequently, those information are kept in memory at runtime.

7.4.2 Execution vectors' classification related results

Section 7.4.1 demonstrates the effectiveness of the proposed MREEF.The evaluated implementation relies upon partial phase recognition and the sensor-based phase characterization scheme (Section 5.3.1) to enable reuse of optimal configuration information for recurring phases. Although effective, it might suffer from the partial recognition threshold selection and can often lead to significant performance degradation. This section presents a modified version of MREEF; it takes the partial phase recognition threshold selection away from users hands by using the EV classification mechanism. Instead of predicting an entire phase as partial phase recognition does, it predicts the behaviour of the system for the next time unit (a second to be more precise) through EVs classification and the use of a principle widely exploited by caching algorithms (see Section 6.2.2 for details).

In the following, experiments we present either use the Last Level Cache per Instruction Ratio (LLCRIR) based phase characterization scheme or the majority-rule-based phase characterization algorithm for phase characterization instead of the

sensor-based phase characterization scheme.

7.4.2.1 Last level cache references per instruction ratio related results

In experiments whose results are presented in this section, we are more interested in knowing the long run behaviour of MREEF. To accomplish this, we consider a 28 node cluster system made up on Grid'5000. At this time all workloads categories/classes *(compute-intensive, memory-intensive, mixed, IO-intensive, network-transmit, and network-receive)* defined in Chapter 5 apply, but in this specific experiment *mixed* and *compute-intensive* workloads are treated alike. Treating mixed and compute intensive workloads differently from memory intensive workloads, permits us to easily notice the impact of reconfiguration decisions taken through MREEF. As discussed earlier, the processor is the only subsystem that is reconfigured. Workloads labelled either as compute-intensive or mixed are executed at the highest available processor's frequency (2.53 GHz on our cluster), while those labelled as memory-intensive are executed at a lower frequency (2.27 GHz) and all other at the lowest available frequency (1.20 GHz). Our test workloads include Block Tri-diagonal solver (BT), Conjugate Gradient (CG), Integer Sort (IS), Scalar Penta-diagonal solver (SP), and Embarrassingly Parallel (EP) from NAS Parallel Benchmark suite [Bailey *et al.* 1991].

We next randomly run workloads just listed 55 times each (nearly 72 hours in total without any idle period) while letting MREEF determine by itself the adequate label through setting the appropriate processor frequency at which each type of workload should be executed. Each node makes its own decisions regardless of the others.

Initially, the processor's frequency on each and every node is the highest available. This means that the first instance of each workload is always considered as being compute intensive because at that time its execution pattern is unknown to the management framework MREEF. Although treating first instances of workloads as compute intensive is arguable, we believe that in doing so we guarantee a certain quality of service to workloads that are actually compute intensive. As the first instance of a workload is treated as compute intensive (corresponds in practice to being executed at the highest frequency), it also serves as a reference execution for performance comparison (energy consumption and execution time reduction/increase) and is referred to as the baseline execution.

Figure 7.10 offers an outline of the difference between the energy consumption (respectively the execution time) of the baseline execution of each workload and the average energy consumption (respectively the average execution time) of all other instances of the corresponding workload. For compute intensive (EP) and mixed (BT and SP) workloads the difference is insignificant, meaning that their energy consumption as well as their execution time is nearly constant among instances. However, we can observe a gab, up to 14% difference between the energy consumption of the baseline execution and the average energy consumption of other instances for memory intensive workloads such as CG and IS. This comes at the expense of

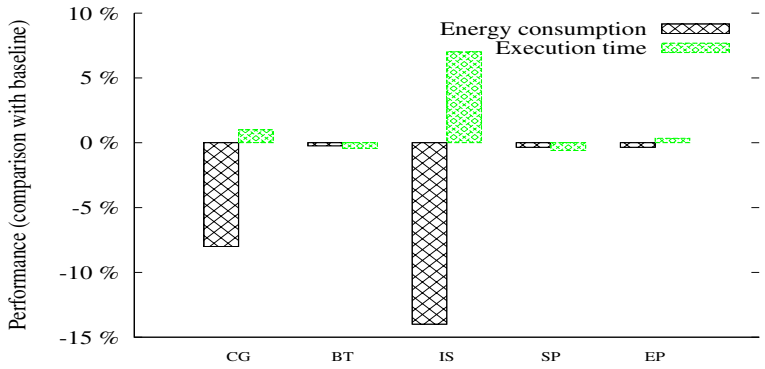


Figure 7.10: Average energy variations and execution time increase with respect to the baseline execution.

increased execution time; however, performance degradation is less than 7%.

What we can learn from this experiment is that even in the long run, our management policy can effectively take advantage of workload variability in a high performance computing system for reducing its power consumption. More interestingly, it can distinguish relevant classes of workloads without any information about actual workloads and any human intervention.

7.4.2.2 Majority-rule-based phase characterization related results

In the previous section, we have shown that even at long run, MREEF performs well. However, our target objective remains reducing the energy consumption while offering "good" performance over a wide range of applications. This section still evaluates our implementation of the methodology for reducing the energy consumption of HPC systems: MREEF. However, this other version differs from the previous in that it uses the majority-rule-based algorithm for phase characterization. In the evaluation presented here, a 34 node cluster system (136 cores in total) set up on the French large-scale experimental platform Grid'5000 is shared by multiple workloads. Class B problem set of benchmarks (CG and MG) along with Molecular Dynamics Simulation (MDS), the Advance Research Weather Research and Forecasting (WRF-ARW) model, Parallel Ocean Program (POP) X1 benchmark, and GeneHunter are used as test workloads.

