Predicting performance and power consumption of parallel applications

Daniele De Sensi
Dept. of Computer Science, Univ. of Pisa
Email: desensi@di.unipi.it

***Abstract*—Current architectures provide many control knobs for the reduction of power consumption of applications, like reducing the number of used cores or scaling down their frequency. However, choosing the right values for these knobs in order to satisfy requirements on performance and/or power consumption is a complex task and trying all the possible combinations of these values is an unfeasible solution since it would require too much time. For this reasons, there is the need for techniques that allow an accurate estimation of the performance and power consumption of an application when a specific *configuration* of the control knobs values is used. Usually, this is done by executing the application with different configurations and by using these information to predict its behaviour when the values of the knobs are changed. However, since this is a time consuming process, we would like to execute the application in the fewest number of configurations possible.**

In this work, we consider as control knobs the number of cores used by the application and the frequency of these cores. We show that on most PARSEC benchmark programs, by executing the application in 1% of the total possible configurations and by applying a multiple linear regression model we are able to achieve an average accuracy of 96% in predicting its execution time and power consumption in all the other possible knobs combinations.

Keywords: performance prediction, power consumption prediction, concurrency throttling, DVFS, linear regression.

I. INTRODUCTION

Power consumption is becoming a key factor in designing applications and computing systems. This is motivated both by economical and environmental reasons. In fact, the energy cost is quickly going to overcome the cost of the physical system itself [1]. Moreover, high power consumption rises the temperature of the system, increasing the probability of failure of the physical components and requiring advanced heat management systems, that increase the cost by one dollar for each dollar spent in electricity [2]. On the other hand, energy consumption has a considerable impact on the environment, since during 2010 the CO_2 emissions of US' data centers were on par with those of an entire country like Argentina [3].

Existing power aware solutions often operate on control knobs to reduce the amount of resources used by an application, thus decreasing its power consumption. Usually this implies reducing the number of cores used by the application (*concurrency throttling*) or scaling down the frequency of these cores (*Dynamic Voltage and Frequency Scaling (DVFS)*). a specific combination of the knobs values is often referred as a "*configuration*". However, decreasing the resources used by the application may produce a degradation in the performances. Accordingly, to have the desired trade-off, it's usually possible to require a maximum allowed power consumption and/or a minimum level of performance. To satisfy such requirements, we should be able to answer to the following questions:

- *Which is the least power consuming configuration that allows the application to finish before a specified time deadline?*
- *Which is the most performing configuration that consumes less power than a specified threshold?*

To answer these questions, we need to know both the performance and the power consumption of the application in all the possible configurations. One solution is to execute the application with a specific configuration and to stop it after we obtained the values for power consumption and performances[1]. If we repeat this process for all the possible configurations, we can then choose the one that satisfies the requirements. However, on current machines this would be too costly since we have an high number of possible configurations and in many cases the search process could last longer than the execution itself. For example, the machine used to validate this work has 24 cores and 13 possible frequency levels, for a total of 312 possible configurations to explore. If we should also consider other possible control knobs, this number would rapidly increase.

A more efficient approach would be to execute the application only in few configurations and to use the collected information to predict the performance and power consumption of the application in all the other knobs combinations. In some of these systems [4], [5] the exploration phase is done while the application is running, without restarting the application between configurations changes. To be precise, they start the execution in an arbitrary configuration and, while running, they change the number of threads of the application and the frequency of the cores. This process continues until they collect enough information to have a sufficiently high prediction accuracy. After this phase, they use the obtained model to execute the remaining part of the computation in the best configuration.

For these reasons, we need prediction algorithms which require to explore the smallest possible number of configurations. Moreover, the algorithm should be simple enough to be

[1]For the sake of clearness, in this work we consider the case in which the algorithm collects data about execution time. However, since we stop the application before it finishes, we can alternatively collect the data about the bandwidth, defined as the number of elements processed per time interval. Since the bandwidth is the inverse of the execution time, the algorithm would still work in the same way.

applied at runtime with minimum impact on the application latency.

Our prediction algorithm is targeted towards such kind of systems, where the duration of the exploration phase is a critical factor. The main contributions of this paper are:

1) The proposal of a strategy for low latency and high accuracy predictions of the execution time and power consumption of all the possible configurations of a parallel application. By exploring just few configurations, we are able to accurately predict the behaviour of the application in all the non explored configurations.
2) Differently from many existing solutions that collect information using specific hardware counters (which may not be available on all architectures), our solution only needs the execution time of the application and its power consumption.

We validate the proposed algorithm on PARSEC [6] applications. PARSEC is a well known benchmark containing applications from many different domains and with different characteristics in terms of parallelization model, working set size and data sharing and exchange between computational nodes, thus allowing the assessment of our approach over a wide range of real world scenarios. Moreover we propose and compare different strategies to select the configurations to be explored, in order to minimise the time required to obtain the model.

The paper is structured as follows. In Section II we analyse some of the existing works in this area. Then, in Section III we will describe our algorithm proposal. The achieved results will be later shown in Section IV. Eventually, in Section V we will draw conclusions and outline some possible future directions for our work.

