Energy driven adaptivity in stream parallel computations

Marco Danelutto, Daniele De Sensi and Massimo Torquati
Dept. of Computer Science, Univ. of Pisa
Email: daniele.desensi@for.unipi.it {marcod,torquati}@di.unipi.it

***Abstract*—Determining the right amount of resources needed for a given computation is a critical problem. In many cases, computing systems are configured to use an amount of resources to manage high load peaks even though this cause energy waste when the resources are not fully utilised. To avoid this problem, adaptive approaches are used to dynamically increase/decrease computational resources depending on the real needs. A different approach based on *Dynamic Voltage and Frequency Scaling* (DVFS) is emerging as a possible alternative solution to reduce energy consumption of idle CPUs by lowering their frequencies. In this work, we propose to tackle the problem in stream parallel computations by using both the classic adaptivity concepts and the possibility provided by modern CPUs to dynamically change their frequency. We validate our approach showing a real network application that performs Deep Packet Inspection over network traffic. We are able to manage bandwidth changing over time, guaranteeing minimal packet loss during reconfiguration and minimal energy consumption.**

Keywords: parallel design patterns, energy efficiency, dynamic adaptation, DVFS

I. Introduction

In the latest years, increasing attention has been paid on the creation of energy efficient computing system. This interest is motivated both by environmental and economical reasons. In fact has been estimated that, during 2010, the data centers in the US consumed more CO_2 than an entire country like Argentina or Netherlands [1] (and close to the CO_2 consumed by the entire airline industry). Furthermore, energy consumption has also a consistent economical impact. In 2010, for example, the energy consumed in US by data centers reached the 3% of the overall energy production, and this power demand is estimated rapidly growing $10\% - 12\%$ a year [1], [2]. Consequently, energy cost in the near future could easily overtake the cost of the physical system itself. In addition to this, high energy consumption causes high temperature in the system, thus requiring more energy for advanced heat management and cooling systems. This often means that for every dollar spent on electricity, an additional dollar is required for cooling [3].

Moreover, energy efficiency is also needed on smaller scale, for example by mobile systems users where power consumption is one of the main concern. Finding efficient power management methods could lead to longer battery life and better user experience.

According to [3], [4], the average utilisation of many systems is usually around $10\% - 50\%$. This opens many possibilities for energy saving by increasing the average utilisation of these systems. This solution is also supported by manufacturers, which provide new architectural mechanisms to control and adapt the architecture to the real needs, for example by scaling the frequency of the CPUs or by turning off cores, cache or RAM modules, allowing thus to use just the resources really needed.

The main contributions of this paper are:

- A reconfiguration strategy proposal for stream parallel computations. The strategy exploits both the possibility to add or remove computation resources at runtime and the possibility to dynamically change their running frequency.
- A model to always choose the best solution in terms of consumed energy.
- The validation of the strategy on top of a framework for Deep Packet Inspection, thus applying the approach over a class of applications characterised by highly varying rates, showing that this approach has a negligible performance impact.

This paper is structured as follows. In Sec. II we first describe the reconfiguration strategy proposed in this work, then, in Sec. III we outline the framework we modified by adding the automatic reconfiguration strategy. In Sec. IV the results obtained running a real network based application are shown. Eventually, in Sec. VI we draw some conclusions proposing also some possible feature directions.

II. Energy driven adaptivity

In this section, we will explain the approach we used to decide when and how to adapt the system.

A. Stream parallel computations

In this work we concentrate mainly on computations working on streams of data. A *stream* can be informally defined as a sequence (possibly infinite) of data items to compute, all of them having the same type. A streaming application may be seen as a graph (or workflow) of computing modules (sequential or parallels) whose arcs connecting them bring streams of data of different types. The typical requirements of such a complex streaming application is to guarantee a given Quality of Service imposed by the application context. In a nutshell, that means that the modules of the workflow describing the application have to be able to sustain a given throughput.

Stream parallel patterns are those natively operating on streams, notably *pipeline* and *task-farm*. Although the approach shown in this work focuses only on *task-farm* based computations, it can be applied in a similar way to the *pipeline* pattern case as well.

The *pipeline* is typically used to model computations expressed in stages. Its parallel semantics ensures that all stages will be executed in parallel on different input data.

The *task-farm* (sometimes also called master-worker or simply farm) is a stream parallel paradigm based on the replication of a purely functional computation (f). Its parallel semantics ensures that it will process tasks such that the single task latency is close to the time needed to compute the function f sequentially, while the throughput (only under certain conditions) is close to $f\frac{1}{N}$ where N is the number of parallel agents used to execute the farm (called *workers*). The concurrent scheme of a farm is composed of three distinct parts: the *emitter*, the pool of workers and the *collector*. The *emitter* gets farm's input tasks and distributes them to workers using a given scheduling strategy (round-robin, auto-scheduling, user-defined). The *collector* collects tasks from workers and sends them to the farm's output stream. We define the current configuration of a farm as a couple $< \overline{\omega}, \overline{\pi} >$ where $\overline{\omega}$ is the number of workers and $\overline{\pi}$ is the frequency of the cores on which the workers are running. We then define a reconfiguration as a change of the resources from $< \overline{\omega}, \overline{\pi} >$ to a different generic configuration $< \omega, \pi >$.