Due to their significantly long execution time, MDS and WRF-ARW are executed on 25 nodes (100 cores), NAS benchmarks on 9 nodes (36 cores in total), and POP X1 and GeneHunter on 4 nodes (16 cores). Some applications are launched simultaneously when available resources can accommodate them. This is because WRF-ARW and MDS run on 25 nodes and they cannot be run at the same time. So in parallel with one of them either NAS Parallel Benchmarks, POP X1, or GeneHunter are executed. As for previous experiments, we compare our management policy with Linux's on-demand system management policy.

Figure 7.11 compares our management policy to the "baseline" execution, in

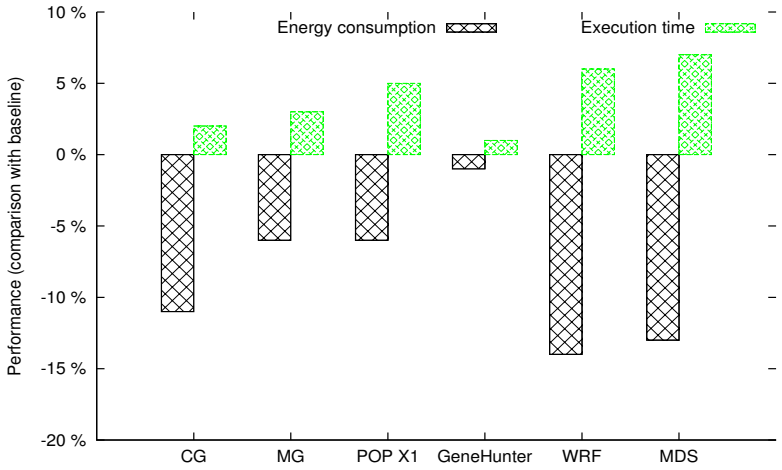


Figure 7.11: Comparison of MREEF (execution time and energy consumption) with the baseline on-demand configuration.

which all nodes are managed by the Linux on-demand governor. As expected, performance degradation depends on the workload at hand, but remains bellow 7%. We notice an energy improvement of up to 15% in comparison with the baseline execution.

Still from Figure 7.11, we can notice that the energy savings for WRF-ARW and MDS are slightly smaller than those in Section 7.4.1.2, that make sense because processor's operating frequencies are not the same in the two experiments. For experiments presented in this section, workloads belonging to the memory-intensive and mixed classes (they are labelled respectively as memory-intensive and mixed) are executed at 2.13GHz and 1.87GHz respectively. While compute-intensive and all other classes are always executed at the highest and lowest available processor frequencies respectively. The rationale behind using multiple and different frequencies is to show that although there might be opportunities for reducing the processor's frequency to save energy, one should not forget that we are in high performance computing environments. Roughly speaking some frequencies might not be appropriate for certain type of workloads.

Figure 7.11 also confirms our assumption that memory-intensive code offers much more energy savings opportunities than compute-intensive or mixed code (MG, POP X1 and GeneHunter). Note in passing that POP's performance also relies on the network[5]. Table 7.1 where rsd. stands for Relative Standard Deviation reveals negligible variabilities in achieved results (execution time and energy consumption), which demonstrates the consistency of our multi-resource energy efficient framework (MREEF).

[5]Further details can be found in its documentation at http://climate.lanl.gov/Models/POP

Table 7.1: Relative standard deviation (rsd.) of the energy consumption and execution time of each workload under our two system configurations.

Workloads	Baseline configuration		MREEF configuration	
	rsd. execution time (%)	rsd. energy consumption (%)	rsd. execution time (%)	rsd. energy consumption (%)
CG	3	2	0.51	4
MG	0.2	12	2	13
POP	2	2	0.53	2
GeneHunter	0.1	0.1	0.12	1
WRF	0.88	0.80	1	1
MDS	0.19	0.52	0.63	0.56

7.5 Extension to cloud environments

HPC were historically known to suffer from performance degradation in cloud deployments, especially those that use virtualisation technologies. However, improvements in virtualisation technologies have significantly reduced the performance gab between physical and virtual deployments. It is therefore not surprising that HPC users are shifting some workloads to cloud in order to benefit from flexibility, cost efficiencies and improved resource sharing that cloud provides. For example, the XLcloud project objective is to propose tools that facilitates HPC in cloud deployments[6]. Despite its cost efficiency, cloud computing suffers from multiple challenging issues including security and power consumption just to name a few. Previous sections demonstrate the effectiveness of our MREEF in "standard" HPC deployments (physical deployments). This section investigates its performance in cloud environment including HPC in cloud deployments.

7.5.1 Context description and results

Dynamic resource allocation in virtualised systems can be combined with Dynamic Voltage Scaling (DVS) or more generally DVFS for energy reduction purposes [Kim *et al.* 2009, von Laszewski *et al.* 2009, Rodero *et al.* 2010]. At its creation, a Virtual Machine (VM) is provided with a certain processing capacity. However, as the workload may vary in time, VM reconfiguration based on monitoring and workload prediction can reduce the power wasted due to workload variability. Workload prediction often follows basic models (request arrival model for web based services for example) and also requires thorough understanding of the service being offered by the virtual machine i.e., requires knowledge of the application being executed in the VM. This rarely reflects the reality, in fact, datacenter operators are often not given the right to look into the actual content of VMs deployed on their infrastructure. The system's load can be used as an indicator for DVFS adaptation, but VMs demands within the same physical node may often be contradictory, requiring more complex analysis before using the processor's load.

[6] http://xlcloud.org

In this section, we address the energy consumption issue in a cloud environment where physical nodes may host VMs with contradictory demands (in terms of processing capabilities). And within which the cloud infrastructure holder is not given the right to look into the content of deployed VMs for energy efficient decision making. Roughly speaking, he/she does not have any information about VMs actual contents. In summary, our scenario boils down to investigating DVFS use in cloud environment when the platform holder lacks information about deployed VMs. To accomplish this, we deploy 8 virtual machines on a single compute node with 8 CPU cores. Each virtual machine as well as the physical host operates under the Linux kernel 2.6.35.

