UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Rapid runtime power and performance profiling of large scale applications

Caracterización rápida y en tiempo de ejecución de grandes despliegues de aplicaciones

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Juan Carlos Salinas Hilburg

DIRECTORES

José Luis Ayala Rodrigo
Marina Zapater Sancho
José Manuel Moya Fernádez

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA.

TESIS DOCTORAL

Caracterización Rápida y en Tiempo de Ejecución de Grandes Despliegues de Aplicaciones

Rapid Runtime Power and Performance Profiling of Large Scale Applications

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Juan Carlos Salinas Hilburg

DIRECTORES

José Luis Ayala Rodrigo

Marina Zapater Sancho

José Manuel Moya Fernández

Universidad Complutense de Madrid

Facultad de Informática

Caracterización Rápida y en Tiempo de Ejecución de Grandes Despliegues de Aplicaciones.

Rapid Runtime Power and Performance Profiling of Large Scale Applications

AUTHOR:

JUAN CARLOS SALINAS HILBURG

ADVISORS:

Dr. JOSÉ LUIS AYALA RODRIGO

Dr. MARINA ZAPATER SANCHO

Dr. JOSÉ MANUEL MOYA FERNÁNDEZ

Rapid Runtime Power and Performance Profiling of Large Scale Applications

by

Juan Carlos Salinas Hilburg

Thesis submitted to the Universidad Complutense de Madrid in fulfillment of the requirements for the degree of

Doctor en Ingeniería Informática

Facultad de Informática

Advisors:

Professor Dr. José Luis Ayala Rodrigo

Professor Dr. Marina Zapater Sancho

Professor Dr. José Manuel Moya Fernández

Universidad Complutense de Madrid

Madrid

Declaration of authorship

The author, Juan Carlos Salinas Hilburg, hereby declares and confirms that this thesis is entirely the result of the work carried out in the Department of Architecture and Technology of Computing Systems of the School of Computer Science at the Complutense University of Madrid. This thesis contains original contribution by the author unless otherwise indicated.

Juan Carlos Salinas Hilburg,

November 24, 2020

Acknowledgement

I would like to thank my family, friends and coworkers. Their support has been excellent through all these years.

Specially, I would like to thank my advisors. Thanks to them I have learned many things. They are not only great and wise professionals, they have a great human quality which is something that helped me to get through this experience. It has been a pleasure and an honor working with them.

Thank you Jose, Marina and Jose Manuel.

* * *

Financial support

This thesis has been partially founded by a research grant from Complutense University of Madrid and Banco Santander, under grant CT45/15-CT46/15. Furthermore, this work has been partially supported by the following projects: i) the Spanish Ministry of Economy and Competitiveness, under contracts TEC2012-33892, IPT-2012-1041-430000 and RTC-2014-2717-3, ii) the EU (FEDER) and the Spanish MINECO, under grant TIN 2015-65277-R, iii) the EC H2020 MANGO project (GA No. 671668), and iv) by the Spanish MICINN, under grant PID2019-110866RB-I00.

Table of Contents

List of Tables

List of Figures

Abbreviations

BBV	Basic Block Vectors
CFG	Control Flow Graph
CG	Call Graph
CGS	Call Graph Set
CPU	Central Processing Unit
CR	Compression Ratio
DEVS	Discrete EVent Systems
DIMM	Dual In-line Memory Module
DVFS	Dynamic Voltage and Frequency Scaling
ETF	Earliest Task First
FIFO	First In First Out
GA	Genetic Algorithms
GE	Grammatical Evolution
GHGE	Global Greenhouse Gas Emissions
GPU	Graphics Processing Unit
HPC	High Performance Computing
HW	Hardware
ICT	Information and Communications Technologies
IPC	Instructions Per Cycle
IR	Intermediate Representation

IRC	In Row Cooling
IT	Information Technology
LLC	Last Level Cache
LTF	Longest Task First
MAE	Mean Absolute Error
MIC	Many Integrated Cores
MILP	Mixed Integer Linear Programming
NMRSE	Normalized Root Mean Square Error
PCA	Principal Component Analysis
PSU	Power Supply Unit
PUE	Power Usage Effectiveness
RAPL	Running Average Power Limit
RMS	Root Mean Square
RMSE	Root Mean Square Error
RPM	Revolutions Per Minute
RR	Round-Robin
SaaS	Software as a Service
SVR	Support Vector Regression

Abstract

Data centers are one of the most power hungry sections of the Information and Communications Technologies (ICT) sector. In the U.S in 2014, data centers consumed around the 1.8% of the total U.S electricity consumption. Worldwide data centers consumed in 2015 around 200 TWh of the global electricity usage. This electricity consumption is expected to increase to around 1200 TWh in 2025, which would represent 4.5% of the global electricity usage. One of the major contributors to the overall data center power is the IT or computing power, therefore there is a special interest to improve its energy efficiency. Scientific community has developed energy efficient techniques to reduce the energy consumption of IT equipment, such as resource management, power budgeting or power capping. These techniques assume the existence of a full dynamic power profiling, obtained through a previous full execution of the application. This full dynamic profiling is not viable in scenarios of long-running applications that are deployed in data centers, since performing a full dynamic profiling of a large batch of long-running applications is a time consuming process thus not energy-efficient. Therefore, in this work we propose the use of an application signature to estimate the energy in a fast way without the need to execute the application from beginning to end. The application signature is a reduced version, in terms of execution time, of the original application. We developed a fast energy estimation framework that uses the application signature to make a quick energy estimation of long-running applications. The framework estimates, without performing a full profile of the application, the dynamic CPU and memory energy of both single-threaded and multi-threaded long-running application versions. Additionally, the fast energy framework is automatic and it has a modular design, allowing to change the functionality of each module without altering the functionality of the whole framework. We validated the accuracy of the fast energy estimation framework with a set of sequential and multi-threaded long-running applications. For the single-threaded version of the applications we obtained an RMS of 10.4% for the CPU energy estimation error and an RMS of 16.8% for the memory energy estimation error. In the

multi-threaded scenario, we used a subset of applications from the sequential version set. We achieved an RMS of 11.4% for the CPU energy estimation error and an RMS of 12.8% for the memory energy estimation error. We defined the concept of Compression Ratio (CR) as the ratio of total execution time of the original application, to the time it takes to estimate the energy through the fast energy estimation framework. A high CR value indicates that the energy is estimated much faster (CR times faster) than executing the whole application. We obtained Compression Ratios in the range from 10.1 to 191.2. Finally, we validated the usefulness of the energy estimation obtained from the application signature by applying three different energy-efficient task scheduling approaches: i) An optimal approach using a Mixed Integer Linear Programming (MILP) technique, ii) An energy-aware heuristic approach that uses a Longest Task First (LTF) algorithm together with an energy-efficient task allocation based on the current servers consumption, and iii) We proposed an implementation of a metaheuristic using a Simulated Annealing process. The results obtained through the energy estimation (obtained through the application signature) values are compared with the real energy values. We obtained energy savings from 8% to 19%, and more importantly the energy savings obtained with the application signature approach are similar to the values obtained with the real energy measurements, with an energy savings difference below 1.5%.

Resumen

Los centros de datos son una de las secciones del sector de Tecnologías de la Información y Comunicaciones (TIC) que tienen mayor consumo energético. Durante el año 2014 en EE. UU., los centros de datos consumieron alrededor del 1.8% del consumo eléctrico total en dicho país. A nivel mundial, los centros de datos representaron en el año 2015 alrededor de 200 TWh respecto al consumo eléctrico mundial. Según estimaciones, este consumo eléctrico puede aumentar hasta unos 1200 TWh en el año 2025, lo que representaría el 4.5% del consumo eléctrico global. Uno de los mayores contribuidores al consumo global en los centros de datos es el representado por los equipos de computación o consumo de *IT*. A nivel computacional, se han desarrollado diversas técnicas para reducir el consumo de *IT* como pueden ser, la gestión de recursos, presupuestos de potencia y la limitación de consumo de los servidores ubicados en los centros de datos. Para poder aplicar estas técnicas se asume la existencia de un perfilado (*profiling*) previo obtenido a través de una ejecución completa de la aplicación. En escenarios donde se ejecutan grandes despliegues de aplicaciones de larga duración no resulta viable realizar un *profiling* previo debido a que es un proceso que demanda elevados tiempos de ejecución y, por lo tanto, no es eficiente energéticamente. Teniendo en cuenta la problemática expuesta anteriormente, en este trabajo se ha desarrollado el concepto de firma de la aplicación cuyo uso tiene la finalidad de estimar la energía sin tener que ejecutar la aplicación en su totalidad. La firma de la aplicación se define como una versión reducida, respecto al tiempo de ejecución, de la aplicación original. Se ha desarrollado un *framework* de estimación rápida de energía que utiliza la firma de la aplicación para estimar la energía sin tener que ejecutar completamente las aplicaciones. El *framework* estima la energía de CPU y memoria, tanto de aplicaciones secuenciales como de aplicaciones de tipo paralelas (multihilo). A su vez, el framework se ejecuta de forma automática y tiene un diseño modular, permitiendo de esta forma reemplazar la funcionalidad interna de un módulo sin necesidad de alterar la funcionalidad de todo el *framework*. Se ha validado la precisión del framework de estimación rápida de energía con un conjunto representativo de ejecuciones

secuenciales y paralelas, obteniendo unos errores RMS de 10.4% y 16.8% de estimación de energía de CPU y memoria respectivamente para el caso de aplicaciones secuenciales. En el caso de aplicaciones paralelas, se ha trabajado con un subconjunto de las aplicaciones del caso secuencial y se han obtenido errores RMS de 11.4% y 12.8% de estimación de energía de CPU y memoria respectivamente. Por otra parte, se ha definido el concepto Ratio de Compresión (*CR*) como el ratio de la ejecución total de la aplicación original respecto al tiempo que tarda el framework en estimar la energía de la aplicación. Un valor alto de Ratio de Compresión indica que el framework estima la energía de forma mucho más rápida (*CR* veces más rápida) que la ejecución total de la aplicación. Se obtienen Ratios de Compresión que están en el rango entre 10.1 hasta 191.2. Finalmente, se ha evaluado la utilidad de la información de energía obtenida mediante la firma de la aplicación gracias a la aplicación de tres propuestas de planificación de tareas: i) utilizando un modelo de Programación Lineal Entera Mixta (*MILP*), ii) haciendo uso de una heurística energéticamente eficiente que utiliza un algoritmo de tipo *LTF* (tareas de larga ejecución se ejecutan primero) junto con una eficiente colocación de tareas en los servidores del centro de datos y, iii) se propone una implementación de una metaheurística basada en un algoritmo de recocido simulado (*Simulated Annealing*). Los resultados de estimación de energía global del centro de datos obtenidos con los datos de la firma de la aplicación se han comparado con los datos reales de energía de las aplicaciones. Se han obtenido unos ahorros de energía entre el 8% y 19%, y lo que es más importante los valores de ahorro de energía obtenidos con la información de la firma son similares, con un error inferior al 1.5%, respecto a los ahorros de energía obtenidos con los valores reales de energía de las aplicaciones.

Chapter 1

Introduction

1.1 Motivation

Data center facilities are high power consumers. It was estimated that data centers already accounted for at least 1.5% of the worldwide total electricity consumption in 2010 [70]. In the U.S at the year 2014, data centers consumed about 70 billion kWh which represented the 1.8% of the total U.S electricity consumption. It is estimated that U.S data centers will consume, in the year 2020, approximately 73 billion kWh [111].

Moreover, worldwide data centers consumed in 2015 around 200 TWh of the global electricity usage and it is expected to increase its electricity consumption to around 1200 TWh in 2025, which would represent 4.5% of the global electricity usage [5]. The sector of Information and Communication Technology (ICT), including data centers, generates up to 2% of the global $C0_2$ emissions [122]. In terms of the global greenhouse gas emissions (GHGE), data centers alone are projected to have the second fastest growing (behind Smart Phones) GHGE footprint from all of the ICT sector [14].

The overall energy use of the IT equipment (servers, storage devices, and network) has increased from 92 TWh (2010) to 130 TWh (2018) [87]. On the one hand, the energy efficiency solutions adopted in data centers in the last decade have enabled a large growth in services without a high increase of the energy use. On the other hand, there are estimations indicating that the computing resources of the data centers are going to double within the next 3 to 4 years. Therefore, the need to improve energy efficiency in data centers will be required to manage the possible energy growth [87]. This idea of improving energy efficiency is backed by several studies where they indicate the necessity of targeting energy efficiency as a key strategic initiative in data centers [120] [9] [13] [57].

There is an special interest to improve energy efficiency of computing (or IT) power since it is the major contributor to overall data center power (another important contributor is the cooling power) [12]. There are proactive approaches to reduce the overall energy consumption, such as, proactive thermal management techniques that are used to reduce the cooling energy and minimize the overheating of IT equipment, thus, minimizing the IT power consumption due to leakage [74]. Other energy-efficient techniques focus mainly in the IT power equipment, such as:

i) ***Resource management:*** where an energy-efficient or energy-aware task scheduling approach is applied for an optimal task or job allocation [134] [1] [8].

ii) ***Power budgeting:*** where policies or rules are established to distribute efficiently the power budget among all the servers from the data center [131] [108] [59].

iii) ***Power capping:*** ensures that the maximum peak power consumption of each server remains below a cap value [107] [19] [86] [118].

Moreover, when data centers are integrated in the Smart Grid, the regulation capabilities of servers enable the participation in demand-response programs, allowing to reduce the energy costs in data centers [24]. The previous techniques are usually used proactively and assume the existence of either a full dynamic power profiling of the applications (obtained through a previous full execution of the application), or power models that predict the power consumption of the applications that are going to be executed in the servers.

The work of this thesis is focused on long-running, data-intensive and iterative applications. This type of application is usually found in many scientific computing or HPC scenarios. Additionally, besides taking too much time to finalize the execution, these applications are CPU and memory intensive, resulting in high energy consumption in data centers, characteristics that make this type of application interesting for energy efficiency purposes.

In scenarios of long-running applications deployed in data centers the process to perform a full dynamic power profiling is not viable since performing a full profile of a large batch of long-running applications is a time consuming process and therefore, is not energy-efficient. These long-running applications, as we have previously mentioned, have the characteristic to be iterative, data-intensive, and often they are formed by computational intensive kernels such as, matrix multiplication. These kernels can be found in scientific applications (e.g., fluid dynamics, climate modeling) [51] and more recently in artificial intelligence applications [135] [123]. Long-running applications are

executed in large-scale data centers, for example, the *Barcelona Supercomputing Center*[*] or in small-scale data centers such as *Madrid Supercomputing and Visualization Center*[†].

Moreover, these types of applications can be executed with multiple instances of the same application in a single-threaded way, and also, can be executed in a multi-threaded way using multiple cores as occur in the High Performance Computing (HPC) scenario. Also, long-running applications can be found in non-cloud and cloud environments. In non-cloud environments the applications are executed directly in the server. This type of environment is usually present in HPC scenarios. In cloud environments the long-running applications can present themselves as a Software as a Service (SaaS) type of applications like in services such as Google Cloud[‡].

Long-running applications are usually deployed in batches and they are commonly executed with different input datasets, hence these applications show a different behaviour each time their input changes. This would require to redo a full dynamic profiling every time that the input changes. Obviously, this is not an energy-efficient process.

Therefore, in this work we propose the use of an application signature to make a fast energy estimation of long-running single and multi-threaded applications without the need to perform a complete execution of the original application. We define the **application signature** as a reduced version, in terms of execution time, of the original application. Also, the **application signature** is used to make a **performance prediction**, which is a prediction of the execution time of the applications before they are executed in the servers of the data centers. We developed a fast energy estimation framework that uses the application signature to estimate in a fast way the energy without executing the application from beginning to end, thus allowing the development of proactive energy-efficient optimization policies adapted to the application.

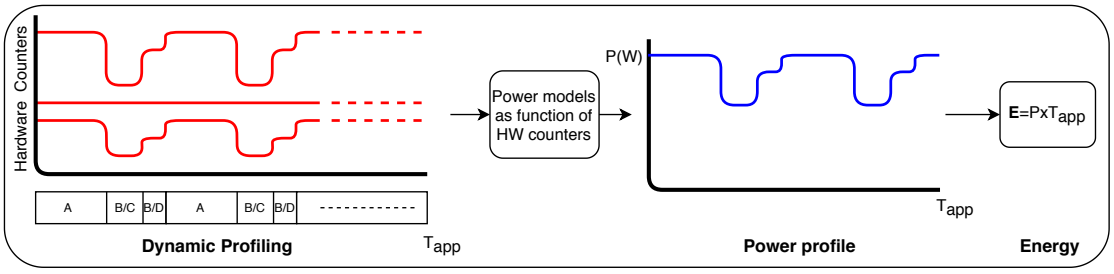
Hence, in this work we use the information provided by the application signature to apply different energy-efficient task scheduling approaches (resource management) in order to reduce the **makespan** of the original batch, and therefore improve the energy efficiency in data centers. The **makespan** is defined as the total execution time of the batch of applications that will run in the data center. Without the application signature those energy-aware scheduling approaches would require a full dynamic profiling from the complete execution of the applications preventing them to be applied in an efficient way.

[*]https://www.bsc.es/
[†]https://www.cesvima.upm.es/
[‡]https://cloud.google.com/

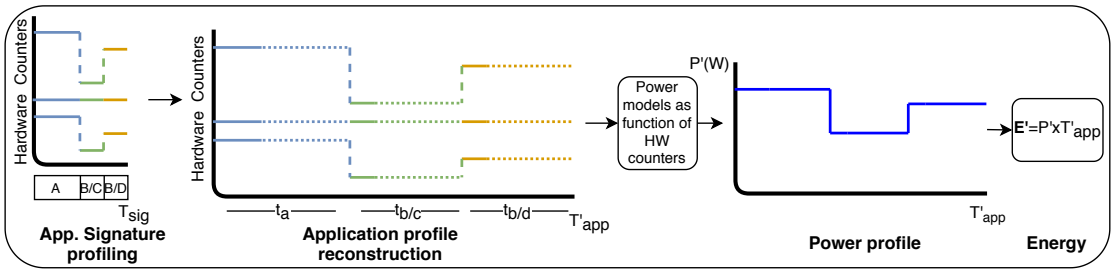
In this work we do not address experimentally the problem of using other energy-efficient techniques such as power budgeting or power capping in an experimental way. Nonetheless, the fast energy estimation framework using the application signature can be used to solve the problems of such techniques.

Finally, although we have not used any stream processing applications, we consider that the fast energy framework should be able to work with them. Moreover, HPC over cloud applications could also be a potential target for the proposed framework. Finally, applications such as data-intensive interactive or latency-sensitive online applications would not be able to work with the developed framework.

1.2 Problem Formulation



(a) Traditional approach: Energy estimation through the execution of the original application



(b) Our approach: Energy estimation through the execution of the application signature

Figure 1.1: Motivational Example: Comparison between traditional energy estimation techniques vs our proposed energy estimation approach

To illustrate our proposed solution using the application signature we present the problem formulation as a motivational example. The code presented in Application 1 shows a simple loop-based application. The application starts by defining a set of variables (n, v, y and z) that represent the ***input data***. The input data can be defined in the *Main* function of the source code (as in this example) or it can be read from an external source such as a file. The application continues with a main loop that iterates until the threshold set by the variable n.

Inside the loop, two functions ($FunctionA$ and $FunctionB$) are called. We can see that the execution can be divided in three ***execution paths***:

- $Main \rightarrow FunctionA$ (Path 1).
- $Main \rightarrow FunctionB \rightarrow FunctionC$ (Path 2).
- $Main \rightarrow FunctionB \rightarrow FunctionD$ (Path 3).

The iterative structure shown in Application 1 has been selected because it appears in many long-running applications [101] [81] [21].

Application 1 Motivational Example: Long-running application

```
1: function MAIN
2:     n ← 1000
3:     v ← [100, 100]
4:     y ← 50
5:     z ← 25
6:     for i ← 0 to n do
7:         FUNCTIONA(v)
8:         FUNCTIONB(y,z)
9:     end for
10: end function
11: function FUNCTIONA
12:     for i ← 0 to v(1) do
13:         Computation
14:         for j ← 0 to v(2) do
15:             Computation
16:         end for
17:     end for
18: end function
19: function FUNCTIONB
20:     FUNCTIONC(y)
21:     FUNCTIOND(z)
22: end function
23: function FUNCTIONC
24:     for i ← 0 to y do
25:         Computation
26:     end for
27: end function
28: function FUNCTIOND
29:     for i ← 0 to z do
30:         Computation
31:     end for
32: end function
```

Figure 1.1a shows the traditional approach of energy estimation by doing a full profiling of the whole execution of the application. First, we execute the whole application and collect a set of hardware counters through a dynamic profiling.

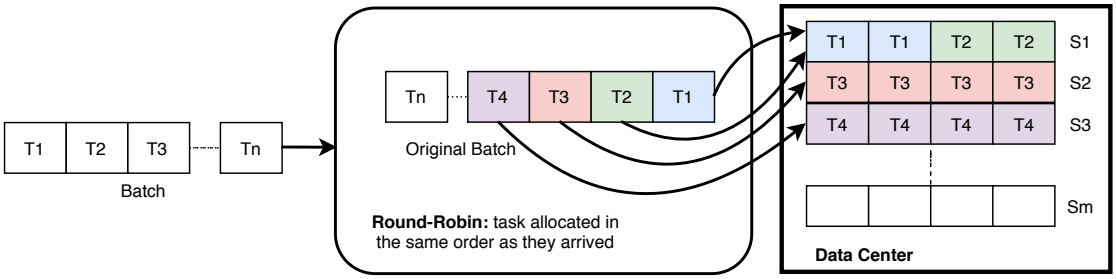
Then, we obtain a power profile (P) by using power models as a function of the hardware counters previously collected. Finally, from the power profile and the execution time (T_{app}) of the application we can calculate the energy. In the case of long-running applications this process is not efficient since the execution times are usually long (hours or days). To solve this problem we use an ***application signature***.

We propose an energy estimation process through the application signature, as shown in Fig. 1.1b. First, we execute the application signature and collect a set of hardware counters. The application signature execution is composed by a shorter execution of each independent path, i.e we execute Path 1 for a short period of time, then execute Path 2, and so on. Therefore, the execution time of the application signature (T_{sig}) is significantly lower than the execution time of the original application. The next step is to reconstruct the application profile. To reconstruct the profiles we extend in time the values of the hardware counters obtained through the execution of the application signature. Each hardware counter profile from each path is extended until it has a length equal to the **estimated execution time path** (t_a, $t_{b/c}$ and $t_{b/d}$).

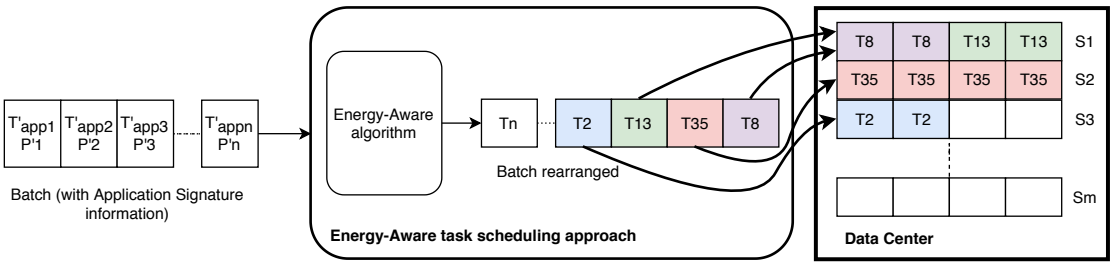
The execution time of each path is estimated through the **estimated executed instructions** (obtained from the source code and binary of the original application) and the **Instructions per Cycle** (IPC) of each path (obtained from the application signature). The total estimated execution time of the application (T'_{app}) can be obtained by adding the estimated execution time of each path. To minimize error, the value of T'_{app} should be the closest possible to T_{app}.

Finally, we use power models as a function of hardware counters with the application reconstructed profile to obtain a power profile (P'). From the power profile we can calculate the energy (E') of the application. Since the application signature executes each independent execution path independently the power profile P' is not the same as the power profile P of the original application. However, the estimated energy E' is equal to the energy E estimated from the original application, since the area under the curve of both power profiles (P and P') should be the same.

As we have previously commented, in order to apply energy-aware task scheduling approaches we need to extract information such as power or execution time from the applications before they are executed in the data center. When is not possible to gather these type of information a Round-Robin process is applied to the batch. Figure 1.2a shows the task allocation process of a batch of T_n tasks using the Round-Robin approach. Each task is allocated to an available server (S_m) in the same order as they originally arrived. The example shows a



(a) Traditional approach: Round-Robin policy



(b) Our approach: Energy-aware task scheduling approach using the application signature

Figure 1.2: Comparison between Round-Robin policy vs an energy-aware task scheduling approach

data center with m servers with each server equipped with 4 cores. The first T_1 task requires 2 cores and is allocated to server S_1, similarly task T_2 requires 2 cores and is allocted in server S_1 since it has enough available resources. When a task can not be allocated in a server it goes into a waiting queue until there is enough resources available to be executed. The Round-Robin policy is simple and therefore, easy to implement. Although, the Round-Robin policy is not an energy-efficient task scheduling process since the tasks are not allocated aiming to reduce the makespan or the power consumption of the servers.

Energy-efficient proactive task scheduling approaches can be used to reduce the energy consumption in data centers by reducing the makespan of the original batch. Figure 1.2b shows that an energy-aware task scheduling approach is used to rearranged the original batch. By changing the tasks execution order the makespan can be minimized. In the figure we can see that the tasks are allocated differently when compared to the Round-Robin policy (tasks T_8 and T_{13} are allocated in server S_1, and so on). This change in the tasks order of execution is the result of applying an energy-aware algorithm to minimize the makespan of the original batch. The energy-aware algorithm needs information, such as the execution time or the mean power consumption from the tasks that will be executed. This information can be obtained through a full dynamic profiling of each task (not very energy-efficient process). By executing the application signature we can obtain the information of

execution time T'_{app_i} and mean power consumption P'_i without the need to perform a full profiling of each task of the batch. Thus, allowing to apply an energy-aware task scheduling process in a more efficient way.

1.3 Thesis Contributions

The contributions of this PhD. thesis are described in 3 sections: i) the application signature as a proof of concept, ii) the implementation of a fast energy estimation framework that uses the application signature and, iii) using energy-aware task scheduling approaches with the information provided by the application signature.

- Application signature as a proof of concept:
  - We present the concept of application signature and its use to obtain a fast CPU and memory energy estimation of long-running applications. A proof of concept of the use of the application signature is proposed for single-threaded and multi-threaded applications.
  - We obtain errors for the estimation of the dynamic CPU and memory energy consumption below 8.0% when the estimated energy is compared against the energy consumption of the complete execution of the application. For the multi-threaded applications we obtain an RMSE equal to 12.7% when we compare the dynamic energy estimated from the application signature against the dynamic energy from the whole multi-threaded execution of the application. We obtain an average value of almost 9.8 for the CR (**Compression Ratio**, defined as a relation between the total execution time and the time the application signature takes to estimate the energy consumption) of all the benchmarks, indicating that using the application signature we are able to make a dynamic energy consumption estimation almost 10 times faster than the original execution of the application.
- Fast energy estimation framework:
  - We propose a fast energy estimation framework for long-running applications that uses the application signature to estimate the dynamic CPU and memory energy without the need to execute the whole application. The framework is able to estimate the energy for both sequential and multi-threaded applications.

  - The framework estimates the energy in an automatic way. The design of the framework is modular, allowing to change the internal functionality of each module, due to preferences or technical availability, without affecting the functionality of the whole framework.
  - We validate the accuracy of the fast energy estimation framework with a set of sequential and multi-threaded long-running applications. For the sequential version of the applications we obtain an RMS of 10.4% for the CPU energy estimation error and an RMS of 16.8% for the memory energy estimation error. In case of the multi-threaded scenario we use a subset of applications from the sequential version set. We achieve an RMS of 11.4% for the CPU energy estimation error and an RMS of 12.8% for the memory energy estimation error. We obtain Compression Ratios in the range from 10.1 to 191.2.
  - We model dynamic CPU and memory power given the per-application hardware counters, using Grammatical Evolution techniques. We obtain absolute power errors equals to 4.4W and 3.7W, for the dynamic CPU and memory power model respectively. We use analytical fan and leakage power models to obtain overall server power. Our models are trained and tested using a wide range of sequential and parallel workloads, under various DVFS setups, improving error by a 32% when compared to a traditional approach.
  - We show that our model is robust enough to predict the power consumption of two different tasks when they run co-assigned in the same server, given the hardware counters of the overall server. Also, we develop a methodology to, given the hardware counters of individual tasks, obtain the hardware counters when both applications are co-allocated (without executing the co-allocated applications).

- Energy-aware task scheduling using the application signature:

  - We validate the usefulness of the energy estimation information (mean power and estimated execution time) obtained from the application signature by applying different energy-efficient task scheduling approaches. The results obtained through the energy estimation values are compared with the real energy values. The values of the energy estimation from the fast energy estimation framework are presented as the application signature information. The real energy values are presented as the oracle information.
  - We use three different task scheduling approaches:

i) An optimal approach using a Mixed Integer Linear Programming (MILP) technique.

ii) An energy-aware heuristic approach that uses a Longest Task First (LTF) algorithm together with an energy-efficient task allocation based on the current servers consumption.

iii) We propose an implementation of a metaheuristic using a Simulated Annealing process.

The resulting overall data center energy consumption from each task scheduling approach is compared against a Round-Robin (RR) approach.

- We obtain energy savings from 8% to 19%, and more importantly the energy savings obtained with the application signature information are similar as the values obtained with the oracle information, with energy savings difference below 1.5%.
- We define the Compression Ratio of the Batch of applications as the ratio of total execution time of the original batch using the Round-Robin approach to the total execution time of extracting and executing the Application Signature of the whole batch. We obtained Compression Ratios around 39.7 to 45.8.

1.4 Thesis Structure

The remainder of this work is organized as follows:

- Chapter 2 explains the application signature as a proof of concept. The concept of the application signature is explained and an experimental proof of concept is applied to validate the use of the application signature as a mean to estimate the energy of the applications. A proper understanding of the proof of concept presented in this chapter makes it easier to follow the content of Chapter 3, where we present the related work.

- Chapter 3 presents the related work associated with this PhD. thesis. We cover works associated with similar approaches that uses an application signature and different energy estimation process applied in data centers.

- Chapter 4 describes the fast energy estimation framework that uses the application signature. We show a full implementation of the framework and also, the results of applying the framework in an online manner with a set of long-running applications. Additionally, in this chapter we present a

methodology to obtain an overall server power model using Grammatical Evolution techniques.

- Chapter 5 shows the results of using the information provided by the application signature for energy-efficient task scheduling approaches.
- Chapter 6 provides the conclusions obtained from this work and also, presents a summary of the future research works derived from this PhD. thesis.

1.5 Publications and Grants

In this section we present the journal and conference publications associated with this work, as well as awarded grants.

1.5.1 Journal papers

We have presented our work in an international journal. Moreover, at the moment of writing this work there is one article in review process:

- J. C. Salinas-Hilburg, M. Zapater, J. M. Moya, J. L. Ayala, "Fast energy estimation framework for long-running applications", Future Generation Computer Systems (2021) (Chapter 4 of this PhD. thesis).
- J. C. Salinas-Hilburg, M. Zapater, J. M. Moya, J. L. Ayala, "Energy-Aware Task Scheduling in Data Centers using an Application Signature", Computers and Electrical Engineering (2020) (Review process) (Chapter 5 of this PhD. thesis).

1.5.2 Conference papers

Additionally, the results of this thesis were presented in the following international peer-reviewed conferences:

- J. C. Salinas-Hilburg, M. Zapater, J. M. Moya, J. L. Ayala, "Fast Energy Estimation Through Partial Execution of HPC Applications", International Conference on Application-specific Systems, Architectures and Processors (ASAP) (2018) (Chapter 2 of this PhD. thesis).
- J. C. Salinas-Hilburg, M. Zapater, J. L. Risco Martín, J. M. Moya, J. L. Ayala, "Unsupervised Power Modeling of Co-Allocated Workloads for Energy Efficiency in Data Centers", Design, Automation and Test in Europe (DATE) (2016) (Chapter 5 of this PhD. thesis).

- J. C. Salinas-Hilburg, M. Zapater, J. L. Risco Martín, J. M. Moya, J. L. Ayala, "Using grammatical evolution techniques to model the dynamic power consumption of enterprise servers", International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS) (2015) (Chapter 5 of this PhD. thesis).

1.5.3 Mobility Grants

The author of this dissertation has been awarded with a mobility research grant from the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC), for a research stay of 3 months in the Performance and Energy Aware Computing lab (PeacLab) at Boston University (2017).

In the next chapter the reader will find a proof of concept of the application signature. We will show the technical details of how the application signature works and also, how is used to estimate the energy of the applications.