II. RELATED WORK

In this section we analyse some of the works addressing the estimation of the performances and power consumption of an application in all its possible configurations.

Li and Martinez [4] propose a system that, at runtime, tries different configurations to find the one that satisfies the given requirements in terms of performances and power consumption. Since an exhaustive search would be too costly, they cut down the search space by using some heuristics. However, even though the search space is reduced, the cost significantly increases with the number of possible configurations. Moreover, the applications runs are only simulated and the values for execution time and power consumption may be different from those obtained in real executions.

In [5] the authors propose a method to determine the optimal degree of parallelism for loop based computations, by executing some iterations of the loop to collect data to train the algorithm. After this phase, the algorithm predicts the execution time of the application in all the configurations and chooses the fastest one to execute the remaining iterations of the loop. Although they succeed in finding the fastest configuration, their solution lacks of generality, since it is limited to loop based applications, with synchronisations performed only at the end of each parallel section. Moreover, DVFS is not considered and the power consumption of the configurations is not explicitly estimated. Differently from our work, this algorithm is only executed on traces collected from simulated executions.

Pusukuri et al. propose in [7] a method to estimate the number of threads such that the shortest execution time is obtained. For a specific application and input, they run the application on that input for a short period of time and with different configurations, collecting information for each execution by using hardware counters. Eventually, when the best configuration is found, they start the application from scratch. Nevertheless, they do not consider the possibility to change the frequencies of the cores. Moreover, power consumption is not investigated since the focus of this work is on finding the most performing solution.

These approaches are close to the one we propose in this work. However, power consumption is often not explicitly modelled thus not allowing the tuning of performance-power trade-offs. Moreover, some of them need to collect data from hardware counters during the exploration phase, while our approach only requires the execution time of the application and its power consumption. In addition to this, our algorithm have been extensively tested on a wide range of applications executed on real hardware, while in many existing works the executions have only been simulated.

Other works [8], [9] present more general models that, after exploring different configurations of different applications, allow the prediction even for applications not analysed during the exploration phase. However, this is usually possible thanks to more complicated models that take into account many different factors, requiring long training phases and thus not usable at runtime. Our algorithm is orthogonal to these since it is much simpler and faster in deriving a model, even if it is only able to predict the behaviour of the same application analysed during the exploration phase. Moreover, our approach is particularly suited for highly dynamic runtime supports [10] since, due to frequent changes in the workload intensity, there is the need to continuously recompute the model with minimum impact on the application performances.

III. ALGORITHM

In this Section, we will show the approach we used to predict the performance and the power consumption of an application in all its possible configurations.

A. Multiple linear regression

In multiple linear regression [11], we model the relationship between two or more independent variables (called *predictors*) and a dependent variable (called *response*) by fitting a linear equation to observed data. In our case, the predictors are the number of cores used by the application and their frequency, while the responses are the execution time or the power consumption.

Suppose we made a set of n *observations* to obtain the values of the responses $y_1, y_2, \ldots, y_n$. Let $x_i = x_{i,1}, x_{i,2}, \ldots, x_{i,p}$ denote the p predictors for the observation i. Then we have:

$$y_i = \beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p} + \epsilon_i \quad (1)$$

where β_i is a regression coefficient and ϵ_i is a term representing a random error due to measurement error or fluctuations in the results.

By fitting a regression model to observations, we determine the β coefficients, thus enabling the prediction of the responses for the unobserved predictors. To fit the model, we use the *least squares method*, minimising the sum of the squares of the residuals, where a residual is the difference between the real value of the dependent variable and the value predicted by the model.

B. Performance and power consumption modelling

By using linear regression, we will perform an interpolation of performances and power consumption values obtained in few configurations to infer an analytical model which allows us to predict the behaviour of all the other configurations. To achieve this goal, we first need to express execution time and power consumption in a form similar to the one of Equation 1, i.e. as a linear combination of the number of cores[2] and their frequency.

Concerning the performances we can use the Amdahl's law [12][3] , defined as:

$$T(t) = T(1)\left(B + \frac{(1-B)}{t}\right) \quad (2)$$

where t is the number of threads, $T(t)$ is the execution time with t threads and B is the percentage of the application that is strictly sequential.

We can generalise this equation to also consider the effects of frequency scaling on the execution time. First of all, we assume that we can't change the frequency of the cores individually but we are forced to change them simultaneously for all the cores on a CPU. This is a reasonable assumption since this is how DVFS currently work on most existing architectures [13].

When the frequency is increased, the execution time with one thread proportionally decrease [14]. Accordingly, if f is the frequency and f_{min} is the minimum available frequency, then we have:

$$\begin{aligned} T(t,f) &= \frac{T(1,f_{min})}{\frac{f}{f_{min}}}\left(B + \frac{(1-B)}{t}\right) = \\ &= \frac{f_{min}}{f}T(1,f_{min})(B) + \\ &+ \frac{f_{min}}{ft}T(1,f_{min})(1-B) = \\ &= \beta_1\frac{f_{min}}{f} + \beta_2\frac{f_{min}}{ft} \end{aligned} \quad (3)$$

For example, if we fix the number of threads and we set a frequency $f = 2f_{min}$ both serial and parallel time will be halved.