We next randomly run each and every one of the following workloads 20 times in different system configurations (nearly 72 hours for each system configuration): (1) a transactional database system emulated through sysbench benchmark[7] and MySQL[8]; (2) a web application emulated through siege benchmark tool[9] and Apache HTTP server [10], (3) Conjugate Gradient (CG) from NAS Parallel Benchmark suite; and (4) an application that performs intensive IO operations (read and write) using IOzone[11]. Among these workloads, CG reflects high performance computing workloads, while the others are representative of cloud workloads.

Initially, to ensure that the workload of a VM does not change during the experiments, we draw a random execution order of these applications in each VM. The draw is performed in such a way that each and every application is exactly executed 20 times in each VM. By randomly selecting workloads, we simulate the behaviour of a production system in the sense that at a given point in time, services offered by VMs hosted on the same node may be of different types.

We further consider three system configurations: (i) *on-demand*, the configuration wherein the operating frequency of the processor is set to the maximum available; (ii) *powersave*, the configuration wherein the operating frequency of the processor is set to the minimum available; and (iii) MREEF, the configuration where it is used to detect phases of execution of the system, characterize them and adapt the processor's operating frequency accordingly. During these experiments, MREEF uses the majority-rule-based phase characterization algorithm for phase characterization and on-line EVs classification for workload prediction.

Figure 7.12, where we compare performance of MREEF with *on-demand* and *powersave* system configurations, offers an outline of our most recent results.

The comparison of MREEF with *powersave* in this case is motivated by the fact that cloud workloads are traditionally not processor hungry. Figure 7.12 indicates that the energy consumption of MREEF is nearly 8% less than that of *on-demand* with nearly no performance degradation. We can also observe that MREEF outperforms the *powersave* system configuration since it does not only improve the

[7]Sysbench: http://sysbench.sourceforge.net
[8]MySQL: http://www.mysql.com
[9]Siege: http://www.joedog.org/siege-home
[10]Apache: http://httpd.apache.org
[11]IOzone: http://www.iozone.org

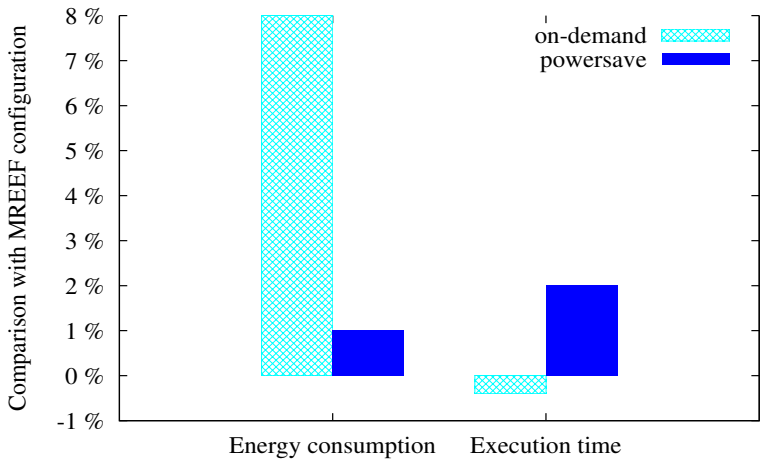


Figure 7.12: MREEF versus powersave and on-demand in a cloud environment.

execution time, but it also consumes less energy. These recent results show that MREEF is a potential solution to the energy consumption problem that cloud HPC systems (cloud systems accommodating HPC workloads) are likely to face. In fact, when provided with MREEF, cloud systems that are over provisioned to accommodate HPC applications will experience less energy inefficiency when the type of workload changes from HPC to cloud and vice versa.

7.6 Conclusions

In this chapter, we presented Multi-Resource Energy Efficient Framework (MREEF), an implementation of our methodology for reducing the energy consumption of HPC systems. We demonstrate how our methodology can be used to effectively reduce the power consumption of real life HPC systems. We considered implementations of MREEF that use different architectures (centralised and decentralised) and implemented stages of the energy reduction methodology differently. The point in implementing stages of the methodology differently is not limited to providing a simple, effective, and user friendly (no specific knowledge needed) implementation. However, it implicitly emphasises that the methodology only provides guidelines for reducing the energy consumption. We also show that in addressing the energy consumption problem in HPC one can go beyond processor frequency scaling to fully take advantage of power saving opportunities offered by other HPC subsystem (disk, network interconnects). Note that in Chapter 6 we showed that provided the right technology, one can equally relies upon the memory subsystem to reduce energy consumption of the overall system.

We conducted experiments considering workloads representative of HPC systems and HPC clusters of varied sizes. Comparison of MREEF's performance with baseline unmanaged execution shows that MREEF can reduce the energy consumption of up to 24% with less than 7% performance degradation with real life workloads.

More interestingly, this is achieved without a priori information about workloads being executed. Results analysis show that MREEF is consistent over time and has little overhead on system's performance (Figure 7.10 and Table 7.1). They also indicate that *compute intensive* and *mixed* workloads may not offer as much energy reduction opportunities as *memory intensive, IO, and communication intensive* workloads. MREEF offers better energy performance when the infrastructure is executing applications which can be broken into execution phases of different types. Experiments on HPC clusters of varied sizes show that, in addition of being completely automated, MREEF can easily be extended to power-aware clusters of larger size for optimizing their energy consumption; given that it does not require any a priori knowledge of applications sharing the infrastructure.

Although initially designed for HPC systems, we show through experiments that MREEF or more generally our methodology for reducing the energy consumption of HPC systems is actually not limited to HPC systems. Results in cloud environments are both encouraging and convincing.

Chapter 8

Conclusions and Perspectives

Contents

This chapter concludes work presented in this document. The first section summarises the main contributions of this thesis, while the second presents future directions.