B. Detection

To check if the system is under-utilised or over-utilised, at regular time intervals, we compute the average utilisation factor of the system defined as:

$$\rho = \frac{T_S}{T_A}$$

where T_S is the average service time of the system (in isolation) and T_A is the average interarrival time of the request to the system. The system will be able to manage all the requests only if $\rho < 1$.

To have a system which is neither over-utilised nor under-utilised we need to keep ρ as much close as possible to 1. Therefore, we want to keep ρ between two bounds ρ_{min} and ρ_{max}, where ρ_{max} is close to 1. The rationale is that, when ρ goes below ρ_{min} the system is under-utilised and we could decrease the resources still being able to manage the same input bandwidth. Similarly, when ρ goes above ρ_{max}, the system starts to become over-utilised and we should increase the resources in order to manage all the requests.

The values of ρ_{min} and ρ_{max} can be specified by the user. The closer ρ_{min} is to ρ_{max}, the more the system will be efficient, but at the same time, it will also incur in an higher number of reconfigurations. If the bandwidth variations are too large, this strategy may lead to a degradation of performance since too many reconfigurations could be performed by the system. In this work, we mitigate this problem by locking a configuration for a certain period of time before considering a possible reconfiguration. However, more sophisticated strategies that model and take into account the cost of the reconfiguration could also be used [5].

C. Choosing the new configuration

We now see how we choose the new configuration when we either detect that the current one is not able to manage the input bandwidth or is not efficient enough.

If the emitter or the collector are over-utilised, we could parallelize them or increase the frequency of the cores where they are mapped. In a similar way, when they are under-utilised we could decrease the frequency to save energy. In this work, for the sake of simplicity, we will never consider the cases in which this situation happens. However, our approach can be easily extended to manage this case too.

Accordingly, the only utilisation factor we will consider, is the one of the set of workers, computed as $\rho_\omega = \frac{\sum_{i=0}^{N} \rho_i}{N}$, where ρ_i is the utilisation factor of a specific worker and N is the number of workers. When $\rho_{\overline{\omega}} < \rho_{min}$, we will decrease the number of workers and/or the frequency of the cores on which they are bound. Similarly, if $\rho_{\overline{\omega}} > \rho_{max}$, we will increase the number of workers and/or the frequency of the cores on which they are bound. With $\rho_{\omega,\pi}$, we denote the utilisation factor of the generic configuration $< \omega, \pi >$ where ω is the number of workers and π the CPU frequency used.

To know if a specific configuration will still be able to manage the input bandwidth, we need to predict the utilisation factor of the system at a configuration different from the current one. Accordingly, we must know how the T_S changes when the frequency and the number of workers changes.

Without loss of generality, for CPU intensive computations, the service time of the set of workers is proportional to their frequency:

$$T_S^{<\overline{\omega},\pi>} = T_S^{<\overline{\omega},\overline{\pi}>} \times \frac{\overline{\pi}}{\pi}$$

The proportionality of the performance to the frequency has also been experimentally shown in [6].

Similarly, when the number of workers changes, the service time of the set of workers will become:

$$T_S^{<\omega,\overline{\pi}>} = T_S^{<\overline{\omega},\overline{\pi}>} \times \frac{\overline{\omega}}{S(\omega)}$$

where $S(\omega)$ is the expected scalability[1] of the application when ω workers are used. For applications characterised by a good scalability, we may consider the ideal case $S(\omega) = \omega$.

It follows that, in general, if both the number of workers and the frequency change:

$$T_S^{<\omega,\pi>} = T_S^{<\overline{\omega},\overline{\pi}>} \times \frac{\overline{\omega} \times \overline{\pi}}{\omega \times \pi}$$

therefore, given a fixed T_A, we have:

$$\rho_{\omega,\pi} = \rho_{\overline{\omega},\overline{\pi}} \times \frac{\overline{\omega} \times \overline{\pi}}{\omega \times \pi} \tag{1}$$

Among all the possible configurations, we are only interested in those $< \omega, \pi >$ such that $\rho_{min} < \rho_{\omega,\pi} < \rho_{max}$. It's important to notice that, since $\rho_{max} < 1$ and since we are assuming that emitter and collector are never over-utilised, all these configurations are able to sustain the input bandwidth.

[1] Defined as $S(\omega) = \frac{T_S^{<1,\pi>}}{T_S^{<\omega,\pi>}}$

Above this restricted set of configurations, we want to find the one which minimises the amount of energy consumed.