Concerning the power consumption of a CPU, it can be modelled as in [15], [16], [17]:

$$P(t,f,v) = vI_{leak} + \alpha c v^2 f t \quad (4)$$

where v is the voltage, I_{leak} is the *leakage current*, α is the activity factor and c is the capacitance. For our purposes, we can consider α, c and I_{leak} as constants.

If we consider a system with multiple CPUs, this formula applies separately for each of them. Let $\overline{k}$ and $\underline{k}$ be the number of active and inactive CPUs respectively. Then, for a given application we have[4] :

$$\begin{aligned} P(t,f,v,\overline{k},\underline{k}) &= \overline{k}(vI_{leak} + \alpha c v^2 f t) + \underline{k} v I_{leak} = \\ &= \overline{k} v I_{leak} + \underline{k} v I_{leak} + \overline{k}\alpha c v^2 f t \end{aligned} \quad (5)$$

However, we would like to remove the model dependence from v, $\overline{k}$ and $\underline{k}$. Concerning v, it is strictly correlated to the frequency, since by increasing the frequency we raise the operating voltage. The relationship between voltage and frequency may be computed once and for all, programmatically or by using the values provided by the CPU vendor. Consequently, we can use a tabular function $V(f)$ to get the voltage value associated to a specific frequency level f.

To get the number of used CPUs, we assume a linear mapping of the threads over the cores. In this way, we start using an additional CPU only if we do not have any available core on the current one. Let $\tilde{k}$ be the number of cores available for each CPU and K be the number of available CPUs. Then we have

$$\begin{aligned} \overline{k} &= \left\lceil \frac{t}{\tilde{k}} \right\rceil \\ \underline{k} &= K - \overline{k} \end{aligned} \quad (6)$$

We can then rewrite Equation 5 as:

[2]It's worth noticing that in this work we only consider scenarios with at most one thread running on each core. Accordingly, to reduce the number of cores used by the application we need to reduce the number of threads it uses.

[3]We decided to use a very general model since we do not want to make assumption about the structure of the application. However, for structured applications models that are more detailed could be used.

[4]In our experiments we considered a different voltage for active and inactive CPUs. However, to simplify the exposition, we only show the formula with a single voltage. The model can be easily modified to consider situations with two different voltages.

$$
\begin{aligned}
P(t,f) = & \left\lceil \frac{t}{\widetilde{k}} \right\rceil V(f) I_{leak} + \\
& + \left(K - \left\lceil \frac{t}{\widetilde{k}} \right\rceil \right) V(f) I_{leak} + \\
& + \left\lceil \frac{t}{\widetilde{k}} \right\rceil \alpha c V(f)^2 f t = \\
& = \beta_0 \left\lceil \frac{t}{\widetilde{k}} \right\rceil V(f) + \\
& + \beta_1 \left(K - \left\lceil \frac{t}{\widetilde{k}} \right\rceil \right) V(f) + \\
& + \beta_2 \left\lceil \frac{t}{\widetilde{k}} \right\rceil V(f)^2 f t =
\end{aligned}
\tag{7}
$$

At this point, we can use multiple linear regression to obtain the β_i values in Equations 3 and 7. It's worth noticing that the β_i values include the constants of the application that we do not know a priori, like the activity factor α or the serial fraction B. Basically, by using linear regression to derive these values, we are able to complete our analytical model.

Eventually, we can use the obtained equations to predict the execution time and the power consumption of the application in all the possible *[t, f]* configurations.

IV. Results

In this Section, we will show the results we obtained by applying our prediction algorithm. Moreover, we will evaluate different alternative solutions for the choice of the configurations to be used as input for the linear regression model.

A. Testing methodology

One of the targets of this work is to assess the prediction accuracy of the proposed algorithm. Accordingly, we need to compare the values obtained by the prediction algorithm with those obtained in a real execution. For this reason, we first need to run the application in all the possible *[Threads, Frequency]* configurations, collecting the real execution times and power consumptions.

To assess the accuracy of the prediction algorithm we used the *holdout* method, described as follows:

1) We select a certain percentage of the total configurations as input for the multiple linear regression algorithm, thus obtaining a model of the execution time and power consumption.
2) After that, we use this model to predict the behaviour of the application in all the configurations not selected at the previous step.
3) Eventually, we compute the error of the predictions by using the Mean Absolute Percentage Error (MAPE), defined as:

$$
\varepsilon = \frac{1}{p} \sum_{i=1}^{p} \left| \frac{A_i - P_i}{P_i} \right| \tag{8}
$$

where p is the number of configurations, A_i is the real execution time (or power consumption) and P_i is the predicted execution time (or power consumption).

The average accuracy is then computed as $100 - \varepsilon$.

B. Test environment

To validate our approach, we used the applications provided by PARSEC. PARSEC [6] is a well-known benchmark suite of parallel applications, diverse in terms of: application domain, programming model (pipeline, data-parallel and unstructured), granularity, working set size, data sharing and data exchange patterns. This heterogeneity allowed us to validate our approach on a wide range of real world applications.