8.1 Conclusions

The main objective of this thesis was to propose a software approach that combines system profiling and green capabilities for improving the energy efficiency of large scale distributed systems. To achieve that objective, we focused on High Performance Computing (HPC) systems that we modelled as a set of computing and storage nodes/systems, excluding network equipments due to their usually constant power consumption.

Firstly, we reviewed the state of the art on techniques for reducing the energy consumption of HPC along with their limitations. Secondly, we proposed a methodology for reducing the energy consumption in the form of three complementary modules or steps including (i) phase detection, (ii) phase characterization, and (iii) phase identification and system adaptation/reconfiguration.

The proposed methodology offers, among other benefits, the advantage that it is independent from any individual application as it examines runtime execution patterns of a system, instead of considering the individual applications sharing the platform. It then allows on-line reconfiguration of subsystems, including processor, memory, storage and communication to meet users' requirements while reducing the overall energy consumption of the computing infrastructure. As the methodology focuses on the system, it does not need any a priori information about applications sharing the computing infrastructure, and can be used by non-experts without great effort.

At phase detection level, we proposed and evaluated two phase-detection methodologies. The first methodology, which we refer to as "*power-based*", detects phase changes in the runtime behaviour of a system relying on its power consumption. The power-based approach sometimes fails because of inaccuracy in power

measurements, hence rendering it difficult to on-line use. The second approach called "EV-based" is an on-line phase detection mechanism that works with the concept of Execution Vector (EV). The EV-based phase detection approach suits best our needs because it is on-line, accurate for both program and system phase detection, and has a negligible overhead. As the concept of phases is relative, the EV-based phase detection, similar to techniques presented in Section 2.3, uses a detection threshold to determine phase changes.

For phase characterisation, we propose three workload/phase characterisation schemes which are: *"sensor-based"*, last level cache references per instructions based (*"LLCRIR-based"*), and principal component analysis based (*"PCA-based"*) phase characterisation techniques. Relying on these phase characterisation schemes, we propose two phase characterisation algorithms: the first basically applies the PCA-based phase characterisation techniques and then the LLCRIR if the PCA-based fails to characterise the phase. The second is a majority-rule-based algorithm; it is majority-rule-based because to a given phase it applies three different phase characterisation schemes and uses the majority rule to determine the appropriate. Although those two phase characterization algorithms nearly have the same characterization accuracy, we use the majority-rule-based algorithm because it can easily take advantage of any improvements in one of our three characterization schemes.

Finally, at the phase identification and system reconfiguration stage, we proposed two on-line phase identification and prediction techniques and an off-line phase identification. The proposed on-line phase identification techniques including "partial phase recognition" and "on-line EV classification" also serve as alternative means to phase prediction. They differ from existing phase prediction techniques in two ways: (i) they do not require any knowledge from the workload being executed and (ii) can predict the upcoming behaviour of a system running multiple workloads. The main drawback of partial phase recognition lies on the fact that users must select the partial recognition threshold. On-line EV classification mitigates user intervention by taking the selection of the partial recognition threshold away from users hands. The off-line phase identification permits to identify phases that have completed. It is used for preventing unnecessary phase characterisation and redundancies among stored phases. Nevertheless, it can also be used in a program simulation context. We also proposed a set of non-conventional power saving schemes and investigated their usefulness in real-life systems. Those include: (a) platform selection through energy prediction, (b) memory size scaling, and (c) CPU core switch on/off.

We proposed a software framework called Multi-Resource Energy Efficient Framework (MREEF) as an implementation of our methodology for reducing the energy consumption of large-scale and distributed systems. It is completely automated, coordinated, fully scalable, and user friendly (easy to use and does not require any a priori knowledge from the users) as suggested by the energy reduction methodology. We demonstrated the effectiveness of MREEF through real life experiments on the French large-scale experimental platform Grid'5000 using real life HPC applications and benchmarks generally accepted by the HPC community. Comparison with baseline execution reveals that MREEF reduces the energy consumption of

the overall infrastructure to up to 24% with less than 7% performance degradation for real life applications. We also conducted experiments through which we showed that MREEF is consistent over time. This means that it accurately identifies variabilities in HPC workloads and takes management decisions accordingly. We also demonstrated that MREEF fully takes advantage of all power aware technologies available to HPC subsystems from the processor to the memory to storage and communication subsystems. Cloud HPC environments are gaining space, MREEF also responds to energy consumption challenges they face; results of its evaluation in cloud HPC environments showed that it fully takes advantage of workloads variability in cloud environments as well as HPC systems.

8.2 Future directions

While in this thesis we focused on proposing a general purpose methodology for reducing the energy consumption of HPC systems, future work is needed to extend the methodology towards a performance-oriented management framework. The added value of such a framework would be the fact that instead of reconfiguring the system through power saving schemes, one will make performance optimisation decisions. Proposed algorithms and techniques can potentially be applied to other type of environments. For example, the last contribution of Chapter 7 investigated their use in cloud HPC environments or more generally cloud environments. Nevertheless, more experiments and developments still need to be conducted in cloud environments. In fact, the extension to cloud environments opens the way to new power saving schemes such as Virtual Machine (VM) migration and/or consolidation.

A venue of future research concerns the introduction of a feedback mechanism. A feedback mechanism along with the energy prediction methodology can help determine whether a power saving scheme will be effective before actually applying that power saving scheme to the system. For example, one may be interested in knowing whether scaling the processor down will actually reduce the systems' energy consumption. Speaking of power saving schemes, a future research direction can investigate how power saving schemes such as memory size scaling can be supported in today's systems. That would require addressing the following points among others:

- Appropriate selection of RAM (Random Access Memory) blocks where the data gets stored such that unused RAM can be turned off without compromising on the timing performance;
- Load prediction and time taken to turn on and use the RAM.