Chapter 2

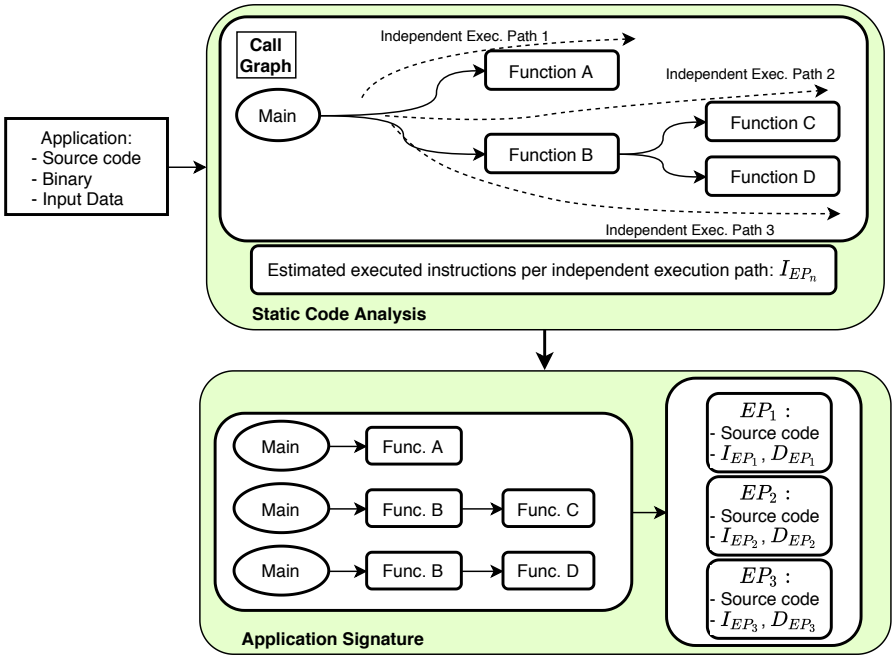
Energy Estimation with an Application Signature

In this chapter we present a proof of concept where we show that is possible to estimate the energy of an application without the need to execute it completely. We define the concept of application signature and how it is used to estimate the energy of the applications in a fast way. Additionally, we present the results of energy estimation from the application signature of a large set of heterogeneous applications.

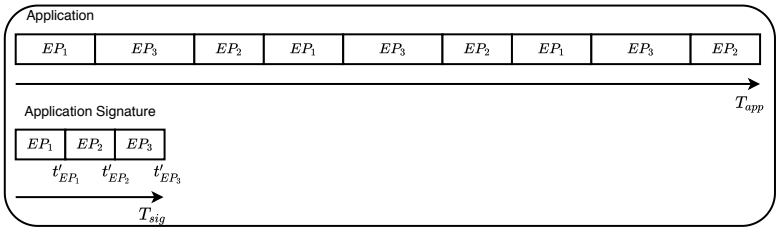
2.1 Application Signature

In this section we present our methodology to construct the application signature. The **application signature** is defined as a reduced version (in terms of the execution time) of the original application. The main purpose of the application signature is to estimate the energy and performance of the whole execution of the application without the need to execute it completely.

Figure 2.1a shows an overview of the application signature structure. The source code/binary of the application together with its correspondent input data is analyzed statically to extract information about the Call Graph (CG) and the estimated executed instructions per independent execution path. We define an ***independent execution path*** as a path from the Call Graph obtained through the following process: i) start the path search at the root node (main function), ii) if an edge can be followed, do so; iii) if not, stop the path search. The application signature presents a two layer architecture. The first layer is composed by the source codes from all the independent execution paths of the application; while the second layer contains static code analysis information



(a) Application signature structure



(b) Original application Vs Application Signature execution times

Figure 2.1: Overview of the application signature structure and execution

required for the energy consumption estimation. This information are the values of I_{EP_1}, I_{EP_2},..., I_{EP_n} that represent the estimated executed instructions from each independent execution path and the values of D_{EP_1}, D_{EP_2},..., D_{EP_n} indicate the input data to each independent execution path. An *application signature execution manager* creates the binaries of each source code and executes each independent execution path with their respective input data. The manager stops the execution of each independent execution path until a desired threshold criteria of executed instructions per path is reached. By doing this we can execute each independent execution path of the application separately and for a very short period of time, much shorter than the original execution. The application signature is directly associated with the input data of the original application, therefore when the application change its input data a new application signature is built.

Figure. 2.1b shows the comparison between the execution time of the application (T_{app}) against the execution time of the application signature (T_{sig}), which must be lower by definition. The relation between the two execution times is the **Compression Ratio** (CR) and indicates the acceleration of the energy estimation process. The value of CR is defined in Eq. 2.1. The value of t'_{EP_j} represents the execution time of the partial execution of the j^{th} independent execution path of the application. By adding the partial execution of all the independent execution paths we estimate the total execution time of the application signature: T_{sig} ($T_{sig} << T_{app}$).

$$CR = \frac{T_{app}}{T_{sig}} \quad where \;\; T_{sig} = \sum_{j=1}^{N} t'_{EP_j} \tag{2.1}$$

2.1.1 Static Code Analysis

The main purpose of a static code analysis is to obtain information without the need to execute the application, such as the number of resources an application will use during runtime. The COSTA tool [3] [2] is a static code analysis tool that can perform such task. It can estimate an upper bound of the resource consumption of the application and, among other parameters, it can estimate the number of instructions executed per independent execution path considering the input data of the application. Also, COSTA builds the Call Graph listing all the functions and the calls between them. Although COSTA is a static code analysis tool intended to work for JAVA bytecode the concepts and techniques can be applied to the static code analysis of applications in other programming languages. Another tool called Mira [89] is a framework for static performance modeling and analysis and can estimate the number of executed instructions for large scale or HPC applications programmed in C/C++ languages.

In this chapter, we made use of the information that a regular static code analysis profiler (such as COSTA or the Mira framework) provides. In particular, we rely on the Call Graph extraction and the estimation of executed instructions per each independent execution path. The Call Graph allows us to know the number and the hierarchy of the whole set of paths. According to this, we can isolate each independent execution path to perform its partial execution.

The static code analysis process takes into account the input data of the application to estimate the number of executed instructions. An example of input data typically found in large scale or HPC applications is the number of timesteps. The number of timesteps usually determines the number of iterations of the main loop of the application which heavily affects the number of estimated executed

instructions. The estimated number of executed instructions together with the Instructions per Cycle calculated from each partial execution of each independent path is used to estimate the total execution time of each independent execution path ($T'_{EP_n} = (I_{EP_n})/(IPC_{EP_n} \times Freq_{CPU})$), as can be seen in Section 2.1.2.

2.1.2 Dynamic Profiling of the Application Signature

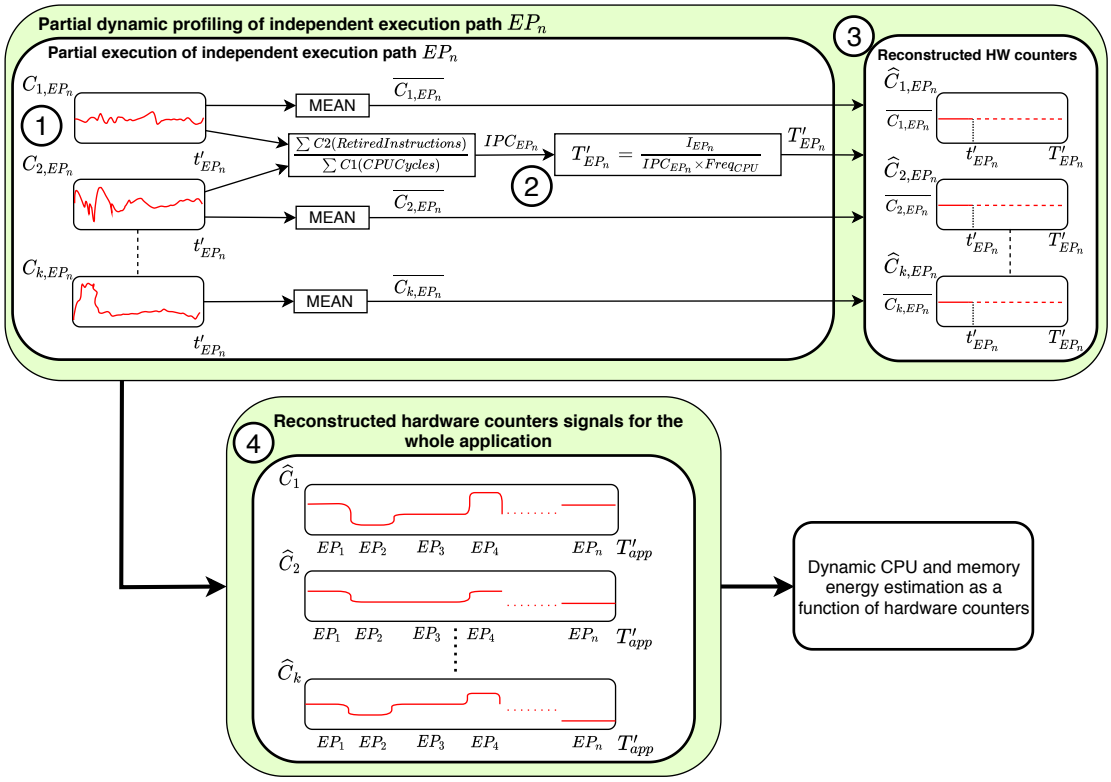


Figure 2.2: Overview of the process for energy estimation using the application signature

In order to estimate the energy consumption using the application signature we need to apply the process shown on Fig. 2.2. The first step is to perform a partial dynamic profiling of the application signature, which is formed by the partial execution of each independent execution path. This partial dynamic profiling takes much less time than the complete dynamic profiling of the whole application. The purpose of the dynamic profiling step is to make a quick acquisition of the hardware counters and calculate the Instructions per Cycle (IPC) of each partial execution. By combining the information of I_{EP_n} and the IPC from each partial execution we are able to estimate the total execution time T'_{app} (i.e. performance) and also obtain an equivalent temporal profile of hardware counters for the complete execution. As we can see in the dynamic profiling block from Fig. 2.2:

1. Each independent execution path is partially executed during a short amount of time t'_{EP_n} and the hardware counters $(C_{1,EP_n}, C_{2,EP_n}, ..., C_{k,EP_n})$ are collected. Each independent execution path is partially executed by the *application signature execution manager* until the instructions retired reach a threshold equal to $\alpha \times I_{EP_n}$, where $\alpha \in [0, 1]$ and I_{EP_n} is the estimated executed instructions for the n^{th} independent execution path. The value of α should be low enough to partially execute the path (providing the expected acceleration), while still targeting a significant amount of execution time to gather the hardware counters.

2. The IPC of the partial execution for the n^{th} independent execution path is obtained and then the estimated total execution time of the path EP_n is calculated as follows: $T'_{EP_n} = (I_{EP_n})/(IPC_{EP_n} \times Freq_{CPU})$ (CPU equation time or basic performance equation).

3. The temporal profile of the hardware counters for the n^{th} independent execution path can be reconstructed by forming signals with duration T'_{EP_n} and amplitude equal to the mean value of each hardware counter $(\widehat{C}_{1,fn}, \widehat{C}_{2,fn}, ..., \widehat{C}_{k,fn})$.

4. The temporal profiles of each hardware counter for the complete application are built by aligning the temporal profiles of the hardware counters from each independent execution path. These hardware counter profiles have a duration equal to the total estimated execution time T'_{app} and amplitude equal to the mean value of the respective hardware counters of each path. These reconstructed profiles of hardware counters for the whole application are equivalent, in terms of reflecting the dynamic CPU and memory energy, to the profiles of hardware counters obtained through a full dynamic profiling of the original application.

2.1.3 Energy Estimation

Figure 2.2 shows that the dynamic CPU and memory energy is estimated through the reconstructed hardware counter signals obtained with the execution of the application signature. The dynamic CPU and memory energy is estimated using the CPU and memory power models explained in Section 4.2. We have two separated power models for dynamic CPU and memory as a function of the hardware counters: $\widehat{P}_x = f(\widehat{C}_1, \widehat{C}_2, ..., \widehat{C}_k)$, where x is CPU for the dynamic CPU power or Mem for the dynamic memory power. The expression for the dynamic CPU and Memory power $(\widehat{P}_x)$ can be found in Section 4.2.

The energy is estimated by adding all the instantaneous dynamic CPU or memory power values and then multiplying by the sampling period sm, as shown in Eq. 2.2, where x is for CPU or Mem. The energy is calculated over a period equals to T'_{app} and for the set B formed by the temporal samples of the estimated power signal ($\widehat{P}_x$). These dynamic CPU and memory energies are obtained through the reconstructed hardware counters signals that comes from the application signature.

$$\widehat{E}_x = \Big[\sum_{n \in B} \widehat{P}_x[n] \Big] \times sm \tag{2.2}$$

2.2 Application Signature for Multi-Threaded Applications

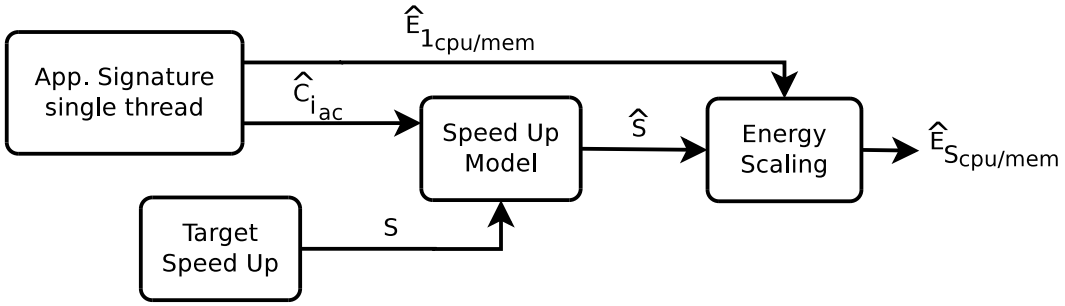


Figure 2.3: Application Signature for Multi-threaded applications

In order to obtain an estimation of the dynamic CPU and memory energy for multi-threaded applications without the need to run the entire application we apply the process shown in Fig. 2.3. First, we extract the application signature of the single-threaded version of the application to obtain the dynamic CPU and memory energy estimation ($\widehat{E_{1cpu/mem}}$) and also the accumulative values of the estimated hardware counters ($\widehat{C}_{i_{ac}}$). As a next step, we use a speed-up model to obtain an estimated acceleration ($\widehat{S}$) as a function of the accumulative values of the estimated hardware counters and the desired speed-up (S). The **speed-up** is defined as the ratio between the runtime of a single-threaded execution divided by the runtime of a multi-threaded execution (i.e, if we execute a parallel application with two threads we should expect that the runtime of the multi-threaded execution is half the runtime of the single-threaded execution, taking into account that the parallel execution has minimal synchronization delays). It should be noted that the target speed-up is equal to the numbers of threads the application will run in order to

perform a parallel execution. Finally, an energy scaling is made to estimate the dynamic CPU and memory energy for the multi-threaded run:

$$\widehat{E_{Scpu/mem}} = \frac{\widehat{E_{1cpu/mem}}}{\widehat{S}} \tag{2.3}$$

The process of estimating the dynamic CPU and memory energy for multi-threaded applications can be applied to applications that have a data parallelism form of parallelization, where the threads are executing the same task and the performance exhibited by each other is almost the same. This form of parallelization can be found in multiple large scale or HPC scenarios and has been addressed by other authors [124].

The speed-up model for the accelerations ranging from 1 to the number of cores (c) is a linear model where the estimated speed-up is equal to the target speed-up, while for accelerations from the number of cores plus one ($c+1$) to the total number of hardware threads (h) we develop a speed-up model as a function the accumulative values of the estimated hardware counters ($\widehat{C}_{i_{ac}}$) and the target speed-up (S), as shown in Eq. 2.4.

$$\widehat{S} = \begin{cases} S & S \in [1, c] \\ \beta_0 + \sum_{i=1}^{l} \widehat{C}_{i_{ac}} * \beta_i + S * \beta_{l+1} & S \in [c+1, h] \end{cases} \tag{2.4}$$

We assume that when a parallel application runs a number of threads equal to the number of cores each thread is going to run separately in each core, so the speed-up must be lineal as a function of the number of cores. In the case of running more than $c+1$ threads each thread is going to share core resources with another thread so this linear behaviour with the number of hardware threads cannot be assumed. The speed-up model for accelerations above $c+1$ is obtained offline by using a partial least-squares regression model taking as input the accumulative values of the hardware counters and the actual speed-up both obtained by running a set of benchmarks with a variable numbers of threads (from $c+1$ to h).

2.3 Experimental Setup

The validation of the aforementioned methodology takes place in an Intel server (S2600GZ) based on the Intel Decathlete 2.0 Open Compute Project server board. The server is equipped with one Intel 6-core SandyBridge-EP processor providing

Table 2.1: Hardware counters used as inputs to the power estimation models

	Description
C1	Clock cycles
C2	Instructions retired
C3	LLC misses
C4	L2D Cache misses
C5	Branch instructions retired
C6	Resource stalls
C7	μops dispatched
C8	L1D Cache misses

Table 2.2: Coefficients values of the Speed-Up model

β_0	β_1	β_2	β_3	β_4
2.78	1.02e-11	-4.34e-12	-3.02e-10	0.18

up to 12 hardware threads, 8 4GB memory modules, 4 hard disk drives, 5 fans and 2 PSUs. The server runs a CentOS 6.5 Linux OS. We use *ocount*, an Oprofile tool, to gather hardware counters during runtime. The hardware counters are polled every second. Table 2.1 shows the hardware counters collected from the execution of the application signature. This set of hardware counters comes from the result of a feature selection method done during the power modeling process presented in Section 4.2. The Linux tool *pstack* is used to gather information about the temporal evolution of the independent execution paths of the running application. The *pstack* tool is executed with the applications *pid* as an input every second (i.e., in every second *pstack* displays a stack trace of the current execution of the application.).

It should be noted that for the experimental setup of the proof of concept presented in this chapter, the independent execution paths that showed data dependencies between them are merged into one execution path. In Chapter 4, a real implementation of the application signature is developed where the benchmarks used to evaluate the accuracy of the application signature are selected to not have data dependencies between the independent execution paths.

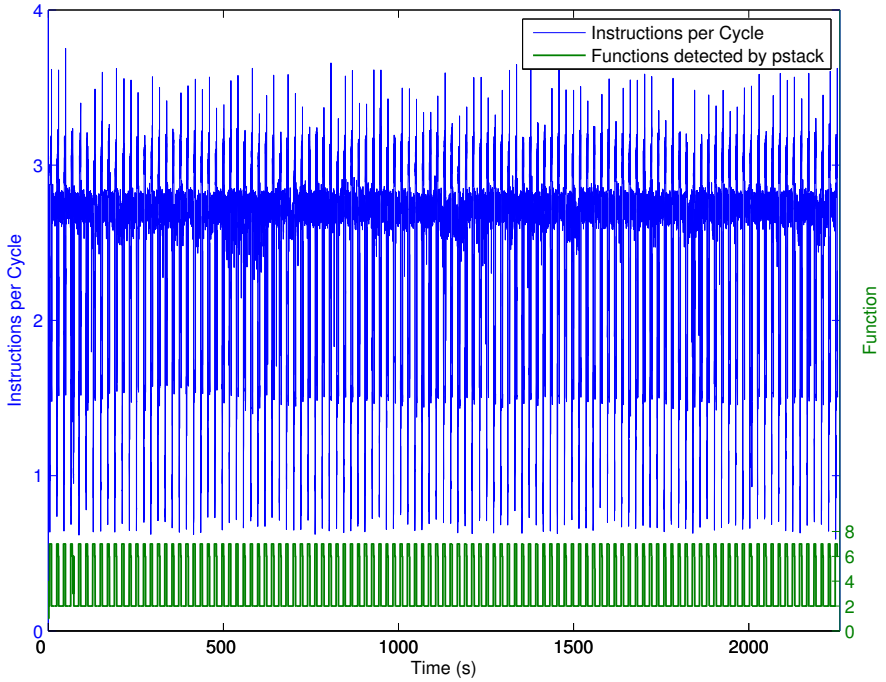
The set of workloads used in this chapter come from the SPEC CPU2006 [54] benchmark suite, the PARSEC suite [16] [17] and benchmarks from the NAS Parallel suite [11]. For the estimated executed instructions per independent execution path we make the assumption of having the executed instructions per independent execution path values with zero % estimation errors. It should be noted that the application signature is extracted from offline data acquired from a complete execution of the heterogeneous set of workloads.

In case of the speed-up model we performed a Principal Component Analysis (PCA) on the set of features (hardware counters and target speed-up) that are the inputs to the acceleration model. The final features selected to build the model are: Clock cycles (C1), Instructions Retired (C2), LLC misses (C3) and target speed-up (S). The values of the speed-up model linear regression coefficients are shown in Table 2.2, where β_0 is the constant offset, the coefficients β_1, β_2 and β_3 are the coefficients that multiply the hardware counters C1, C2 and C3, respectively, and the coefficient β_4 multiplies the value of the target speed-up. The training set was formed by the following benchmarks: *blackscholes*, *swaptions*, *freqmine*, *streamcluster*, *BT*, *LU*, *FT*, and *CG*. The benchmarks for the test set are the following: *canneal*, *fluidanimate* and *SP*. The benchmarks selected both for the train and test set were selected manually to represent an heterogeneous set of benchmarks with different speed-ups and accumulated hardware counters behaviours. In order to evaluate the accuracy of the speed-up model, we calculate the error of the estimated speed-up against the actual speed-up from each benchmark of the training and test set. The overall speed-up model *RMSE* for the training set and test set are 0.69 and 0.71, respectively.

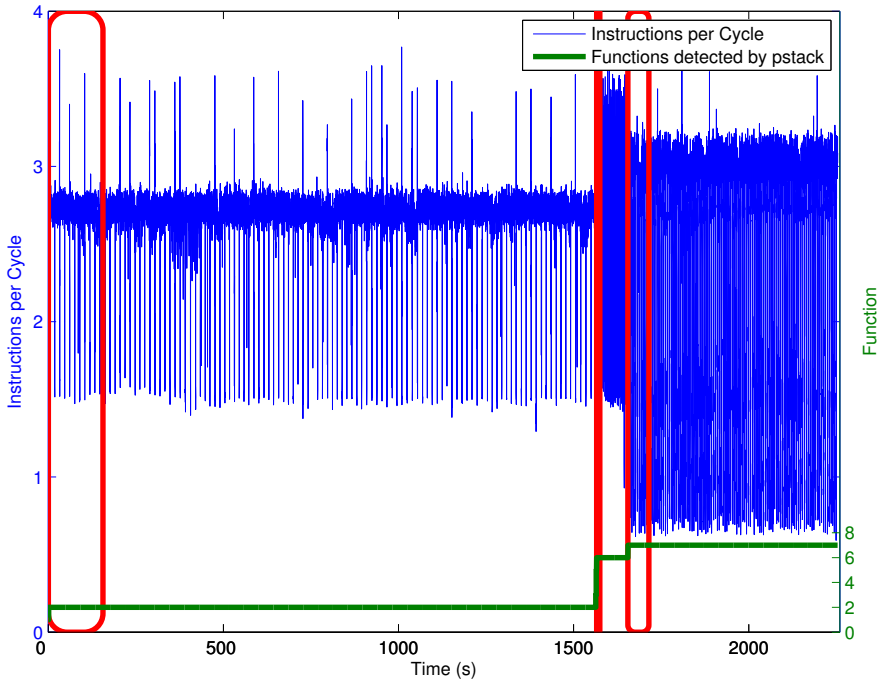
2.4 Results

Figure 2.4a shows the Instructions per Cycle from the complete execution of the benchmark *Calculix*, where the green line shows the temporal evolution of the final functions of each independent execution path as given by the *pstack* output. Each different value of the green line represents a different running independent execution path from the benchmark. In order to ease the validation of the proof of concept using the application signature, the temporal profile of the complete execution is reorganized to group the execution of each function, as shown in Fig. 2.4b. For example, the independent execution path that leads to the function that goes from 0 seconds to 1500 seconds is formed by all the execution blocks from the original execution of the *Calculix* function *e3cd*. Although we are showing the values for the Instruction per Cycle this process is performed for all of the hardware counters. The application signature samples are represented by the red rectangles shown in Fig. 2.4b.

The Table 2.3 shows the complete set of benchmarks evaluated in this chapter. The benchmarks from *bwaves* to *tonto* are sequential applications from the SPEC CPU2006 suite, benchmarks from *blackscholes* to *fluidanimate* are parallel applications from the PARSEC suite and benchmarks from *BT* to *SP* are parallel applications from the NAS Parallel suite. For each benchmark, the



(a) Original execution



(b) Reorganized execution

Figure 2.4: Instructions per Cycle (blue) and the temporal evolution of the independent execution paths (green). Application signature samples (red). Benchmark: Calculix

Table 2.3: Results of the validation process

	Exec. Paths	Total runtime (s)	Signature runtime (s)	$Error_{CPU}$ (%)	$Error_{Mem}$ (%)	CR
bwaves	4	1503	151	6.11	6.13	9.95
cactusADM	1	1657	166	18.22	18.25	9.98
calculix	7	2253	226	0.64	0.64	9.96
dealII	15	919	94	1.62	1.62	9.70
GemsFDTD	10	1296	131	0.66	0.66	9.89
gromacs	13	1408	142	0.20	0.20	9.91
lbm	1	756	76	0.28	0.28	9.94
leslie3d	11	1674	169	0.20	0.20	9.90
libquantum	10	966	98	1.17	1.17	9.85
mcf	13	628	64	7.34	7.34	9.81
milc	13	859	88	4.15	4.16	9.76
namd	15	1256	128	0.37	0.37	9.81
omnetpp	21	771	80	1.07	1.08	9.63
povray	27	520	56	1.63	1.63	9.28
sjeng	26	1459	149	1.36	1.36	9.79
tonto	15	1399	142	1.35	1.35	9.85
blackscholes	2	479	48	0.27	0.27	9.97
freqmine	12	2114	213	0.19	0.20	9.92
swaptions	4	824	83	0.52	0.52	9.92
canneal	10	258	27	2.42	2.42	9.55
streamcluster	4	964	96	0.53	0.54	10.04
fluidanimate	5	800	80	0.30	0.31	10.00
BT	17	674	68	0.20	0.21	9.91
CG	2	240	24	0.74	0.74	10.00
FT	8	159	17	3.65	3.67	9.35
LU	6	610	61	0.56	0.57	10.00
SP	9	513	52	0.76	0.77	9.86

table shows the total execution time (T) in seconds, the number of independent execution paths detected by the *pstack* tool during runtime and the execution time of the application signature (T_{sig}) in seconds. Attending to the number of paths detected, we observe a heterogeneous distribution, with *benchmarks* such as *cactusADM* and *lbm* with only 1 independent path detected, to benchmarks like *povray* with 27 paths detected.

To validate our approach we apply step by step the energy estimation process using the application signature to each benchmark listed in the Table 2.3. The first step is to choose the value of α to establish the threshold of retired instructions for the partial execution of each independent execution path. The value of α is equal to 0.1, i.e., the partial execution of each path must be stop once the retired instructions reach 10% of the total estimated executed instructions of that independent path. We choose that value of α to have a CR that could reach to 10 so the application signature could estimate the dynamic CPU and memory energy with a runtime 10 times shorter than the actual execution of the whole application. Table 2.3 shows relative errors for the dynamic CPU ($Error_{CPU}$) and memory ($Error_{Mem}$) estimated energy defined in the Eq. 2.5, where x is for CPU or Mem. The values of E_{CPU} and E_{Mem} represent the dynamic CPU and memory energy of the whole execution of the application. The values of $\widehat{E}_{CPU}$ and $\widehat{E}_{Mem}$

are the estimated energy from the application signature.

$$Error_x = \frac{\left|E_x - \widehat{E}_x\right|}{E_x} \times 100 \tag{2.5}$$

The overall error for the dynamic energy of CPU and memory is below 8.0% except for the *cactusADM* benchmark with an 18% CPU and memory energy error. The error value from the *cactusADM* is explained by the behaviour of the Instructions per Cycle not being steady during the execution of the only independent execution path detected leading to function *bench_staggeredleapfrog2*. We can see that the estimated energy using the application signature performs with almost the same accuracy when calculating both CPU and memory energy consumption.

In the case of the parallel applications (from *blackscholes* to *SP*) shown in the Table 2.3 the application signature is applied to a single-threaded execution. The CR is almost the same for all the benchmarks with a value around 10, meaning that the application signature estimates the energy with an execution almost 10 times faster than the original execution of the benchmark. These values of CR are expected since we choose a value of α equals to 0.1.

Table 2.4: Results of energy estimation using the application signature for multi-threaded applications ($Error_{CPU}$ (%), $Error_{Mem}$ (%))

	Numbers of threads										
	2	3	4	5	6	7	8	9	10	11	12
blackscholes	1.47	0.20	0.45	0.62	4.26	0.94	3.49	7.17	7.27	14.31	14.79
	1.72	0.29	1.18	0.32	3.21	2.05	4.65	8.38	8.48	15.68	16.21
freqmine	20.26	25.71	27.37	25.50	19.77	2.56	2.99	3.35	6.80	8.50	10.11
	20.67	26.55	28.64	27.11	21.56	1.11	4.63	5.01	8.58	10.35	12.03
swaptions	0.02	3.50	1.53	12.06	15.08	10.80	1.77	7.49	24.72	20.71	23.48
	0.34	2.84	0.49	10.99	13.86	9.46	0.09	5.99	23.76	19.60	22.44
canneal	3.99	1.80	1.22	0.55	12.54	6.90	0.69	5.02	17.19	20.64	4.13
	4.08	1.96	1.46	0.24	12.26	6.58	0.30	5.50	17.79	21.31	3.69
streamcluster	0.70	3.48	6.39	10.86	24.95	4.62	1.27	6.45	10.46	11.83	14.78
	0.58	3.27	6.08	10.48	24.63	4.21	1.75	7.00	11.07	12.51	14.24
fluidanimate	4.83	-	11.05	-	-	-	6.78	-	-	-	-
	4.49	-	10.15	-	-	-	5.65	-	-	-	-
BT	3.07	0.53	1.53	9.49	25.12	0.11	0.96	0.57	9.59	3.45	0.96
	3.46	1.24	0.53	8.39	24.24	0.87	1.98	1.58	10.77	4.56	2.04
CG	2.13	0.99	3.48	9.04	24.94	5.13	0.90	5.93	9.46	6.02	18.76
	2.11	0.99	3.52	9.09	25.00	5.12	0.92	5.99	9.53	6.28	18.60
FT	1.61	0.70	1.11	3.30	19.53	19.27	20.26	23.19	27.02	25.35	17.55
	1.96	0.03	2.12	2.07	18.62	20.48	21.41	24.30	28.17	26.38	18.42
LU	3.54	0.25	1.02	7.58	23.99	20.49	15.42	13.52	4.96	13.39	26.99
	3.87	0.85	0.18	6.62	23.23	19.78	14.61	12.63	3.88	12.44	26.30
SP	3.55	0.25	4.43	11.14	31.49	19.64	17.64	17.13	13.35	19.32	27.52
	3.90	0.91	3.59	10.19	30.81	18.89	16.84	16.27	12.36	18.42	26.75

The results for the energy estimation from multi-threaded applications are shown in Table 2.4 for both the dynamic CPU (upper-row) and Memory

(lower-row) energy errors. For example, the dynamic CPU energy error for the benchmark *blackscholes* is equal to 14.79% when 12 threads are executed. This means that our approach can estimate with a 14.79% error the dynamic CPU energy when 12 threads of *blackscholes* are executed without the need to run the whole application and using the application signature from the single-thread version of the application. Running the application signature from the single-thread version of the application is in the majority of the cases a good choice when compared to estimate the energy by running the multi-threaded application since the CR is fairly high.

It should be noted that in the case of our experiments when an application was executed with more than 10 threads almost no application show a speed-up over 10. Therefore, a CR of 10 clearly compensates estimating the energy with the application signature from the single-thread version of the application. The benchmark *fluidanimate* can only be executed with a number of threads equals to a power of two. We can see that the benchmark *freqmine* has high errors in the range from 2 to 6 threads because the actual speed-up is higher than the target speed-up, so the linear speed-up model for this range of target threads is not accurate. Also, as an overall result the dynamic energy errors for an execution of 6 threads is high because the actual speed-up of all the benchmarks when are run with 6 threads is notably lower than a speed-up equals to 6. The overall RMSE of all the errors shown in Table 2.4 is 12.7%.

2.5 Conclusions

In this chapter, we proposed the use of an application signature to make a fast estimation of the dynamic CPU and memory energy consumption of large scale or HPC applications. The use of the application signature is validated in an offline manner with a set of sequential CPU-intensive, memory-intensive and multi-threaded applications, showing an overall error below 8.0% when compared to the dynamic energy of the whole execution of the application, when the applications are single-threaded.

For a multi-threaded scenario the RMSE is equal to 12.7% when the dynamic energy extracted through the application signature is compared against the complete parallel execution of the original application. The application signatures presented an average Compression Ratio equals to 9.8, which allowed to estimate the dynamic energy consumption almost 10 times faster when compared with the execution time of the whole application.

The results show that it is possible to estimate the energy of the applications without the need to execute them completely by using an application signature.

Moreover, the results obtained in this chapter lead us to developed the fast energy estimation framework explained in Chapter 4.

Additionally, the results of this chapter were presented in the following international peer-reviewed conference:

- J. C. Salinas-Hilburg, M. Zapater, J. M. Moya, J. L. Ayala, "Fast Energy Estimation Through Partial Execution of HPC Applications", International Conference on Application-specific Systems, Architectures and Processors (ASAP) (2018).

In the next chapter we present the related work associated with this work. We will show similar approaches to our application signature concept and other methodologies to estimate the energy in data centers.

Chapter 3

Related Work

In this chapter we present the related work associated with this work. The related work is divided in 4 sections: i) application signature, where we show works that use similar approaches to our application signature concept, ii) energy, power and performance estimation, where we describe different solutions to estimate energy, power or performance of the applications that run in data centers, iii) server power modeling, shows the state of the art regarding CPU, memory and server power models. Additionally, the state-of-the-art for power modeling co-allocated scenarios, iv) dynamic profiling, showing how traditional approaches extract information through the full execution of the applications and also identifying the program phases. Finally, v) energy-aware task scheduling, where we present different task scheduling approaches that are energy-efficient.

3.1 Application Signature

In the previous chapter (Chapter 2) we explained our proof of concept for the application signature. There are other works that predict or estimate performance by using a similar concept of our application signature or in some cases, they use partial execution of the applications.