To find the configuration which consumes the lowest amount of energy, we need first to understand how the energy changes when we change the frequencies and/or the number of workers of the farm. We are not interested in knowing the exact amount of energy consumed but only a proportional estimation such that we can compare two different configurations between each other. According to [7], the power consumption can be estimated as:

$$P \propto (\pi \times \gamma^2)$$

where π is the operating frequency and γ is the supply voltage. The voltage depends on the frequency of the processor and can be automatically changed by the system when the frequency changes.

This implies that, at a given configuration $< \omega, \pi >$, the energy consumption will be:

$$P \propto (\pi \times \gamma^2 \times \omega) \tag{2}$$

Accordingly, from the restricted set of configurations, we will pick the one such that $\pi \times \gamma^2 \times \omega$ is minimum.

Putting all the pieces together we obtain the following algorithm to choose the new configuration of the system:

for all ω **do**
  for all π **do**
    $\rho_{\omega,\pi} = \rho_{\overline{\omega},\overline{\pi}} \times \frac{\overline{\omega} \times \overline{\pi}}{\omega \times \pi}$
    if $\rho_{min} < \rho_{\omega,\pi} < \rho_{max}$ **and** $\pi \times \gamma^2 \times \omega < min$ **then**
      $min = \pi \times \gamma^2 \times \omega$
      nextconf = $< \omega, \pi >$
    end if
  end for
end for
return nextconf

Fig. 1. Algorithm for the selection of a new configuration.

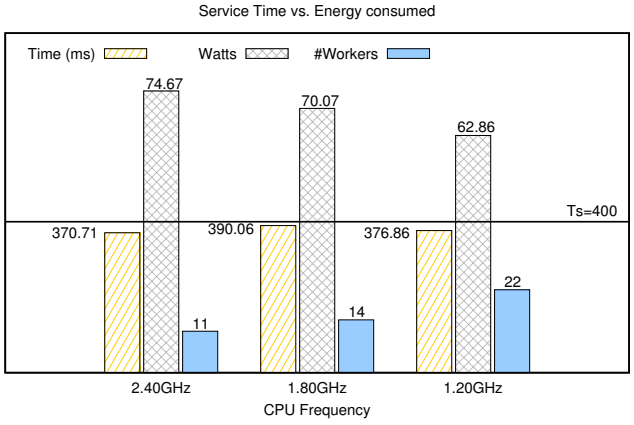

Fig. 2. Service Time vs Energy (in Watts) vs Parallelism degree (Workers) for the execution of the standard *ijk* matrix multiplication algorithm for a single square matrices of size 1024×1024 (double precision elements) on a dual Xeon E5-2695 v2 @2.40GHz platform, varying the CPU frequency.

From the previous reasoning it follows that, since the power is dominated by γ^2 and since it depends on the frequency, in many cases it could be more energy efficient to run an high number of workers at lower frequencies instead of a minor number of workers running at higher frequencies. As an example, consider the case in which a matrix multiplication algorithm has to be computed for a number of input matrices. Suppose also that the requirement is to satisfy a given service time value T_S for the computation of each single input. In Fig. 2 is sketched the results obtained for this case when the input matrices are of size 1024×1024 and the target service time is $T_S = 400ms$. In the graph we plotted the average execution time (in milliseconds) for computing one matrix, the energy consumed during the execution and the minimum number of workers that allows to satisfy the requirement. It can be seen that by using 22 worker threads (the platform considered in this test has 24 cores) and a CPU frequency of 1.2GHz (the lowest possible for the considered platform) it is possible to satisfy the given T_S minimising also the energy consumed for computing each single output matrix.

TABLE I. NOTATION USED

Symbol	Meaning
ρ	Utilisation
ρ_{min}/ρ_{max}	Minimum/Maximum allowed utilisation
ω	Number of workers
π	Frequency
$\rho_{\omega,\pi}$	Utilisation at configuration $< \omega, \pi >$
$S(\omega)$	Speedup with ω workers
γ	Voltage

III. APPLICATION

In this section, we describe how we applied the concepts shown in Sec. II to PEAFOWL, a framework which allows to write parallel Deep Packet Inspection (DPI) applications [8] with relatively low programming effort.

We first introduce PEAFOWL, showing its peculiarities and describing the application we used for our experiments. Then, we briefly describe FASTFLOW, the framework used by PEAFOWL to manage the parallel execution. Eventually, we provide some details on how we implemented the reconfiguration in PEAFOWL, thanks to the mechanisms provided by FASTFLOW.