Phase detection often serves as a start point for program simulation since it can be used to identify simulation points. We propose the EV-based phase detection methodology in Chapter 4 and show how it can be used for selecting program simulation points. Further research efforts can be directed toward program simulation to investigate the use of EV for program simulation. One of the basic questions will

consist of finding out how accurate is the estimated result from selected simulation points relative to the full simulation result.

Publications

[1] G. L. Tsafack, L. Lefèvre, and J.-P. Gelas, "Dtn based management framework for green on/off networks," in *RESCOM 2011*, 2011.

[2] G. L. Tsafack, L. Lefèvre, and J.-P. Gelas, "On applying dtns to a delay constrained scenario in wired networks," in *14th International Symposium on Wireless Personal Multimedia Communications, WPMC 2011*, pp. 1–5, IEEE, Oct 2011.

[3] G. L. Tsafack, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "Dna-inspired scheme for building the energy profile of hpc systems," in *Proceedings of the First international conference on Energy Efficient Data Centers*, E2DC'12, (Berlin, Heidelberg), pp. 141–152, Springer-Verlag, 2012.

[4] G. L. Tsafack, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "A runtime framework for energy efficient hpc systems without a priori knowledge of applications," in *ICPADS 2012 : 18th International Conference on Parallel and Distributed Systems*, (Singapore, Singapore), pp. 660–667, IEEE, Dec 2012. (Cited on page 6.)

[5] G. L. Tsafack, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "Beyond cpu frequency scaling for a fine-grained energy control of hpc systems," in *SBAC-PAD 2012 : 24th International Symposium on Computer Architecture and High Performance Computing*, (New York City, USA), pp. 132–138, IEEE, oct 2012. (Cited on page 6.)

[6] M. el Mehdi Diouri, G. L. Tsafack, O. Glück, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "Energy efficiency in high-performance computing with and without knowledge of applications and services," *IJHPCA*, vol. 27, no. 3, pp. 232–243, 2013.

[7] G. L. Tsafack, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "Exploiting performance counters to predict and improve energy performance of {HPC} systems," *Future Generation Computer Systems*, 2013.

[8] G. L. Tsafack, L. Lefèvre, and P. Stolf, "A three step blind approach for improving hpc systems' energy performance," in *EE-LSDS 2013 : Energy Efficiency in Large Scale Distributed Systems conference*, (Vienna, Austria), Springer, april 2013. (Cited on pages 6 and 23.)

[9] G. L. Tsafack, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "A user friendly phase detection methodology for hpc systems' analysis," in *GreenCom 2013 : The 2013 IEEE International Conference on Green Computing and Communications*, (Beijing, China), IEEE, Aug 2013. (Cited on pages 6 and 25.)

[10] B. Robert, G. Da Costa, G. L. Tsafack, L. Lefèvre, A. Oleksiak, and J.-M. Pierson, *High-Performance Computing on Complex Environments*, ch. Energy Aware Approaches for HPC Systems. Wiley, 2014. to appear.

Bibliography

[Bailey *et al.* 1991] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan and S. K. Weeratunga. *The nas parallel benchmarks.* Rapport technique, The International Journal of Supercomputer Applications, 1991. (Cited on pages 31, 36, 49, 58, 69, 80, 87 and 89.)

[Balasubramonian *et al.* 2000] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu and Sandhya Dwarkadas. *Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures.* In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, MICRO 33, pages 245–257, New York, NY, USA, 2000. ACM. (Cited on pages 19, 27 and 39.)

[Basmadjian & De Meer 2012] R. Basmadjian and H. De Meer. *Evaluating and modeling power consumption of multi-core processors.* In Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on, pages 1–10, 2012. (Cited on page 12.)

[Bellosa 2000] Frank Bellosa. *The benefits of event: driven energy accounting in power-sensitive systems.* In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system, EW 9, pages 37–42, New York, NY, USA, 2000. ACM. (Cited on page 14.)

[Bhattacharjee & Martonosi 2009] Abhishek Bhattacharjee and Margaret Martonosi. *Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors.* In Proceedings of the 36th annual international symposium on Computer architecture, ISCA '09, pages 290–301, New York, NY, USA, 2009. ACM. (Cited on page 13.)

[Bijl 1919] Van Der Bijl. *Theory And Operating Characteristics Of The Thermionic Amplifier.* Proceedings of the Institute of Radio Engineers, vol. 7, no. 2, pages 97–128, 1919. (Cited on page 12.)

[Binder *et al.* 2004] Kurt Binder, Jürgen Horbach, Walter Kob, Wolfgang Paul and Fathollah Varnik. *Molecular dynamics simulations.* Journal of Physics: Condensed Matter, vol. 16, no. 5, page S429, 2004. (Cited on page 80.)

[Bolze *et al.* 2006] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi and Iréa

Touche. *Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed.* Int. J. High Perform. Comput. Appl., vol. 20, no. 4, pages 481–494, November 2006. (Cited on pages 71, 78 and 83.)

[Cameron *et al.* 2005] Kirk W. Cameron, Rong Ge and Xizhou Feng. *High-Performance, Power-Aware Distributed Computing for Scientific Applications.* Computer, vol. 38, no. 11, pages 40–47, November 2005. (Cited on page 17.)

[Casas *et al.* 2007] Marc Casas, Rosa M. Badia, Jesús Labarta, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, B. Mohr, F. Peters (eds, Marc Casas, Rosa M. Badia and Jesús Labarta. *Automatic phase detection of MPI applications.* In In: Proceedings of the 14th Conference on Parallel Computing (ParCo 2007), Aachen and Juelich, 2007. (Cited on pages 19 and 20.)

[Chen *et al.* 2005] Guangyu Chen, Konrad Malkowski, Mahmut T. Kandemir and Padma Raghavan. *Reducing Power with Performance Constraints for Parallel Sparse Applications.* In IPDPS. IEEE Computer Society, 2005. (Cited on page 83.)