In the work presented by Wong et al. [124] [125] the application signature is used to predict the performance of multi-threaded applications. They extract the application signature by executing the whole application on a platform *A*. Then, the application signature is used to predict the performance on a different platform *B*. The methodology to create the signature and predict the performance is called Parallel Application Signatures for Performance Prediction (PAS2P) and consists of two stages:

1. Signature generation: in this stage the whole application is instrumented

and executed in a base machine (platform A). The data collected through the profiling of the application is used to characterize the computation and communication parts of the parallel application. This allows to identify and extract the most relevant program phases. A ***program phase*** is defined as a set of intervals or sections in time within a program's execution profile that have similar behavior. The point of this stage is to extract the phases, the number of times they occur (weights) and their execution time. These three components form the application signature.

2. Performance prediction: in order to predict the performance of the application in a different machine (platform B) the signature (each phase) is executed and the execution time of each phase is measured. To estimate the performance of the entire application all the measured execution times from each phase are aggregated and multiplied by their respective weight.

They are able to predict performance with an average accuracy greater than 97%, using a set of well-known parallel applications such as the NAS Parallel suite [11] applications. The work by Canillas et al. [20] uses the approach described in [124] [125] (Wong et al.) to extract the application signature and then predict the execution time for a range of different input data sizes of the original application. To do this they execute the application signature with different input data sizes and derive a model of the execution time as a function of the input data size using a classical regression approach.

Yang et al. [126] uses a partial execution of the application to predict the performance. They have the same goal as the work by Wong et al., to predict the performance of the application in a different machine (platform B). The application is completely executed in a reference platform A and then, in platform B the application is partially executed to predict the performance in that machine. They argue that most of the long-running scientific type of applications have a highly repetitive nature and therefore, is likely to extract enough information in a short partial execution. They can predict the performance through three ways:

- Base Prediction Model via Cumulative Averages: the simplified process to predict the performance is as follows:

    1. The application is completely executed in a reference machine platform A and its execution time is measured (T_{ref}).
    2. The application is partially executed a number of m times in the reference machine (platform A). The execution time is measured in each partial execution and then the average time execution is calculated (t_{ref_avg}).

3. The same process is applied in the target machine (platform B), resulting in an average execution time equals to t_{tar_avg}.
4. A relative performance R_{tar_ref} is calculated through the ratio t_{tar_avg}/t_{ref_avg}.
5. Finally, the performance is calculated by multiplying the relative performance with the execution time of the application in the reference machine: $T_{estimated} = R_{tar\_ref} \times T_{ref}$.

- Prediction via Filter Model: the filter model captures and filters out the initial fluctuations of the partial executions. This makes possible to keep using the base prediction model by adjusting and enhancing the average partial execution times.

- Prediction via Sliding Window: this method enhances the previous methods (cumulative averages and filter model) by using a sliding window through the execution of each partial execution.

For steady and repetitive applications (best case scenario) they obtain 97% performance prediction. In the worst cases scenarios they obtain performance prediction errors in the range of 5 to 37%.

The work presented by Sodhi et al. [117] extracts a performance skeleton of an application. The performance skeleton is a synthetically generated application that has the same fundamental execution characteristics as the original application and is also as short-running as possible. The execution time of the skeleton is scaled down by a factor of K when compared with the original execution of the application. The simplified steps to create the skeleton are as follows:

1. Execute the application completely to capture: CPU activity, memory address trace and communications patterns.

2. Summarize the execution into a compact execution signature: identifying phases of similar activities and capturing loops structure.

3. Construct the application skeleton from the execution signature: reducing the number of loops iterations by a factor of K and converting the application behaviour into C code segments.

They obtain an average error in predicting the performance equal to 6%, using a set of parallel applications from the NAS Parallel suite [11] applications.

Another interesting work based on an application signature is from Jayakumar et al. [63]. They present a performance prediction framework that has a static

code analysis step together with a runtime analysis step, similar to our work. The framework predicts the performance of HPC applications using single small scale executions. Each execution profile is analyzed to extract the respective phases, then each phase is compared against kernel execution profiles stored in a database. A prediction engine collects this information to obtain a performance prediction. They refer to application signatures to the execution profiles of the small scale complete executions. They obtain performance predictions with errors in the range 0.4-18.7%. Although this is a very robust work there are some main differences we want to avoid in our present work: 1) even if the small scale executions are short in time they execute more than one which is not viable if we have a large batch of applications, and 2) their framework needs to have more than one version of the application with different inputs in order to obtain the small scale executions, hence making it difficult to automate the performance prediction since some user interaction is needed to obtain different sets of inputs. In our present work, we avoid the complete execution of the whole application, moreover our fast energy estimation framework only needs the original application and the original input. Our framework does not need to rely on different versions of the same application (with a set of different inputs) nor a database of previous executed kernels.

The concept of Dwarf Code is presented in the work by Zhang et al. [132]. The Dwarf Code is defined as a shorter running benchmark that mimics the behaviour of the original parallel application and is used to predict the performance of the application in a target machine (platform B). The Dwarf Code is extracted in a reference machine (platform A) through a complete profiling of the original application. They idea is to generate a sequential version of the original application with the execution phases compressed in a small code (dwarf). They obtain performance prediction errors below 10%.

The work presented by Combs et al. [33] shows that the power consumption behaviour of large scale applications can be captured with power signatures. They defined power signatures as a representation of a power trace that preserves information about application-specific power behavior. The power signatures are extracted through a full profiling of the applications and then calculate a series of statistical features that represent the extracted power profile.

All these works present a similar concept of our application signature and they inspired our concept of application signature, as well as the fast energy estimation framework. Although, they relay on a previous execution of the whole original application in order to either build the application signature or to predict the performance. In scenarios of long-running applications, performing a full execution to either build an application signature or use another technique for performance estimation is not viable since is not an energy efficient process. Moreover, in these

previous works when the input dataset of the application changes the whole process to extract the application signature must be redone. An accurate performance estimation is key to estimate correctly the energy. In our work, the application signature is used to estimate the energy (or performance and mean power) of the application and is obtained without the need to execute the whole original application.

3.2 Energy, Power and Performance Estimation

In Chapter 4 we will describe the fast energy estimation framework that uses the application signature to estimate the energy of the long-running applications. There are works that proposed different methodologies to predict either power, energy or performance for long-running, scientific and HPC applications through collected data from the complete execution of the applications.

Shoukourian et al. [115] proposed a methodology to build power and energy consumption models of multi-threaded applications taking as input to the models the available history of the power and energy data from parallel applications. The model was validated for strong and weak scaling applications. The model takes as input an application energy tag and the number of nodes the application will used during execution. They achieve energy prediction errors below 5.2%.

The work described by Chetsa et al. [27] shows a methodology to estimate the energy in large scale systems. They build a DNA-like structure of an application that represent the phases of the program. The application is completely executed in a reference platform to extract the DNA-like structure. Moreover, to estimate the energy in a target platform the application is executed and its DNA-like is extracted in runtime. When the extracted DNA-like structure matches with a given percentage of the DNA-like structure from the reference platform the execution is stopped. The energy is estimated by establishing a relationship between the measured energy from the reference platform and the measured energy of the partial execution from the target platform. They present results showing the methodology can save up to 19% of energy with less than 4% performance loss.

A server power model is presented in the work by Arjona et al. [6] that is used to estimate the energy consumption of data centers. First, they perform a thorough breakdown of the power of the subsystems inside the server. Next, they use this information to model the server power and build an energy estimation model. Although, the methodology needs as a parameter the total active cycles of the execution to estimate the performance which is obtained through a full execution of the application. They obtain energy estimation errors below 7%.

The work by Sirbu et al. [116] shows a data-driven model to predict the power consumption of applications through a dedicated monitoring framework from the Eurora system. The system is capable to collect the power with high resolution (5-second intervals) of all the processing components (CPU, GPU and MIC). They collected the power data from one year and build a power model using Support Vector Regression (SVR). The regression features for the SVR model are the type of processing unit used by the job (CPU/MIC/GPU), runtime, name, number of nodes and the number of the same node components used by other jobs. The regression target of the model is the power. Therefore, the power prediction derives from historical trace data of the users rather than architectural metrics such as hardware counters. They obtain a power NMRSE (normalized root mean squared error) below 20%. Lee et al. [75] predicts power and performance in multicore-based systems. They need to train the system with several executions of the applications to find the optimal configuration for each prediction.

A performance model for long-running scientific applications is presented by Sadjadi et al. [105]. The performance model is constructed by doing a full profile of the execution of the application without using intrusive technique such as instrumentation or code inspection. They measured performance errors within 10%. Another performance model for iterative applications is presented in the work by Lu et al. [82] by using a fine-grained basic block level profiling of the applications, with performance prediction errors below 13%. Sato et al. [109] present a methodology to build a performance model by using a previous trace data of the applications.

Zhang et al. [133] estimates the performance for multi-threaded applications using the LLVM framework. The methodology build a sequential version of the original multi-threaded application to estimate separately the computation and communication times. The goal is to execute the sequential version of the application in a reference node to estimate the performance in a set of target nodes. The performance error obtained is equal to 10.86%. A similar approach is presented by Zhai et al. [128] using deterministic replay to acquire the sequential computation time, obtaining performance prediction errors below 7%.

The work by Escobar et al. [39] presents a performance prediction of parallel applications using a fractal model. In order to find the model a series of small scale execution of the original application must be performed. The fractal model can be used to predict the performance of larger data inputs. This method presents performance prediction errors around 12%. In a previous work [40], the same authors use a similar approach of running small scale executions to predict the performance. They extract phases of the small scale executions and match those phases with a set of kernels. The performance prediction errors range from 1%

to 15%. A similar work is presented by Huang et al. [55], where they developed a performance model using polynomial regression based on a previous execution with a set of sample small inputs to predict the performance for a larger input. They obtain a prediction accuracy of 97%.

There are works that predict performance for large scale applications in runtime. Curtis et al. [34] built a performance model using multivariate regression techniques that is used runtime when there is concurrency changes during the execution. Another runtime performance estimation is done by Li et al. [78] where they use artificial neural networks to predict in an online way the execution time of each separated function of the application.

Static code analysis can also be applied to estimate either the energy or performance of the applications. Liqat et al. [80] show a methodology to estimate the energy of single-threaded small programs through a pure static code analysis using the LLVM intermediate representation (IR). The energy estimation errors are in the range of 1% to 17%. Grech et al. [49] also present a methodology to estimate the energy of single-threaded small programs using the LLVM IR, obtaining energy estimation errors up to 20%. Moreover, there are interesting approaches that combine static code and dynamic analysis of the applications to estimate the energy or the execution time. Liqat et al. [79] present an approach to estimate the energy of single-threaded small programs by combining static code analysis and a dynamic profiling of the basic blocks of the program, obtaining energy estimation errors below 12%. Additionally, the work by Mera et al. [90] estimates the performance of single-threaded programs using static code analysis and a dynamic profiling of the program to calculate a cost function. This cost function allows to estimate the performance of the program according to a set of input parameters. On the one hand, using static code analysis to estimate either energy or performance is an interesting and in some cases useful approach since there is no need to execute completely the application. On the other hand, static code analysis is usually bounded for small programs used in embedded systems and also, perform a static code analysis on multi-threaded long-running applications is often not achievable.

Our proposed energy estimation framework using the application signature does not rely on collected power or energy data from the whole execution. This allows to develop and implement proactive energy optimization policies that are not feasible through a full dynamic profiling of long-running applications. Table 3.1 shows a summary of how our proposal outperforms previous works in the field. As aforementioned, the main advantage of our fast energy estimation framework is that there is no need for a previous dynamic profiling of the application, unacceptable in long-running applications. Additionally, our

Table 3.1: Comparison of our present work against other works

	No need for a previous profiling or a historical trace data	Energy Estimation	Power Estimation	Performance Estimation	Support for Multi-threaded Applications
Shoukourian et al. [115]	✗	✓	✓	✓	✓
Chetsa et al. [27]	✗	✓	✓	✓	✓
Arjona et al. [6]	✗	✓	✓	✓	✓
Sirbu et al. [116]	✗	✗	✓	✗	✓
Lee et al. [75]	✗	✓	✓	✓	✓
Sadjadi et al. [105]	✗	✗	✗	✓	✓
Lu et al. [82]	✗	✗	✗	✓	✓
Sato et al. [109]	✗	✗	✗	✓	✓
Zhang et al. [133]	✗	✗	✗	✓	✓
Zhai et al. [128]	✗	✗	✗	✓	✓
Escobar et al. [39] [40]	✗	✗	✗	✓	✓
Huang et al. [55]	✗	✗	✗	✓	✓
Liqat et al. [80]	✓	✓	✓	✓	✗
Grech et al. [49]	✓	✓	✓	✓	✗
Liqat et al. [79]	✗	✓	✓	✓	✗
Mera et al. [90]	✗	✗	✗	✓	✗
Wong et al. [124] [125]	✗	✗	✗	✓	✓
Canillas et al. [20]	✗	✗	✗	✓	✓
Yang et al. [126]	✗	✗	✗	✓	✓
Sodhi et al. [117]	✗	✗	✗	✓	✓
Zhang et al. [132]	✗	✗	✗	✓	✓
Jayakumar et al. [63]	✗	✗	✗	✓	✓
Combs et al. [33]	✗	✗	✗	✓	✓
Our present work	✓	✓	✓	✓	

framework is able to estimate energy and not only power or performance separately, bringing the possibility to apply energy-aware optimization policies.

3.3 Server Power Modeling

In order to estimate the energy of the applications we use power models as a function of hardware counters that are described in Section 4.2. In this section we will explain traditional strategies to collect or model the power from the servers of the data center.

Energy-minimization resource management techniques usually require the *a priori* knowledge of the power consumption attained by servers when running a specific workload. Utilization and Instructions per Cycle (IPC) have traditionally been used as metrics to predict the power [10, 18], using linear or quadratic models [43]. However, these metrics are not sufficient to derive accurate models for the power consumption of highly-multithreaded enterprise servers [127]. To overcome this limitation, hardware counters have been used to model CPU and memory power [77], as they provide information on workload or application characteristics. Previous works perform classical feature selection (e.g. correlation analysis, or Principal Component Analysis) to extract relevant parameters for a set of applications. Then, models are fitted using classical regression methods such as linear regression or multiple linear regression [91]. These methods require user interaction to discover the relevant features, train and test the models, therefore they are not automatic. Our server power modeling, on the contrary, leverages the usage of unsupervised techniques that automatically perform feature selection for power prediction. Moreover, we also consider the contribution of leakage and fan power to overall server power.

Other power-saving mechanisms, such as power-capping, reduce the voltage-frequency setup (DVFS) of servers to minimize power. Due to the lack of models able to predict power consumption under arbitrary workloads and frequency constraints, some approaches experimentally derive the power-frequency curves of each application [24]. Others enforce a power cap by using feedback controllers [100]. Newer mechanisms such as Intel's Running Average Power Limit (RAPL) [35] automatically implement that feature on newer Sandy Bridge systems. However, the former case requires profiling all the incoming tasks, whereas the latter sets an arbitrary power limit by decreasing frequency, potentially degrading performance. Our server power modeling approach, on the contrary, predicts the power consumption of tasks under various frequency setups, by using application characteristics measured via hardware counters, overcoming the limitations of previous approaches. Moreover, it can be

combined with frequency optimization policies to leverage energy efficiency.

Our server power modeling approach, also tackles power prediction in task co-allocation. Research in this area is mainly focused on consolidation in virtualized environments [36]. However, due to the high number of cores in today's servers, it is common in HPC clusters to find a certain degree of task co-allocation (i.e. two different multi-threaded tasks running on the same server). Previous work addresses this challenge by trying to attribute the total CPU power to each of the tasks [129], or by deriving the power consumption of co-allocated tasks given the power of individual tasks using regression techniques [30], obtaining errors around 10% in overall server power. As opposed to our approach, theirs require profiling all new incoming tasks to obtain per-application power profiles. Because our power model is robust to task co-allocation, we only need application parameters. Thus, power consumption for co-allocated tasks can be predicted using the parameters of individual applications, reducing error when compared to previous approaches.

In our work, we use a server power model to estimate the energy of the applications. Instead of feeding the power model with the information of a full dynamic profile we feed the power model with the information (hardware counters) gathered with the execution of the application signature.

3.4 Dynamic Profiling

As we have previously indicated the traditional approach to either estimate energy, power or performance is to make a full dynamic profiling of the application. Dynamic profiling has been a useful technique to characterize and optimize the performance of the applications for many decades [4] [25]. The dynamic profiling can be used to characterize and analyze energy [92] [110], power [66] [48] or performance [84] [23]. The dynamic profiling is not limited to the application level, it can also be used to profile complete data centers infrastructures [103] [29]. In our work, we do not perform a full dynamic profiling of the application. Although, we do perform a dynamic profiling of the application signature which is a short running version of the original application.

Traditional approaches to estimate energy, power or performance usually use the concept of program phases to build models or characterize applications. As we previously defined, a program phase is a set of running-time sections that have the same behaviour. The concept of program phases has been used since many years [114] and is still used today to leverage the build of energy or power models. The program phases are extracted through a full dynamic profiling of the applications and can be obtained through three ways:

i) Basic Block Vectors (BBV): a basic block is defined as a section of code with one entry point and one exit point. A Basic Block Vector is a vector containing the set of basic blocks of the application [112] [113]. The set of BBV can be used to simulate programs in fast way in simulators such as SimPoint [52] [97]. Additionally, the BBV can be obtained through specific tools, such as Valgrind [93]. Finally, the BBV extraction is not limited to single-threaded applications as also can be obtained for multi-threaded applications [64] [65].

ii) Time-series of different metrics: the execution phases can be extracted directly through the time-series profile of the application. The phases can be identified through the time-series of the architectural features such as the hardware counters [60] [62] [61] [68]. Moreover, the phases can be extracted in an online way [26].

iii) Signal processing: finally, the phases can be detected by using signal domain transformations of the time-series collected through the execution of the application. For example, there are several works that show how to extract the program phases through the wavelet domain [56] [22] [28].

These previous works extract program phases to characterize (to build power or performance models) the applications by doing a full execution of the applications. In our work, we estimate the energy by using an application signature which is built from independent execution paths (defined in Section 2.1) of the whole application. Each independent execution path can be seen as a phase of the application in a coarse-grained fashion. Moreover, it is important to emphasize that each independent execution path is extracted without the need to execute the application completely.

3.5 Energy-Aware Task Scheduling

In Chapter 5 we use the information provided by the application signature to apply different energy-aware task scheduling approaches. In this section, we show different works and techniques used to derive energy-aware task scheduling policies.

There is an extended research on using energy efficient task scheduling approaches for energy savings in data center. The work proposed by Wang et al. [121] presents scheduling heuristics to reduce energy consumption from the execution of parallel tasks in a cluster. They developed two algorithms: the Power Aware Task Clustering (PATC) and the Power Aware List-based Scheduling (PALS). The PALS algorithm employs the ETF (Earliest Task First)

heuristic which takes into account the execution time of the tasks. Etinski et al. [41] developed a parallel job scheduling policy for improving the efficiency of power budget techniques. The policy is called MaxJobPerf and is based on integer linear programming.

The work by Auweter et al. [8] presents an energy-aware task scheduler to improve energy savings of supercomputers. They introduce a prediction model that forecast performance and power of large-scale applications. Finally, in the work by Mämmelä et al. [85] is shown an energy-aware scheduler that can be applied to HPC data centers. They used energy-aware variations of the FIFO (First In First Out) and Backfilling schedulers and also, presents a very detailed power consumption model.

In all of the previously commented works they assume the existence of either energy, power or performance of the tasks that will be executed in the data center. Whereas, in our work we use the information provided by the application signature to estimate the energy and apply an energy-efficient task scheduling approach.

The task scheduling approaches can be implemented in the form of Integer Linear Programming, or by using metaheuristics or heuristics methodologies. In the case of Integer Linear Programming based approach there is a great amount of research. Goldman et al. [47] presents a Mixed Integer Linear Programming task scheduling approach for parallel independent tasks. They present the MILP formulations for either fragmented or non-fragmented systems. A fragmented system is one where each thread of the task does not need to be using a continuous set of resources, i.e the threads of the parallel task does not need to be running on the same processor. The work developed by Chretien et al. [31] shows a task scheduling approach using successive Linear Programming approximations. They used an iterative Linear Programming scheme to find the optimal makespan.

Metaheuristics approaches can find near optimal solutions in much less computation time than Integer Linear Programming scheduling approaches. Lei et al. [76] proposed a scheduling approach based on a co-evolutionary algorithm for green data centers. In the work by Fidanova [44] a Simulated Annealing technique is used to schedule task efficiently in a grid computing scenario. They compare the results of the scheduling with another metaheuristic called Ant Colony Optimization. Kashani et al. [67] shows a task scheduling method based on Simulated Annealing to minimize the makespan in distributed systems.

Finally, heuristic methods allows to find good solutions with much less computation time than Integer Linear Programming and metaheuristic approaches. Reda et al. [102] presents a Sort-Mid algorithm for efficient scheduling in grid computing. The algorithm uses the execution times and the

number of resources the tasks will use during the execution. The work presented by Garefalakis et al. [46] shows a cluster scheduler for long-running applications. They implement both an Integer Linear Programming and a heuristic based scheduling approach.

In our present work we use and implement three different scheduling approaches based on Mixed Integer Linear Programming, Simulated Annealing and a heuristic approach based on the Longest Task First method. The main goal of the experimental work presented in Chapter 5 is to validate the use of the application signature with different scheduling approaches.

In the next chapter we will described the full implementation of the fast energy estimation framework that uses the concept of the application signature to estimate the energy of the applications. Moreover, we will show the methodology to obtain the server power models.

Chapter 4

Fast Energy Estimation Framework

In this chapter we present the design and implementation of the fast energy estimation framework. The framework is able to estimate the dynamic CPU and memory energy of large scale long-running applications. In Chapter 2 we present a proof of concept where we show how it is possible to estimate the energy using an application signature without the need to execute the original application completely. The results obtained in Chapter 2 motivate the development of the fast energy estimation framework presented in this chapter.

Additionally, we present the process to obtain the server power models used in this work. We explain the use of Grammatical Evolution techniques to obtain the dynamic CPU and memory power models.

4.1 Fast Energy Estimation Framework Modules

The overall proposed framework is shown in Fig. 4.1. The framework takes the source code, the binary and the input data of the original application to estimate, through an automatic process, the dynamic CPU and memory energy of the application without the need of a full execution. This framework uses the concept of application signature developed in Chapter 2. The overall framework runs in a target platform, where the original application will be executed. The framework is composed of the following independent modules:

1. *Call Graph Set*
2. *Estimation of Executed Instructions*

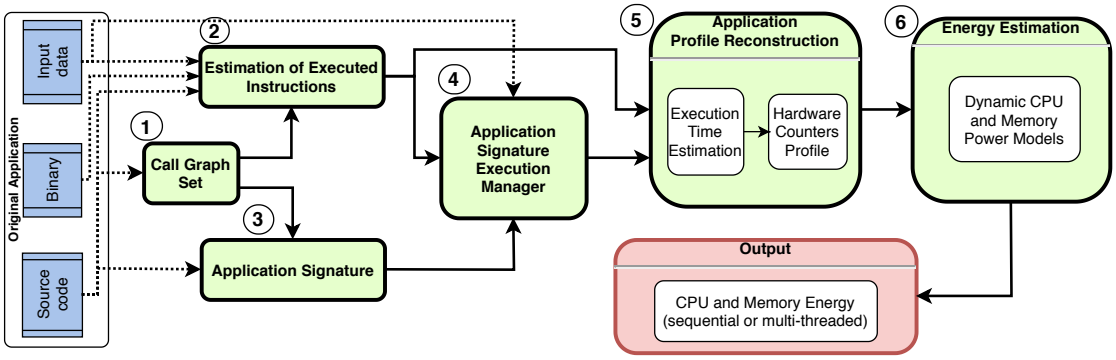


Figure 4.1: Overview of the fast energy estimation framework modules

3. *Application Signature*
4. *Application Signature Execution Manager*
5. *Application Profile Reconstruction*
6. *Energy Estimation* modules.

The fast energy estimation framework is designed in a modular way. This allows to change the functionality of any module without compromising the functionality of the whole framework. In this section we propose an implementation for each module which works for any compiled language.

1. The **Call Graph Set module** takes as ***input*** the source code of the original application and the ***output*** is the Call Graph Set (CGS) which is a set of Call Graphs for each independent execution path of the original application. The Call Graph is a Control Flow Graph (CFG) that represents the calls between the functions of the application. As we previously defined in Section 2, the ***independent execution path*** is an execution path from the Call Graph of the original application obtained through the following process:

   i) start the path search at the root node (main function).

   ii) if an edge can be followed, do so.

   iii) if not, stop the path search.

2. The **Module of Estimation of Executed Instructions** estimates the number of executed instructions for each independent execution path without executing the whole application, i.e, via a static profiling approach. The module takes as ***inputs*** the source code, binary and the input dataset of the original application, and the Call Graph Set. The ***output*** of this module

is the estimated executed instructions of each independent execution path. From the source code we extract the upper bounds of each loop of the original application and from the binary we extract the CPU instructions from each independent execution path. Additionally, this module takes into account the input dataset (e.g. number of timesteps, or the size of a matrix) of the original application since this information affects the upper bound values of the loops and therefore the number of executed instructions.

3. The **Application Signature module** creates the application signature taking as ***inputs*** the source code of the original application and the Call Graph Set. The ***output*** is the application signature and is composed by the binaries of each independent execution path, as shown in Fig. 4.4.

4. The **Application Signature Execution Manager module** takes as ***inputs*** the estimated executed instructions of each independent execution path, the application signature and the input dataset of the original application. This module executes the binaries from the application signature taking as input to each binary the input dataset of the original application and gathers a set of hardware counters profiles for each executed binary. The ***output*** of this module is the hardware counters profiles of each independent execution path obtained from the application signature execution.

5. The **Application Profile Reconstruction module** builds the application profile of the whole execution of the application. This module takes as ***input*** the hardware counters profiles and the estimated executed instructions of each independent execution path. The ***output*** of this module is the reconstructed application profile. The reconstructed application profile is composed by the reconstructed hardware counters profiles of each independent execution path. This reconstructed application profile is equivalent, in terms of energy, to the original application profile obtained through a dynamic profiling of the whole execution of the original application.

6. The **Energy Estimation module** estimates the dynamic CPU and memory energy of the application taking as ***input*** the reconstructed application profile. This module has two power models: the dynamic CPU and memory power models, both as a function of the hardware counters profiles. The ***output*** of this module is the CPU and memory estimated energy obtained from the CPU and memory estimated power profiles.

It should be noted that each step of the framework is performed automatically and none of the steps requires a manual interaction by the user. Additionally, it is important to point out that the input dataset of the original application is used by the framework to 1) estimate the executed instructions, and 2) execute the application signature. Hence, the fast energy estimation framework is dependant on, and is tuned to, the input dataset of the application. This fast energy estimation framework can be applied, without difference in the energy estimation accuracy, to any compiled languages that provides the binary of the original application since this is a key element to estimate the executed instructions. Interpreted languages, such as Python, that are able to be compiled and create a binary can also be used by this fast energy estimation framework.

It is important to notice that it is very interesting and useful to estimate or predict the power profiles of the applications. However, we center our focus on the energy since it encapsulates power and performance. The energy metric is directly related to energy savings in data centers which has an impact on the carbon footprint and the operational cost of data centers. Furthermore, energy estimation can be used to deploy energy-aware proactive task scheduling policies, aiming to reduce the total energy consumption of the data center, which is explained in Chapter 5.

In the following sections we show a detailed explanation of the design and implementation of each module of the fast energy estimation framework.

4.1.1 Call Graph Set

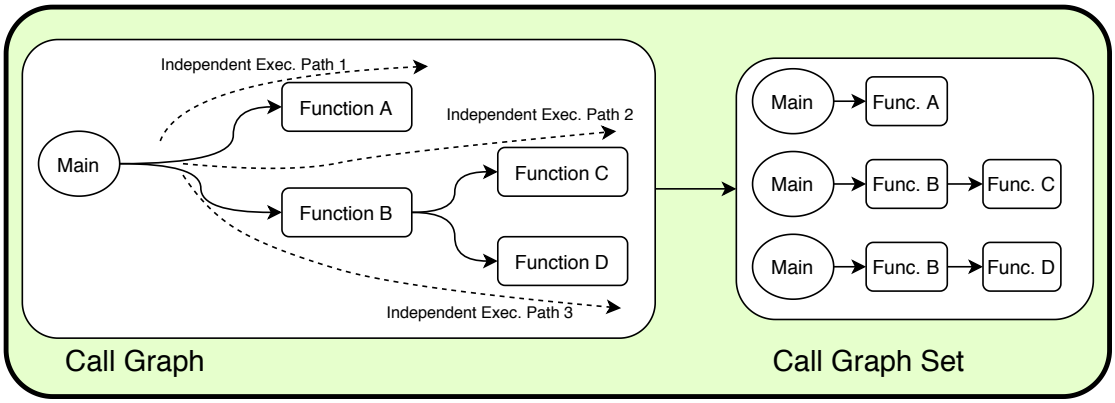


Figure 4.2: Call Graph and Independent Execution Path

The Call Graph, as shown in Fig. 4.2, provides information of the calls between the functions of the application. This module is equipped with the open-source tool *Doxygen* to extract the complete Call Graph of the original

application. Doxygen is known to have great accuracy and it is capable of working with several programming languages.

The Call Graph of each independent execution path is obtained by applying a partitioning algorithm to the complete Call Graph obtained through Doxygen. We save every Call Graph of each independent execution path in a Call Graph Set (CGS). As previously mentioned, the Call Graph is a directed graph where each node is a function and each edge is a call between functions. We apply a graph processing process similar to the Depth-First Search algorithm to the Call Graph of the original application to extract the independent execution paths, as shown in Algorithm 1.

The algorithm uses a recursive function ($EP - recursive$) that takes as input the Call Graph of the application (G), a node (s) from the Call Graph and a *path* composed by a series of connected nodes. The process starts at the root node (main function), for each neighbour node (w) the algorithm checks if the node has a neighbour (line 7). If this is false an independent execution path is found and the path is stored in the CGS (line 15). In case the node has a neighbour, the recursive function $EP - recursive$ is called again until no neighbours are found (line 9). For each call to $EP - recursive$, a node s is added to the *path* (line 6). The algorithm stops when all the nodes are marked as visited (line 3 and line 4). The process to extract each independent execution path explained in Algorithm 1 is automatic.

Figure 4.2 shows an example of this process. By applying the graph processing algorithm to the original Call Graph we obtain three independent execution paths:

- Main → Function A
- Main → Function B → Function C
- Main → Function B → Function D

4.1.2 Estimation of Executed Instructions

The module of Estimation of Executed Instructions estimates the CPU executed instructions for each independent execution path from the Call Graph Set. This is an independent module, hence the process proposed to estimate the executed instructions does not interfere with the construction of the application signature nor the application signature execution manager. The process is automatic and has the following phases, as shown in Fig. 4.3:

Phase 1: The binary of the original application is disassembled to obtain the CPU instructions of each function and the loop regions are delimited with *entry* and

Algorithm 1 Call Graph Set

```
1: path = []
2: function EP-RECURSIVE(G,s,path)
3:     if all g of G are visited then                ▷ g are all the nodes from G
4:         stop
5:     else
6:         path = path + s
7:         if s has a neighbour then
8:             for all w of s: do                    ▷ w are the neighbours of s
9:                 EP-RECURSIVE(G,w,path)
10:            end for
11:            mark s as visited
12:            update.path⇒ erase nodes marked as visited
13:        else
14:            mark s as visited
15:            CGS = append.path
16:            update.path⇒ erase nodes marked as visited
17:        end if
18:    end if
19: end function
```

exit tags. This process is done with the disassembler tool from the *MAQAO* [37] framework. Once the loops inside each function are delimited, we count the CPU instructions inside each function and each loop regions.

Phase 2: The upper bound limits of the loops are calculated. To do this, the source code of the original application is modified, compiled and then executed following these steps in an automatic way:

- A *print* function is added to print and label each loop. Also, the variables of the loop are printed.
- A *break* (or *exit*) function is added at the end of each loop to stop the execution of each loop at the first iteration.
- The modified source code is compiled with the same conditions (compiler flags) as the original application to create a new binary.
- The binary of the modified source code is executed with the input data of the original application. The output of this execution provides information about the beginning and the end of the execution of each loop. Also, the information of the upper bound limits of each loop is shown with the following

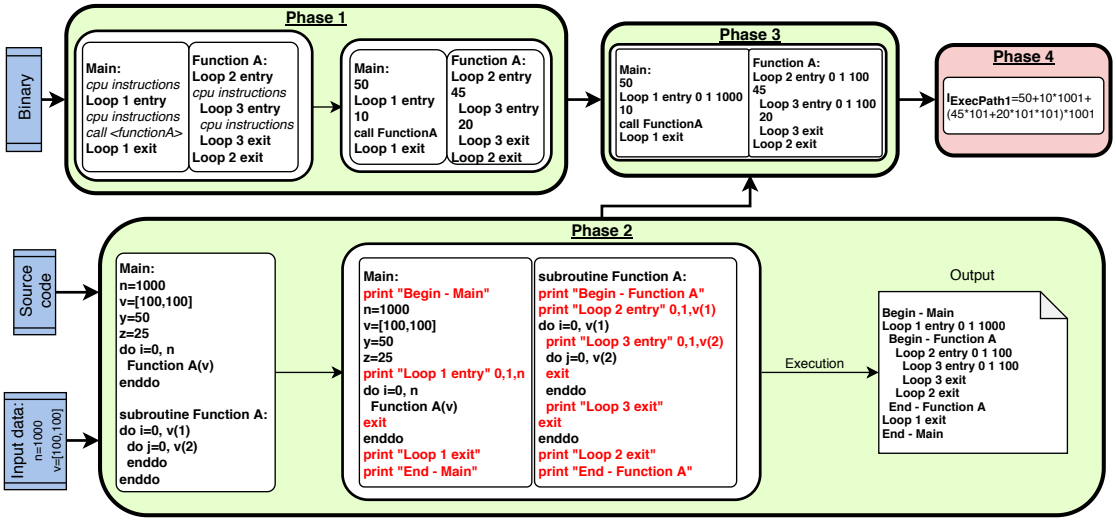


Figure 4.3: Overview of the estimation of executed instructions process (for the independent execution path 1 obtained from the Call Graph Set)

format: *loop n entry a inc b*, where n defines the loop number within the function, a is the beginning value of the iteration, b is the end value of the iteration and *inc* is the increment of the iteration values of the loop. The output of this execution is saved to a file.