A. The PEAFOWL DPI framework

PEAFOWL is a flexible and extensible DPI framework which can be used to identify the application protocols carried by network packets and to extract and process data and metadata carried by those protocols. Conceptually, PEAFOWL is structured as a task-farm, where the emitter reads the packets and dispatches them to the workers. Each worker identifies the protocol and processes the content consequently. After that, the packet and the result of the processing are forwarded to the collector. Eventually, the collector decides whether to drop or forward the packet or to perform any other kind of operation. The application programmer, by providing callback functions, has the possibility to customise the code executed by the emitter, the workers and the collector, providing thus the possibility to write many different DPI applications and to use different technologies to send and receive packets to and from the network (see Fig. 3).

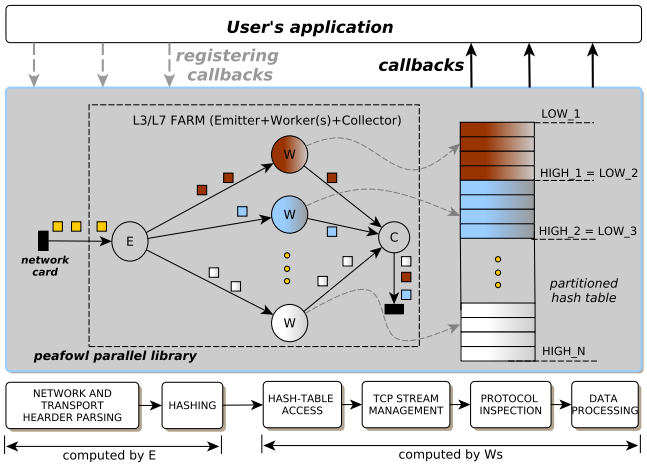

Fig. 3. The architecture model of the PEAFOWL framework.

In order to correctly process network packets, PEAFOWL divides the packets in so called "flows" (groups of packets with same source and destination). These flows are stored into an hash table, so that PEAFOWL can correlate the packets with the previous information collected for the same flow. The flows and the hash table are partitioned among the set of workers, in such a way that each of them will only manage a contiguous partition of the table. In this way, each worker accesses only to its partition without any need of synchronisation with the other workers. However, this requires the emitter to send to each worker only the packets belonging to flows it can manage. This is done by a simple hash function like $h(x) = p(x)\ mod\ N$, where $p(x)$ is a function computed over some data contained in the packet header and N is the current number of workers. By using PEAFOWL, we had the possibility to easily implement an application that analyses the HTTP packets travelling over the network, searching for well known patterns identifying possible security threats. As shown in Sec. IV, we managed to validate our approach on an application which resembles a real case, by inspecting traffic received at a variable rate, requiring thus different computation capabilities over a period of time [8].

FASTFLOW*:* The non functional part of PEAFOWL concerning the parallel execution and the reconfiguration is implemented using the FASTFLOW parallel framework, a stream parallel programming framework providing the application programmer with customizable and efficient parallel patterns for shared memory multi-core platforms [9]. FASTFLOW provides different, fully customizable and composable patterns including a *task-farm* skeleton, with an arbitrary number of "Worker" threads, each one independently executing tasks appearing on the input stream. Moreover, FASTFLOW provides the possibility to "pause" the farm and dynamically add or remove its workers. Communication channels between threads are implemented using lock-free Single-Producer Single-Consumer FIFO queues [10], with messages carrying data pointers rather than plain data copies.

B. Dynamic reconfiguration

When implementing the model described in Sec. II, an important problem to solve is the computation of the average service time in isolation (T_S). In fact, in our application T_S is influenced by many factors, like the percentage of traffic composed by HTTP packets, the length of the packets, the percentage of packet re-transmitted after losses and others. Since these factors may significantly change during the execution, it would be unfeasible to compute in advance or to predict the service time of an isolated worker. For this reason, instead of computing ρ as the rate between the average service time in isolation and the interarrival time, we compute it as the percentage of time spent processing tasks:

$$\rho = \frac{T_{work}}{T_{tot}}$$

When $T_S < T_A$ this new formulation is equivalent to the one described in Section II and since in our case we always keep $T_S < T_A$, we can always use this formulation. Every T_{check} seconds, PEAFOWL checks the value of ρ over the last T_{tot} seconds ($T_{check} < T_{tot}$) and, as soon as ρ goes out of the bounds, a reconfiguration is performed. The values of T_{check} and T_{tot} can be specified by the user as parameters. The smaller T_{tot} is, the more ρ will be influenced by anomalous spikes in the rate, causing in some situations useless reconfigurations. The bigger it is, the less reconfigurations are executed.

When a change in the number of worker is needed, PEAFOWL temporary halts the farm, stops or starts some of the workers and then changes the partition of the hash table to reflect the new number of workers. To efficiently change the partitions of the table, instead of having one separated sub table for each worker, we have a shared table partitioned by means of two indexes LOW_i and $HIGH_i$ (see Fig. 3), such that the worker i will only access the portion of the table between these two indexes. When the partitions has to be changed, we can simply change the value of the indexes. After that, the emitter is updated by changing the value of N in the hash function it computes to distribute the packets to the workers. Eventually, PEAFOWL starts again the farm.