[Choi *et al.* 2006] Kihwan Choi, R. Soma and M. Pedram. *Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times.* Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 24, no. 1, pages 18–28, November 2006. (Cited on pages 3 and 18.)

[Conant *et al.* 2002] Gavin Conant, Steve Plimpton, William Old, Andreas Wagner and Pam Fain. *Parallel genehunter: Implementation of a linkage analysis package for distributed-memory architectures.* In In Proceedings of the First IEEE Workshop on High Performance Computational Biology, International Parallel and Distributed Computing Symposium, page electronic., 2002. (Cited on pages 69 and 80.)

[Contreras 2005] Gilberto Contreras. *Power prediction for intel xscale processors using performance monitoring unit events.* In In Proceedings of the International symposium on Low power electronics and design (ISLPED, pages 221–226. ACM Press, 2005. (Cited on page 14.)

[Costa & Hlavacs 2010] Georges Da Costa and Helmut Hlavacs. *Methodology of measurement for energy consumption of applications.* In GRID, pages 290–297. IEEE, 2010. (Cited on page 15.)

[Dhodapkar & Smith 2002a] Ashutosh S. Dhodapkar and James E. Smith. *Dynamic microarchitecture adaptation via co-designed virtual machines.* In Intl. Solid-State Circuits Conference, Digest of Technical Papers, ISSCC '02, pages 198–199, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on page 19.)

[Dhodapkar & Smith 2002b] Ashutosh S. Dhodapkar and James E. Smith. *Managing multi-configuration hardware via dynamic working set analysis*. In Proceedings of the 29th annual international symposium on Computer architecture, ISCA '02, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on page 19.)

[Dias de Assuncao *et al.* 2010] Marcos Dias de Assuncao, Jean-Patrick Gelas, Laurent Lefèvre and Anne-Cécile Orgerie. *The Green Grid5000: Instrumenting a Grid with Energy Sensors*. In 5th International Workshop on Distributed Cooperative Laboratories: Instrumenting the Grid (INGRID 2010), Poznan, Poland, May 2010. (Cited on page 79.)

[Economou *et al.* 2006] Dimitris Economou, Suzanne Rivoire and Christos Kozyrakis. *Full-system power analysis and modeling for server environments*. In In Workshop on Modeling Benchmarking and Simulation (MOBS), 2006. (Cited on page 14.)

[Freeh & Lowenthal 2005] Vincent W. Freeh and David 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, PPoPP '05, pages 164–173, New York, NY, USA, 2005. ACM. (Cited on page 17.)

[Fürlinger & Moore 2008] Karl Fürlinger and Shirley Moore. *Detection and Analysis of Iterative Behavior in Parallel Applications*. In Marian Bubak, G. Dick van Albada, Jack Dongarra and Peter M. A. Sloot, editeurs, ICCS (3), volume 5103 of *Lecture Notes in Computer Science*, pages 261–267. Springer, 2008. (Cited on page 20.)

[Ge *et al.* 2005] Rong Ge, Xizhou Feng and Kirk W. Cameron. *Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters*. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, pages 34–, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 3.)

[Ge *et al.* 2007] Rong Ge, Xizhou Feng, Wu chun Feng and Kirk W. Cameron. *CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters*. In ICPP, page 18. IEEE Computer Society, 2007. (Cited on pages 18 and 61.)

[Gunaratne *et al.* 2005] Chamara Gunaratne, Ken Christensen and Bruce Nordman. *Managing energy consumption costs in desktop PCs and LAN switches with proxying, split TCP connections, and scaling of link speed*. Int. J. Netw. Manag., vol. 15, no. 5, pages 297–310, September 2005. (Cited on page 85.)

[Hastie *et al.* 2001] Trevor Hastie, Robert Tibshirani and Jerome Friedman. The elements of statistical learning. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001. (Cited on page 52.)

[Hsu & Kremer 2003] Chung-Hsing Hsu and Ulrich Kremer. *The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction.* In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, PLDI '03, pages 38–48, New York, NY, USA, 2003. ACM. (Cited on page 17.)

[Huang *et al.* 2003] Michael C. Huang, Jose Renau and Josep Torrellas. *Positional adaptation of processors: application to energy reduction.* In Proceedings of the 30th annual international symposium on Computer architecture, ISCA '03, pages 157–168, New York, NY, USA, 2003. ACM. (Cited on pages 19 and 27.)

[Intel 1996] Intel. Pentium pro processor user's manual, volume 1–3. INTEL, Santa Clara, California , USA, 1996. (Cited on page 40.)

[INTEL 2003] INTEL. *Intel XScale Microarchitecture for the PXA255 Processor: User's Manual Intel Corporation*, 2003. (Cited on page 14.)

[Isci & Martonosi 2003a] 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, 2003. (Cited on page 20.)

[Isci & Martonosi 2003b] Canturk Isci and Margaret Martonosi. *Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data.* In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 14.)

[Isci *et al.* 2006] Canturk Isci, Gilberto Contreras and Margaret 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 Microarchitecture, MICRO 39, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on pages 18 and 31.)

[ITU 2009] ITU. *background paper for the ITU Symposium on ICTs and Climate Change*, jul 2009. (Cited on page 4.)

[Joseph & Martonosi 2001] Russ Joseph and Margaret Martonosi. *Run-time power estimation in high performance microprocessors.* In Proceedings of the 2001 international symposium on Low power electronics and design, ISLPED '01, pages 135–140, New York, NY, USA, 2001. ACM. (Cited on page 14.)

[Kadayif *et al.* 2001] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykirsnan, M. J. Irwin and A. Sivasubramaniam. *vEC: virtual energy counters.* In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for

software tools and engineering, PASTE '01, pages 28–31, New York, NY, USA, 2001. ACM. (Cited on page 14.)