Among the different input sizes provided by PARSEC, we chosen the *native* one, in order to have a real world behaviour of the applications. We executed our algorithm on all the applications provided except X264, which we were not able to run on our system. For all the benchmarks we executed the *pthread* version, except for FREQMINE which only provides the *OpenMP* version.

The number of threads to activate have been selected with the `-n t` parameter provided by the `parsecmgmt` tool. This parameter however ensures that *at least* `t` threads will be activated. In most cases, only one or two threads more will be created, for scheduling and collecting data from the other threads [7]. However their impact on the performances and power consumption is negligible and they can be executed on a core together with a working thread. For this reason, we consider the number of used cores equal to that specified by the `-n` parameter. Particular cases are those of FERRET and DEDUP applications, which respectively activate $(4 \times t) + 3$ and $(3 \times t) + 2$ threads. In these cases, we also consider the additional threads since they are not scheduling threads but they actively contribute to the processing of the input data. Moreover, some benchmarks only allow some values for the `-n` parameter. To be precise, FACESIM only allows $1, 2, 3, 4, 6, 8, 16$ values, while FLUIDANIMATE only allows $1, 2, 4, 8, 16$ values.

All experiments were conducted on an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyperthreaded cores, running with Linux x86 64. This machine has 13 possible frequency levels: from 1.2GHz to 2.4GHz with 0.1GHz steps. Since in these tests we do not use hyperthreading and we perform a mapping of at most one thread for each core, we will have at most 24 threads running on the machine, leading to 312 possible configurations (13 frequency steps times 24 threads). To change the frequency of the cores, we used the `cpupower` utilities.

To get the power consumption, we used the Running Average Power Limit (RAPL) feature [18], introduced by Intel in its newer architectures. It provides sensors for measuring the power consumption of an entire CPU package, of the set of cores only, of some uncore devices or of the memory controller. In our case, we considered the power consumption of the set of cores on the CPUs. However, our approach would also work by considering the power consumption of the entire CPU, since they often differ only by a constant factor. It's important to notice that our approach is not bound to the power reading mechanisms provided by Intel and any other method to obtain the power consumption could be used too.

In all the presented results, we only considered the time spent in the so-called *region of interest* (ROI), i.e. the time

spent in the parallel sections of the applications, without considering initialisation and cleanup phases. This approach is commonly used [9], [7] to avoid distortions of the measurements. The same approach has also been adopted for power consumption results.

To better understand the variability in the behaviour of the applications of the benchmark, in Figures 1 and 2 we show their scalabilities with respect to the number of cores as well as their power consumption. As we can see, they cover a wide range of situations, with maximum scalability ranging from 11 to 22 and maximum power consumption ranging from 39 to 106 Watts. Concerning the scalability with respect to the frequency, we do not show the full results here due to space constraints. However, they also cover a wide spectrum of situation, with a maximum scalability ranging from 1.5 to 2.

This variability allowed us to assess our algorithm on a large set of real world applications.

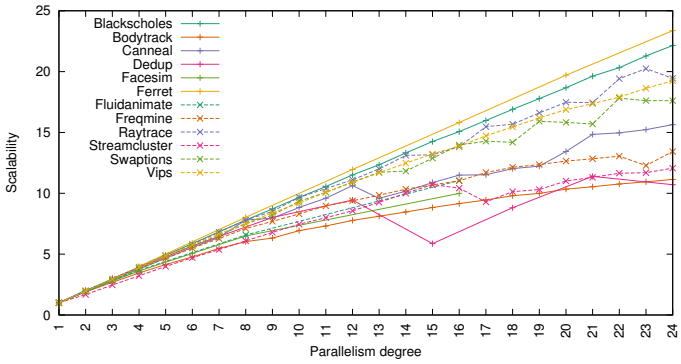

Fig. 1. Scalability of PARSEC applications with respect to the parallelism degree. Frequency is fixed to 2.4GHz.

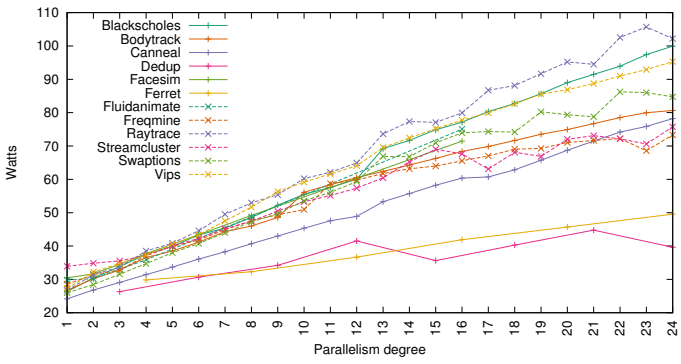

Fig. 2. Power consumption of PARSEC applications with respect to the parallelism degree. Frequency is fixed to 2.4GHz.

C. Choice of the configurations to be explored

One of the first problems to address, is how to choose the configurations to interpolate to obtain our models.

Firstly, we can imagine our configurations as points on a plane, where on the x-axis we have the number of cores and on the y-axis the frequencies.