It should be noted that the modified source code is executed with the input dataset of the original application since in many iterative applications the upper bound limits of the loops are defined via the input data (i.e. matrix size, number of time steps, etc.). Moreover, the steps explained in **Phase 2** are exclusively performed to calculate the upper bound limits of the loops and should not be confused with the process of building and executing the application signature. Also, it should be noted that in **Phase 2** each loop is executed one iteration.

Phase 3: The information of the output from the execution of the modified source code (upper bound of each loops) and the information from the disassembled binary (CPU instructions of each independent execution path) is combined. The final output has the number of CPU instructions of each function and the number of CPU instructions within each loop with the respective upper bound limit information.

Phase 4: The estimated executed CPU instructions for an independent execution path are calculated (I_{EP_n}). To do this, the CPU instructions inside the loops are multiplied by their respective upper bound limits.

The previous phases are applied to each independent execution path of the Call Graph Set. Therefore, the output of this module is the estimated number of

instructions for every independent execution path. To find the total number of CPU instructions ($\widehat{I}_{app}$) that the application will execute we sum the estimated CPU instructions (I_{EP_n}) of all the independent execution paths, as shown in Equation 4.1.

$$\widehat{I}_{app} = I_{EP_1} + I_{EP_2} + ... + I_{EP_n} \tag{4.1}$$

To calculate the upper bound limits of the loops we consider the case of loops with a constant termination. In general terms, the for-loop cases shown in Loop 1 and Loop 2 represent two typical nested for-loop cases found in many iterative applications. The first case is a nested for-loop where the upper bound limit of each for-loop is a constant that is defined either in the source code of the application or in the input data of the application. The second for-loop case is a nested for-loop where the upper bound limit of the inner loops depends on the iteration value of the outer loops. In the case Loop 1 the number of iterations is straightforward: $M+1$ for the i for-loop, $(M+1)\times(N+1)$ for the j for-loop and $(M+1)\times(N+1)\times(P+1)$ for the k for-loop. In the second case Loop 2 the number of iterations is less straightforward since the inner loops depend on the iteration of the outer loops but it can be calculated as: $M+1$ for the i for-loop, and for the inner loops is shown in Equation 4.2. For example, for the j for-loop the number of iterations can be expressed as: $\sum_{x=M+1}^{N+1} x = \frac{(N+1)(N+2)}{2} - \frac{(M+1)(M)}{2}$* We can also have more nested loops with this iteration value dependence and have a general case equal to: $\sum_{x=m}^{n} x^p$, that can be solved using the Faulhaber's formula.

$$\begin{aligned} \sum_{x=m}^{n} x &= \frac{n(n+1)}{2} - \frac{m(m-1)}{2} \\ \sum_{x=m}^{n} x^2 &= \frac{n(n+1)(2n+1)}{6} - \frac{m(m-1)(2m-1)}{6} \end{aligned} \tag{4.2}$$

The previously explained process has some limitations regarding conditional branches (i.e, IF/ELSE statements). On the one hand, the estimation of instructions executed takes into account the branches that are directly dependant of the input data of the application. On the other hand, IF/ELSE statements that depend on data inside the functions are bypassed and therefore all the instructions for those branches (both the IF and the ELSE) are counted in the estimation process. This leads to an overestimation of the number of executed

*It is important to notice that the iterations begin at zero.

Loop 1 For-loop case type 1

```
for i ← 0 to M do
    for j ← 0 to N do
        for k ← 0 to P do
        end for
    end for
end for
```

Loop 2 For-loop case type 2

```
for i ← 0 to M do
    for j ← 0 to N − i do
        for k ← 0 to P − j do
        end for
    end for
end for
```

instructions. In our experiments, this overestimation leads to a maximum of 2.6% error when the total estimated executed instructions are compared against the real executed instructions of the application.

4.1.3 Application Signature

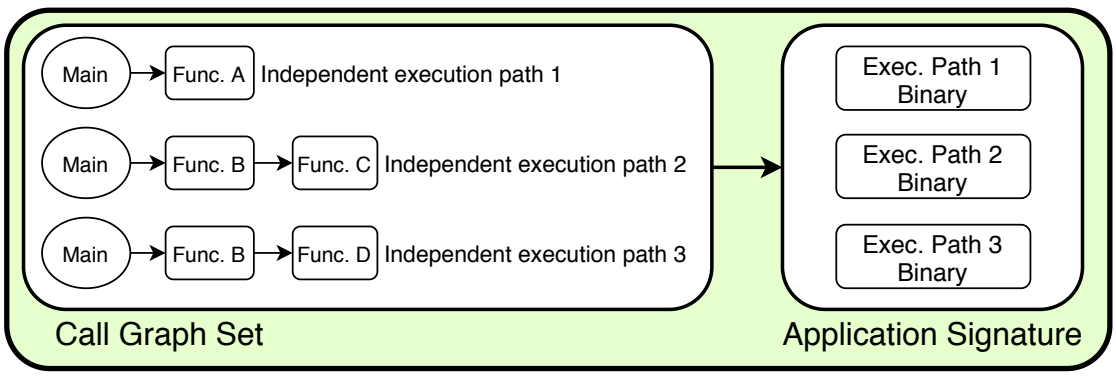


Figure 4.4: Application Signature from the Call Graph Set

The application signature is built in an automatic way by taking the information of each independent execution path from the Call Graph Set. To build the application signature we apply the following process:

- An independent execution path from the Call Graph Set is selected.
- A temporary copy of the source code of the original application is made.

- We modify the copy of the source code to remove all the independent execution paths except the one selected from the Call Graph Set. We do this by removing the call to functions that are not in the selected independent execution path.
- The modified source code is compiled with the same compiler and flags as the original application to create the binary of the selected independent execution path.
- The temporary copy of the source code is erased.
- We repeat the process for all the independent execution paths from the Call Graph Set.

From the process explained previously it can be seen that a binary is generated for each independent execution path. It is done this way to ease the use of the stop criterias (explained in the following section) and execute each independent execution path in a parallel manner by the Application Signature Execution Manager.

In the case shown in Fig. 4.4 the Call Graph Set is composed by three independent execution paths: Main → Function A, Main → Function B → Function C and Main → Function B → Function D. Finally, by applying the process previously explained the application signature is set up with three binaries from each independent execution path.

From Fig. 4.4 we can see that each independent execution path of the application signature starts from the Main function. In iterative applications, it is usually found that the Main function contains the code to load the input data and initialize the required set of variables. This variable initialization allows to execute each independent path successfully. However, there is a limitation for some specific applications that exhibit data dependency between functions from different independent execution paths. This problem will be tackled as a future line of work by using a modular compiler such as LLVM [72].

LLVM* is a modular and reusable compiler infrastructure that provides the tools to optimize, transform and analyze source code in an easy way. Using LLVM would made our fast energy estimation framework general for other programming languages. With the help of LLVM, we can refine the process to build the application signature by performing a data dependency analysis between the functions of the applications. The data dependency analysis will allow to add functions to independent execution paths that were not originally on that independent path.

*https://llvm.org

The benchmarks used to evaluate the accuracy of the framework in this Chapter have strongly independent execution paths, meaning that there are not data dependencies between them. Moreover, the input data of the benchmarks we use to evaluate the real implementation of the framework are defined in the Main function (for example, there is no independent function for data initialization). Therefore, each independent execution path is executed taking into account the original input data since each path starts at the Main function.

As we have previously commented, it is outside of the scope of this thesis to detect the data dependencies between independent execution paths. Although, a slicer tool called *llvm-slicer* can be used to detect data dependencies between independent paths, especially to detect the data dependencies coming from data initialization functions. As we have mention earlier, a proper future development of the whole framework is proposed to be implemented using LLVM and its tools, for example the *llvm-slicer* tool.

4.1.4 Application Signature Execution Manager

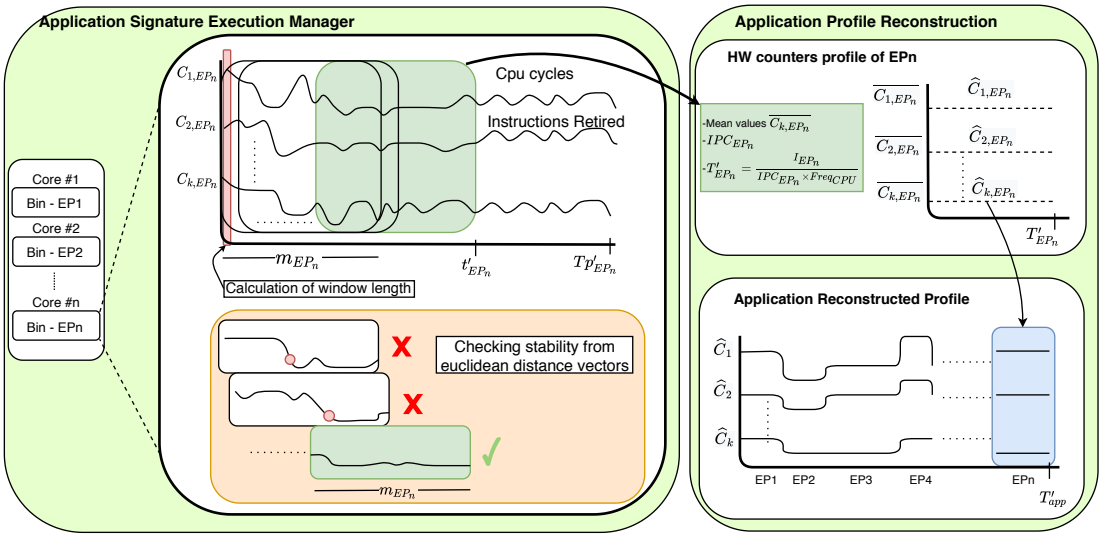


Figure 4.5: Application signature execution manager and application profile reconstruction

The application signature manager has three main objectives: (i) executes the binaries that compose the application signature, (ii) stops at the right time the execution of each binary and (iii) gathers a set of hardware counters from each execution. The inputs of this module are the application signature, the estimated executed instructions of each independent execution path and the input data of the original application.

Figure 4.5 shows the execution process of the application signature. Each binary from each independent execution path (*Bin - EPn*) is executed in a different core (parallel). In Fig. 4.5 we highlight the execution process of the n^{th} independent execution path (EP_n). During the execution we collect a set of hardware counters C_{i,EP_n} (see Table 4.1 from the **Experimental Setup** section).

In the case of extracting and executing the application signature for multi-threaded applications there is two options:

1. Extract and execute the application signature for the single-threaded version of the application and use a speed-up model to scale the energy estimation for the multi-threaded version of the application. This is explained in Section 2.2 in Chapter 2.

2. Extract and execute the application signature directly for the multi-threaded version of the application. In this option each binary from each independent execution path must be executed with the same number of threads as the original application.

The results for the multi-threaded scenario presented in Section 4.4.1.3.2 are obtained using the second option to extract and execute the application signature for multi-threaded applications.

In order to stop the execution of each binary and get t'_{EP_n}, the value of the execution time of the n^{th} independent execution path, two conditions or criterias are applied:

1. The execution of the binary stops once the executed CPU instructions reach a percentage p of the estimated CPU instructions I_{EP_n} (obtained from the module of Estimation of Executed Instructions). In some cases this could lead to potentially large execution times since the selected value of the percentage of executed CPU instructions can be a large number. To alleviate this problem, a second condition is applied.

2. The execution of the binary stops once the collected hardware counter profiles reach a stable value.

The Application Signature Execution Manager stops the execution whenever one of the two conditions are met. To apply the second stop criteria we calculate the length of a sliding window for each independent execution path:

- At the beginning of the execution we gather the instructions retired and CPU cycles from the hardware counters to calculate the IPC ($IPCb_{EP_n}$). This process is done in the red rectangle region shown in Fig. 4.5.

- We estimate a partial execution time Tp'_{EP_n} of the n^{th} independent execution path by taking into account the percentage p (first condition or criteria) of executed CPU instructions and the previously calculated $IPCb_{EP_n}$.

$$Tp'_{EP_n} = \frac{I_{EP_n}}{IPCb_{EP_n} \times Freq_{CPU}} \times (p/100) \tag{4.3}$$

- The estimated execution time is divided in proportional s segments. The sliding window length m_{EP_n} is equal to the length of the segment:

$$m_{EP_n} = Tp'_{EP_n}/s \tag{4.4}$$

Once we have the length m_{EP_n} of the sliding window we perform the following steps to stop the execution:

- The sliding window of fixed length of m_{EP_n} samples is selected. The values of all the k hardware counter signals inside the window form m_{EP_n} vectors of k samples (each sample is one temporal value of each hardware counter).
- The euclidean distance between the vectors is calculated to compose a new signal inside the window. Figure 4.5 shows the euclidean distance of the execution of the n^{th} independent execution path (orange region).
- An algorithm to find abrupt changes is applied to the euclidean distance signal.
- If an abrupt change (red dot) is found we move the sliding window one sample and start the process again. Finding an abrupt change means that the hardware counters signals are not stable.
- If an abrupt change is not found we stop the execution of the application signature and select the values of the hardware counters signals that are located inside this final sliding window. From Fig. 4.5 the execution stops at the green window.

The algorithm to find abrupt changes is based from the work of Lavielle [73] and Killick et al. [99]. The algorithm to find abrupt change points has the following steps:

- The signal is divided into two phases by selecting an intermediate instant point.
- A statistical property (mean) is computed for each section.

- A deviation from the statistical property is calculated for each instant from each section. The residual error is calculated for each section by summing all the deviations.

- The residual error from each section is added to obtain a total residual error.

- A new division point is selected until the total residual error finds a minimum.

Finally, the output of this module is the hardware counter profiles from the region of the last window that fulfill any of the two stop criteria. In the case shown in Fig. 4.5 the output is the hardware counter profiles from the green window.

4.1.4.1 Application Signature Execution Time

The execution time of the application signature (T_{sig}) depends on two factors: i) the value of the execution time of the n^{th} independent execution path (t'_{EP_n}), ii) the number of available cores. There are three possible scenarios that affect the execution time of the application signature:

a) If the number of independent execution paths is equal or lower to the number of available cores the value of T_{sig} is equal to the longest independent execution path: $max(t'_{EP_1}, t'_{EP_2}, \ldots, t'_{EP_n})$.

b) If the number of independent execution paths higher than the available cores, a simple FIFO task-scheduling process is applied. A task-queue is composed by a number of binaries from the application signature that wait for a core to be available. In this case, the value of T_{sig} depends on the task-scheduling process.

c) The worst case scenario, in terms of execution time, is when only one core is available. In this scenario the value of T_{sig} is equal to the sum of all the execution times of all the binaries from the application signature: $sum(t'_{EP_1}, t'_{EP_2}, \ldots, t'_{EP_n})$.

It should be noted that this approach assumes a perfectly parallel execution of the independent execution paths on each core. This is a fair assumption since the independent execution paths do not share data between them (there is no data dependency) and also, there is no message passing between the execution of each independent execution path.

4.1.5 Application Profile Reconstruction

Figure 4.5 shows the process to reconstruct the application profile. The module takes as input the hardware counters profile from the region of the green window obtained from the Application Signature Execution Manager. These hardware counter profiles are averaged, as shown in Equation 4.5. The subindex i indicates each of the k hardware counters and EP_n indicates the n^{th} independent execution path from the application. The mean is calculated over a period equal to the length of the window m_{EP_n} and for the set A formed by the temporal samples of the hardware counter profiles.

$$\overline{C_{i,EP_n}} = \frac{1}{m_{EP_n}} \sum_{j \in A} C_{i,EP_n}[j] \tag{4.5}$$

The IPC of each independent execution path are calculated, as shown in Equation 4.6. The values of C_{RI,EP_n} and C_{Clk,EP_n} are the values of the Retired CPU Instructions and CPU Clock Cycles, respectively, for the execution of the n^{th} execution path.

$$IPC_{EP_n} = \frac{\sum\limits_{j \in A} C_{RI,EP_n}[j]}{\sum\limits_{j \in A} C_{Clk,EP_n}[j]} \tag{4.6}$$

To reconstruct the application profile we estimate the execution time of each independent execution path. Once we have estimated the execution time we reconstruct the hardware counter profiles of the overall application.

4.1.5.1 Execution Time Estimation

The execution time of each independent path is estimated by applying the basic performance equation, shown in Equation 4.7. Where IPC_{EP_n} is the IPC of the n^{th} execution path. The value I_{EP_n} corresponds to the estimated executed CPU instructions of the n^{th} execution path. The value of $Freq_{CPU}$ is calculated as the mean value of the CPU Clock Cycles ($\overline{C_{Clk,EP_n}}$). The value of T'_{app} is the total estimated execution time for the whole application.

$$\begin{aligned} T'_{EP_n} &= \frac{I_{EP_n}}{IPC_{EP_n} \times Freq_{CPU}} \\ T'_{app} &= T'_{EP_1} + T'_{EP_2} + \cdots + T'_{EP_n} \end{aligned} \tag{4.7}$$

4.1.5.2 Hardware Counter Profile Reconstruction

Figure 4.5 shows the hardware counter profiles of each independent execution path ($\widehat{C}_{i,EP_n}$). These hardware counter profiles are built by creating a signal with amplitude equal to the average values of hardware counter profiles of each independent execution path ($\overline{C_{i,EP_n}}$) and with a duration equal to to the estimated execution time of each independent execution path (T'_{EP_n}), as shown in Equation 4.8.

$$\widehat{C}_{i,EP_n}[n] = \overline{C_{i,EP_n}} \qquad n \in [0, T'_{EP_n}] \tag{4.8}$$

The application reconstructed profile $\widehat{C}_i$ is obtained by concatenating each reconstructed hardware counter profile from all the independent execution paths ($\widehat{C}_{i,EP_n}$) and for all the k hardware counters set, as shown in Eq. 4.9 and Fig. 4.5. The value of sm is the sampling period used during the hardware counter acquisition process.

$$\begin{aligned} \widehat{C}_i[n] &= \left\{ \widehat{C}_{i,EP_1}[n_{EP_1}], \widehat{C}_{i,EP_2}[n_{EP_2}], \ldots, \widehat{C}_{i,EP_n}[n_{EP_n}] \right\} \\ n \in [0, T'_{app}] \; and \; & \begin{cases} n_{EP_1} & \in [0, T'_{EP_1}] \\ n_{EP_2} & \in [T'_{EP_1} + sm, T'_{EP_2}] \\ \vdots & \\ n_{EP_n} & \in [T'_{app} - T'_{EP_n} + sm, T'_{EP_n}] \end{cases} \end{aligned} \tag{4.9}$$

4.1.6 Energy Estimation

In order to estimate the energy of the original application we use power models as a function of the reconstructed hardware counter profiles built from the application signature execution.

4.1.6.1 Power Models

In Section 4.2 we will show the validation of the dynamic CPU and memory power models as a function of hardware counters using a Grammatical Evolution technique. The power models of the dynamic CPU and memory are shown in Equation 4.10, which is the same as Equation 4.20, but instead of using the hardware counters profiles of the original application we use the reconstructed hardware counters profiles $\widehat{C}_i$ (see Table 4.1). The values of x_n and y_n are the

coefficient values (see Table 4.4) of the power models for CPU and memory, respectively. The absolute power errors of the models are 4.4W and 3.7W, for the CPU and memory power model respectively.

$$\begin{aligned}\widehat{P}_{CPU,dyn} =& x_0 \cdot \frac{x_1 \cdot \widehat{C}_8 + 1}{x_2 \cdot \widehat{C}_4 + 1} + \\ & x_3 \cdot \frac{(x_4 \cdot \widehat{C}_6 + 1)(x_5 \cdot \widehat{C}_3 + 1)(x_6 \cdot \widehat{C}_2 + 1)^2}{x_7 \cdot \widehat{C}_7 + 1} - x_8 \\ \widehat{P}_{Mem,dyn} =& y_0 \cdot \frac{y_1 \cdot \widehat{C}_8 + 1}{y_2 \cdot \widehat{C}_5 + 1} + \\ & y_3 \cdot (y_4 \cdot \widehat{C}_8 + 1)(y_5 \cdot \widehat{C}_2 + 1)(y_6 \cdot \widehat{C}_3 + 1) - y_7\end{aligned} \tag{4.10}$$

As previously commented, without loss of generality we use CPU and memory power models as a function of hardware counters. However, our framework and methodology is not limited to these power models. The CPU and memory power can also be obtained through alternatives approaches such as RAPL [35] (when available) without compromising the applicability and design of the framework.

4.1.6.2 Overall Energy Estimation

The energy of CPU and memory is estimated by summing all the instantaneous dynamic CPU and memory power values and then multiplying by the sampling period sm, as shown in Eq. 4.11, where x is for CPU or Mem. The energy is estimated over a period equal to T'_{app} (the total estimated execution time of the whole execution) and for the set B formed by the temporal samples of the estimated CPU and memory power profiles ($\widehat{P}_x$).

$$\widehat{E}_x = \Big[\sum_{n \in B} \widehat{P}_x[n] \Big] \times sm \tag{4.11}$$

The estimated energy error is defined in Equation 4.12, where x can represent both the CPU or memory (Mem) error sub-index. The values of E_{CPU} and E_{Mem} represent the dynamic CPU and memory energy of the whole execution of the original application, respectively. The values of $\widehat{E}_{CPU}$ and $\widehat{E}_{Mem}$ are the estimated energy calculated from the fast energy estimation framework.

$$Error_x = \frac{\left| E_x - \widehat{E}_x \right|}{E_x} \times 100 \tag{4.12}$$

We also define the estimated executed instructions error with Equation 4.13, where I_{app} is the total executed instructions of the original application and $\widehat{I}_{app}$ are the estimated executed instructions.

$$Error_{Inst} = \frac{\left| I_{app} - \widehat{I}_{app} \right|}{I_{app}} \times 100 \tag{4.13}$$

4.1.6.3 Compression Ratio of the Framework

In Chapter 2 we defined the concept of Compression Ratio using the application signature. In this section we define the Compression Ratio of the framework CR_{fr} as the ratio of total execution time of the original application (T_{app}) to the execution time of the fast energy estimation framework (T_{fr}), as shown in Eq. 4.14. The value of T_{fr} is the sum of the execution time of each module from the fast energy framework (Fig. 4.1). A high Compression Ratio value indicates that the framework estimates the energy much faster than executing the whole application.

$$CR_{fr} = \frac{T_{app}}{T_{fr}} \tag{4.14}$$

4.2 Sever Power Modeling

Estimating power in highly-multithreaded enterprise servers running arbitrary workloads under a given frequency setup is not a trivial task. Simplified linear or quadratic models [43] exhibit high errors in these scenarios. They disregard the impact of leakage and fan power, and assume the same trend for CPU and memory power. Due to the complexity of modeling resource contention when various CPU or memory intensive tasks run on the same server, task co-allocation in non-virtualized scenarios is either not considered or limited to a particular set of known workloads [30], usually profiled off-line [130]. The validity of these approaches is limited, not matching real data center conditions.

In this section we propose a methodology to model the overall power of servers running single or co-allocated applications, given the parameters (HW counters) of each individual task. Fig. 4.6 shows how the models developed in this paper can be used together to predict overall server power. To predict the power attained by a server when jointly running Tasks 1 and 2, we first gather the HW counters of each application separately (i.e., when not sharing the server with other tasks). Then, we develop a model to estimate the HW counters when both tasks run together in

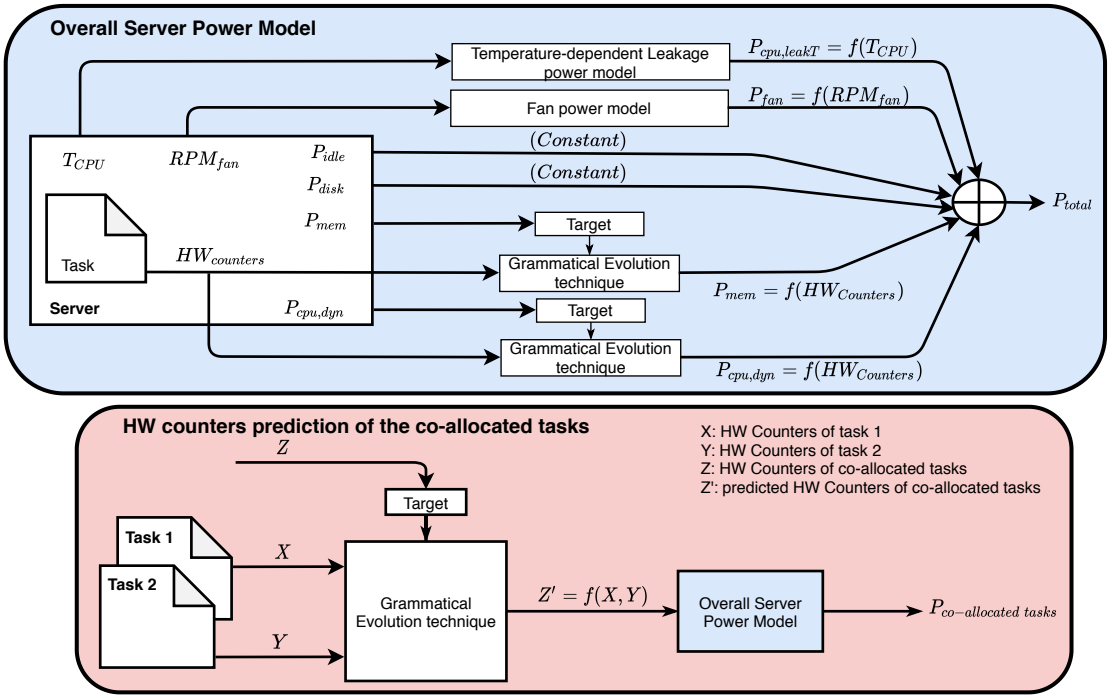


Figure 4.6: Overview of the co-allocated tasks power model methodology

the same server (i.e. co-allocated tasks, as described in Sec. 4.2.2). Finally, we use the predicted HW counters to model the power of the co-allocated tasks by using the overall server power model (described in Sec. 4.2.1).

4.2.1 Server Power Models

The proposed server power model isolates and separately quantifies the main contributors to power consumption: i) CPU power P_{CPU}, ii) dynamic memory power $P_{Mem,dyn}$, iii) fans P_{fan} and iv) disks P_{disk}. Eq. 4.15 describes the power consumption breakdown. CPU power can be further divided into three contributors:

1. P_{idle}: is the power consumption of the CPU and other components of the server (motherboard, service processor, network interface, etc) when no workload is executed; it contains a temperature-independent leakage component plus the power consumption due to the Operating System running.

2. $P_{CPU,leakT}$: represents temperature-dependent leakage, which has an exponential dependence with temperature [127].

3. $P_{CPU,dyn}$: the dynamic power consumption of the CPU due to a workload execution. As mentioned in the Experimental Setup Section 4.3.2, we are not able to measure directly CPU power consumption (P_{CPU}), so we calculate this term by subtracting all other power values (fan, memory and disk power) from overall server power (P_{total}).

The quantization error of P_{total} is equal to 4W and this value establishes a floor to the accuracy of the P_{CPU} value.

$$P_{total} = \underbrace{P_{idle} + P_{CPU,leakT} + P_{CPU,dyn}}_{P_{CPU}} + P_{fan} + P_{Mem,dyn} + P_{disk} \tag{4.15}$$

Leakage power was modeled using the same methodology than in a previous work by Zapater *et al.* [127]. To this end, we run a CPU-intensive workload under various fan speeds (fan speed can be manually set through the server BIOS), while collecting CPU power and temperature. Since leakage power depends exponentially on CPU temperature, changing fan speed under a constant workload varies CPU temperature, and thus, leakage. We regress leakage power to the second order Taylor series expansion of an exponential (see Eq. 4.16), obtaining the coefficients shown in Table 4.4. This process is done through a normalization of the T_{CPU} values. The leakage power model has an RMSE (Root Mean Square Error) of 1.14W and a MAE (Mean Absolute Error) of 1.03W.

$$P_{leak} = \alpha_0 + \alpha_1 \cdot T_{CPU} + \alpha_2 \cdot T_{CPU}^2 \tag{4.16}$$

Fan power consumption was modeled by changing fan speed while gathering fan power values via the deployed sensor, as mentioned in the Experimental Setup Section 4.3.2. As fan power has a cubic relation with fan speed, we regress fan power to a third order polynomial, as shown in Eq. 4.17. The regression coefficients obtained are shown in Table 4.4. The fan power model has an RMSE of 0.09W and a MAE of 0.05W.

$$P_{fan} = \beta_0 + \beta_1 \cdot rpm_{fan} + \beta_2 \cdot rpm_{fan}^2 + \beta_3 \cdot rpm_{fan}^3 \tag{4.17}$$

All the benchmarks used in this thesis presented a constant or very low variation of power drain of disks. Hence, for the sake of simplicity we assume a constant power drain of disks, with a mean power consumption of 16.75W and a standard deviation of 1.3W. Nonetheless, this is not a limitation for the presented power model since a more detailed disk power model can be developed in the future to enhance the model.

Power consumption when no workload is running (P_{idle}) is also constant through all the experiments, with a mean power consumption of 50.0W and a standard deviation of 2.0W.

4.2.1.1 Dynamic CPU and Memory Power Models

In this section, we present a methodology to obtain the models for the dynamic memory ($P_{Mem,dyn}$) power consumption and the dynamic CPU power consumption ($P_{CPU,dyn}$) as a function of HW counters. We propose an unsupervised modeling technique based on Grammatical Evolution (GE) that automatically extracts relevant features, while obtaining a mathematical expression for dynamic CPU and memory power. We compare our solution to a classical modeling approach based on classical feature selection and regression.

In summary, we claim that to model the overall server power we only need to collect the following parameters: HW counters, CPU temperature and fan speed, as shown in Eq. 4.18:

$$P_{total} = f(HW_{counters}, T_{CPU}, rpm_{fan}) \tag{4.18}$$

4.2.1.1.1 Classical Approach

We first present a partial least squares regression to model dynamic CPU and memory power ($P_{CPU,dyn}$, $P_{Mem,dyn}$). We use Principal Component Analysis (PCA) for feature selection, reducing the set of HW counters. In our experiments, the first 3 principal components explain 70% of the variance, therefore we plot the first 3 components and choose the HW counters that are highly independent from each other. The final HW counter set is composed of: C1, C2, C10, C11 (from Table 4.3). We train and validate our model with the training and test sets described in the Experimental Setup Section 4.3.2. Equation 4.19 shows the dynamic CPU and memory power linear regression expression. The values of γ_n and δ_n are the regression coefficients and C_m correspond to the HW counters. Regression coefficients for both models are summarized in Table 4.4.

$$\begin{aligned} P_{CPU,dyn} =& \gamma_0 + \gamma_1 \cdot C_1 + \gamma_2 \cdot C_2 + \gamma_3 \cdot C_{10} + \gamma_4 \cdot C_{11} \\ P_{Mem,dyn} =& \delta_0 + \delta_1 \cdot C_1 + \delta_2 \cdot C_2 + \delta_3 \cdot C_{10} + \delta_4 \cdot C_{11} \end{aligned} \tag{4.19}$$

4.2.1.1.2 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary technique, inspired on the biological process of generating a protein from DNA, that performs symbolic regression [104]. Given a set of composition rules that describe the mathematical relations and the variables of a model, GE generates and evolves a model until it fits the training data with the minimum error possible. As opposed to a classical approach, GE performs automatic feature selection, reducing the initial set of HW counters. We feed the GE algorithm with all the HW counters and variables we want to model (i.e. the target): CPU dynamic power ($P_{CPU,dyn}$) and dynamic memory power ($P_{Mem,dyn}$). This process is done separately for both magnitudes, obtaining one model for $P_{CPU,dyn}$ as a function of HW counters and another for $P_{Mem,dyn}$, as shown in Eq. 4.20 and Table 4.4. The absolute power errors of the models are 4.4W and 3.7W, for the dynamic CPU and memory power model respectively.

$$\begin{aligned} P_{CPU,dyn} =& x_0 \cdot \frac{x_1 \cdot C14 + 1.0}{x_2 \cdot C4 + 1} + \\ & x_3 \cdot \frac{(x_4 \cdot C10 + 1)(x_5 \cdot C3 + 1.0)(x_6 \cdot C2 + 1)^2}{x_7 \cdot C11 + 1} - x_8 \\ P_{Mem,dyn} =& y_0 \cdot \frac{y_1 \cdot C14 + 1}{y_2 \cdot C5 + 1} + \\ & y_3 \cdot (y_4 \cdot C14 + 1)(y_5 \cdot C2 + 1)(y_6 \cdot C3 + 1) - y_7 \end{aligned} \tag{4.20}$$

4.2.2 Power Prediction for Co-assigned Tasks

To predict the overall power of co-allocated applications we first predict the HW counters when two tasks run co-allocated. To this end, we model each counter of the CPU and memory models as a function of those same HW counters on the individual tasks. For the case of the classical model (Classical Approach), HW counters C1, C2, C10 and C11 were selected to obtain the models of $P_{CPU,dyn}$ and $P_{Mem,dyn}$. Thus, we predict the HW counters of the co-allocated tasks ($\widetilde{Ci}$) as a function of the HW counters of single tasks (Ci_{tj}), using classical regression to fit the regression coefficients η (see Table 4.4), as shown:

$$\begin{bmatrix} \widetilde{C1} \\ \widetilde{C2} \\ \widetilde{C10} \\ \widetilde{C11} \end{bmatrix} = \begin{bmatrix} \eta_{00} & \eta_{01} & \eta_{02} & \eta_{03} & \eta_{04} & \eta_{05} & \eta_{06} & \eta_{07} & \eta_{08} \\ \eta_{10} & \eta_{11} & \eta_{12} & \eta_{13} & \eta_{14} & \eta_{15} & \eta_{16} & \eta_{17} & \eta_{18} \\ \eta_{20} & \eta_{21} & \eta_{22} & \eta_{23} & \eta_{24} & \eta_{25} & \eta_{26} & \eta_{27} & \eta_{28} \\ \eta_{30} & \eta_{31} & \eta_{32} & \eta_{33} & \eta_{34} & \eta_{35} & \eta_{36} & \eta_{37} & \eta_{38} \end{bmatrix} \begin{bmatrix} 1 \\ C1_{t1} \\ C1_{t2} \\ C2_{t1} \\ C2_{t2} \\ C10_{t1} \\ C10_{t2} \\ C11_{t1} \\ C11_{t2} \end{bmatrix} \tag{4.21}$$

For the GE model, the counters C2, C3, C4, C5, C10, C11 and C14 were automatically chosen as features to model $P_{CPU,dyn}$ and $P_{Mem,dyn}$. We feed the GE algorithm with these counters of the single tasks to obtain the counters of the co-allocated tasks (target). We derive 7 models, one per HW counter, as shown next:

$$\begin{aligned}
\widetilde{C2} &= m_0.C11_{t1} + m_1.C11_{t2} - m_2 \\
\widetilde{C3} &= n_0 \frac{(n_1.C3_{t2}+1)(n_2.C11_{t1}+1)}{n_3.C2_{t1}+1} - n_4 \\
\widetilde{C4} &= p_0.C4_{t2} + p_1.C4_{t1} + p_2 \\
\widetilde{C5} &= q_0.C5_{t2} + q_1 \frac{q_2.C5_{t1}+1}{q_3.C14_{t1}+1} - q_4 \\
\widetilde{C10} &= r_0(r_1.C10_{t2}+1) + \\
&\quad r_2 \frac{(r_3.C2_{t2}+1)(r_4.C14_{t1}+1)(r_5.C10_{t1}+1)}{r_6.C2_{t1}} - r_7 \\
\widetilde{C11} &= \frac{s_0(s_1.C2_{t2}+1)(s_2.C5_{t2}+1)(s_3.C4_{t2}+1)(s_4.C3_{t2}+1)(s_5.C10_{t1}+1)}{s_6.C14_{t1}+1} + \\
&\quad s_7 \frac{s_8.C11_{t1}+1}{s_9.C4_{t2}+1} - s_{10} \\
\widetilde{C14} &= w_0.C4_{t2} + w_1 \frac{(w_2.C11_{t1}+1)(w_3.C14_{t1}+1)(w_4.C4_{t1}+1)}{w_5.C5_{t2}+1} - \\
&\quad w_6 \frac{(w_7.C4_{t2}+1)(w_8.C3_{t2}+1)}{w_9.C2_{t2}+1} + w_{10}
\end{aligned} \tag{4.22}$$

4.3 Experimental Setup

In this section we present the experimental setup for the validation of the fast energy estimation framework and the server power models.