[Kappiah *et al.* 2005] N. Kappiah, Vincent W. Freeh and D.K. Lowenthal. *Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs.* In Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, pages 33–33, 2005. (Cited on pages 3, 17 and 18.)

[Kim *et al.* 2009] Kyong Hoon Kim, Anton Beloglazov and Rajkumar Buyya. *Power-aware provisioning of Cloud resources for real-time services.* In Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science, MGC '09, pages 1:1–1:6, New York, NY, USA, 2009. ACM. (Cited on page 92.)

[Kimura *et al.* 2006] Hideaki Kimura, Mitsuhisa Sato, Yoshihiko Hotta, Taisuke Boku and Daisuke Takahashi. *Emprical study on Reducing Energy of Parallel Programs using Slack Reclamation by DVFS in a Power-scalable High Performance Cluster.* In CLUSTER. IEEE, 2006. (Cited on page 83.)

[Kimura *et al.* 2010] Hideaki Kimura, Takayuki Imada and Mitsuhisa Sato. *Runtime Energy Adaptation with Low-Impact Instrumented Code in a Power-Scalable Cluster System.* In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 378–387, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 3, 17, 27 and 39.)

[Lewis *et al.* 2012] Adam Wade Lewis, Nian-Feng Tzeng and Soumik Ghosh. *Runtime energy consumption estimation for server workloads based on chaotic time-series approximation.* ACM Trans. Archit. Code Optim., vol. 9, no. 3, pages 15:1–15:26, October 2012. (Cited on page 15.)

[Lim *et al.* 2006] Min Yeol Lim, Vincent W. Freeh and David K. Lowenthal. *Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs.* In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, New York, NY, USA, 2006. ACM. (Cited on pages 3, 18, 27 and 39.)

[Liu & Zhu 2010] Yongpeng Liu and Hong Zhu. *A survey of the research on power management techniques for high-performance systems.* Softw. Pract. Exper., vol. 40, no. 11, pages 943–964, October 2010. (Cited on page 4.)

[Lively *et al.* 2011] Charles Lively, Xingfu Wu, Valerie Taylor, Shirley Moore, Hung-Ching Chang and Kirk Cameron. *Energy and performance characteristics of different parallel implementations of scientific applications on multicore systems.* Int. J. High Perform. Comput. Appl., vol. 25, no. 3, pages 342–350, August 2011. (Cited on page 83.)

[Maede & Diffenderfer 2003] Russell L. Maede and Robert Diffenderfer. Foundation of electronics: Circuits and devices (4th ed.). Thomson Delmar Learning, 2003. (Cited on page 11.)

[Merkel & Bellosa 2006] Andreas Merkel and Frank Bellosa. *Balancing Power Consumption in Multiprocessor Systems.* In First ACM SIGOPS EuroSys Conference, Leuven, Belgium, April 18–21 2006. (Cited on page 13.)

[Merritt 2013] Rick Merritt. *Energy efficient Ethernet standard gains traction.* EE Times, July 2013. (Cited on page 16.)

[MIPS 1996] MIPS. Mips r10000 microprocessor user's manual. MIPS Technologies, Inc., Mountain View, California , USA, 1996. (Cited on page 40.)

[Perelman *et al.* 2003] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood and Brad Calder. *Using SimPoint for accurate and efficient simulation.* In Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM. (Cited on page 42.)

[R Core Team 2013] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013. (Cited on page 80.)

[Ratanaworabhan & Burtscher 2008] Paruj Ratanaworabhan and Martin Burtscher. *Program Phase Detection based on Critical Basic Block Transitions.* In Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software, ISPASS '08, pages 11–21, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on page 19.)

[Rodero *et al.* 2010] I. Rodero, J. Jaramillo, A. Quiroz, M. Parashar, F. Guim and S. Poole. *Energy-efficient application-aware online provisioning for virtualized clouds and data centers.* In Proceedings of the International Conference on Green Computing, GREENCOMP '10, pages 31–45, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on page 92.)

[Rountree *et al.* 2009] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh and Tyler 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. (Cited on pages 3 and 17.)

[Sembrant *et al.* 2011] Andreas Sembrant, David Eklov and Erik Hagersten. *Efficient software-based online phase classification.* 2012 IEEE International Symposium on Workload Characterization (IISWC), vol. 0, pages 104–115, 2011. (Cited on page 20.)

[Sembrant *et al.* 2012] Andreas Sembrant, David Black-Schaffer and Erik Hagersten. *Phase behavior in serial and parallel applications.* In IISWC, pages 47–58. IEEE Computer Society, 2012. (Cited on page 20.)

[Sherwood *et al.* 2001] Timothy Sherwood, Erez Perelman and Brad Calder. *Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications.* In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, PACT '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on pages 19 and 39.)

[Sherwood *et al.* 2002] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. *Automatically characterizing large scale program behavior.* SIGARCH Comput. Archit. News, vol. 30, no. 5, pages 45–57, October 2002. (Cited on page 42.)

[Sherwood *et al.* 2003] Timothy Sherwood, Suleyman Sair and Brad Calder. *Phase tracking and prediction.* In Proceedings of the 30th annual international symposium on Computer architecture, ISCA '03, pages 336–349, New York, NY, USA, 2003. ACM. (Cited on pages 19, 39 and 61.)

[Singh *et al.* 2009] Karan Singh, Major Bhadauria and Sally A. McKee. *Real time power estimation and thread scheduling via performance counters.* SIGARCH Comput. Archit. News, vol. 37, pages 46–55, July 2009. (Cited on pages 13 and 14.)

[Skamarock *et al.* 2005] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang and J. G. Powers. *A description of the advanced research WRF version 2.* NCAR Tech Note, vol. NCAR/TN-468+STR, 2005. (Cited on pages 42, 80 and 87.)