An important observation is that if two points are close to each other, then we are collecting data about two configurations which are very similar, i.e. they have a similar number of threads or a similar frequency. In this situation, most likely the second point will not add any significant information. Consequently, the more evenly distributed the points are, the more information they provide to the algorithm, thus allowing an higher accuracy.

For this reason, we decide which configurations to choose by using *low discrepancy* generators [19]. Such generators cover the domain more evenly with respect to pseudo-random generators, as shown in Figure 3.

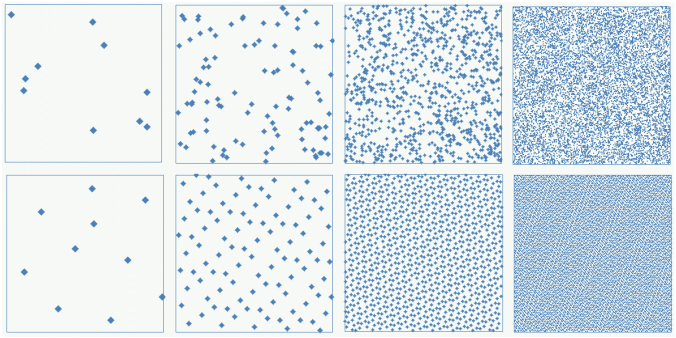

Fig. 3. Comparison between the points generated by a pseudorandom generator (top) and those generated by a low discrepancy generator (down). From left to right, we have 10, 100, 1000 or 10000 generated points.

Moreover, they also have an advantage over deterministic methods since the latter give good equidistribution properties only when the number of points is known a priori, while in low discrepancy generators the equidistribution improves as more points are added. This is a frequent situation since we may want to add points to the interpolation process until the accuracy doesn't get higher than a specified threshold.

In this work, in order to find the most suited generator for our purpose, we compare the pseudorandom generator[5] with the *Niederreiter base 2* [20], *Sobol* [21], *Halton* [22] and *Reverse Halton* [23] low discrepancy generators.

Since the points generated by the pseudorandom generator may change between different executions, thus leading to different average accuracies, we averaged the average accuracy over 200 runs, computing also the 95% confidence interval. On the other hand, this was not necessary for low discrepancy generators since for a given number of points their behaviour is deterministic.

The first result we obtained from the comparison is that, when a sufficient number of points is used (> 20), no significant differences in accuracy are present between the different generators. This is because with such a number of points, even if they are not truly uniformly distributed, the algorithm has enough information to correctly predict the data.

In Figure 4 we show the comparison of the accuracies when we interpolate 4 configurations to try to predict the other 308 configurations.

As we can see from the results, *Niederreiter*, *Halton* and *Reverse Halton* generators always perform better with respect to the pseudorandom one. The best generator among those proposed is *Halton*, with an average accuracy improvement

[5]To be precise, a *multiply-with-carry* generator.

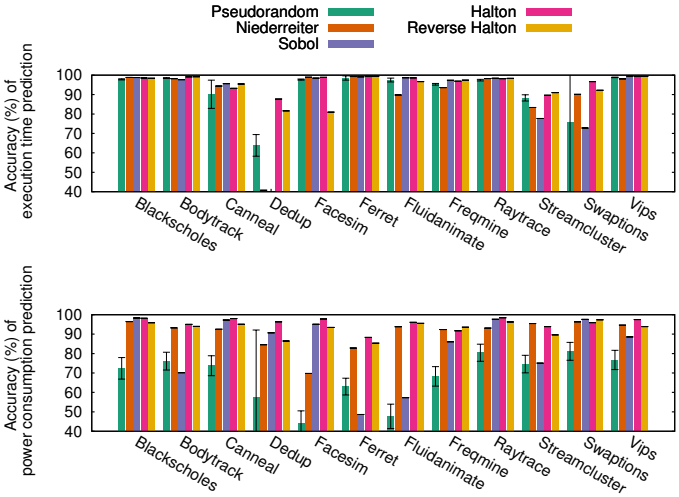

Fig. 4. Comparison of the prediction accuracy between pseudorandom and low discrepancy generators. 4 configurations have been used to obtain the model. For pseudorandom generator, we averaged the results over 200 runs. Error bars represent the 95% confidence interval.

of 26.61% in predicting power consumption and 12.74% in predicting the execution time. The maximum improvement obtained by using this generator is 53.78% for power consumption and 24.43% for execution time.

On the contrary, *Sobol* performs significantly worst in almost all the cases. This effect is caused by the generation of few points and it disappears as the number of explored points increases.

Increasing the number of configurations, the difference between the generators decreases, as shown in Figure 5 for the power consumption prediction accuracy. From the results we can see that to achieve an accuracy of $\sim$ 95% the *Pseudorandom* generator needs ~ 20 points, while *Halton* only requires 4 points. If we assume to spend a constant amount of time in each configuration, by using the *Halton* generator we can achieve a fivefold reduction in the time required to obtain the model with respect to the *Pseudorandom* generator. For the completion time we got similar results, even though the advantage of the *Halton* generator over the *Pseudorandom* one is not so emphatic.