4.3.1 Fast Energy Estimation Framework

Table 4.1: Hardware counters collected during the execution of the application signature

	Description		Description
C1	Clock cycles	C5	Branch instructions retired
C2	Instructions retired	C6	Resource stalls
C3	LLC misses	C7	μops dispatched
C4	L2D Cache misses	C8	L1D Cache misses

The validation of the fast energy estimation framework takes place in an Intel enterprise server (S2600GZ) based on the Intel Decathlete 2.0 Open Compute Project server board. The server has one Intel 6-core SandyBridge-EP processor with 12 hardware threads, 8 4GB memory modules, 4 hard disk drives, 5 fans and 2 power supply units. The server runs a CentOS 6.5 Linux OS. The *ocount* tool, included in the open-source *Oprofile* tool, is used to gather hardware counters during runtime. The hardware counters are polled every 100 ms. Table 4.1 shows the hardware counters collected during the execution of the application signature. The hardware counters C2 to C8 were obtained from the automatic feature selection of the Grammatical Evolution modeling approach (Section 4.2.1.1). The *Doxygen* tool is used to extract the Call Graphs of the applications. The open-source framework *MAQAO* [37] is used to disassemble the binary of the original application and also, to detect the regions of the loops in the disassembled binary.

To validate the accuracy of the fast energy estimation framework we use an heterogeneous set of long-running iterative workloads composed of: the applications *BT* and *SP* from the NAS Parallel suite [11], *Stream* [88], *Dgemm* [83] and *Linpack* [38] benchmarks. For the multi-threaded scenario we use the *OpenMP* versions of *BT* and *SP*. These applications are formed by computational intensive kernels, are iterative and according to their input are long-running. We use two different inputs for each application, shown in Table 4.2. The input datasets are chosen in order to have feasible experiments (regarding execution time) but also maintaining long-running execution times.

Our experiments are centered around long-running, data-intensive and iterative applications. In this work, we have not evaluated the accuracy of the framework with stream processing applications, although the fast energy framework should work with them. Stream processing applications process data using different types of short applications or kernel operations. As long as we have the binary and source code of this short applications or kernel operations, we can apply the fast energy estimation framework, and benefit from the compression ratio and fast energy/performance estimation. On the contrary, the framework is not conceived to work with interactive or latency-sensitive applications. The main problem for this type of applications is the estimation of the executed instructions (which is necessary for the overall execution time estimation) that cannot be accomplished following our methodology and, probably, requires a complete redesign of the approach.

Table 4.2: Input dataset for each application

Applications		Inputs
BT	BT-B	Class B
	BT-D	Class D
SP	SP-B	Class B
	SP-D	Class D
Stream	Stream1	Stream Array Size=10^9
	Stream2	Stream Array Size=1.05^9
Dgemm	Dgemm1	Input matrix size N=1024
	Dgemm2	Input matrix size N=2048
Linpack	Linpack1	Time Steps ntimes=20000 Size of matrices=300
	Linpack2	Time Steps ntimes=30000 Size of matrices=200

4.3.2 Server Power Modeling

The experiments to obtain the server power models take place on the same Intel S2600GZ server mentioned in the Experimental Setup from the Fast Energy Estimation Framework (Section 4.3.1). Figure 4.7 shows a diagram of the server internals. We use the IPMI to poll three server sensors: i) overall server power consumption (W), ii) CPU temperature (ºC) and iii) fan speed (RPM). Since the server is not equipped with power sensors for CPU, fans, memory or disks, we deploy intrusive current measurement sensors in three board components: i) one fan, ii) one memory DIMM and iii) the 4 disks. To obtain power values, we use

the commercial chip from Texas Instruments INA219. To monitor the fan and disks we placed the sensor in series with the power supply of each component. Only one power sensor is used for the fans, as the server fan control firmware always drives all fans to the same speed. Thus, the total fan power can be obtained by multiplying the measured fan power by the number of fans. Regarding memory, to measure the power of one memory DIMM, we inserted a memory extender that incorporates a shunt resistor to enable power measurement. We characterized memory power by running several memory-intensive experiments with the synthetic benchmark Randmem *, and changing the DIMM we were monitoring. This way, we experimentally validated that memory power consumption is equally spread across all DIMMs, regardless of the workload run. Therefore, to measure overall memory power we can multiply the value from the power sensor by the number of DIMMs in the server.

To gather the HW counters of the applications, we use *ocount*. Table 4.3 shows the HW counters collected in each execution of the applications executed to build the server power models. This set of HW counters are proved to be correlated with the power consumption of running tasks [32]. All the parameters (CPU temperatures, fan speed, HW counters, and overall, fan, disk and memory power) are gathered every 10 seconds.

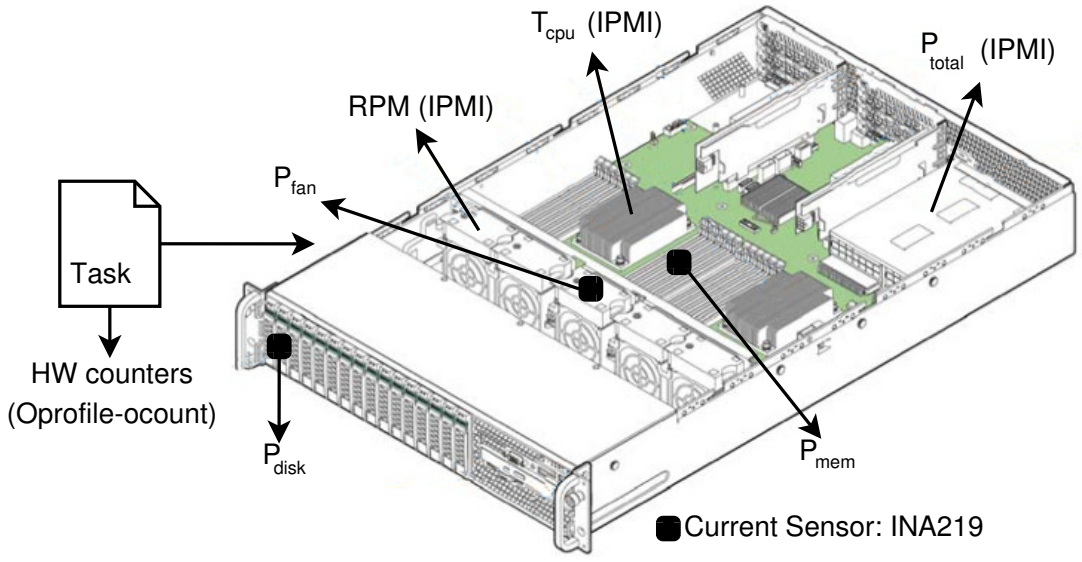


Figure 4.7: Experimental Setup. Parameters collected during the experiments[n]
[n] This figure is based on a image found in [58].

Table 4.4 shows all the coefficient values for all the models developed to build the overall server power model.

*https://github.com/greenlsi/randmem

Table 4.3: HW counters collected to build the server power models

	Description
C1	Clock cycles
C2	Instructions retired
C3	LLC misses
C4	L2D Cache misses
C5	Branch instructions retired
C6	Mispredicted branches retired
C7	Speculative cache-line split load μops dispatched to the L1D
C8	Speculative cache-line split Store-address μops dispatched to L1D
C9	Number of times the divider is active
C10	Resource stalls
C11	μops dispatched
C12	Memory transactions
C13	μops with memory accessed retired
C14	L1D Cache misses

To develop our overall server power model we run a subset of workloads from the SPEC CPU 2006 [54] benchmark suite, the PARSEC [16] [17] benchmark suite and Randmem. The SPEC and PARSEC benchmarks are selected according to their CPU and memory power characteristics, i.e. to train our model we choose 6 benchmarks with very different CPU and memory power profiles that sweep a wide range of values. The training set consists on: *lbm*, *calculix*, *gcc*, *bodytrack*, *blackscholes*, *streamcluster* and *randmem*. To validate our model we use a different set of benchmarks: *mcf*, *perlbench*, *bzip2*, *freqmine*, *raytrace* and *facesim*. Each SPEC/PARSEC benchmark is run for a different number of copies/threads: 1, 2, 3, 6 and 12. We use Randmem to stress various memory usages, ranging from 1 to 32GB. All experiments were run for the following frequency setups available in the server: 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000 MHz and the Turbo Boost frequency of 2001 MHz. Turbo Boost technology allows processors to run at a higher frequency than 2001 MHz, in our case at 2400MHz. We measure the real frequency by running several benchmarks while collecting Clock cycles (C1) every second.

To predict the HW counters of a co-allocated task given the HW counters of the individual tasks we run a set of experiments with co-allocated applications. We choose only two applications running together since is a typical scenario for large scale or HPC applications and long-running applications. The training set is formed by the following co-allocated set of benchmarks: *perlbench-mcf*, *bzip2-lbm*, *perlbench-calculix*, *lbm-mcf*, *gcc-perlbench* and *calculix-mcf*. The test set is

Table 4.4: Coefficient Values for all models

	Coefficient Values										
	0	1	2	3	4	5	6	7	8	9	10
α	27.50	-1.02	0.01	-	-	-	-	-	-	-	-
β	0	$4.21E{-4}$	$-6.11E{-8}$	$1.14E{-11}$	-	-	-	-	-	-	-
γ	4.10	$-1.04E{-10}$	$8.92E{-11}$	$2.58E{-10}$	$8.94E{-12}$	-	-	-	-	-	-
δ	5.13	$-4.75E{-10}$	$2.33E{-11}$	$6.06E{-10}$	$1.06E{-10}$	-	-	-	-	-	-
x	53.10	$1.00E{-10}$	$1.05E{-10}$	9.83	$4.03E{-12}$	$3.72E{-10}$	$2.49E{-12}$	$1.12E{-12}$	58.19	-	-
y	24.80	$1.00E{-10}$	$1.77E{-11}$	5.76	$1.00E{-10}$	$2.49E{-12}$	$3.72E{-10}$	24.80	-	-	-
η_0	$1.70E9$	0.78	1.29	0.06	0.08	-0.02	-0.08	-0.05	-0.10	-	-
η_1	$7.10E9$	0.01	0.32	1.47	0.63	-0.46	-0.43	-0.53	0.25	-	-
η_2	$-1.35E9$	-0.19	-0.07	-0.34	0.05	1.22	1.17	0.34	-0.01	-	-
η_3	$9.82E9$	0.37	1.32	1.17	0.12	-1.12	-1.23	-0.23	0.55	-	-
m	0.44	0.86	$5.15E9$	-	-	-	-	-	-	-	-
n	$1.44E9$	$6.24E{-10}$	$4.68E{-12}$	$5.34E{-12}$	$1.39E9$	-	-	-	-	-	-
p	0.62	0.50	$4.70E8$	-	-	-	-	-	-	-	-
q	1.27	$1.60E10$	$2.83E{-11}$	$2.23E{-10}$	$9.62E9$	-	-	-	-	-	-
r	$8.24E10$	$1.69E{-11}$	$1.88E10$	$5.41E{-12}$	$2.23E{-10}$	$1.73E{-11}$	$5.34E{-12}$	$1.02E11$	-	-	-
s	$5.97E10$	$5.41E{-12}$	$3.03E{-11}$	$2.65E{-10}$	$6.24E{-10}$	$1.73E{-11}$	$2.23E{-10}$	$2.75E11$	$4.68E{-12}$	$2.65E{-10}$	$3.06E11$
w	1.44	$1.47E9$	$4.68E{-12}$	$2.23E{-10}$	$2.59E{-10}$	$3.03E{-11}$	$9.92E8$	$2.65E{-10}$	$6.24E{-10}$	$5.41E{-12}$	$2.71E7$

formed by: *perlbench-lbm*, *calculix-lbm*, *gcc-bzip2*, *calculix-bzip2* and *mcf-gcc*. Each experiment is run for 1, 2 and 3 copies of each application (e.g *Experiment 1:* 1 copy of *perlbench* and 1 copy of *mcf*, *Exp. 2:* 2 copies of *perlbench* and 2 copies of *mcf*, and so on.). Experiments were run under two frequencies: 1700 and 2001 MHz. The train and test set were selected by randomly combining benchmarks of their respective sets.

Grammar 1 Grammar in BNF format used to obtain the power models

<expr> ::= <expr><op><expr> | <cte>*<var>*<var> | <cte>*<var>/<var> | <cte>*<var> | <cte>
<op> ::= +|-|*|/
<var> ::= C1 | C2 | C3 | ... | C14
<cte> ::= <dig>.<dig>
<dig> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Grammar 2 Grammar in BNF format used to obtain the co-allocated hardware counters

<expr> ::= <expr><op><expr> | <cte>*<var>*<var> | <cte>*<var>/<var> | <cte>*<var> | <cte>
<op> ::= +|-|*|/
<var> ::= $C2_{t1}$ | $C2_{t2}$ | $C3_{t1}$ | $C3_{t2}$ | $C4_{t1}$ | $C4_{t2}$ | $C5_{t1}$ | $C5_{t2}$ | $C10_{t1}$ | $C10_{t2}$ | $C11_{t1}$ | $C11_{t2}$ | $C14_{t1}$ | $C14_{t2}$
<cte> ::= <dig>.<dig>
<dig> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The configuration of the parameters for the Grammatical Evolution technique is the following:

- Population size: 200 individuals
- Chromosome length: 100 codons (or genes)
- Mutation probability: inversely proportional to the number of rules.
- Crossover probability: 0.9
- Maximum wraps: 3
- Codon size: 8 bits (values from 0 to 255)

The previous configuration is used for both the power models (Equation 4.20) and the co-allocated hardware counters models (Equation 4.22). Moreover, the grammar used for both modeling approaches is presented in Grammar 1 and 2.

The only difference between both grammars is the variable set definition ($\langle var \rangle$). In the case of the power models the variables are the set of hardware counters from Table 4.3. The variables from the co-allocated hardware counters models are the hardware counters from each task obtained through the co-allocated execution.

Both grammars are defined to reach non-linear expressions to obtain more complex relations between the variables. In the Appendix A the reader can find a more detailed explanation of the GE technique and how to interpret the grammars defined in Grammar 1 and 2.

4.4 Results

In this section we present the results of using the fast energy estimation framework with a set of sequential and multi-threaded long-running applications. Additionally, we present the results of the validation of the server power models.

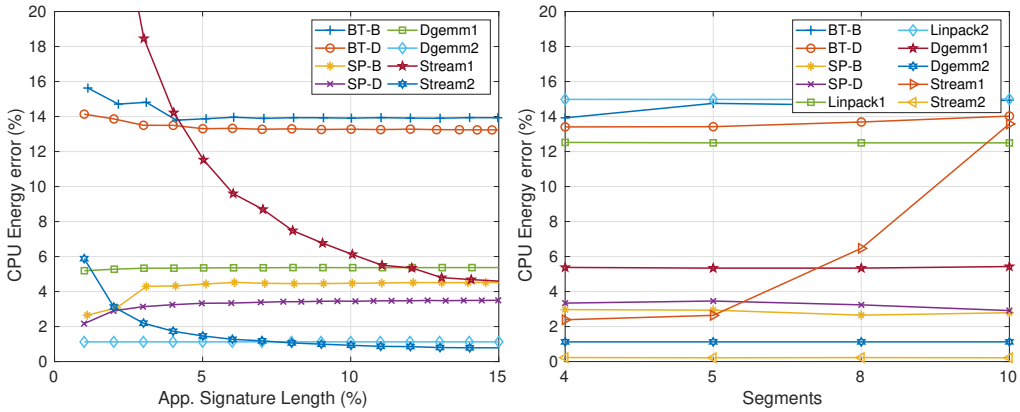
4.4.1 Fast Energy Estimation Framework

We present the results of using the framework with a set of long-running applications. First, we show the effect of the application signature length in the energy estimation process. Second, we show how the segment division (to calculate the sliding window length: m_{EP_n}) affects the estimated energy. Finally, we show the overall results for the sequential and multi-threaded scenarios.

On one hand, each step or module of the framework could lead to errors in the multiple phases of energy estimation. On the other hand, these errors are minimal and are aggregated on the final result of the estimated energy value. It is important to mention that the estimated energy value is compared against the real energy value obtained through a full execution of the application.

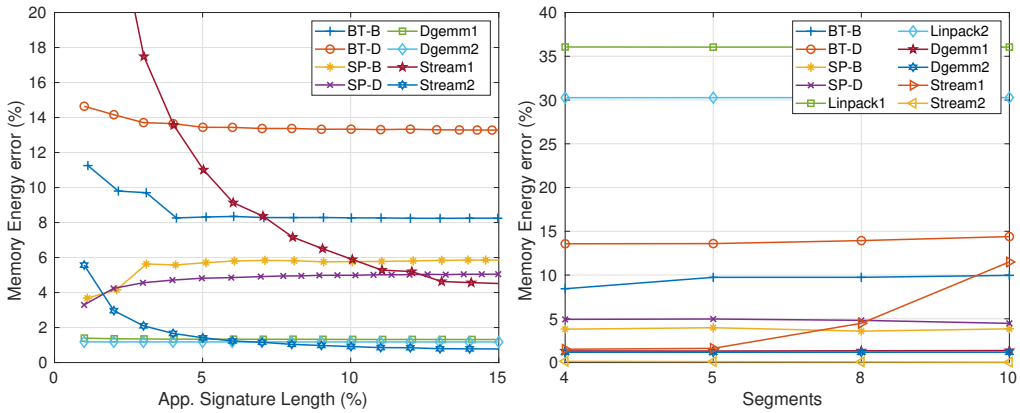
4.4.1.1 Compression Ratio and Energy Error vs Application Signature Length

In this section we present a study of the effect of the application signature length (in terms of execution time) on the energy estimation error and the CR_{fr}. The results are shown in Fig. 4.8. As previously explained, one of the criteria to stop the execution of the application signature is to execute the application signature until the number of executed instructions reaches a percentage p of the estimated executed instructions (Section 4.1.4). Therefore, the x-axis presents the percentage of executed instructions of the application signature. In this case we are not taking into account the second criteria of stability of the hardware counter signals.



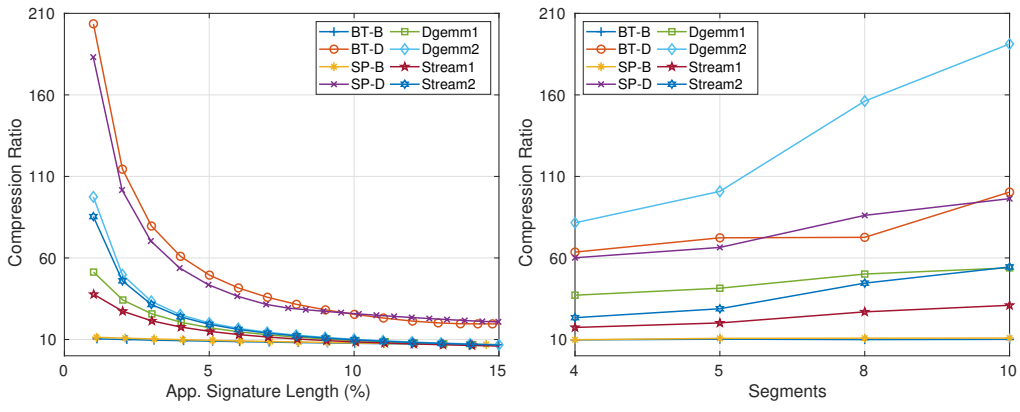
(a) CPU E. error Vs App. Sig. Length

(b) CPU E. error Vs Segment Divsion



(c) Mem E. error Vs App. Sig. Length

(d) Mem E. error Vs Segment Divsion



(e) CR_{fr} Vs App. Sig. Length

(f) CR_{fr} Vs Segment Divsion

Figure 4.8: CR_{fr} and Energy Est. Error vs (Application Signature Length - Segment Division (s))

Figure 4.8a shows the distribution of the energy estimation error for the CPU energy case as a function of the application signature length. Overall, we can see that as the application signature length increases the energy estimation error decreases and converges to a steady value. The application *SP-B* and *SP-D* exhibits a different behaviour since the energy estimation error increases as the application signature length increases and then stabilizes. This unexpected behaviour occurs because the values of the IPC obtained from the application signature execution at the beginning of the execution are closer to the IPC values of the execution of the original application. As the application signature length increases the values of the IPC also increases and therefore, the performance estimation error increases together with the energy estimation error.

Figure 4.8c shows the distribution of the energy estimation error for the memory energy case as a function of the application signature length. The behaviour is the same as the CPU and only presents differences in the energy error estimation values. For both the CPU and memory the energy estimation error is below 15% when the application signature length represents only 5% of the estimated executed instructions. Figure 4.8e shows the CR_{fr} as a function of the application signature length. As expected, the CR_{fr} decreases when the application signature length increases. By looking at Figs. 4.8a, 4.8c and 4.8e we can see how selecting a threshold to stop the execution of the application signature between 5% and 10% of the estimated executed instructions we obtain low energy estimation errors together with high CR_{fr} values. These results give the user the possibility to tune the estimation parameters in order to find the desired trade-off between execution time vs precision of the estimation.

4.4.1.2 Compression Ratio and Energy Error vs Segment Division

In this section we present the results of the energy estimation errors and CR when both stop criteria are applied to the execution of the application signature. We use a threshold of 5% (p) for the criteria of the executed retired instructions and for the hardware counters stability criteria we select four cases of segment divisions (s value in Equation 4.4 from Section 4.1.4): 4, 5, 8 and 10. The effects on the energy estimation error from the segment division are shown in Fig. 4.8. The x-axis shows the segment division (s). Figure 4.8b and Fig. 4.8d shows that for almost all applications the energy estimation errors are steady for different segment divisions, except for the application *Stream1* where increasing the segment division penalizes the energy estimation error. This occurs because the application *Stream1* has a phase at the beginning of the execution where the IPC are different than the IPC of the rest of the execution. When a high segment division is selected the stop criteria algorithm stops the application signature execution at the beginning phase. This

results on a increase of the total execution time error and therefore an increase of the energy estimation error.

Figure 4.8f presents the values of the CR_{fr}. In general, when the segment division increases the CR_{fr} increases, as expected. For the case of a segment division equal to 10 the CR_{fr} values are higher than the CR_{fr} values from only using the stop criteria of the retired instructions with a threshold of 5% (Fig. 4.8e). Therefore, we can see that when using both stop criteria the CR_{fr} increases while not greatly affecting the energy estimation errors.

4.4.1.3 Evaluation of the Fast Energy Estimation Framework

In this section we present the accuracy results of the fast energy estimation framework in terms of the energy error estimation. Also, we present the accuracy of the module of estimation of executed instructions. The dynamic CPU and memory energy estimations are obtained using the following criteria to stop the execution of the application signature: 1) 5% of the estimated executed CPU instructions, and 2) a segment division equal to 10 to detect the stability of the hardware counter profiles. We select those values because, as seen on Section 4.4.1.2, they allow to have low energy estimation errors together with high Compression Ratios.

4.4.1.3.1 Energy Estimation for the Sequential Scenario

Table 4.5 shows the energy estimation errors ($Error_{CPU}$ (%) and $Error_{Mem}$ (%)) for the single-threaded (sequential) cases. In case of the energy estimation errors for the estimated CPU energy all the values are below 15%. The highest energy estimation error is presented for the memory energy error of the application *Linpack1* with a value of 36.5%. The RMS of the CPU energy estimation errors is equal to 10.4% and for the memory is equal to 16.8%. Furthermore, we can see that the mean power absolute errors (absolute value of the difference between the mean power value of the original application and the estimated mean power value obtained from the framework), $ErrorAbs_{CPU}$ and $ErrorAbs_{Mem}$, for all the applications are low, below 1.7W. Additionally, these mean power absolute errors are below the CPU and memory power model errors (4.4W and 3.7W, respectively). It should be noted that the energy estimation module could be replaced with a more precise CPU and memory power models. As a consequence, the energy estimation framework would still be valid and the energy estimation errors would be lower.

Overall, the total execution time error (absolute value of the difference between the total execution time of the original application and the estimated

Table 4.5: Evaluation of the fast energy estimation framework. Sequential scenario

Apps	$Error_{CPU}$ (%)	$ErrorAbs_{CPU}$ (W)	$Error_{Mem}$ (%)	$ErrorAbs_{Mem}$ (W)	$Error_{Time}$ (%)
NAS BT-B	14.9	1.08	9.9	0.61	1.6
NAS BT-D	14.0	0.50	14.3	0.51	7.0
NAS SP-B	2.7	0.12	3.8	0.03	3.8
NAS SP-D	2.9	0.30	4.4	0.13	6.0
Linpack1	13.1	1.30	36.5	0.71	27.9
Linpack2	16.4	0.81	31.8	1.65	4.8
Dgemm1	5.4	0.20	1.3	0.17	2.7
Dgemm2	1.1	0.04	1.1	0.04	1.6
Stream1	13.5	0.01	11.4	0.12	13.2
Stream2	0.2	0.01	0.1	0.02	0.3

Apps	$Error_{Inst}$ (%)	CR_{fr}	T_{fr} (s)	T_{sig} (s)	**Exec. Paths**
NAS BT-B	1.0	10.1	60.3	5.6	17
NAS BT-D	0.1	100.4	547.2	490.6	17
NAS SP-B	0.2	10.9	42.0	4.8	9
NAS SP-D	0.5	96.4	403.9	365.1	9
Linpack1	1.8	33.4	42.5	20.8	5
Linpack2	2.6	13.8	35.5	13.7	5
Dgemm1	0.6	53.9	31.9	15.7	1
Dgemm2	0.1	191.2	216.6	199.5	1
Stream1	0.4	30.9	48.2	32.1	1
Stream2	0.1	54.4	282.7	265.9	1

total execution time obtained from the framework), $Error_{Time}$, is below 14%, except for the application *Linpack1* with an error equal to 27.9%. In case of this application, the memory energy estimation error (36.5%) comes from a high total execution time error (27.9% shorter than the original execution of the application). For the CPU energy estimation error (13.1%) the high mean power absolute error (1.30W over the power value of the original execution of the application) compensate the high total execution time error. The application *Linpack2* also has a high memory energy estimation error with a value equal to 31.9%. In this case, the error is originated from a high mean memory power absolute error (1.65W below the power value of the original execution of the application).

Table 4.5 shows the relative errors of the executed instructions estimation for all the applications ($Error_{Inst}$ (%)). All the estimation errors are below 3% with *Linpack2* presenting the maximum estimation error (2.6%). This means that the estimation of executed instructions process is highly accurate taking into account the we estimate the executed instructions via a static profiling and, therefore, we avoid to execute the whole original application.

The Compression Ratio values for all the energy estimations are presented

in Table 4.5. The lowest value of CR_{fr} comes from the application *BT-B* and is equal to 10.1. Nonetheless, this means that the energy estimated using the framework was obtained with an execution 10.1 times faster than executing the whole application. The application *Dgemm2* has the highest CR_{fr} value with 191.2.

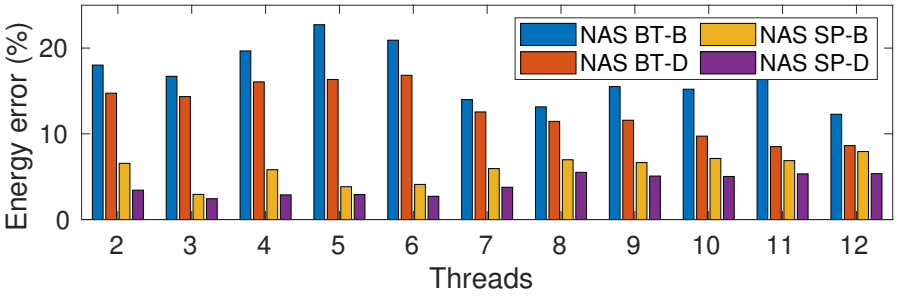
In addition, Table 4.5 presents the execution time of the whole framework (T_{fr}) and the execution time of the application signature (T_{sig}) (this value is included in T_{fr}). For example, in case of *BT-B* the framework spends 54.7s (60.3s-5.6s) in all the modules (Call Graph Set, Application Signature, Estimation of Executed Instructions, Application Profile Reconstruction and Energy Estimation) and only 5.6s executing the application signature.

Finally, Table 4.5 shows all the independent execution paths that form the application signature for each application. The application *BT* presents the highest number of independent execution paths with 17 paths. In the case of *Dgemm* and *Stream* there is only one independent execution path and it corresponds to the main function.

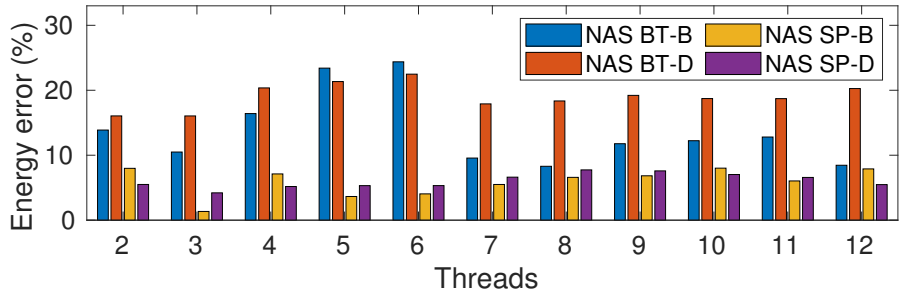
4.4.1.3.2 Energy Estimation for the Multi-Threaded Scenario

Figure 4.9a and Fig. 4.9b show the relative energy estimation error for different multi-threaded cases (from 2 to 12 threads) for the OpenMP version of the NAS Parallel applications *BT* and *SP*. We compare the energy estimation of the multi-threaded scenario using the fast energy estimation framework with the energy extracted from the real executions of the different multi-threaded cases. Almost all the energy estimation errors are below 20% for the CPU and memory energy estimation. The application *BT* presents the highest energy estimation errors around 5 and 6 threads for both CPU and memory energy estimations. The RMS of the CPU energy estimation errors is equal to 11.4% and for the memory is equal to 12.8%. Figure 4.9c and Fig. 4.9d show the mean power absolute errors for both CPU and memory. All the error values, for both CPU and memory, are below 4.0W. Additionally, all the error values are below the CPU and memory power model errors, 4.4W and 3.7W, respectively.