Concerning the frequencies, PEAFOWL changes them by calling the *cpufreq-set*[2] command, available on most Linux distribution. This does not require to halt the farm thus, in principle, it does not cause any packet loss. Accordingly, the possibility of changing the frequencies should reduce the number of situations in which a change in the number of workers is needed, thus reducing the losses due to the pauses of the framework. To be sure that we are changing the frequency only on the processors on which the workers run, each element of the farm will be pinned to a specific processor by changing its affinity. Beside the possibility of changing the frequencies, *cpufreq-set* provides the opportunity to specify a so-called *governor*, i.e. a CPU frequency scaling algorithm which will automatically decide which frequency to use according to the current load on the CPU. Since we have the possibility to delegate the frequencies management to the *governors*, we implemented and tested the following strategies:

Power focused*:* Is the main proposal of this paper. We described it in Section II and it tries to minimise the consumed energy. It uses the *userspace* governor, which delegates to the application the task of changing the frequencies.

[2]we considered Linux-based OS

***Cores focused**:* As in *Power Focused* policy, we can reconfigure both the number of workers and the frequency, but we always choose the configuration which minimises the number of workers.

***CPUfreq on demand**:* We can only choose the number of workers while the frequencies will be chosen automatically by the *on-demand* governor. This governor changes the frequencies based on the current CPU load. The idea is that the governor will change the frequencies to adapt the system when light changes happen in the input rate. If this is not sufficient, the framework will still find the utilisation factor outside of the specified bounds and will increase or decrease the number of workers.

***CPUfreq conservative**:* Similar to *CPUfreq on demand* but using the *conservative* governor. This governor differs in behaviour from *on demand* governor since it increases and decreases the frequency more gracefully.

***No frequency**:* We can only reconfigure the number of workers, the CPUs will always run at maximum frequency.

In the next section, we compare the different aspects of these strategies.

IV. Results

In this section, we validate our approach by showing the results we obtained running our reconfiguration strategy in a real application.

The platform used for the experiments is a NUMA workstation having two Intel Xeon E5-2650 @ 2.00GHz nodes with a total of 16 cores (2-way hyperthreading). Each NUMA node has 16GB of main memory, 20MB of shared L3 cache, 256KB and 32KB of core private L2 and L1 caches, respectively. The machine provides the possibility to use DVFS, with frequencies from 1.20GHz to 2.00GHz with steps of 0.10GHz and only by changing the frequencies of all the cores of a socket at the same time (chip-wide).

To test our application in a realistic situation, we took the rates of the traffic travelling on a backbone link of a Tier1 ISP between San Jose, CA and Los Angeles, CA [11]. These rates have a resolution of one second, cover a 1-hour duration and we used them to send some synthetic traffic to our application. Originally the application was written to receive data from the network. Unfortunately, on our machine we can only change the frequency of all the processors on the NUMA node at once. This implies that when we reduce the frequency of the workers we also reduce the frequency of the emitter, which is reading the packets from the network, causing then an increase in its service time with consequent packet losses. To avoid this situation, we could reserve a NUMA node for the execution of the emitter and the collector, keeping its frequency always at maximum. However, on our machine, this implies that we would have only 8 cores available for the execution of the workers, thus not allowing us to validate our strategy at its full potential. For this reason in our tests, instead of reading the packets from the network, we will read them from the main memory. Using this solution, we found that the emitter is able to manage the required rate also when its frequency changes, thus allowing the application to use all the 16 cores provided by the machine. At the same time, this solution is functionally equivalent to the one that reads the packets from the network [8]. Furthermore, some machines already provide the possibility to individually change the frequency of each core (e.g. IBM Power8 processor [12]). Using such kind of machine, we would have been able to test our application by reading the packets from the network and having at the same time the possibility to run the emitter at the right frequency without thus impairing the overall performances.

A. Model assessment

First of all, we checked the correctness of the assumptions we made about the trend of the utilisation factor and the energy when the number of workers and the frequency change.

To check the accuracy in the prediction of ρ (equation 1) we ran our test application and, when a reconfiguration was required, we took the error between the utilisation factor $\rho_{\omega,\pi}$ predicted in configuration $< \overline{\omega}, \overline{\pi} >$ and the real utilisation factor after moving to configuration $< \omega, \pi >$. This error is shown in Fig. 4, where on the labels we have the destination configuration $< \omega, \pi >$.