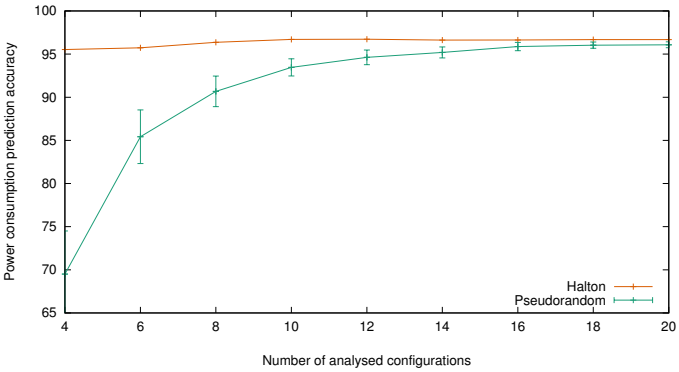

Fig. 5. Comparison of the power consumption prediction accuracy between pseudorandom and Halton generators while increasing the number of explored configurations. For pseudorandom generator, we averaged the results over 200 runs. Error bars represent the 95% confidence interval.

TABLE I. AVERAGE ACCURACY OF EXECUTION TIME AND POWER CONSUMPTION PREDICTIONS. BETWEEN BRACKETS, THE STANDARD DEVIATION FROM THE MEAN.

APPLICATION	AVERAGE EXECUTION TIME ACCURACY (STANDARD DEVIATION)	AVERAGE POWER CONSUMPTION ACCURACY (STANDARD DEVIATION)
BLACKSCHOLES	98.53 (1.23)	98.17 (1.51)
BODYTRACK	99.14 (0.71)	95.04 (3.95)
CANNEAL	93.24 (4.99)	98.02 (1.57)
DEDUP	87.66 (11.81)	96.33 (3.76)
FACESIM	98.86 (0.95)	97.78 (2.69)
FERRET	99.44 (0.42)	96.47 (3.66)
FLUIDANIMATE	98.57 (1.21)	96.07 (3.00)
FREQMINE	96.93 (2.69)	91.73 (8.73)
RAYTRACE	98.18 (1.85)	98.45 (1.26)
STREAMCLUSTER	89.65 (10.35)	93.85 (5.69)
SWAPTIONS	96.75 (3.38)	96.66 (3.66)
VIPS	99.33 (0.60)	97.49 (2.25)
AVERAGE	96.35 (3.34)	96.33 (3.47)

In Table I we show the average accuracy and the standard deviation for execution time and power consumption predictions, obtained by using the *Halton* generator and 4 configurations. For the execution time prediction, we achieve a maximum of 99% for of BODYTRACK, FERRET and VIPS. For power consumption prediction, we get a maximum accuracy of 98% for BLACKSCHOLES, CANNEAL and RAYTRACE. In both cases, we achieve an average accuracy of 96%. It's important to notice that there is no strict relationship between the linearity of the scalability and the accuracy of the model. For example, by looking at Figure 1 we can observe that BODYTRACK exhibits a worst scalability than CANNEAL, while achieving a better prediction accuracy.

D. Search of the best configurations

In this section, we would like to understand if the method succeeds in finding the most performing configuration under a power budget or the least consuming configuration under performance constraints. Indeed, albeit the accuracy of our method is sufficiently high, we may still not succeed in finding these configurations. This is because, even with a small percentage of error, we may still miss the best configuration.

We define the following types of optimal configurations, corresponding to different types of trade-offs between power consumption and execution time:

- $minPower(\tau)$ This is the configuration that minimises the power consumption while terminating in a time less than τ.
- $minTime(\pi)$ This is the configuration that minimises the execution time while not consuming more than a power π.

Concerning $minPower(\tau)$, we tested it under different time requirements. Let T_{min} and T_{max} be respectively the minimum and the maximum execution time of an application. Then, we set:

$$\tau = T_{min} + ((T_{max} - T_{min}) * i) \quad (9)$$

where i varies between 0.1 and 1, with steps of length 0.1. For example, if an application has a minimum execution

TABLE II. COMPARISON BETWEEN OUR ALGORITHM AND THE IDEAL ONE IN FINDING THE $minPower(\tau)$ CONFIGURATION. THE FIRST COLUMN REPRESENTS THE i VALUE OF EQUATION 9. FOR EACH TEST WE INDICATE WHETHER IT IS A *Miss*, A *Success* OR A *Loss*. FOR LOSSES, WE REPORT THEIR VALUES.

i	BLACKSC.	BODYTRACK	CANNEAL	DEDUP	FACESIM	FERRET	FLUIDAN.	FREQMINE	RAYTRACE	STREAMCL.	SWAPT.	VIPS
0.1	SUCC.	7.23	SUCC.	MISS	6.08	SUCC.	5.78	12.22	SUCC.	5.73	SUCC.	SUCC.
0.2	SUCC.	SUCC.	4.76	MISS	1.54	SUCC.	4.43	SUCC.	SUCC.	4.47	SUCC.	SUCC.
0.3	2.27	SUCC.	SUCC.	MISS	1.1	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	MISS	SUCC.
0.4	SUCC.	SUCC.	SUCC.	MISS	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.
0.5	SUCC.	2.34	0.91	3.45	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	5.05	SUCC.	SUCC.
0.6	SUCC.	SUCC.	SUCC.	3.45	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.
0.7	SUCC.	SUCC.	SUCC.	3.45	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.
0.8	SUCC.	SUCC.	SUCC.	MISS	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.	SUCC.
0.9	SUCC.	SUCC.	SUCC.	MISS	SUCC.	SUCC.	SUCC.	MISS	SUCC.	1.41	SUCC.	MISS
1	SUCC.	4.34	7.19	SUCC.	SUCC.	SUCC.	2.76	11.83	3.52	1.41	3.89	SUCC.