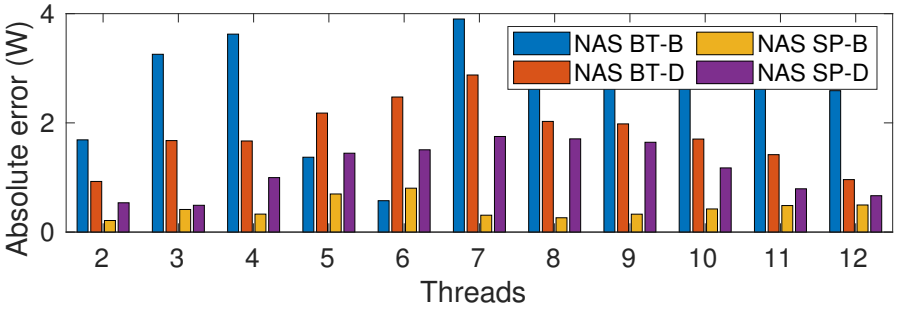
The CR values are presented in Fig. 4.10a and 4.10b. For the applications *BT-D* and *SP-D* the overall CR values are over 15. This indicates that the CPU and Energy estimation is calculated 15 times faster than executing the original multi-threaded application. The applications *BT-B* and *SP-B* present lower CR values, as shown in Fig 4.10b. since these applications are shorter in terms of execution time to *BT-D* and *SP-D*. The overall CR values are over 2, indicating that the energy estimation for these applications is achieved at least twice as fast as executing the original application.



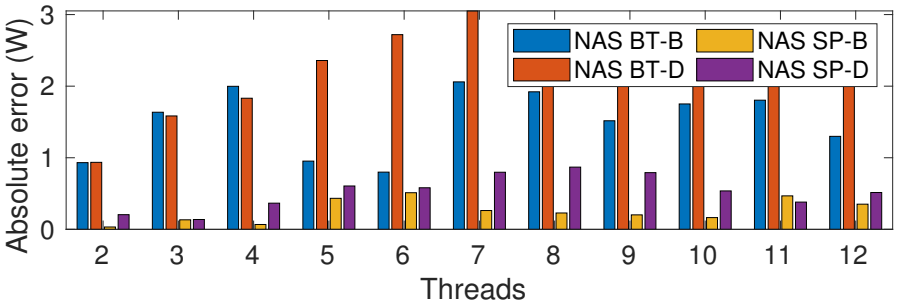
(a) CPU energy error (%) for the parallel scenario



(b) Memory energy error (%) for the parallel scenario

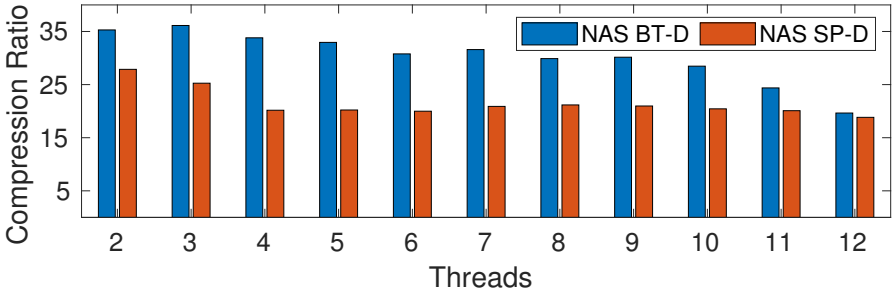


(c) CPU absolute error (W) for the parallel scenario

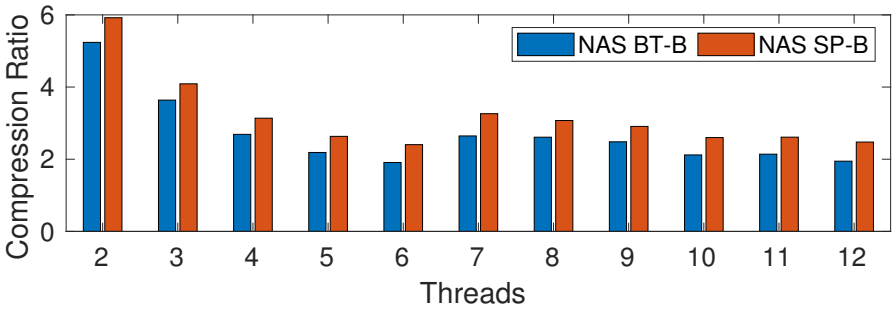


(d) Memory absolute error (W) for the parallel scenario

Figure 4.9: Energy and absolute errors for the parallel scenario



(a) Compression Ratio for the parallel scenario (Class D)



(b) Compression Ratio for the parallel scenario (Class B)

Figure 4.10: Compression Ratio for the parallel scenario

4.4.2 Server Power Modeling

In this section we present the results obtained for the overall server power model and for the power prediction of co-assigned tasks obtained by following the process in Fig. 4.6. We compare the GE technique against a classical modeling approach using PCA analysis. The main advantage of the GE technique is the automatic feature selection, as opposed to the classical approach where the feature selection is not done in an automatic way.

Table 4.6: RMSE and MAE for training and test set in classical and grammatical evolution models

	Training set				Test set			
	RMSE (W)		MAE (W)		RMSE (W)		MAE (W)	
	GE	Classical	GE	Classical	GE	Classical	GE	Classical
Overall Power (OP)	7.15	8.75	5.56	7.20	7.40	10.90	5.82	7.68
OP of co-allocated applications	8.06	13.15	6.13	10.46	9.52	12.64	7.48	9.25
OP of co-allocated applications (2vs1)	-	-	-	-	7.77	8.76	6.26	6.92
OP of co-allocated applications (4vs2)	-	-	-	-	12.86	15.91	10.14	12.50

4.4.2.1 Overall Server Power Model

The first row of Table 4.6 shows the RMSE and MAE of the overall server power model without task co-allocation, for both the GE and classical models. We observe that the GE technique has a MAE of 5.82W and a RMSE of 7.40W for the test set, improving by 32.11% the RMSE value with respect to the classical approach.

Figure 4.11 shows the value of the RMSE of the overall server power model for each frequency and copy/thread configuration. For instance, the RMSE value of the test set when the frequency is 1300MHz and we have one copy/thread is around 8W. The RMSE value decreases when the frequency increases, for 1, 2, 3 and 6 copies/threads configuration. This means that overall server power model is more accurate for higher frequencies than for lower frequencies. This can be explained by the fact that lower CPU frequencies are related to lower power values which are close to the overall idle power value. At overall idle power values the quantization error of P_{total} is more noticeable, thus obtaining a lower accuracy for lower CPU frequencies. Again, higher frequencies increase the power consumption of applications, increasing the accuracy of our model. As the main usage of this modeling would be to estimate if a workload exceeds a certain cap, obtaining higher errors at low power values is not an important limitation. We see that accuracy is lower in the case of 12 copies/threads. This is mainly caused by the impact of resource contention, which is very extreme in these cases, as we are assigning highly CPU and memory bounded applications to all HW threads in the system.

However, even in these adverse conditions our model is able to obtain errors below 10%. Another interesting case is the 2001MHz frequency (i.e. the Turbo Boost frequency, that corresponds to 2400MHz), where the RMSE rises as the number of copies/threads increases. This is caused both by resource contention and due to the underfitting of the model. In this sense, for the 2001MHz frequency we have less samples in our training sets compared to other frequencies (due to applications running faster and finishing early). Moreover, as the there is an important gap (400MHz) between the 2000MHz and the 2001MHz frequency, instead of 100MHz, our models tend to overfit lower frequencies, obtaining less accuracy for the 2001MHz. This issue is inherent to any regression technique that tries to minimize the RMSE across samples. We observe the same behavior for the classical modeling approach, but the GE techniques improve the error obtained.

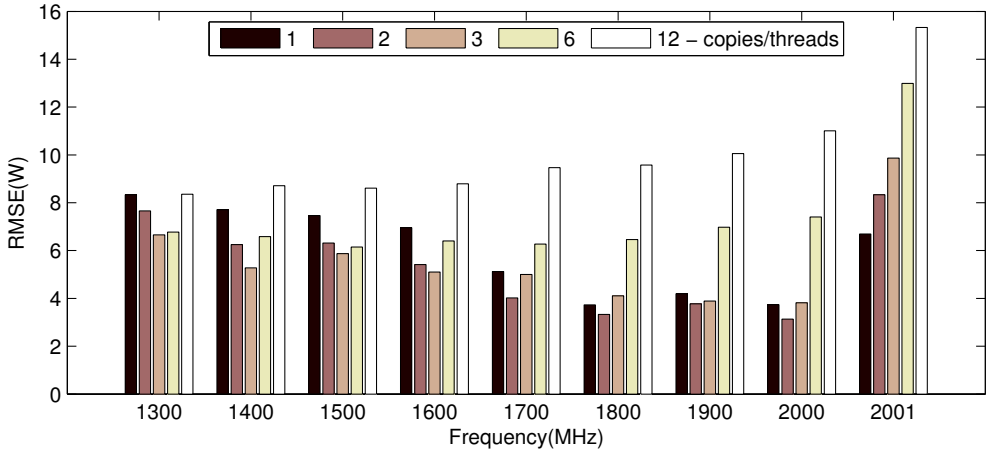


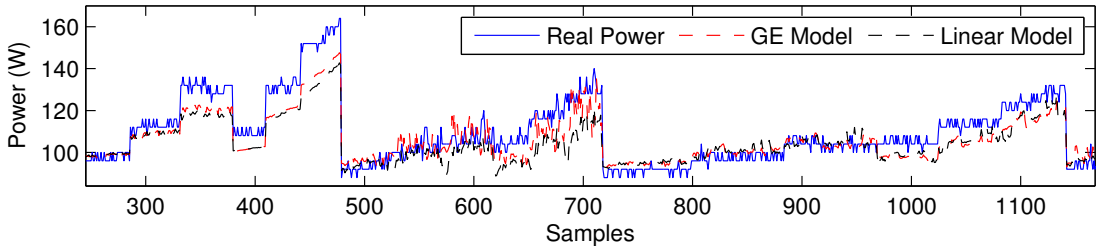
Figure 4.11: RMSE of the overall server power model for each frequency configuration (Test set)

4.4.2.2 Overall Power of Co-assigned Applications

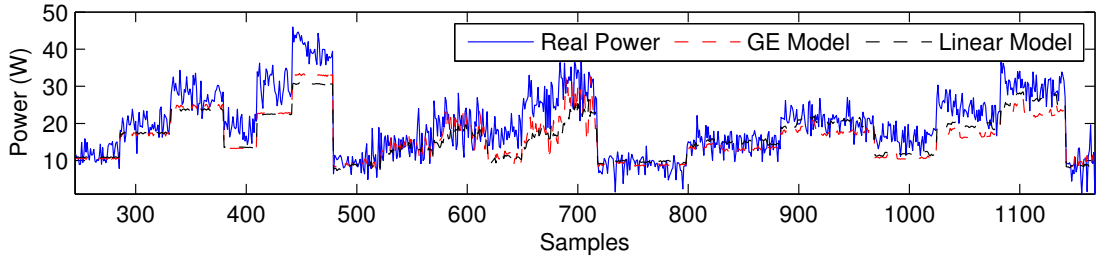
The third row of the Table 4.6 shows the RMSE and MAE in overall server power when two tasks are running at the same time in the server. As expected, the GE model outperforms the classical approach, with a MAE of 7.48W and a RMSE of 9.52W for the test set. Fig. 4.12 shows the prediction of the overall, dynamic CPU and memory power for the co-allocation scenario. We can see that both the GE and the classical model are able to predict fairly accurately the overall server power using HW counters prediction.

The last two rows of the Table 4.6 show the RMSE and MAE for an asymmetric task co-allocation. In the *2vs1* scenario, we run 2 copies/threads of one benchmark of the test set together with 1 copy/thread of another benchmark of the test set. For example, 1 copy of *perlbench* together with 2 copies of *lbm*, and also 2 copies of *perlbench* with 1 copy of *lbm*, and so on. In the *4vs2* scenario the process is identical. There are no training sets for the *2vs1* and *4vs2* scenarios, as the model is robust enough to predict the HW counters for an asymmetrical scenario and no re-training is needed. The RMSE value for the *2vs1* scenario is fairly low when compared to the *4vs2* scenario. The overall RMSE values are below 12W when the unsupervised GE model is used, which represents a 7.3% of the total server power.

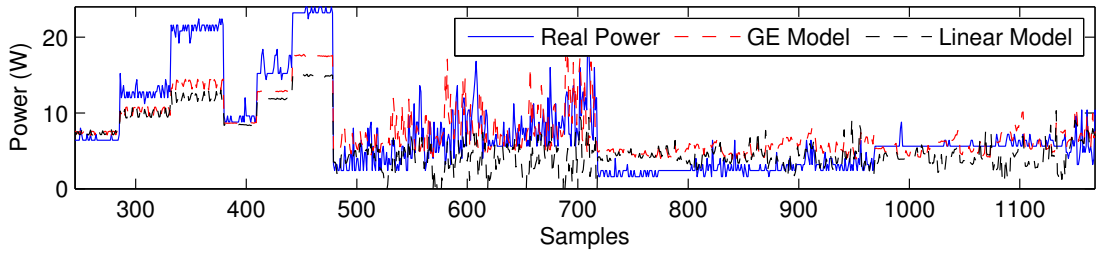
The results of the models show errors with enough accuracy to apply different state-of-the-art policies to reduce the IT power consumption, thus improving energy efficiency in data centers. Policies like power-capping or resource management could benefit from the models obtained in this work.



(a) Overall Server Power



(b) Dynamic CPU Power



(c) Memory Power

Figure 4.12: Test set samples (subset). Power prediction of co-allocated tasks using HW counter prediction

4.5 Conclusions

In this chapter, we presented a fast energy estimation framework for large scale long-running applications. The framework estimates the dynamic CPU and memory energy of long-running applications through the application signature without the need to execute the original application completely. We validated the framework with a representative set of long-running applications. For the sequential versions of the applications we obtained RMS values of 10.4% and 16.8% for the CPU and memory energy estimation errors, respectively.

In case of the multi-threaded scenario we use a subset of the original set of applications and obtained RMS values of 11.4% and 12.8% for the CPU and memory energy estimation errors, respectively. We have measured the speed of the

energy estimation process with the Compression Ratio parameter. We obtained Compression Ratios from 10.1 to 191.2, indicating that the energy estimation framework can estimate the energy 191.2 times faster than executing the whole original application.

Moreover, we present a model based on Grammatical Evolution (GE) techniques to predict dynamic CPU and memory power in enterprise servers as a function of HW counters. Using leakage and fan power models we develop an overall server power model that is robust enough to predict the power of co-allocated tasks without the need to train the model under that situation. In order to predict the power consumption of co-allocated tasks, we develop a methodology to predict the HW counters of two co-allocated tasks given the HW counters of the individual tasks. All the models only need to be trained one time per server.

Furthermore, we can predict the power consumption of a task that was not previously trained in the models. The proposed models are generated in an unsupervised way, leveraging the usage of the automatic feature selection and extraction of GE techniques. Models have been trained and tested using real traces from a multi-threaded enterprise server, running a wide range of sequential and parallel applications, under various DVFS setups. Results have been compared against a classical feature selection and least-squares modeling approach. The overall RMSE values of the models are below 12W when the unsupervised GE model is used, which represents a 7.3% of the total server power.

The results of this chapter were presented in an international journal:

- J. C. Salinas-Hilburg, M. Zapater, J. M. Moya, J. L. Ayala, “Fast energy estimation framework for long-running applications”, Future generation computer systems (2021).

Additionally, results of this chapter were also presented in two international conferences:

- J. C. Salinas-Hilburg, M. Zapater, J. L. Risco Martín, J. M. Moya, J. L. Ayala, “Unsupervised Power Modeling of Co-Allocated Workloads for Energy Efficiency in Data Centers”, Design, Automation and Test in Europe (DATE) (2016).

- J. C. Salinas-Hilburg, M. Zapater, J. L. Risco Martín, J. M. Moya, J. L. Ayala, "Using grammatical evolution techniques to model the dynamic power consumption of enterprise servers", International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS) (2015).

In the next chapter we validate the use of the application signature for energy-aware task scheduling approaches. We show that is possible with negligible error to use the information provided by the application signature to apply energy-aware task scheduling approaches to minimize the energy consumption in data centers.

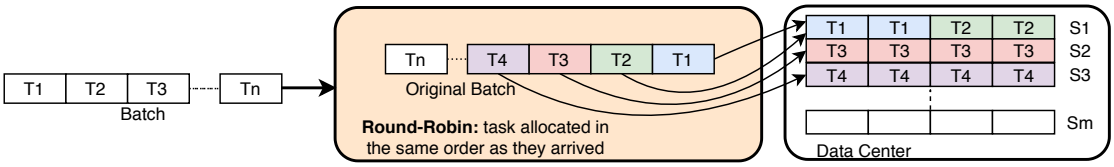
Chapter 5

Task scheduling with the application signature

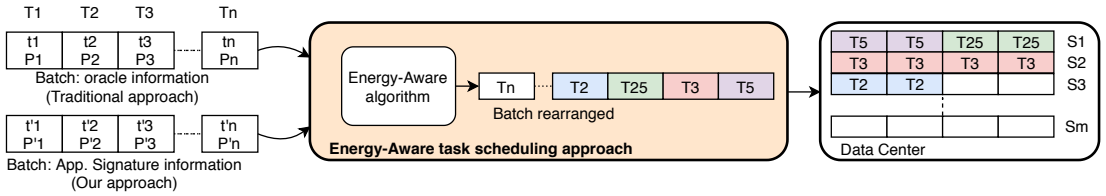
In this chapter we use the information of the application signature, obtained through the fast energy estimation framework (Chapter 4), to apply different energy-efficient task scheduling approaches in order to reduce the **makespan** of the original batch, and therefore improve the energy efficiency in data centers. As we have previously commented in the Introduction chapter (Chapter 1) the **makespan** is defined as the total execution time of the batch of applications that will run in the data center. Traditionally, the makespan is reduced by applying an efficient scheduling process with the information (such as power or runtime) obtained through a full profiling of the application. This is a time consuming process and hence, not energy-efficient. Therefore, the application signature is used to obtain the information needed by proactive energy-efficient scheduling approaches without performing a full profile (or execution) of the applications. It should be noted that the main goal of this chapter is to show that the information of the application signature can be used to apply proactive energy-aware scheduling approaches. Thus, it is not the main goal of this chapter to propose new or better energy-aware scheduling approaches.

5.1 Task Scheduling with the Application Signature

Task scheduling techniques to improve energy efficiency are widely used in today's data centers. There are energy-efficient proactive task scheduling approaches that require some information of the tasks that will be executed in the data center. This information can be the execution time or even the power that the tasks



(a) Round-Robin task scheduling policy



(b) Energy-Aware task scheduling approach

Figure 5.1: Comparison between task scheduling approaches: Round-Robin vs Energy-Aware

will consume during the execution. Traditionally, the previous information can be obtained through a full profiling of the tasks. Nevertheless, this process is not feasible in long-running tasks scenarios where the process to gather the tasks information is not efficient. A Round-Robin policy is applied when is not possible to obtain any information of the tasks. Figure 5.1a shows the task allocation of a batch of T_n tasks using the Round-Robin policy, where each task is allocated to an available server in the same order as they arrived. The example shows a data center with m servers and each server has 4 cores. The first T_1 task of the batch requires 2 cores and is allocated to server S_1, similarly task T_2 requires 2 cores and since server S_1 has enough resources available the task T_2 is allocated there. When a task can not be allocated in a server it goes to a queue and waits until there is enough resources available to be executed. The Round-Robin policy is easy to implement, although is not efficient in terms of energy since the tasks are not allocated aiming to reduce the makespan or the power consumption of the servers. In this work we use the Round-Robin policy as a baseline to evaluate the energy savings of energy-aware task scheduling approaches.

As we have previously mentioned, efficient proactive task scheduling approaches can be used to reduce the energy consumption in data centers, for example by reducing the makespan of the original batch. Figure 5.1b shows that an energy-aware task scheduling approach can be used to rearranged the original batch and therefore minimize the makespan. In the example we can see that the tasks are allocated in a different manner when compared to the Round-Robin policy. This change in the order of execution of the original batch is the result of applying an energy-aware algorithm to minimize the makespan of the original

batch. The energy-aware algorithm needs information (execution time t_i of the mean power consumption P_i) from the tasks that will be executed. Traditionally, this information is obtained through a full dynamic profiling of each task. In the present chapter we call this knowledge as the **oracle** information of the tasks. We propose the use of an application signature to leverage or approximate the oracle information. By executing the application signature we can obtain the information of execution time t'_i and mean power consumption P'_i without the need to perform a full profiling of each task of the batch. The oracle information is our gold-standard and it allows to evaluate the accuracy of the energy savings when using the information of the application signature. Finally, it is important to notice that each task scheduling approach performs a static scheduling of the whole batch. Additionally, each task of the batch is allocated and executed in a non-preemptive manner.

5.1.1 Using the Application Signature for Energy-Aware Task Scheduling

Algorithm 2 shows the process of using the application signature information for energy-aware task scheduling in data centers. The process starts when a batch formed by n tasks arrives at the data center (2). Each task of the batch is sent to an available server of the data center using a Round-Robin policy to extract and execute the application signature and therefore, estimate the execution time and the mean power (4). The information (execution time and mean power) obtained from the application signature of each task is saved in a list (5). Furthermore, an energy-aware task scheduling approach can use the applications signature information saved in the list to rearrange the batch (changing the order of execution of each task) aiming to reduce the makespan (7). Finally, the rearranged batch is executed resulting in a lower makespan than the execution of the original batch arrange (8). In section 5.2 we will show how an energy-aware approach can use the tasks information to reduce the energy consumption in data centers.

5.1.2 Compression Ratio of the Batch

The Compression Ratio of the batch CR_B measures the acceleration of the execution time and mean power information extraction of the whole batch using the application signature. The Compression Ratio is calculated as the ratio of total execution time or makespan of the original batch using the Round-Robin approach (T_{RR}) to the total execution time of extracting the execution time and

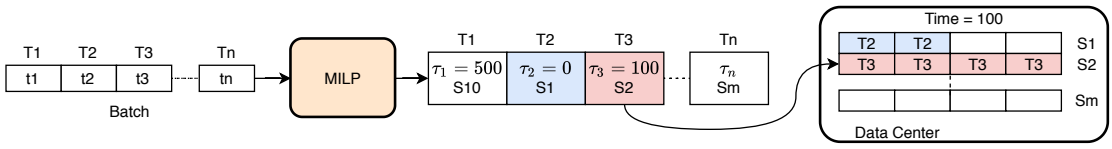
Algorithm 2 Application Signature for Energy-Aware Task Scheduling

Require: batch of n tasks, set of batches, m servers
1: **while** set of batches not empty **do**
2: batch= New batch arrives
3: **while** batch not empty **do**
4: [Exec. Time_n, Mean Power_n]=app_signature(task_n,server_m)
5: App. Signature Info List = append values of Exec. Time_n and Mean Power_n
6: **end while**
7: rearrange_batch=energy_aware_task_scheduling_rearrange(App. Signature Info)
8: allocation→ energy_aware_task_scheduling_execute(rearrange_batch,m servers)
9: **end while**

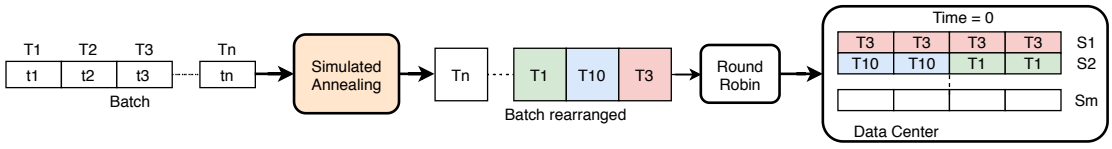
mean power with the Application Signature (T_{AS}) of the whole batch, as shown in Eq. 5.1.

$$CR_B = \frac{T_{RR}}{T_{AS}} \tag{5.1}$$

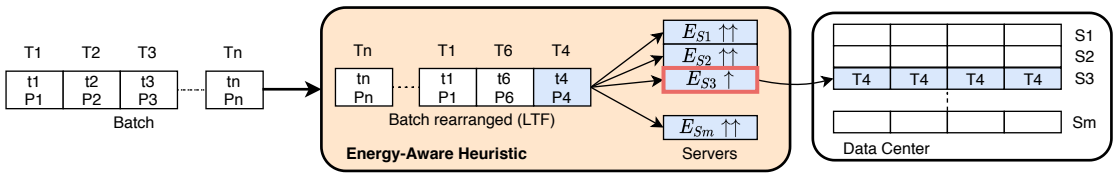
5.2 Task Scheduling Approaches



(a) MILP



(b) Simulated Annealing



(c) Energy-Aware Heuristic

Figure 5.2: Energy-Aware task scheduling approaches: input, output and task allocation

In this section we explain the three task scheduling approaches to validate the use of the application signature. The task scheduling approaches presented

rearrange the execution order of the tasks from a batch and allocate the workload to the servers in order to improve the energy efficiency in data centers. Three scheduling approaches were selected and implemented based on: Mixed Integer Linear Programming, a metaheuristic approach with Simulated Annealing and an energy-aware heuristic. These scheduling approaches were selected because they can be efficiently used with the real (oracle) values of either execution time or mean power, and this would not be possible without a full dynamic profiling of the applications.

The goal of the present work is not to outperform existing energy-aware task scheduling approaches, but to show that with the information provided by the Application Signature we open the possibility to perform energy optimization in data centers with scheduling techniques more powerful than applying reactive heuristics. The three task scheduling approaches cover in great detail different options of searching the optimal value. The optimum of energy savings is found with the Mixed Integer Linear Programming approach. This value serves as reference to evaluate the accuracy of the other approaches (heuristic and metaheuristics). Finally, the following approaches assume that the number of copies or threads that each task will execute is a known information.

5.2.1 Mixed Integer Linear Programming Formulation

A mixed integer linear programming (MILP) formulation is presented for the task scheduling problem of a batch execution in a data center. In the present work the MILP formulation for the task scheduling of sequential and parallel tasks is based on the work from Goldman et al. [47]. We use this formulation since is a generic MILP task scheduling formulation and it solves our task scheduling problem very efficiently. The original MILP formulation does not consider that all the resources of the data center are distributed along different servers. Therefore, we propose a modification of the original MILP formulation in order to send each task to a different server. The batch B_{MILP} is formed by a number of n independent parallel tasks T_i $(1 \leq i \leq n)$. Each task requires c_{T_i} cores and has an execution time equal to t_i. Each parallel task can only be executed in one server S_i, and there is m number of servers. Each server has c_{server} cores.

Figure 5.2a shows the task scheduling process using the MILP formulation. The **input** for the MILP problem is the execution time t_i of each task T_i of the batch B_{MILP}. The **output** for the MILP task scheduling approach is the batch of tasks with their respective starting time (τ_i) and the assigned server (S_i) where the task is going to be executed. For example, at the time instant of 100 the task T_3 should begin its execution. Therefore, the 4 copies or threads (c_{T_3}) of task T_3

are allocated in the server S_2. The 2 copies or threads (c_{T_2}) of task T_2 were already running in the server S_1 since its starting time is equal to 0. The task T_1 needs to wait until the time instant 500.

The objective of the MILP optimization process is to minimize the makespan (C) of the whole batch B_{MILP} of tasks. The complete MILP formulation is as follows:

Minimize C
Subject to:

1. $\tau_j \geq 0 \ \forall T_j$
2. $x_{kj},\ y_{kj} \in \{0,1\} \ \forall T_j, T_k$
3. $z_{ji} \in \{0,1\} \ \forall T_j, S_i$
4. $\sum_{i=1}^{s} z_{ji} = 1 \ \forall T_j, S_i$
5. $\tau_j \leq \tau_k + t_k + (3 - x_{kj} - z_{ji} - z_{ki})X \ \forall T_j, T_k, S_i$
6. $\tau_k + t_k \leq \tau_j + t_j + (3 - x_{kj} - z_{ji} - z_{ki})X \ \forall T_j, T_k, S_i$
7. $\tau_k + t_k + d \leq \tau_j + (2 + y_{kj} + x_{kj} - z_{ji} - z_{ki})X \ \forall T_j, T_k, S_i$
8. $\tau_j + t_j + d \leq \tau_k + t_k + (3 - y_{kj} + x_{kj} - z_{ji} - z_{ki})X \ \forall T_j, T_k, S_i$
9. $\sum_{\substack{j=1 \\ j \neq k}}^{n} c_{T_j} \times x_{kj} \leq c_{server} - c_{T_k} \ \forall T_k$
10. $C \geq \tau_k + t_k \ \forall T_k$

Constraint (1) indicates that the starting time (τ_i) of each task must be a positive number. Constraints (2)-(3) show the decision variables of the MILP formulation. The decision variables x_{kj} and y_{kj} indicate the condition when a task starts the execution before or after another task. The decision variable z_{ji} indicates the condition when a task is running on server S_i. Constraint (4) guarantees that each parallel task runs in one server. Constraints (5)-(8) allows to calculate, with no overlaps, the starting time (τ_i) of each task. The constraint (9) assures the validity of the whole task scheduling process. This is done by checking that all the tasks that are concurrently running in each server do not consume more than c_{server} cores. Finally, constraint (10) shows the objective function as given by $C =$

$max\{\tau_j + t_j\}$. The value of X is a constant and must follow the rule: $X > \sum_{k=1}^{n} t_k$. The constant d is a delay that guarantees the no presence of strict inequalities in the MILP formulation. The value of d is selected such as: $d < min_{1 \leq i \leq n}\ t_i/2$.

As we previously commented the output of the MILP task scheduling approach is the starting time τ_i and assigned server S_i of each task T_i of the original batch B_{MILP}. The value of the starting time τ_i of each task is directly obtained at the end of the MILP calculation. For the assigned server S_i we use the final state of the z_{ji} decision variable, where each row represent a task T_i and each column represent a server S_i. A value of 1 in a column indicates that the task is assigned to that server.

5.2.2 Simulated Annealing

A metaheuristic approach is proposed to minimize the makespan (C) of the task scheduling process based on the Simulated Annealing technique. As opposed to the MILP technique, the Simulated Annealing process does not guarantee to find the global optimum whereas is used to find an approximate global optimum in large search spaces. Furthermore, the Simulated Annealing technique has the advantage, like other metaheuristics, to be more scalable than the MILP technique. The Simulated Annealing technique works by modeling the physical process whereby a solid material is slowly cooled until it reaches a frozen state, which happens at a minimum system energy [69].

We propose the use of a Simulated Annealing algorithm to rearrange the tasks T_i from a batch of n tasks in order to minimize the makespan (C) of the task scheduling process. Figure 5.2b shows the Simulated Annealing task scheduling approach. The **input** for the Simulated Annealing algorithm is the execution time t_i of each task of the batch B_{SA}. The **output** is the batch rearranged, where the tasks from the batch are executed in a different order from the original batch using a Round-Robin approach. In the example, we can see that after the Simulated Annealing process is finished the positions for tasks T_3, T_{10} and T_1 are different from the original batch. Each task of the rearranged batch is allocated to the server using a Round-Robin process. Since task T_3 is the first in the batch its 4 copies or threads (c_{T_3}) are allocated in first available server which is server S_1. The 2 copies or threads ($c_{T_{10}}$) of task T_{10} are allocated in the server S_2, followed by the 2 copies or threads (c_{T_1}) of task T_1. Tasks T_{10} and T_1 are allocated in the server S_2 since there is enough available cores for both of them. The rest of the tasks of the rearranged batch are allocated in the rest of available servers.

The Simulated Annealing algorithm is shown in Algorithm 3. The input of the algorithm is the original batch B, which has the information of the execution times

t_i of each task T_i. The algorithm starts by setting an initial temperature $Temp_0$. The temperature variables $Temp$ and $Temp_0$ are defined in the context of the Simulated Annealing process. They should not be confused with the temperature values of the servers in the data center. The algorithm has two while loops: i) the inner while loop ((5)-(15)) rearranges the original batch B_{SA} to improve the makespan C. This is done with the same temperature $Temp$ and a number of n iterations, and ii) the outer while loop ((2)-(17)) decrease the temperature $Temp$ value until a number of iterations equal to *max_iterations*. Inside the inner while loop a random trial point is generated using the annealing function (6). In this process the concept of point refers to changes in the tasks T_i positions from the batch B_{SA}, generating a new rearranged batch $B_{rearranged}$. Next, the algorithm determines if the new point is better or worse (in terms of minimizing the makespan C) than the current point with a probability of acceptance P, as shown from lines (7) to (13). To calculate the makespan C the algorithm uses the objective function *makespan_function* taking as input the rearranged batch $B_{rearranged}$. Finally, the temperature changes according to a function (16). Where, $Temp_0$ is the initial temperature and k is the iteration number until reanniling.

Algorithm 3 Simulated Annealing Algorithm

Require: batch: B_{SA}

1: set initial temperature $Temp = Temp_0$
2: **while** k $\leq$ max_iterations **do**
3: k=k+1
4: temperature_iteration=0
5: **while** temperature_iteration$\leq$n **do**
6: $B_{rearranged}$=annealing_function(B_{SA},Temp)
7: C=makespan_function($B_{rearranged}$)
8: $\Delta = C\text{-}C_{old}$
9: **if** $\Delta < 0$ **then**
10: B_{SA}=$B_{rearranged}$
11: **else**
12: B_{SA}=$B_{rearranged}$ with probability $P = \frac{1}{1+exp(\frac{\Delta}{Temp})}$
13: **end if**
14: temperature_iteration=temperature_iteration+1
15: **end while**
16: $Temp = Temp_0 \times 0.95^k$
17: **end while**

The two important functions of the Simulated Annealing algorithm are the annealing function and the objective function. The annealing function (*annealing_function*) process is shown in Algorithm 4. The function takes as

input the batch B_{SA} and the temperature $Temp$. The goal of the function is to change the tasks T_i positions within the batch B_{SA}. The number of tasks position changes are proportional to the actual temperature $Temp$ state and are done in a random manner ($random\_integer$).