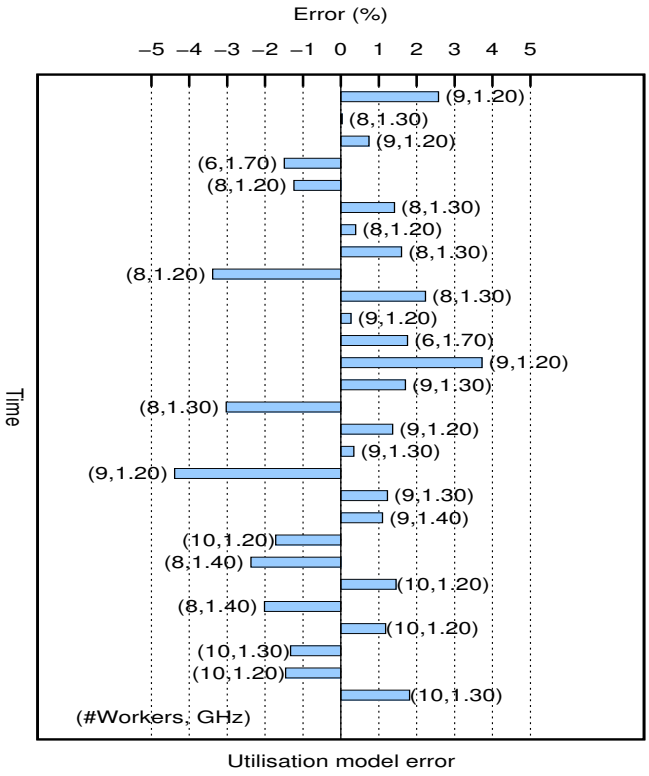

Fig. 4. Absolute error (in percentage) of the utilisation model proposed.

The model demonstrated an error $< 4.5\%$, allowing thus to predict quite accurately the utilisation factor of the application at a configuration different from the current one. This error accounts both the fact that the scalability at n workers is not exactly n, and the fact that the service time is not entirely depending on the CPU, slightly reducing the benefit of frequency changes. As future work, the model could be improved to take into account both these considerations.

To compute the consumed energy, we used RAPL energy sensors (*Running Average Power Limit*), available in Intel

Sandy Bridge architectures. RAPL sensors provide the possibility to read the energy consumed by an entire CPU package, by the processors cores only or by the memory controller. As shown in [13], these counters are quite accurate also for fine-grained energy measurements. To check the consistence of the model, we sent data to our application at maximum rate, disabling reconfiguration capabilities and taking the average Watts consumed in one hour execution for each configuration $< \omega, \pi >$. Since we only have the possibility to read the energy chip-wide, we first computed the energy of the system when the application was not running, and then we subtracted this quantity to the result, in order to remove from the result the energy consumed by cores not used by the application.

Concerning the model, we know that power consumption is proportional to $\omega \times \pi \times \gamma^2$ (equation 2). However, voltage in turn depends on the operating frequency. If frequency is directly proportional to supply voltage, the previous relation predicts cubic power reduction when considering frequency, but this applies only within a narrow, process specific, supply voltage range. In real situations, according to [14], the frequency may have a sublinear proportion to voltage.

On our machine, we experimentally found that the power is proportional to $\omega \times \pi^{1.3}$. In Fig. 5 we plot on the X-axis all possible configurations for the target platform varying both the frequency and the number of workers. For readability, for each ω, only the first label $< \omega, \pi >$ is shown (the range is $[1.20 - 2.40]$GHz with steps of 0.10GHz).

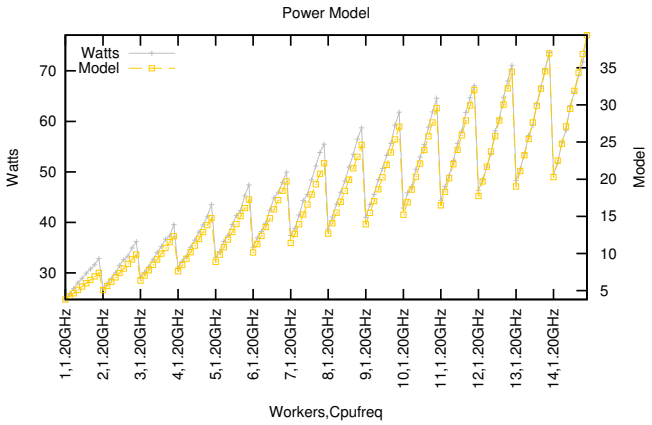

Fig. 5. Comparison between the real power consumed vs the power predicted by the model.

On the left Y-axis we have the average Watts consumed by the application at a specific configuration and on the right Y-axis the result of our estimation at the same configuration. As we can see, the model proportionally fits the real consumed energy, thus allowing the framework to take correct decisions in order to minimise the energy consumed.

B. Tests

To validate the proposed model, first we have to ensure that the system is able to guarantee an high utilisation independently from the rate of the data arriving to the application. This means that PEAFOWL must be able to keep the utilisation factor in the range $[\rho_{min}, \rho_{max}]$.