TABLE III. COMPARISON BETWEEN OUR ALGORITHM AND THE IDEAL ONE IN FINDING THE $minTime(\pi)$ CONFIGURATION. THE FIRST COLUMN REPRESENTS THE i VALUE OF EQUATION 9. FOR EACH TEST WE INDICATE WHETHER IT IS A *Miss*, A *Success* OR A *Loss*. FOR LOSSES, WE REPORT THEIR VALUES.

i	BLACKSC.	BODYTRACK	CANNEAL	DEDUP	FACESIM	FERRET	FLUIDAN.	FREQMINE	RAYTRACE	STREAMCL.	SWAPT.	VIPS
0.1	MISS	MISS	MISS	SUCC.	SUCC.	SUCC.	6.98	MISS	MISS	MISS	SUCC.	SUCC.
0.2	MISS	MISS	0.82	MISS	SUCC.	SUCC.	SUCC.	MISS	SUCC.	MISS	SUCC.	SUCC.
0.3	SUCC.	MISS	SUCC.	MISS	MISS	8.23	21.67	MISS	SUCC.	10.63	SUCC.	SUCC.
0.4	SUCC.	MISS	6.47	71.75	SUCC.	SUCC.	11.21	MISS	SUCC.	7.38	SUCC.	SUCC.
0.5	MISS	2.45	SUCC.	60.63	SUCC.	SUCC.	12.65	10.74	SUCC.	11.31	8.78	4.14
0.6	3.99	3.54	4.93	62.15	SUCC.	5.8	6.06	11.34	SUCC.	9.2	7.52	6.25
0.7	SUCC.	6.03	6.96	66.7	SUCC.	5.53	5.32	11.55	4.42	10.52	6.48	8.79
0.8	SUCC.	5.65	SUCC.	68.74	SUCC.	4.65	4.11	11.76	MISS	15.89	11.83	6.82
0.9	MISS	8.21	1.64	21.4	SUCC.	4.75	SUCC.	17.29	6.85	13.18	3.85	6.73
1	4.01	11.41	3.93	10.39	SUCC.	4.34	SUCC.	16.31	SUCC.	8.59	7.83	9.86

time of 20 and a maximum of 220, then we will compute $minPower(40), minPower(60), \ldots, minPower(220)$. Basically, we are slicing the range of execution times in equal intervals. By doing so, we are able to test a wide range of situations and to cover the entire spectrum of execution times, avoiding biases due to specific choices of τ. A similar approach has also been adopted in finding $minTime(\pi)$ configurations.

In the following tests, we will compare the results obtained by our algorithm with those obtained through an ideal and optimal algorithm. The ideal algorithm has the knowledge of all the power consumption and execution times values in all the 312 possible configurations, so it is able to always choose the optimal configuration. On the other hand, our algorithm only knows the values about 4 configurations and try to predict all the other values. The configurations have been produced with the *Halton* generator.

For each test, three different situations may happen:

1) The algorithm chooses a solution that according to the predictions satisfies the requirements but actually it doesn't. We will denote this situation as a MISS. For example, this happens when we want to find the minPower(20) configuration but the algorithm find a configuration with an execution time greater than 20.
2) The algorithm succeeds in finding a configuration that satisfies the requirement. However, the found configuration is worst than the one found by the ideal algorithm. For example, consider the scenario where we need to find the minTime(40) configuration. In this case, our algorithm will find a configuration that satisfies the required power consumption bound but has an execution time higher than that of the configuration found by the ideal algorithm. We will call this situation as a LOSS and we will indicate in our results the percentage of this loss. For example, for $minTime(\pi)$ configurations the loss will be $\frac{(FoundTime - IdealTime)}{IdealTime} * 100$, where $FoundTime$ is the execution time of the configuration found by our algorithm while $IdealTime$ is the execution time of the optimal solution.
3) Our algorithm finds the ideal configuration. We will call this situation a SUCCESS.

In Table II, we show the accuracy of our algorithm with respect to the ideal one when searching for $minPower(\tau)$ configurations. The first column of the table represents the i value of equation 9. We successfully satisfied the requirements in 92.5% of all the tests. In 71.66% of the cases we found the optimal configuration (SUCCESS), while in 20.83% of tests we satisfied the requirements but we didn't found a solution as good as the one found by the ideal algorithm (LOSS). However, those solutions were in average only 5.3% worst than the corresponding optimal solutions. In the remaining 7.5% of the cases, we were not able to satisfy the requirements and the algorithm chosen a configuration with an execution time greater than τ (MISS).