Algorithm 4 Annealing Function (*annealing_function*)

Require: batch: B_{SA}, Temp
1: **for** i=1:Temp **do**
2: r1=random_integer(1,length(B_{SA}))
3: r2=random_integer(1,length(B_{SA}))
4: B_{tmp}=B_{SA}
5: $B_{SA}(r1)$=$B_{tmp}(r2)$
6: $B_{SA}(r2)$=$B_{tmp}(r1)$
7: **end for**

The objective function (*makespan_function*) returns the makespan C of the task scheduling process, as shown in Algorithm 5. The input for the objective function is the batch B_{SA},a matrix with n rows (number of tasks in the batch B_{SA}) and three columns. The first column ($B_{SA}(i,1)$) is the task T_i starting time (or the task position), the second column ($B_{SA}(i,2)$) is the execution time t_i of the task T_i and the third column ($B_{SA}(i,3)$) is the number of cores (c_{T_i}) requested by the task T_i. Lines (7) through (37) are the core of the algorithm, where each task is assigned to a server until the batch is empty. The whole process is stopped when the batch is empty and all the tasks have completed their execution ((34)-(36)). The condition in line (8) through (30) guarantees that a task is scheduled if there are any task in the batch and the task starting time is less or equal than the makespan C. Then, for each task in the batch a task is assigned to a server ((9)-(29)). Lines (10) through (12) indicates that the task allocation process is temporarily stopped until the makespan C is equal or greater to the starting time of the next task on the batch. Lines (13) through (22) shows that a task is assigned to the first available server (17). The batch is updated by removing the task that has already been assigned to a server, as shown in lines (23) through (27). Line (28) shows the update of the starting time for the tasks that are in queue waiting for execution. Line (31) shows that the makespan C is updated every iteration and this is the final output of the algorithm.

5.2.3 Energy-Aware Heuristic

As we have previously commented the objective of this chapter is to reduce the makespan of the batch execution with efficient task scheduling approaches by using

Algorithm 5 Objective Function (*makespan_function*)

Require: Batch: B_{SA}

```
C=0
num_servers=m
num_cores=c_server
servers=ones(num_servers,num_cores) /*A matrix of ones. Saves the current temporal state of each core of the server*/
used_cores=zeros(num_servers) /*A vector of zeros. Saves the available cores of each server*/
flag=0 /*Flag to label if a task is allocated to a server*/
while true do
    if length(B_SA)>0 and B_SA(1,1) ≤ C then
        for k=1:length(B_SA) do
            if B_SA(k,1) >C then
                break
            end if
            for i=1:length(servers) do
                if used_cores(i)+B_SA(k,3) ≤ num_cores then
                    available_cores=find_empty_cores(servers(i,:)≤0);
                    for j=1:B_SA(k,3) do
                        servers(i,available_cores(j))=B_SA(k,2);
                    end for
                    flag=1
                    break
                end if
            end for
            if flag=1 then
                flag=0
                B_SA(k,:)=[]
                break
            end if
            B_SA(k,1)=C+k
        end for
    end if
    C=C+1
    servers=servers-1 /*Reduce in one the current temporal state of each core of each server*/
    used_cores=sum(servers>0) /*Update the available cores of each server*/
    if length(B_SA)<1 and servers<0==ones(num_servers,num_cores) then
        break
    end if
end while
```

Algorithm 6 Energy-Aware Heuristic

Require: Batch: B_{EA}
1: B_{LTF} = LTF($B_{EA}(:,1)$)
2: **for** i=1:length(B_{LTF}) **do**
3: minEnergy=MAX
4: allocatedServer=NULL
5: **for** j=1:length(servers) **do**
6: **if** servers(j) has enough resources for $B_{LTF}(i,3)$ **then**
7: energy=estimateEnergy(servers(j),$B_{LTF}(i,1)$,$B_{LTF}(i,2)$)
8: **if** energy < minEnergy **then**
9: minEnergy=energy
10: allocatedServer=server(j)
11: **end if**
12: **end if**
13: **if** allocatedServer ≠ NULL **then**
14: allocate $B_{LTF}(i)$ to allocatedServer
15: **end if**
16: **end for**
17: **end for**

the application signature. That explains why the execution time information is used in the MILP and Simulated Annealing approach. In this section, we leverage the use of an energy-aware heuristic to improve the energy efficiency in Data Centers. In this case, we want to introduce a heuristic that is conceived to be used with the execution time and mean power information to fully take advantage of the information provided by the application signature. Considering both time and mean power information is better exploited in heterogeneous scenarios, where we have different types of servers with different types of power consumption behaviour.

The energy-aware heuristic is based from the work of A. Beloglazov et al. [15]. In that work the authors propose an algorithm to allocate in an energy efficient manner virtual machines in a cloud oriented scenario. We adapted their algorithm to our non-cloud oriented scenario and added an heuristic to sort the original batch of tasks. Figure 5.2c shows the overall energy-aware heuristic approach for task scheduling in data centers. The **input** is the batch B_{EA} of tasks T_i. From the batch B_{EA} we have the information of execution times t_i and the mean CPU and memory power (both represented as P_n) of each task of the batch. As opposed to the MILP and Simulated Annealing approach the energy-aware heuristic is both proactive and reactive. The first step of the process is the proactive part of the approach, where the heuristic called Longest Task First (LTF) is used to sort the tasks of the original batch in a descending way according to the execution time t_i of each task to improve the overall makespan C. This means that the tasks

with higher execution times t_i are executed first. In our example the task T_4 has a higher execution time than the rest of the tasks, therefore it will be executed first. The second step of the process is the reactive part, where the selected task will be send to the server where the energy consumption is increased the least. Following our example from Fig. 5.2c the values of mean power P_4 and execution time t_4 of task T_4 are used to estimate the increase of energy in each server of the data center. The server that shows the minimum energy increase is server S_3, thus the 4 copies or threads of task T_4 are allocated in S_3 of the data center.

The energy-aware heuristic algorithm is shown in Algorithm 6. The input of the algorithm is the batch B_{EA}, a matrix with a row number equal to the number of tasks and 3 columns. The first column $B_{EA}(i,1)$ is the execution time t_i, the second column $B_{EA}(i,2)$ is the values of mean power from both CPU and memory and the third column $B_{EA}(i,3)$ is the number of cores (c_{T_i}) requested by the correspondent task. The first step of the algorithm is to sort the original batch in a descending way according to the execution time of each task using the LTF heuristic (1). Then, from line (2) to line (17) each task of the sorted task list is allocated to a server where the energy is increased the least. If a server has enough resources for the task selected from the rearranged batch B_{LTF} (6) then the energy of that task running on that server is estimated (7).

To estimate the energy we take into account the current power state of the server and update the mean dynamic CPU and memory power with the mean power values from the selected task. As it is explained in the Experimental Setup section (Section 5.3) we use a data center simulator where the server model defined is the same as the server model used to derived the power models in Chapter 4, Section 4.2. Therefore, in practical terms, the mean dynamic CPU and memory power values from the selected task are added to the $P_{CPU,dyn}$ and $P_{Mem,dyn}$ of the Equation 4.15 (Section 4.2.1) to update the server power.

Once the current power state is updated the energy is estimated taking into account the execution time t_i of the selected task. The variables *minEnergy* and *allocatedServer* are updated every time a lower energy value is found ((9)-(10)). The process is repeated for every server that has enough resources ((5)-(16)) and the selected task is allocated to the server that shows the minimum increase in energy consumption (14).

5.3 Experimental Setup

5.3.1 Data Center Simulator

A data center simulator is used in order to evaluate the energy savings of each task scheduling approach when the information of the application signature is used. The SFIDE data center simulator [96] is a simulator based on discrete event system modeling (DEVS). The modular architecture of the simulator allows an easy implementation of the different task scheduling approaches. The SFIDE simulator was validated against real server and data center traces of the same type of workload we use in this work. The SFIDE data center simulator is composed by two main modules: Room and Cooling. The Room module is where the task allocation and processing takes place. The Cooling module regulates the temperature of the data center according to a cooling model. In this work we do not implement a cooling model and therefore to calculate the overall data center power we assume a fixed Power Usage Effectiveness (PUE) value. The Room module is formed by the following modules: Allocator, In Row Cooling Units (IRC), Rack and Server. The Allocator module takes a batch of tasks and assign each task to a server. In the present work we implement different Allocator modules for each task scheduling approaches. The IRC module groups a set of Racks and each Rack contains a set of Servers. The IRC module computes the overall status of all the servers from all the racks. For both the Simulated Annealing and the Energy-Aware heuristic approaches we implemented the algorithms as an independent modules in the SFIDE data center simulator. The MILP formulation is implemented using IBM ILOG CPLEX, version 12.8.

5.3.1.1 Server Power Model and Overall Data Center Power

The server model defined in the simulator is the same server, the Decathlete (Intel S2600GZ), used to build the power models explained in Chapter 4, Section 4.2. Instead of being formed by 1 SandyBridge-EP processors the server is defined in the simulator with 2 SandyBridge-EP processor, both with 6 cores and therefore each server has a total of 12 cores (c_{server}). The simulations are done with a fixed fan speed for all servers equal to 6000 RPM and a fixed ambient temperature value equal to 22°C.

The SFIDE simulator is able to calculate the overall data center power, including IT power and cooling power. We established a fixed PUE equal to 1.5 to obtain the overall data center power P_{DC}, as shown in Equation 5.2. The value of P_{IT} is the power from all the servers of the data center. The PUE values for data centers around the world are in the range of 1.1 to more than 1.5 with

an average PUE value of 1.58 in 2018 [71]. Therefore, a PUE value of 1.5 is a reasonable value for a current energy-efficient data center. The simulator outputs the data center power P_{DC} each time there is an event (a task is send to a server, a task ends its execution, etc.), allowing to obtain a power profile of the overall data center power consumption. Therefore, the energy consumption of the data center can be calculated using the data center power profile and the makespan of the executed batch of tasks.

$$P_{DC} = PUE \times P_{IT} \tag{5.2}$$

5.3.2 Simulation Scenarios and Task Batch Composition

We validate our results using two data center scenarios: i) a small scale scenario formed by 1 rack and 5 servers, which creates a scenario with a total of 60 available cores; ii) a large scale scenario formed by 5 racks and 10 servers per rack, generating a total of 600 available cores. To validate the efficiency of the application signatures for energy savings we use an heterogeneous set of long-running tasks composed of: the applications *BT* and *SP* from the NAS Parallel suite [11], *Stream* [88], *Dgemm* [83] and *Linpack* [38] benchmarks. The applications and their respective application signatures used in the simulations are the same applications/signatures used to evaluate the energy estimation framework (Chapter 4, Section 4.4.1.3), using the following criteria to stop the execution of the application signature: 1) 5% of the estimated executed CPU instructions, and 2) a segment division equal to 10 to detect the stability of the hardware counter profiles. The batch for both scenarios are generated randomly with sequential (1 Thread) and parallel tasks (2, 4 and 6 threads) from the workload set previously commented. The batch size for both the small and large scale scenario, are equal to 66 and 900 tasks respectively.

5.4 Results

In this section we present the results of applying the three different task scheduling approaches using the information from the application signature. We compare the energy savings against a Round-Robin task scheduling approach (baseline) of the original task list. Additionally, we evaluate the error of the energy savings values from the application signature with energy savings values obtained from the oracle values of task execution times and mean CPU and memory power. The energy savings are calculated as follows:

$$Energy\ Saving\ (\%) = \frac{Energy_{RoundRobin} - Energy_{TaskSchedApproach}}{Energy_{RoundRobin}} \times 100 \quad (5.3)$$

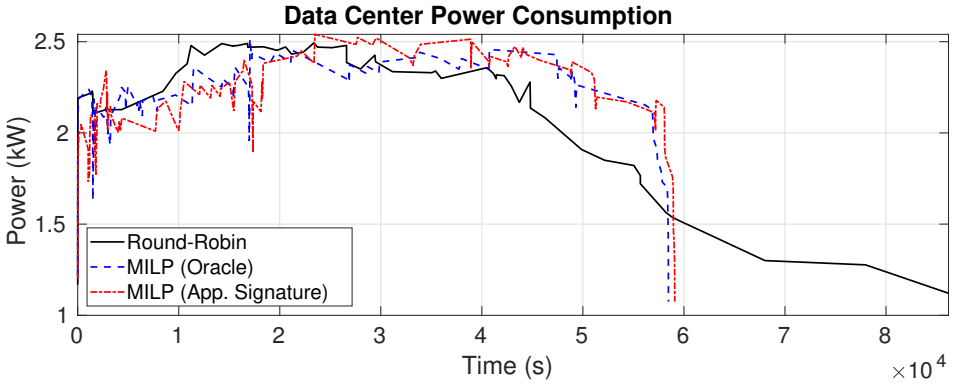
Table 5.1: Energy savings results for the small and large scale scenario when compared to the baseline Round-Robin policy

	Small Scale Scenario			
	Energy Saving (%)		Makespan (s)	
	Oracle	Application Signature	Oracle	Application Signature
MILP	19.4	18.3	58467	59090
Metaheuristic	17.7	16.3	59591	62188
Heuristic	17.0	15.5	61548	64117
	Large Scale Scenario			
	Energy Saving (%)		Makespan (s)	
	Oracle	Application Signature	Oracle	Application Signature
MILP	-	-	-	-
Metaheuristic	13.3	12.5	74119	75815
Heuristic	8.6	8.2	82899	82979

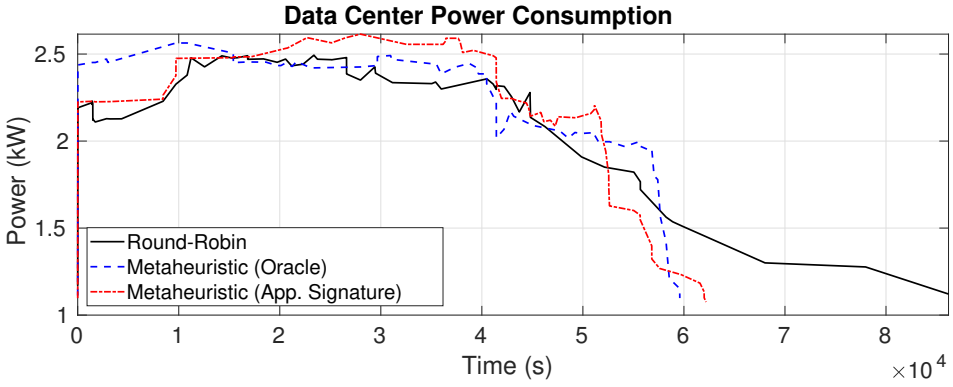
5.4.1 Small Scale Scenario

The small scale scenario consists on 5 servers (60 cores) and a batch of 66 tasks. Table 5.1 shows the energy savings and makespan of every task scheduling approach for the small scale scenario when compared to the baseline Round-Robin policy. The results presented in Table 5.1 are calculated taking into account that the application signature of all the tasks in the batch is already extracted. This means that both energy savings and makespan values do not include the energy and makespan of the application signature calculation process. This is a fair approach since we are comparing the results against an oracle that extract the information (execution time and mean power) from a full profiling of the original applications.

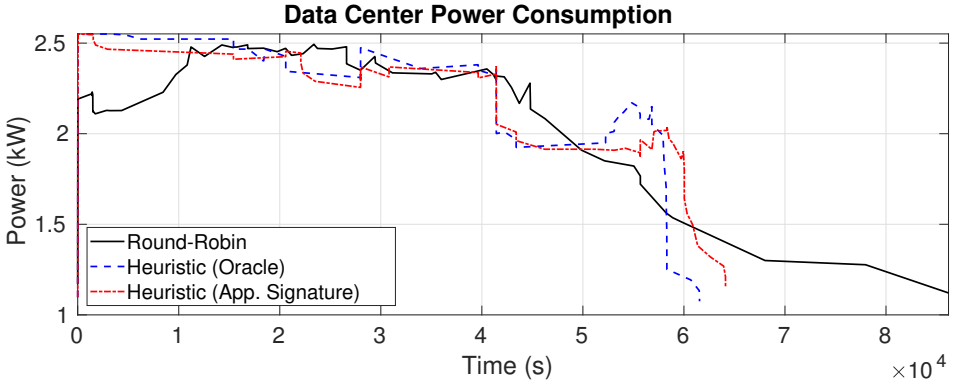
The MILP technique offers the highest energy savings when compared with the baseline task scheduling approach (Round-Robin), both for the application signature and the oracle. The metaheuristic and heuristic energy savings are below the MILP technique, although presenting energy savings higher than 15% when compared with the Round-Robin approach. The result is expected since the MILP technique searches the global optimum while the metaheuristic and the heuristic find an approximate global optimum. Furthermore, we can see the



(a) Power profile - MILP



(b) Power profile - Metaheuristic



(c) Power profile - Heuristic

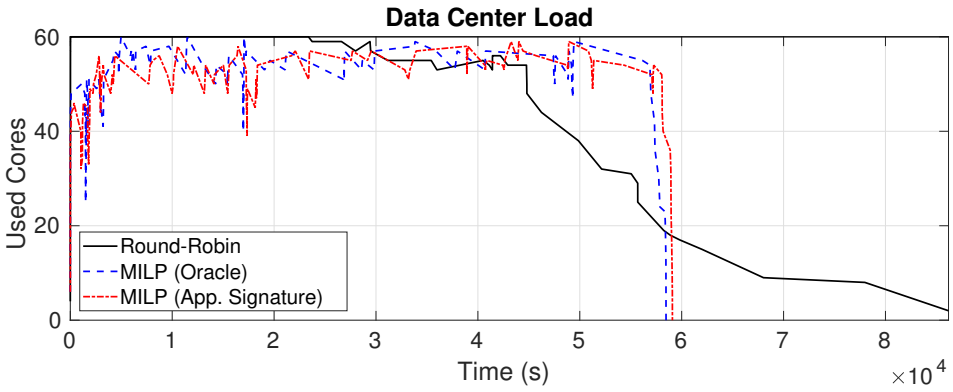
Figure 5.3: Small scale scenario: Power profiles

energy savings values obtained by using the information from the application signature are close to the energy savings values by using the information of the original application (oracle). The difference of the energy savings between the values from the application signature and the oracle is below 1.5%. Additionally, Table 5.1 shows the makespan values for all the task scheduling approaches. As expected, when using the application signature information, the lowest makespan value is from the MILP approach with a value equal to 59090 seconds.

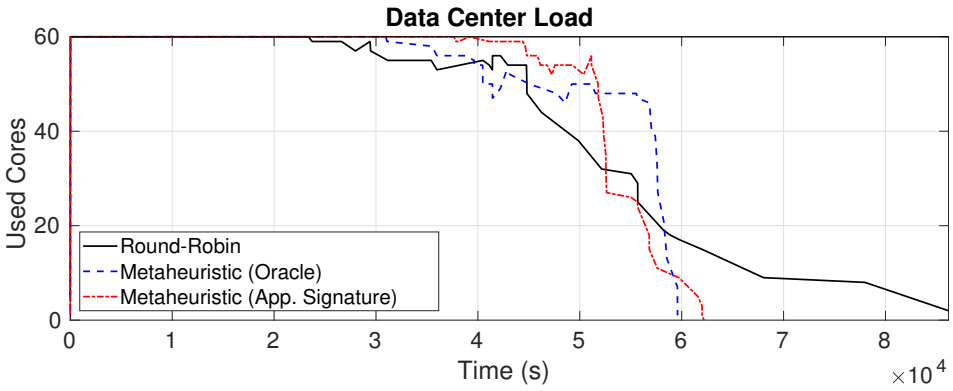
Figures 5.3a, 5.3b and 5.3c show the power profile of each task scheduling approach when compared with the Round-Robin approach. As we can see, the overall energy savings between all the task scheduling approaches comes from the makespan optimization since every batch ends before the Round-Robin policy. The power profiles from each task scheduling approach are different since the tasks are scheduled in different order. For example, the MILP approach (Fig. 5.3a) shows, at the beginning of power profile, an average power lower than the metaheuristic and heuristic approach. The heuristic approach (Fig. 5.3c) shows a power peak at the beginning of the power profile indicating that this approach is scheduling high power consuming tasks at the beginning of the batch execution.

Figures 5.4a, 5.4b and 5.4c show the load profile, or the number of used cores, of each task scheduling approach. In case of the MILP approach, the number of used cores are between 50 and 60 cores during the whole batch execution and it almost never gets to a value equal to 60 cores. The metaheuristic approach shows the opposed behaviour, as shown in Fig. 5.4b. The number of used cores is equal to 60 cores during almost all the batch execution, therefore all the data center resources are being used during the execution. In the case of the heuristic approach (Fig. 5.4c), the number of used cores are close to 60 at the beginning of the batch execution, it slightly decreases until the end of the execution where the number of used cores peaks again.

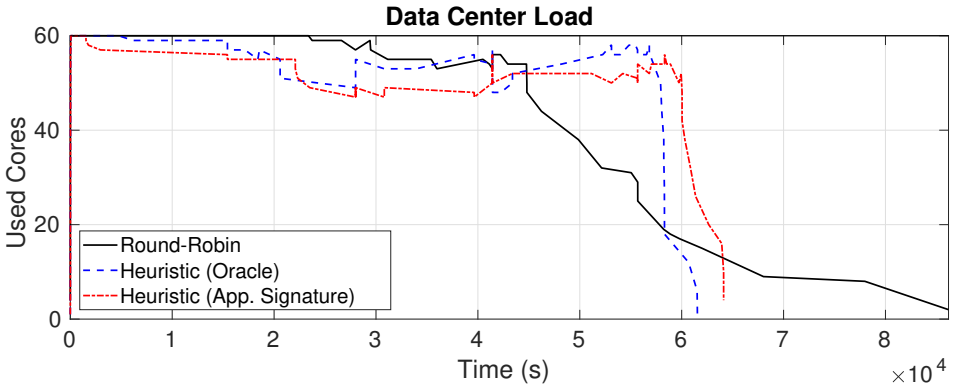
The differences between the load profiles reside on the task scheduling process of each approach. In the output of the MILP approach each task of the batch has an initial execution time together with a small delay (parameter d of the MILP formulation) respect of the previous executed task. This explains the noisy form of the load profile seen in the Fig. 5.4a. As opposed to the MILP approach, the output of the metaheuristic only rearranges the original batch and then applies a Round-Robin approach to send the tasks to available servers, the tasks from the batch do not have the small delay shown in the MILP approach. This leads to a more stable load profile (Fig. 5.4b), where during almost all the batch execution the data center is completely loaded because the Allocator (from SFIDE) is assigning each task to a server once a server is available. The heuristic approach has a similar stable load profile as the metaheuristic approach, as seen in Fig. 5.4c. In this case,



(a) Data Center Load - MILP



(b) Data Center Load - Metaheuristic

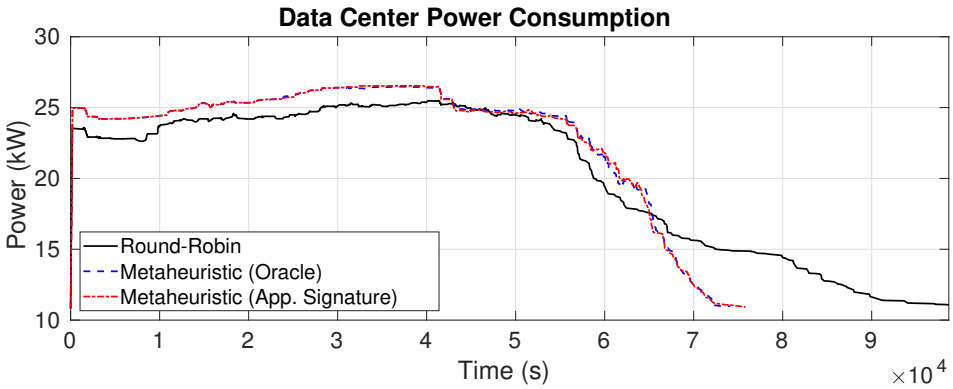


(c) Data Center Load - Heuristic

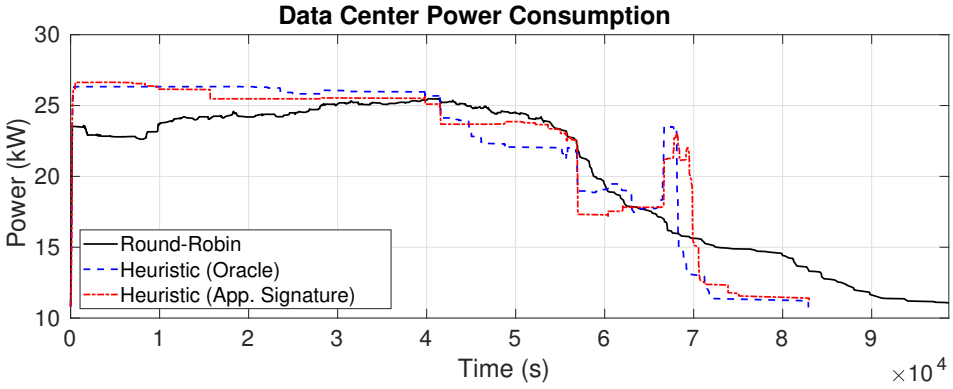
Figure 5.4: Small scale scenario: Load profiles

the data center is not completely loaded during the batch execution. The task scheduling process of the heuristic is allowing tasks with low number of threads (for example, 1 or 2 threads) get assigned at the same time to different servers not allowing tasks with high number of threads getting assigned to an available server. Thus, leaving a number of cores unused during the execution of the batch.

5.4.2 Large Scale Scenario



(a) Power profile - Metaheuristic



(b) Power profile - Heuristic

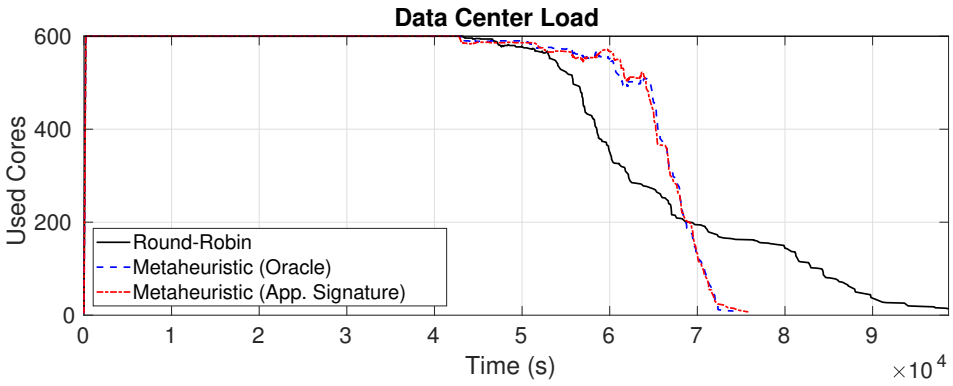
Figure 5.5: Large scale scenario: Power profiles

The large scale scenario is setup by 50 servers (600 cores) and a batch of 900 tasks. Table 5.1 shows the energy savings and makespan of every task scheduling approach for the large scale scenario. In this scenario the MILP approach is not used since is not scalable and the process to find the global optimum would take too much time. We obtain energy savings higher than 8% when compared with the Round-Robin task scheduling approach. The metaheuristic approach presents the highest value of energy saving and is equal to 12.5% when the information of

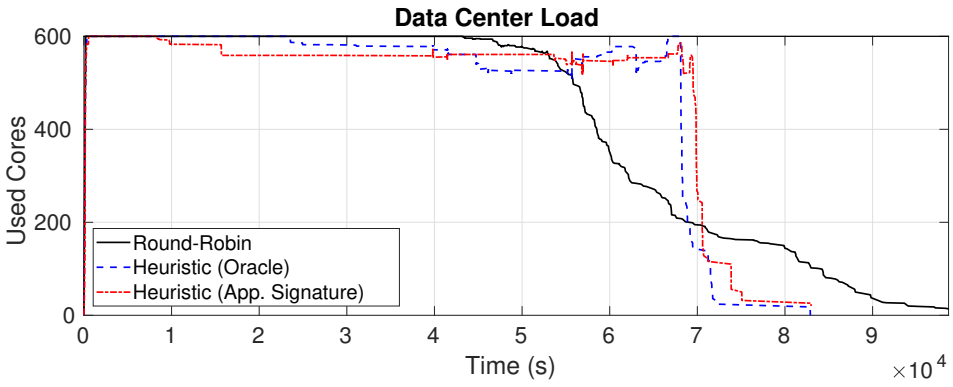
the application signature is used. The difference of the energy savings between the values from the application signature and the oracle is below 0.8%.

Moreover, the makespan is shown in Table 5.1 for the metaheuristic and heuristic approaches. As expected, the metaheuristic present better results than the heuristic since it can find a closer approximate value to the global optimum, which is to minimize the makespan C.

Figures 5.5a and 5.5b show the power profile of each task scheduling approach when compared with the Round-Robin approach for the large scale scenario. The power profile of the metaheuristic approach shows a stable signal during the whole execution of the batch, when using the information from both the oracle and the application signature. The main difference is that the batch execution ends earlier when the information from the oracle is used. The heuristic task scheduling approach power profiles (Fig. 5.5b) for both the oracle and the application signature are very similar. Both power profiles show a rise of the overall data center power around the time 7×10^4 s indicating that a set of power consuming tasks were waiting to be assigned to available servers.



(a) Data Center Load - Metaheuristic



(b) Data Center Load - Heuristic

Figure 5.6: Large scale scenario: Load profiles

The load profiles for both the metaheuristic approach and the heuristic approach are shown in Figures 5.6a and 5.6b. The loads profiles are similar to those obtained from the small scale scenario. In the metaheuristic approach the load profile obtained when using the information from the application signature is very similar to the load profile when using the information of the oracle. The load profile from the heuristic task scheduling approach when using the information of the application signature is slightly different from the oracle, since the load profile from the oracle shows a more loaded data center until around the time 4×10^4 s. This indicates the task assigning process when using the application signature is different from the oracle. Nonetheless, the results show that using the information of the application signature will result in energy savings similar to the energy savings obtained when using the oracle information.

5.4.3 Compression Ratio of the Batch

In this section we compare the execution time of the application signature extraction process against the execution time of the whole batch using the Round-Robin approach. For the small scale scenario the execution time of the application signature extraction of the whole batch (66 tasks) takes 1881 seconds, a 2.18% (86187 seconds) of the execution time of the whole batch using the Round-Robin approach. This leads to a Compression Ratio CR_B of the whole batch equal to 45.8, when we compare the execution time of application signature process to the execution time of the whole batch using the Round-Robin task scheduling approach. The large scale scenario has similar results with an execution time of the application signature process equal to 2483 seconds, a 2.51% (98667 seconds) of the execution time of the whole batch using the Round-Robin task scheduling approach. Resulting in a CR_B equal to 39.7. These results show that using the application signature process is viable, when compared to the Round-Robin approach, since the Compression Ratios have high values.

5.4.4 Overall Results

Finally, the overall results shows that the information provided by the application signature allows to apply the different energy-aware task scheduling approaches with similar results from the oracle information. Furthermore, we can use or develop many other energy-aware scheduling approaches that uses the information from the application signature and that were only possible to use with a priori full dynamic profiling of the applications.

5.5 Conclusions

In this chapter, we presented the use of an Application Signature for energy-aware task scheduling approaches. We use the information given by the Application Signature together with energy-aware task scheduling approaches to obtain energy savings in data centers in the scenario of large scale long-running applications. We validate our results by implementing three energy-aware scheduling approaches based on: Mixed Integer Linear Programming, Simulated Annealing and an energy-aware heuristic. Additionally, we use a heterogeneous set of long-running applications with different batch sizes. A simulator was used to evaluate the results in a small and large scale scenario. The energy savings from each scheduling approach were obtained by comparing the energy values of a batch execution against a baseline scheduling approach based on a Round-Robin policy. The energy savings values obtained with the Application Signature were compared against the energy savings obtained through the real (oracle) energy values of the applications. The difference between the energy savings obtained with the Application Signature and the oracle values is below 1.5%. Furthermore, we obtained energy savings around 8.2% to 19.4%. Finally, we obtained Compression Ratios around 39.7 to 45.8.

The results of this chapter were presented in an international journal and it is currently in review process:

- J. C. Salinas-Hilburg, M. Zapater, J. M. Moya, J. L. Ayala, "Energy-Aware Task Scheduling in Data Centers using an Application Signature", Computers and Electrical Engineering (2020) (Review process).

In the next chapter we present a summary of this work together with the contributions and results from this PhD. thesis. Additionally, we present the future lines of research derived from this work.

Chapter 6

Conclusions and Future Work

This PhD. thesis has presented a solution to perform the fast energy estimation of long-running applications through an application signature. In this chapter we present the summary and conclusions of this work highlighting the contributions and the obtained results. We conclude this chapter with a description of the future work derived from this research.

6.1 Summary and Conclusions

As previously commented in the introduction of this PhD thesis (Chapter 1), data centers are huge power consumers and despite the energy-efficient solutions adopted in the last decade there is still a need to improve energy efficiency in data centers during the following years. The computing or IT component of the data centers is the major contributor to the overall data center power and therefore, there are proactive energy aware approaches focused on reducing the energy consumption of the IT component in data centers. These proactive approaches are usually represented by energy-efficient task scheduling approaches, where the jobs or tasks are allocated efficiently with the aim to reduce the overall energy consumption of the data center.

These energy-efficient task scheduling approaches assume the existence of a full dynamic power profiling obtained through a complete execution of the applications. This is not feasible in scenarios of time consuming long-running applications. Therefore, in this work we present a solution to estimate the energy of the applications without the need to execute the application completely by using an application signature. The application signature is a short version, in terms of execution time, of the original application. With the application signature we are able to estimate the energy in a fast way and also, apply

proactive energy-aware task scheduling approaches in an efficient manner.