[Spiliopoulos *et al.* 2011] V. Spiliopoulos, S. Kaxiras and G. Keramidas. *Green governors: A framework for Continuously Adaptive DVFS.* In Green Computing Conference and Workshops (IGCC), 2011 International, pages 1–8, 2011. (Cited on pages 18 and 61.)

[Spiliopoulos *et al.* 2012] V. Spiliopoulos, A. Sembrant and S. Kaxiras. *PowerSleuth: 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, 2012. (Cited on page 15.)

[SUN 1995] SUN. *The UltraSPARC processor - Technology white paper: The UltraSPARC architecture*, 1995. (Cited on page 14.)

[Tsafack *et al.* 2012a] Ghislain Landry Tsafack, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf and Georges Da Costa. *Beyond CPU Frequency Scaling*

for a Fine-grained Energy Control of HPC Systems. In SBAC-PAD 2012 : 24th International Symposium on Computer Architecture and High Performance Computing, pages 132–138, New York City, USA, oct 2012. IEEE. (Cited on page 6.)

[Tsafack *et al.* 2012b] Ghislain Landry Tsafack, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf and Georges Da Costa. *DNA-inspired scheme for building the energy profile of HPC systems.* In Proceedings of the First international conference on Energy Efficient Data Centers, E2DC'12, pages 141–152, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on page 6.)

[Tsafack *et al.* 2012c] Ghislain Landry Tsafack, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf and Georges Da Costa. *A runtime framework for energy efficient HPC systems without a priori knowledge of applications.* In ICPADS 2012 : 18th International Conference on Parallel and Distributed Systems, pages 660–667, Singapore, Singapore, Dec 2012. IEEE. (Cited on page 6.)

[Tsafack *et al.* 2013a] Ghislain Landry Tsafack, L. Lefévre, J.M. Pierson, P. Stolf and G. Da Costa. *Exploiting performance counters to predict and improve energy performance of {HPC} systems.* Future Generation Computer Systems, 2013. (Cited on page 6.)

[Tsafack *et al.* 2013b] Ghislain Landry Tsafack, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf and Georges Da Costa. *A User Friendly Phase Detection Methodology for HPC Systems' Analysis.* In GreenCom 2013 : The 2013 IEEE International Conference on Green Computing and Communications, Beijing, China, Aug 2013. IEEE. (Cited on pages 6 and 25.)

[Tsafack *et al.* 2013c] Ghislain Landry Tsafack, Laurent Lefevre and Patricia Stolf. *A Three Step Blind Approach for Improving HPC Systems' Energy Performance.* In EE-LSDS 2013 : Energy Efficiency in Large Scale Distributed Systems conference, Vienna, Austria, april 2013. Springer. (Cited on pages 6 and 23.)

[von Laszewski *et al.* 2009] G. von Laszewski, Lizhe Wang, A.J. Younge and Xi He. *Power-aware scheduling of virtual machines in DVFS-enabled clusters.* In Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, pages 1–10, 2009. (Cited on page 92.)

[Weaver 2013] Vincent M. Weaver. *Linux perf_event Features and Overhead.* In The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath 2013, Austin, Texas, 2013. (Cited on page 80.)

[Welbon *et al.* 1994] E.H. Welbon, C. C. Chan-Nui, D.J. Shippy and D. A. Hicks. *The POWER2 performance monitor.* IBM Journal of Research and Development, vol. 38, no. 5, pages 545–554, 1994. (Cited on page 40.)

[Weste & Eshraghian 1985] Neil H. E. Weste and Kamran Eshraghian. Principles of cmos vlsi design: a systems perspective. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985. (Cited on page 11.)

[Wimmer *et al.* 2009] Christian Wimmer, Marcelo Silva Cintra, Michael Bebenita, Mason Chang, Andreas Gal and Michael Franz. *Phase detection using trace compilation.* In Ben Stephenson and Christian W. Probst, editeurs, PPPJ, pages 172–181. ACM, 2009. (Cited on page 20.)

System Profiling and Green Capabilities for Large Scale and Distributed Systems

Nowadays, reducing the energy consumption of large scale and distributed infrastructures has truly become a challenge for both industry and academia. This is corroborated by the many efforts aiming to reduce the energy consumption of those systems. Initiatives for reducing the energy consumption of large scale and distributed infrastructures can without loss of generality be broken into hardware and software initiatives.

Unlike their hardware counterpart, software solutions to the energy reduction problem in large scale and distributed infrastructures hardly result in real deployments. At the one hand, this can be justified by the fact that they are application oriented. At the other hand, their failure can be attributed to their complex nature which often requires vast technical knowledge behind proposed solutions and/or thorough understanding of applications at hand. This restricts their use to a limited number of experts, because users usually lack adequate skills. In addition, although subsystems including the memory are becoming more and more power hungry, current software energy reduction techniques fail to take them into account. This thesis proposes a methodology for reducing the energy consumption of large scale and distributed infrastructures. Broken into three steps known as (i) phase identification, (ii) phase characterization, and (iii) phase identification and system reconfiguration; our methodology abstracts away from any individual applications as it focuses on the infrastructure, which it analyses the runtime behaviour and takes reconfiguration decisions accordingly.

The proposed methodology is implemented and evaluated in high performance computing (HPC) clusters of varied sizes through a Multi-Resource Energy Efficient Framework (MREEF). MREEF implements the proposed energy reduction methodology so as to leave users with the choice of implementing their own system reconfiguration decisions depending on their needs. Experimental results show that our methodology reduces the energy consumption of the overall infrastructure of up to 24% with less than 7% performance degradation. By taking into account all subsystems, our experiments demonstrate that the energy reduction problem in large scale and distributed infrastructures can benefit from more than "the traditional" processor frequency scaling. Experiments in clusters of varied sizes demonstrate that MREEF and therefore our methodology can easily be extended to a large number of energy aware clusters. The extension of MREEF to virtualized environments like cloud shows that the proposed methodology goes beyond HPC systems and can be used in many other computing environments.