Similarly, in Table III, we show the accuracy of our algorithm in selecting the $minTime(\pi)$ configuration under different π constraints. We found a configuration that respected the requirements in 83.2% of the cases. In 31.6% of the cases we found the same solution found by the ideal algorithm (SUCCESS). In 51.6% of the cases our solution was slightly worse than the optimal one, with an average degradation of 15.46% (LOSS). The remaining 16.6% of cases were those in which we chosen a solution that according to our prediction satisfied the requirement but actually it wasn't (MISS). Although we achieved a good average accuracy, for some applications the results are quite below the average (e.g. for DEDUP). This is caused by the fluctuations and outliers in its scalability behaviour (Figure 1), which let the performance much more difficult to predict. However, this happen only in few cases and to be solved it would require more complex

prediction models, which are outside the scope of this work.

V. Conclusions and future work

In this work we proposed a methodology for the prediction of the performance and power consumption of an application under different control knobs combinations. By exploring few *[Threads, Frequency]* configurations, we were able to predict its behaviour in all the other configurations. To do that we used a multiple linear regression model. This model interpolates the collected data to infer an analytical model that will then be used to perform our predictions. Differently from many existing solutions based on hardware counters readings, we only needed information about execution time and power consumption, thus making our approach more portable. In order to minimise the amount of configurations needed by the interpolation process, we compared different selection strategies. We shown that by picking equidistributed points we can achieve better accuracies compared to the case of a pseudorandom selection of the points. By using this intuition, we have been able to achieve an average prediction accuracy of 96% over the PARSEC applications, by interpolating only the $\sim 1\%$ of the possible configurations. Eventually, we analysed the accuracy of our algorithm in finding the best configurations for trade-offs between power consumption and performances, comparing it with an ideal and optimal algorithm.

As a future work, we would like to extend this approach by considering both hyperthreading and different mappings of the threads on the cores. Moreover, we would like to integrate this prediction algorithm into highly dynamic runtime supports since, thanks to its low overhead, it could be used to reevaluate the optimal configuration with minimum impact on the application performances.

Acknowledgements

I would like to thank Prof. Marco Danelutto and Massimo Torquati for their support and their valuable advices.

This work has been partially supported by EU FP7-ICT-2013-10 project REPARA (No. 609666) "*Reengineering and Enabling Performance And poweR of Applications*", EU H2020-ICT-2014-1 project REPHRASE (No. 644235) "*REfactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach*" and University of Pisa Project PRA_2016_64 "*Through the fog*".

References

[1] Intel, "Reducing data center energy consumption," White Paper.

[2] P. Ranganathan, "Recipe for efficiency: Principles of power-aware computing," *Commun. ACM*, vol. 53, no. 4, pp. 60–67, Apr. 2010.

[3] E. J. Lucente, "The coming "c" change in data centers," 2010, http://www.hpcwire.com/2010/06/15/the_coming_c_change_in_datacenters/.

[4] J. Li and J. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, Feb 2006, pp. 77–87.

[5] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 277–286, Mar. 2008.

[6] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[7] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via os level monitoring," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 116–125.

[8] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 250–259.

[9] R. Cochran, C. Hankendi, A. Coskun, and S. Reda, "Identifying the optimal energy-efficient operating points of parallel workloads," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 608–615.

[10] M. Danelutto, D. D. Sensi, and M. Torquati, "Energy driven adaptivity in stream parallel computations," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, 2015, pp. 103–110.

[11] D. C. Montgomery and E. Peck, *Introduction to linear regression analysis*, ser. Wiley-Interscience Publication. New York: John Wiley & sons, 1992, a Wiley-Interscience publication.

[12] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[13] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, ser. HotPower'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[14] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: Understanding the runtime effects of frequency scaling," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 35–44.

[15] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, Apr 1995.

[16] P. Alonso, M. Dolz, R. Mayo, and E. Quintana-Ort, "Modeling power and energy of the task-parallel cholesky factorization on multicore processors," *Computer Science - Research and Development*, vol. 29, no. 2, pp. 105–112, 2014.

[17] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec 2003.

[18] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, Jan. 2012.

[19] W. J. Morokoff and R. E. Caflisch, "Quasi-random sequences and their discrepancies," *SIAM J. Sci. Comput*, vol. 15, pp. 1251–1279, 1994.

[20] P. Bratley, B. L. Fox, and H. Niederreiter, "Implementation and tests of low-discrepancy sequences," *ACM Trans. Model. Comput. Simul.*, vol. 2, no. 3, pp. 195–213, Jul. 1992.

[21] I. Antonov and V. Saleev, "An economic method of computing lp-sequences," {*USSR*} *Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252 – 256, 1979.

[22] J. Halton, "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numerische Mathematik*, vol. 2, no. 1, pp. 84–90, 1960.

[23] B. Vandewoestyne and R. Cools, "Good permutations for deterministic scrambled halton sequences in terms of l2-discrepancy," *J. Comput. Appl. Math.*, vol. 189, no. 1-2, pp. 341–361, May 2006.