For our experiments, we set $\rho_{min} = 80\%$ and $\rho_{max} = 90\%$. In Fig. 6-D we can see that the strategy proposed is able to keep the utilisation factor within the selected range. This is possible because the configuration of the application is changed according to the input rate, as shown in Fig. 6-A. The step function in Fig. 6-A, represents the product between the number of activated workers and the operating frequency. As described in Sec. II this is proportional to the bandwidth the application is able to sustain and this is proved by the similar behaviour of the two functions.

Similarly, in Fig. 6-C, we show how the energy consumption changes to fit time by time the rate received by the application and how it is mainly influenced by the operating frequency (Fig. 6-B).

When a change in the number of workers is needed, PEAFOWL waits for all the workers to finish to process the already enqueued tasks before changing the partitions and starting again the workers. During this period, no packets are read, thus causing some losses. We experimentally found that on our test machine this correspond to an average time less than $10\,ms$ causing, at this rate, $\sim 3\% - 4\%$ losses.

This is required because we must ensure that the packets received before the reconfiguration are processed according to old partitions and the packets received after the reconfiguration are processed according to new partitions. Moreover we must ensure that all the workers, at each time, have the same image of the partitions. Otherwise, we could have inconsistent situations in which a worker needs to process a packet that, after the re-partitioning, belongs to a network flow managed by a different worker. However, this could be optimised by keep reading the packets and enqueuing them according to the new partitions. This would require the worker to distinguish the packets received before and after the re-partitioning (for example by putting a mark in the stream) and proper synchronisation mechanisms between the workers (i.e. a barrier) to change the partitions only when they all receive the first packet after the mark. It is important to notice that losses happen only when the number of workers is changed, while no losses are experienced when only the frequency changes.

We then compared the behaviour of our approach (*power focused*) with the other strategies proposed in Sec. III. For the sake of readability, instead of showing the plots for each strategy, we present the results in table II.

TABLE II. STRATEGIES COMPARISON.

Strategy	Avg. Watts	Avg. Workers	Avg. Utilisation
Power focused	40.25	8.77	84.61%
Cores focused	45.62	5.96	84.84%
No frequency	48.38	5.99	80.10%
CPUFreq Conservative	51.19	5.99	80.24%
CPUFreq On demand	50.98	6.02	80.08%

The first thing to point out, is that *power focused* strategy is the one that performs better in terms of consumed energy while also being the one that uses the higher number of workers. As also shown in Fig. 2, this effect is caused by the possibility to run these workers at lower frequencies and achieving the same results that could be obtained by running a minor number of workers at higher frequencies, and thus consuming more energy. With the strategy proposed, we are

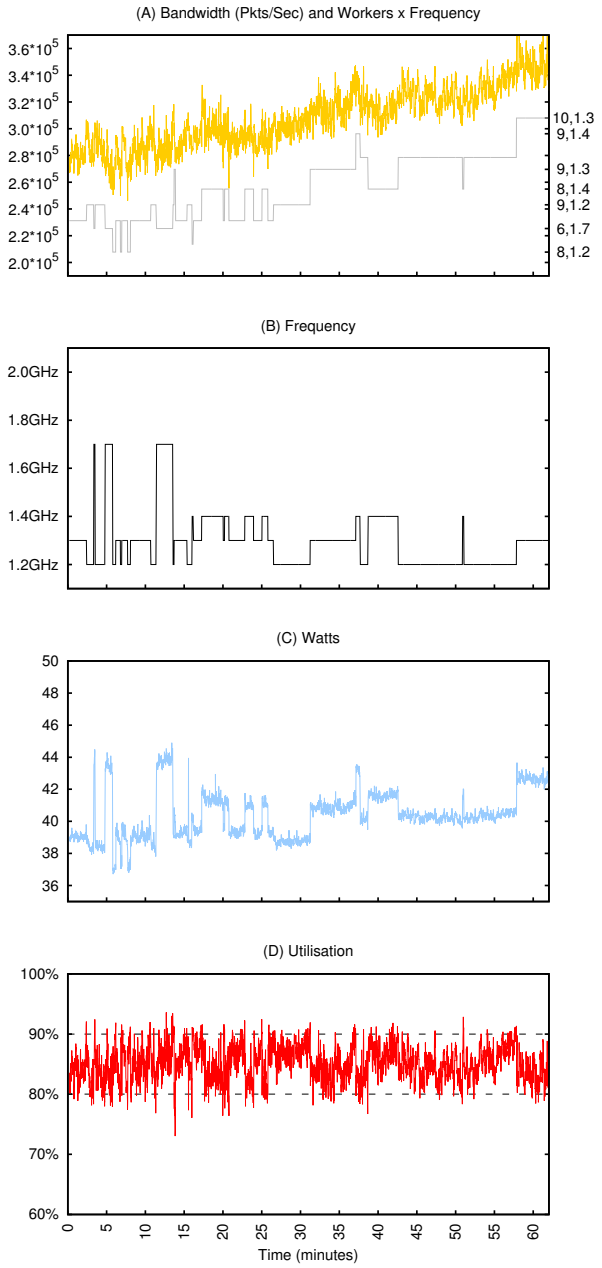

Fig. 6. Comparison between processed bandwidth and current system configuration (A). Frequency behaviour (B). Consumed power (C). Utilisation of the application (D). Values have been sampled once per second. Tics on the time axis represent 5 minutes intervals.