The summary of the contributions derived from this PhD thesis are the following:

- In Chapter 2 we presented the application signature as a proof of concept:
  - We explained the concept of application signature and how is used to make a fast energy estimation for sequential and multi-threaded applications.
  - In the case of sequential applications we obtained CPU and memory energy estimation errors below 8.0% when the estimated energy (through the application signature) is compared against the energy of the complete execution of the application. For the multi-threaded scenario we obtain a RMSE equal to 12.7% when we compare the estimated energy against the consumed energy from the whole multi-threaded execution of the application. We obtained an average value of almost 9.8 for the Compression Ratio of all the benchmarks used in the experiments. This indicates that using the application signature we are able to make an energy estimation almost 10 times faster than the complete execution of the application.
- The fast energy estimation framework is presented in Chapter 4:
  - We described the fast energy estimation framework for long-running applications that uses the application signature to estimate the energy without the need to execute the whole application.
  - The framework estimates the energy in an automatic way. The design of the framework is modular, allowing to change the internal functionality of each module, due to preferences or technical availability, without affecting the functionality of the whole framework. Moreover, the framework is able to work for any compiled language.
  - The accuracy of the fast energy estimation framework is validated with a set of sequential and multi-threaded long-running applications. For the sequential version we obtained an RMS of 10.4% for the CPU energy estimation error and an RMS of 16.8% for the memory energy estimation error. In case of the multi-threaded scenario we used a subset of applications from the sequential version set, achieving an RMS of 11.4% for the CPU energy estimation error and a RMS of

12.8% for the memory energy estimation error. We obtained Compression Ratios in the range from 10.1 to 191.2.

- In this chapter we presented a methodology to obtain the overall server power model. We model the dynamic CPU and memory power as a function of the hardware counters using Grammatical Evolution techniques. We obtained absolute power errors equal to 4.4W and 3.7W, for the dynamic CPU and memory power model respectively. Our models were trained and tested using a wide range of sequential and multi-threaded applications, under various DVFS setups, improving error by a 32% when compared to a traditional approach.
- We showed that our model is robust enough to predict the power consumption of two different tasks running co-assigned in the same server, given the hardware counters of the overall server. Additionally, we developed a methodology to, given the hardware counters of the individual tasks, obtain the hardware counters when both applications are co-allocated (without the need to execute the co-allocated applications).

- In Chapter 5 we used the application signature to apply energy-aware task scheduling approaches:

  - We validated the usefulness of the energy estimation information obtained through the execution of the application signature by applying different energy-aware task scheduling approaches. The results obtained through the energy estimation values (from the application signature) are compared against the real energy values (oracle).
  - We used three different energy-aware task scheduling approaches:

    i) An optimal approach using a Mixed Integer Linear Programming (MILP) technique.

    ii) An energy-efficient heuristic that uses a Longest Task First (LTF) algorithm together with an energy-aware task allocation based on the current servers consumption in the data center.

    iii) We proposed an implementation of an energy-aware metaheuristic using a Simulated Annealing technique.

    The overall data center energy consumption from each task scheduling approach is compared against a Round-Robin (RR) approach.

– We obtained energy savings from 8% to 19%, and more importantly the energy savings obtained with the **application signature** information are similar as the values obtained with the **oracle** information, obtaining a difference in the energy savings below 1.5%.

6.2 Future Work

As we have commented earlier the main goal of this PhD. thesis is to estimate the energy of long-running applications avoiding a complete execution. We presented and developed an application signature as a solution to this problem and we limited our experiments to a non-cloud environment (among other considerations). Therefore, taking into account the results obtained in this research work we present different lines of future research works.

6.2.1 Enhance and Broadening of the Scope of the Fast Energy Estimation Framework

As we can see in Chapter 4 the fast energy estimation framework has its limitations. We propose to use the LLVM compiler to address these limitations. The goal of this future work is to implement the fast energy estimation framework using the LLVM compiler. The modular nature of our fast energy estimation framework makes easy to implement each module in the LLVM compiler and this way we can leverage limitations as data-dependency or to make our framework code independent.

The energy estimation framework can be enhanced to be parametric with respect to the input data of the application, allowing to extract/build the application signature only once. In order to do this, a study must be done to see how to parameterize the input data of the applications in a homogeneous way, since each application can have different types of input data.

Moreover, the framework can be used in a similar way as other approaches presented in the related work (Chapter 3). In those approaches the full execution information of the application in a machine A is used to estimate the performance in a machine B. Therefore, the application signature can be extracted in a machine A and then executed in a machine B to estimate performance and power in that target machine without the need to extract/build the application signature again.

Finally, although in this thesis there is no work done in cooling optimization the energy estimation framework using the application signature can be used to optimize the cooling part of the data center. For example, the fan speed of each server can be set to an optimal value by knowing beforehand the mean power (obtained through the application signature) of the applications that will run.

Furthermore, this can be co-optimized together with an energy efficient task scheduling approach by allocating tasks in the servers with the objective to reduce the energy consumption of all the servers and the cooling (fan speed) part of the data center.

6.2.2 Supporting a Cloud Scenario

As we have previously mentioned our present work is focused on a non-cloud scenario. The work by Arroba et al. [7] presents a consolidation of virtual machines in an energy-efficient way. The application signature could improve that methodology by providing the information of energy of the applications before they are executed in the cloud. Moreover, there is an increasing use of scientific applications in cloud scenarios [94] [50]. This type of applications can be presented in cloud systems such as Google Cloud [98], Amazon Elastic Compute Cloud (Amazon EC2) [42] and in Microsoft Azure [53]. These scientific long-running applications rely on the use of machine learning or deep learning techniques that usually are time consuming. Therefore, a natural line of research is to adapt our work to be useful in cloud scenarios whereheterogeneous systems (CPU/GPU/MIC) are found.

6.2.3 Supporting Anomaly Detection and Prediction

The work by Tuncer et al. [119] describe a system to detect and predict anomalies in HPC scenarios. The system uses machine learning techniques to perform a diagnosis and takes as input different metrics from the execution of the applications. Moreover, the system needs to be trained in an offline manner resulting in a time consuming process if the applications are long-running. The application signature can leverage this problem by performing the training in a fast way since there is no need to execute the whole application. Additionally, we can use the application signature to predict possible anomalies before launching the original application.

Bibliography

[1] Aikema, D., Kiddle, C., and Simmonds, R. Energy-cost-aware scheduling of hpc workloads. In *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks* (2011), IEEE, pp. 1–7.

[2] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science 413*, 1 (2012), 142–159.

[3] Albert, E., et al. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Int. Symp. of Formal Methods for Components and Objects (FMCO)* (2007), pp. 113–132.

[4] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS) 15*, 4 (1997), 357–390.

[5] Andrae, A. Total consumer power consumption forecast. *Nordic Digital Business Summit 10* (2017).

[6] Arjona Aroca, J., Chatzipapas, A., Fernández Anta, A., and Mancuso, V. A measurement-based analysis of the energy consumption of data center servers. In *Proceedings of the 5th international conference on Future energy systems* (2014), pp. 63–74.

[7] Arroba, P., Moya, J. M., Ayala, J. L., and Buyya, R. Dynamic voltage and frequency scaling-aware dynamic consolidation of virtual machines for energy efficient cloud data centers. *Concurrency and Computation: Practice and Experience 29*, 10 (2017), e4067.

[8] Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., and Wilde, T. A case study of energy aware scheduling on supermuc. In *International Supercomputing Conference* (2014), Springer, pp. 394–409.

[9] Avgerinou, M., Bertoldi, P., and Castellazzi, L. Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency. *Energies 10*, 10 (2017), 1470.

[10] Ayoub, R., and et al. Temperature aware dynamic workload scheduling in multisocket cpu servers. *IEEE TCAD* (2011).

[11] Bailey, D., et al. The Nas Parallel Benchmarks. *The International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[12] Barroso, L. A., Clidaras, J., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[13] Bates, N., Ghatikar, G., Abdulla, G., Koenig, G. A., Bhalachandra, S., Sheikhalishahi, M., Patki, T., Rountree, B., and Poole, S. Electrical grid and supercomputing centers: An investigative analysis of emerging opportunities and challenges. *Informatik-Spektrum 38*, 2 (2015), 111–127.

[14] Belkhir, L., and Elmeligi, A. Assessing ict global emissions footprint: Trends to 2040 & recommendations. *Journal of cleaner production 177* (2018), 448–463.

[15] Beloglazov, A., Abawajy, J., and Buyya, R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems 28*, 5 (2012), 755–768.

[16] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[17] Bienia, C., and et al. The PARSEC benchmark suite: characterization and architectural implications. In *PACT* (2008), pp. 72–81.

[18] Bohra, A., and et al. Vmeter: Power modelling for virtualized clouds. In *IEEE (IPDPSW)* (2010), pp. 1–8.

[19] Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., and Benini, L. Scheduling-based power capping in high performance computing systems. *Sustainable Computing: Informatics and Systems 19* (2018), 1–13.

[20] Canillas, J. M., Wong, A., Rexachs, D., and Luque, E. Predicting parallel applications performance using signatures: The workload effect. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)* (2011), IEEE, pp. 299–300.

[21] Casas, M., Badia, R. M., and Labarta, J. Automatic phase detection and structure extraction of mpi applications. *The International Journal of High Performance Computing Applications 24*, 3 (2010), 335–360.

[22] Casas, M., Servat, H., Badia, R. M., and Labarta, J. Extracting the optimal sampling frequency of applications using spectral analysis. *Concurrency and Computation: Practice and Experience 24*, 3 (2012), 237–259.

[23] Caubet, J., Gimenez, J., Labarta, J., DeRose, L., and Vetter, J. A dynamic tracing mechanism for performance analysis of openmp applications. In *International Workshop on OpenMP Applications and Tools* (2001), Springer, pp. 53–67.

[24] Chen, H., and et al. Dynamic server power capping for enabling data center participation in power markets. In *ICCAD* (2013), pp. 122–129.

[25] Chen, H., Hsu, W.-C., Lu, J., Yew, P.-C., and Chen, D.-Y. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2003), pp. 79–90.

[26] Chetsa, G. L. T., Lefevre, L., Pierson, J.-M., Stolf, P., and Da Costa, G. A user friendly phase detection methodology for hpc systems' analysis. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing* (2013), IEEE, pp. 118–125.

[27] Chetsa, G. T., Lefèvre, L., Pierson, J.-M., Stolf, P., and Da Costa, G. Exploiting performance counters to predict and improve energy performance of hpc systems. *Future Generation Computer Systems 36* (2014), 287–298.

[28] Cho, C.-B., and Li, T. Using wavelet domain workload execution characteristics to improve accuracy, scalability and robustness in program phase analysis. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software* (2007), IEEE, pp. 136–145.

[29] Cho, H. K., Moseley, T., Hank, R., Bruening, D., and Mahlke, S. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2013), IEEE, pp. 1–10.

[30] Choi, J., et al. Power consumption prediction and power-aware packing in consolidated environments. *IEEE Trans. on Computers* (2010).

[31] Chrétien, S., Nicod, J.-M., Philippe, L., Rehn-Sonigo, V., and Toch, L. Job scheduling using successive linear programming approximations of a sparse model. In *European Conference on Parallel Processing* (2012), Springer, pp. 116–127.

[32] Cochran, R., Hankendi, C., Coskun, A., and Reda, S. Identifying the optimal energy-efficient operating points of parallel workloads. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2011), IEEE, pp. 608–615.

[33] Combs, J., Nazor, J., Thysell, R., Santiago, F., Hardwick, M., Olson, L., Rivoire, S., Hsu, C.-H., and Poole, S. W. Power

signatures of high-performance computing workloads. In *2014 Energy Efficient Supercomputing Workshop* (2014), IEEE, pp. 70–78.

[34] Curtis-Maury, M., Blagojevic, F., Antonopoulos, C. D., and Nikolopoulos, D. S. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems 19*, 10 (2008), 1396–1410.

[35] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)* (2010), pp. 189–194.

[36] Dhiman, G., and et al. A system for online power prediction in virtualized environments using gaussian mixture models. In *DAC* (2010).

[37] Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., Jalby, W., et al. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose* (2005), vol. 200.

[38] Dongarra, J. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing* (London, UK, UK, 1988), Springer-Verlag, pp. 456–474.

[39] Escobar, R., and Boppana, R. Performance prediction of parallel cpu and gpu applications using fractals. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2018), IEEE, pp. 610–617.

[40] Escobar, R., and Boppana, R. V. Performance prediction of parallel applications based on small-scale executions. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)* (2016), IEEE, pp. 362–371.

[41] Etinski, M., Corbalan, J., Labarta, J., and Valero, M. Parallel job scheduling for power constrained hpc systems. *Parallel Computing 38*, 12 (2012), 615–630.

[42] Evangelinos, C., and Hill, C. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2. *ratio 2*, 2.40 (2008), 2–34.

[43] Fan, X., and et al. Power provisioning for a warehouse-sized computer. In *ISCA* (New York, NY, USA, 2007), pp. 13–23.

[44] Fidanova, S. Simulated annealing for grid scheduling problem. In *IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)* (2006), IEEE, pp. 41–45.

[45] García, P. A. Proactive power and thermal aware optimizations for energy-efficient cloud computing. 2017.

[46] Garefalakis, P., Karanasos, K., Pietzuch, P., Suresh, A., and Rao, S. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–13.

[47] Goldman, A., and Ngoko, Y. A milp approach to schedule parallel independent tasks. In *2008 International Symposium on Parallel and Distributed Computing* (2008), IEEE, pp. 115–122.

[48] Govindan, S., Choi, J., Urgaonkar, B., Sivasubramaniam, A., and Baldini, A. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), pp. 317–330.

[49] Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Morse, J., and Eder, K. Static analysis of energy consumption for llvm ir programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems* (2015), pp. 12–21.

[50] Guyon, D., Orgerie, A.-C., Morin, C., and Agarwal, D. How much energy can green hpc cloud users save? In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (2017), IEEE, pp. 416–420.

[51] Haidar, A., Jagode, H., Vaccaro, P., YarKhan, A., Tomov, S., and Dongarra, J. Investigating power capping toward energy-efficient scientific applications. *Concurrency and Computation: Practice and Experience 31*, 6 (2019), e4485.

[52] Hamerly, G., Perelman, E., Lau, J., and Calder, B. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism 7*, 4 (2005), 1–28.

[53] Hassan, H. A., Mohamed, S. A., and Sheta, W. M. Scalability and communication performance of hpc on azure cloud. *Egyptian Informatics Journal 17*, 2 (2016), 175–182.

[54] Henning, J. L. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News 34*, 4 (Sept. 2006), 1–17.

[55] Huang, L., Jia, J., Yu, B., Chun, B.-G., Maniatis, P., and Naik, M. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in neural information processing systems* (2010), pp. 883–891.

[56] Huffmire, T., and Sherwood, T. Wavelet-based phase classification. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques* (2006), pp. 95–104.

[57] Hussain, S. M., Wahid, A., Shah, M. A., Akhunzada, A., Khan, F., ul Amin, N., Arshad, S., and Ali, I. Seven pillars to achieve energy efficiency in high-performance computing data centers. In *Recent Trends and Advances in Wireless and IoT-enabled Networks*. Springer, 2019, pp. 93–105.

[58] Intel. Server Board S2600GZ/GL. Technical Product Specification, 2014 (Revision 2.1).

[59] Isci, C., Buyuktosunoglu, A., Cher, C.-Y., Bose, P., and Martonosi, M. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006), IEEE, pp. 347–358.

[60] Isci, C., Contreras, G., and Martonosi, M. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006), IEEE, pp. 359–370.

[61] Isci, C., and Martonosi, M. Identifying program power phase behavior using power vectors. In *2003 IEEE International Conference on Communications (Cat. No. 03CH37441)* (2003), IEEE, pp. 108–118.

[62] Isci, C., and Martonosi, M. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* (2006), IEEE, pp. 121–132.

[63] Jayakumar, A., Murali, P., and Vadhiyar, S. Matching application signatures for performance predictions using a single execution. In *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), IEEE, pp. 1161–1170.

[64] Kambadur, M., Tang, K., and Kim, M. A. Harmony: Collection and analysis of parallel block vectors. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)* (2012), IEEE, pp. 452–463.

[65] Kambadur, M., Tang, K., and Kim, M. A. Parashares: Finding the important basic blocks in multithreaded programs. In *European Conference on Parallel Processing* (2014), Springer, pp. 75–86.

[66] Kansal, A., and Zhao, F. Fine-grained energy profiling for power-aware application design. *ACM SIGMETRICS Performance Evaluation Review 36*, 2 (2008), 26–31.

[67] Kashani, M., and Jahanshahi, M. Using simulated annealing for task scheduling in distributed systems. *Computational Intelligence, Modelling and Simulation, International Conference on 0* (09 2009), 265–269.

[68] Khoshbakht, S., and Dimopoulos, N. A new approach to detecting execution phases using performance monitoring counters. In *International Conference on Architecture of Computing Systems* (2017), Springer, pp. 85–96.

[69] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *science 220*, 4598 (1983), 671–680.

[70] Koomey, J., et al. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times 9*, 2011 (2011), 161.

[71] Koronen, C., Åhman, M., and Nilsson, L. J. Data centres in future european energy systems—energy efficiency, integration and policy. *Energy Efficiency 13*, 1 (2020), 129–144.

[72] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[73] Lavielle, M. Using penalized contrasts for the change-point problem. *Signal Processing 85*, 8 (Aug. 2005), 1501–1510.

[74] Lee, E. K., Kulkarni, I., Pompili, D., and Parashar, M. Proactive thermal management in green datacenters. *The Journal of Supercomputing 60*, 2 (2012), 165–195.

[75] Lee, Y.-H., and Kim, J. Fast and accurate on-line prediction of performance and power consumption in multicore-based systems. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (2013), IEEE, pp. 1879–1886.

[76] Lei, H., Wang, R., Zhang, T., Liu, Y., and Zha, Y. A multi-objective co-evolutionary algorithm for energy-efficient scheduling on a green data center. *Computers & Operations Research 75* (2016), 103–117.

[77] Lewis, A., and et al. Run-time energy consumption estimation based on workload in server systems. In *HotPower* (Berkeley, CA, USA, 2008).

[78] Li, J., Ma, X., Singh, K., Schulz, M., de Supinski, B. R., and McKee, S. A. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *2009 IEEE international symposium on performance analysis of systems and software* (2009), IEEE, pp. 89–100.

[79] Liqat, U., Banković, Z., Lopez-Garcia, P., and Hermenegildo, M. V. Inferring energy bounds via static program analysis and evolutionary modeling of basic blocks. In *International Symposium on Logic-Based Program Synthesis and Transformation* (2017), Springer, pp. 54–72.

[80] Liqat, U., Georgiou, K., Kerrison, S., López-García, P., Gallagher, J. P., Hermenegildo, M. V., and Eder, K. Inferring parametric energy consumption functions at different software levels: Isa vs. llvm ir. In *International Workshop on Foundational and Practical Aspects of Resource Analysis* (2015), Springer, pp. 81–100.

[81] Llort, G., Gonzalez, J., Servat, H., Gimenez, J., and Labarta, J. On-line detection of large-scale parallel application's structure. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (April 2010), pp. 1–10.

[82] Lu, G., Zhang, W., He, H., and Yang, L. T. Performance modeling for mpi applications with low overhead fine-grained profiling. *Future Generation Computer Systems 90* (2019), 317 – 326.

[83] Luszczek, P. R., Bailey, D. H., Dongarra, J. J., Kepner, J., Lucas, R. F., Rabenseifner, R., and Takahashi, D. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006), vol. 213, Citeseer.

[84] Malony, A. D., and Huck, K. A. General hybrid parallel profiling. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2014), IEEE, pp. 204–212.

[85] Mämmelä, O., Majanen, M., Basmadjian, R., De Meer, H., Giesler, A., and Homberg, W. Energy-aware job scheduler for high-performance computing. *Computer Science-Research and Development 27*, 4 (2012), 265–275.

[86] Marathe, A., Bailey, P. E., Lowenthal, D. K., Rountree, B., Schulz, M., and de Supinski, B. R. A run-time system for power-constrained hpc applications. In *International conference on high performance computing* (2015), Springer, pp. 394–408.

[87] Masanet, E., Shehabi, A., Lei, N., Smith, S., and Koomey, J. Recalibrating global data center energy-use estimates. *Science 367*, 6481 (2020).

[88] McCalpin, J. D. A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers, 1995.

[89] Meng, K., and Norris, B. Mira: A framework for static performance analysis. In *Cluster Computing (CLUSTER), 2017* (2017).

[90] Mera, E., López-García, P., Puebla, G., Carro, M., and Hermenegildo, M. V. Combining static analysis and profiling for estimating execution times. In *International Symposium on Practical Aspects of Declarative Languages* (2007), Springer, pp. 140–154.

[91] Mobius, C., and et al. Power consumption estimation models for processors, virtual machines, and servers. *IEEE TPDS* (2014).

[92] Mukhanov, L., Nikolopoulos, D. S., and de Supinski, B. R. Alea: Fine-grain energy profiling with basic block sampling. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), IEEE, pp. 87–98.

[93] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices 42*, 6 (2007), 89–100.

[94] Netto, M. A., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L., and Buyya, R. Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges. *ACM Computing Surveys (CSUR) 51*, 1 (2018), 1–29.

[95] O'Neill, M., and Ryan, C. Grammatical evolution. *IEEE Transactions on Evolutionary Computation 5*, 4 (2001), 349–358.

[96] Penas, I., Zapater, M., Risco-Martín, J. L., and Ayala, J. L. Sfide: a simulation infrastructure for data centers. In *Proceedings of the Summer Simulation Multi-Conference* (2017), pp. 1–12.

[97] Perelman, E., Hamerly, G., Van Biesbrouck, M., Sherwood, T., and Calder, B. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review 31*, 1 (2003), 318–319.

[98] Posey, B., Deer, A., Gorman, W., July, V., Kanhere, N., Speck, D., Wilson, B., and Apon, A. On-demand urgent high performance computing utilizing the google cloud platform. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)* (2019), IEEE, pp. 13–23.

[99] R. Killick, P. Fearnhead, E. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association 107*, 500 (2012), 1590–1598.

[100] Raghavendra, R., and et al. No "power" struggles: Coordinated multi-level power management for the data center. In *ASPLOS* (2008).

[101] Ramesh, S., Perarnau, S., Bhalachandra, S., Malony, A. D., and Beckman, P. Understanding the impact of dynamic power capping on application progress. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), pp. 793–804.

[102] Reda, N. M., Tawfik, A., Marzok, M. A., and Khamis, S. M. Sort-mid tasks scheduling algorithm in grid computing. *Journal of advanced research 6*, 6 (2015), 987–993.

[103] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., and Hundt, R. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro 30*, 4 (2010), 65–79.

[104] Ryan, C., Collins, J. J., and Neill, M. O. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming* (1998), Springer, pp. 83–96.

[105] Sadjadi, S. M., Shu Shimizu, Figueroa, J., Rangaswami, R., Delgado, J., Duran, H., and Collazo-Mojica, X. J. A modeling approach for estimating execution time of long-running scientific applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), pp. 1–8.

[106] Sancho, M. Z. Proactive and reactive thermal aware optimization techniques to minimize the environmental impact of data centers. 2015.

[107] Sarood, O., and et al. Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems. In *IEEE CLUSTER* (2013).

[108] Sarood, O., Langer, A., Gupta, A., and Kale, L. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 807–818.

[109] Sato, K., Komatsu, K., Takizawa, H., and Kobayashi, H. A history-based performance prediction model with profile data classification for automatic task allocation in heterogeneous computing systems. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications* (2011), IEEE, pp. 135–142.

[110] Schubert, S., Kostic, D., Zwaenepoel, W., and Shin, K. G. Profiling software for energy consumption. In *2012 IEEE International Conference on Green Computing and Communications* (2012), IEEE, pp. 515–522.

[111] Shehabi, A., et al. United States Data Center Energy Usage Report. *Lawrence Berkeley National Laboratory, Berkeley, California. LBNL-1005775* (2016).

[112] Sherwood, T., Perelman, E., and Calder, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques* (2001), IEEE, pp. 3–14.

[113] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices 37*, 10 (2002), 45–57.

[114] Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. Discovering and exploiting program phases. *IEEE micro 23*, 6 (2003), 84–93.

[115] Shoukourian, H., Wilde, T., Auweter, A., and Bode, A. Predicting the energy and power consumption of strong and weak scaling hpc applications. *Supercomputing Frontiers and Innovations 1*, 2 (2014).

[116] Sîrbu, A., and Babaoglu, O. *Power Consumption Modeling and Prediction in a Hybrid CPU-GPU-MIC Supercomputer*. Springer International Publishing, Cham, 2016, pp. 117–130.

[117] Sodhi, S., Subhlok, J., and Xu, Q. Performance prediction with skeletons. *Cluster Computing 11*, 2 (2008), 151–165.

[118] Tsuzuku, K., and Endo, T. Power capping of cpu-gpu heterogeneous systems using power and performance models. In *2015 International Conference on Smart Cities and Green ICT Systems (SMARTGREENS)* (2015), IEEE, pp. 1–8.

[119] Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. Diagnosing performance variations in hpc applications using machine learning. In *International Supercomputing Conference* (2017), Springer, pp. 355–373.

[120] Wahlroos, M., Pärssinen, M., Rinne, S., Syri, S., and Manner, J. Future views on waste heat utilization–case of data centers in northern europe. *Renewable and Sustainable Energy Reviews 82* (2018), 1749–1764.

[121] Wang, L., Khan, S. U., Chen, D., KołOdziej, J., Ranjan, R., Xu, C.-Z., and Zomaya, A. Energy-aware parallel task scheduling in a cluster. *Future Generation Computer Systems 29*, 7 (2013), 1661–1670.

[122] Whitehead, B., et al. Assessing the environmental impact of data centres part 1: Background, energy use and metrics. *Building and Environment 82* (2014), 151–159.

[123] Wilson, A. G., Hu, Z., Salakhutdinov, R., and Xing, E. P. Deep kernel learning. In *Artificial Intelligence and Statistics* (2016), pp. 370–378.

[124] Wong, A., et al. Parallel application signature for performance analysis and prediction. *IEEE Trans. on Parallel and Distributed Systems* (2015).

[125] Wong, A., Rexachs, D., and Luque, E. Extraction of parallel application signatures for performance prediction. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)* (2010), IEEE, pp. 223–230.

[126] Yang, L. T., et al. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing, 2005.* (2005).

[127] Zapater, M., and et al. Leakage-aware cooling management for improving server energy efficiency. *TPDS PP*, 99 (2014).

[128] Zhai, J., Chen, W., Zheng, W., and Li, K. Performance prediction for large-scale parallel applications using representative replay. *IEEE Transactions on Computers 65*, 7 (2015), 2184–2198.

[129] Zhai, Y., and et al. Happy: Hyperthread-aware power profiling dynamically. In *USENIX Annual Conference* (2014), pp. 211–218.

[130] Zhan, X., and et al. Techniques for energy-efficient power budgeting in data centers. DAC.

[131] Zhan, X., and Reda, S. Techniques for energy-efficient power budgeting in data centers. In *Proceedings of the 50th Annual Design Automation Conference* (2013), DAC '13, ACM, pp. 176:1–176:7.

[132] Zhang, W., Cheng, A. M., and Subhlok, J. Dwarfcode: a performance prediction tool for parallel applications. *IEEE Transactions on Computers 65*, 2 (2015), 495–507.

[133] Zhang, W., Hao, M., and Snir, M. Predicting hpc parallel program performance based on llvm compiler. *Cluster Computing 20*, 2 (2017), 1179–1192.

[134] Zheng, X., and et al. Markov model based power management in server clusters. In *GREENCOM* (2010), pp. 96–102.

[135] Zhongzhi, S. *Advanced artificial intelligence*, vol. 4. World Scientific, 2019.

Appendix A

Grammatical Evolution Technique

In this appendix we describe the mapping process in the Grammatical Evolution technique. We use this technique in this thesis to develop the power (Section 4.2.1.1.2) and co-allocated hardware counters (Section 4.2.2) models. For a more detailed explanation on the principles of Grammatical Evolution, the reader is referred to the work by Ryan et al. [104].

In Grammatical Evolution (GE), a mapping process is applied to extract the mathematical expression given by an individual. The individual is defined as the phenotype. The mapping process sets the rules to obtain a mathematical expression and this is done by using grammars expressed in Backus Naur Form (BNF) [95]. In general terms, a BNF specification is a set of rules expressed as:

$$< symbol >::=< expression > \tag{A.1}$$

The rules defined by the BNF are expressed as a sequence of terminals and non-terminals symbols. Non-terminal symbols can appear in either the right or the left part of the rule, as shown in the Equation A.1. Additionally, non-terminal symbols are enclosed between the pair <>. The Equation A.1 defines that the $< symbol >$ will be replaced by an expression ($< expression >$). Moreover, the terminal symbols never appear on the left side of the rules and they are usually represented by operations, functions (such as log, sin, etc) and numbers.

A grammar is composed by the tuple N, T, P, S, where: i) N is the non-terminal set, ii) T is the terminal set, iii) P is the production rules for the assignment of elements on the sets N and T, and iv) S is a start symbol that must appear in N. The "|" symbol separates the different options within a production rule.

A grammar example in BNF format is shown in Grammar 3. The final mathematical expression is composed by the elements of the set of terminals T,

Grammar 3 Example of a grammar in BNF format

N = {expr, op, preop, var, num, dig}
T = {+, -, *, /, sin, cos, log, x, y, z, 0, 1, 2, 3, 4, 5, (,), .}
S = {expr}
P = {I, II, III, IV, V, VI}
<expr> ::= <expr><op><expr> | <preop>(<expr>) | <var>
<op> ::= +|-|*|/
<preop> ::= sin|cos|log
<var> ::= x|y|z|<num>
<num> ::= <dig>.<dig> | <dig>
<dig> ::= 0 | 1 | 2 | 3 | 4 | 5

which have been combined with the rules of the defined grammar.

The Grammatical Evolution technique defines a population of individuals. Each individual is represented by a chromosome and a fitness value. The chromosomes are formed by a number of genes (or codons). Then, the original population is evolved by a genetic algorithm with the objective of improving the fitness function.

The Grammatical Evolution technique uses the genes (or codons) from the chromosomes to map symbols onto the expressions. The genes are numbers defined in a number of bits, for example 8 bits. This means that each gene from the chromosome can have a value from 0 to 255. Then, the algorithm combines the chromosomes (crossover process) of the population in order to improve the fitness function. Additionally, a mutation of the individuals can occur with a certain level of probability (mutation probability). Hence, we can see that process to select the final features is done automatically by the algorithm. Finally, in order to replace the expressions the GE technique uses the modulus operator as follows:

$$\text{Rule} = \text{Codon (or gene) Value MOD Number of Rule Choices} \tag{A.2}$$

The process of obtaining a mathematical expression using the GE technique can be better explained with an example. In the following example, we explain the mapping process using the grammar shown in Grammar 3. The reader can find a further detailed explanation of this example in the works by Zapater et al. [106] and Arroba et al. [45].

Suppose we have a BNF grammar (Grammar 3) and the following 7-gene chromosome: 21-64-17-62-38-254-2. In this case, the gene is an 8 bit integer. The start symbol is $S = \langle expr \rangle$, therefore the decoded expression will begin with the following non-terminal:

$$Solution = \langle expr \rangle$$

We use the first value of the gene (21) of the chromosome in rule `I` of the grammar. The number of choices in that rule is 3. Therefore, a mapping function is applied: `21 MOD 3 = 0` and the first option is selected (since we obtain a value of 0) $\langle expr \rangle \langle op \rangle \langle expr \rangle$. As a consequence, the current expression is the following:

$$Solution = \langle expr \rangle \langle op \rangle \langle expr \rangle$$

Then, we proceed to decode the first non-terminal of the current expression, which again is $\langle expr \rangle$. We use the value of the next gene (64) to replace the expression: `64 MOD 3 = 1`, hence, the second option $\langle preop \rangle (\langle expr \rangle)$ is selected and the current expression is updated:

$$Solution = \langle preop \rangle (\langle expr \rangle) \langle op \rangle \langle expr \rangle$$

The next gene value 17, is taken for decoding. In this case, the first non-terminal in the current expression is $\langle preop \rangle$, we apply the mapping function (modulus operator) and the third option is selected. Since $\langle preop \rangle$ has three options we still have to do a MOD 3 operation. We obtain the following expression:

$$Solution = log(\langle expr \rangle) \langle op \rangle \langle expr \rangle$$

The mapping process for the next genes are the following:

- Decode $\langle expr \rangle$ -> `62 MOD 3 = 2` -> $Solution = log(\langle var \rangle) \langle op \rangle \langle expr \rangle$
- Decode $\langle var \rangle$ -> `38 MOD 4 = 2` -> $Solution = log(z) \langle op \rangle \langle expr \rangle$
- Decode $\langle op \rangle$ -> `254 MOD 4 = 2` -> $Solution = log(z) * \langle expr \rangle$
- Decode $\langle expr \rangle$ -> `2 MOD 3 = 2` -> $Solution = log(z) * \langle var \rangle$

We can see that at this point we have run out of genes and still do not have a mathematical expression. The GE technique solves this problem by starting from the first gene of the chromosome. The technique of using the genes more than once is called wrapping. By applying a wrapping process to our example we decode $\langle var \rangle$ with the gene 21: `21 MOD 3 = 1`. Then, the final solution is the following:

$$Solution = log(z) * y$$

The fitness function express the error of the estimation process. As we previously commented, the GE technique evolves the solution to improve the fitness function (reduce the error). In our work, we use the Root Mean Square Error as a fitness function f presented in Equation A.3:

$$\begin{aligned} f &= \sqrt{\frac{1}{N} \cdot \sum_{n} {e_{\mathrm{n}}}^2} \\ e_{\mathrm{n}} &= |X(n) - \widehat{X}(n)|, \qquad 1 \leq n \leq N \end{aligned} \tag{A.3}$$

We use the GE technique to either estimate the power or the co-allocated hardware counters, then in Equation A.3 the value of X represents either the real samples of power or the co-allocated hardware counters and, the value of $\widehat{X}$ represents the power estimation or the estimated co-allocated hardware counters. The estimation error e_{n} represents the deviation between the real samples X and the estimation obtained by the model $\widehat{X}$. The value of n represents each sample of the entire set of N samples used to train the GE algorithms.