able to save up to 12% of power with respect to the strategy where we can change the frequency but we always try to minimise the number of workers and up to 17% of the energy with respect to a strategy where frequency is not considered at all (*no frequency*). The reason why *no frequency* strategy consumes more energy than *power focused* can be explained by analysing the utilisation factors. When using *no frequency*, the utilisation factor is lower and, as shown in Fig. 7, it is characterised for most of the time by an utilisation $< \rho_{min}$, due to the impossibility to find a configuration characterised by a $\rho_{\omega,\pi}$ such that $\rho_{min} < \rho_{\omega,\pi} < \rho_{max}$. Accordingly, PEAFOWL has to choose a sub optimal configuration which ensures the required performances but wastes energy. Regarding *CPUFreq* strategies, we can see that they behave very similar to *no frequency*. This is most probably due to the mechanism used by FASTFLOW to manage the situations when a worker finds no task in the input queue. In this case indeed, in order to achieve high performances, the worker keep actively polling the queue waiting for a task. This will cause an high CPU utilisation, and thus the *CPUFreq* governor will always maintain the highest frequency. This problem is not present in our internal strategies since we do not check the CPU utilisation but the real utilisation of the application by mean of probes in the framework.

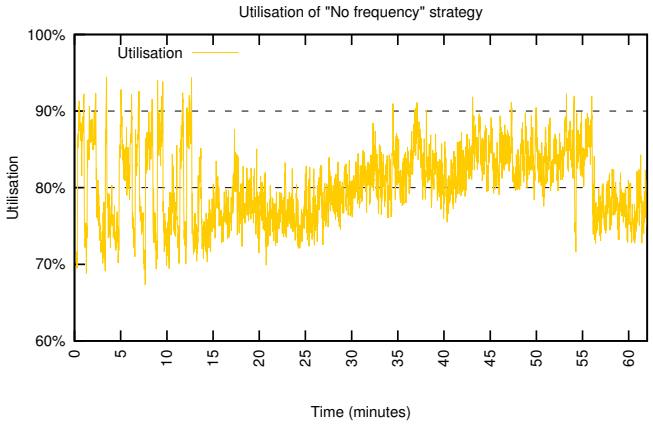

Fig. 7. Utilisation of *no frequency* strategy. Values have been sampled once per second. Tics on the time axis represent 5 minutes intervals.

V. RELATED WORK

We analyse in this section some research works addressing the problem of power and performance efficiency in parallel computations. There has been relatively little work on the interactions between processes/threads variations and energy-efficiency.

Li and Martinez [15] optimise parallel workloads running on multi-core systems by dynamically changing the number of active processors and the voltage and frequency levels at which the system runs. As in our work, they apply DVFS chip-wide and consider the problem of maximising power savings while delivering a given level of performance. Differently from our work, they do not provide a prediction model on which the dynamic re-configuration is based; they propose simple heuristics that can be used to cut down on the search effort along both dimensions of the optimisation problem.

In [16] the authors consider the case in which the voltage and frequency levels are changed independently in each core. They examine different DVFS policies for high performance and power efficiency. Their solutions are primarily based on the exhaustive search of the solution space.

Teodorescu et al. [17] develop an optimisation algorithm based on linear programming to provide power management for a chip-multi-processor (CMP) based on both DVFS and thread mapping. The work is based on the assumption that

power consumption of a CMP at each DVFS level can be estimated accurately. This assumption is effective only when the system is running workloads that are the same or very similar with the one used to do power estimation.

Other research works, focusing only on dynamic reconfiguration and performance optimisation in distributed parallel computations are those related to the ASSIST run-time [18], [19] and to the Behavioural Skeleton approach [20], [21].

VI. Conclusion and future work

In this work we presented a novel approach for reconfiguration of stream-based parallel applications structured as task-farm. This approach tries to minimise the amount of consumed energy by operating both on the number of replicas (i.e. the number of workers ω) and on their frequency (π). Among all the possible configurations ($< \omega, \pi >$), we always choose the one which our model estimates will consume less energy while being able to respect the required QoS. To validate the proposed model, we applied it to a Deep Packet Inspection (DPI) application which analyses network traffic received at variable rates. We shown that the DPI application is able to maintain an high utilisation factor of the system by reconfiguring itself in a way it always uses only the needed resources. Consequently, the power consumed by the application is proportional to the input rate. We also compared our strategy with other possible strategies that can be used to reconfigure the application, showing that the one we proposed performs better in terms of consumed energy.

This work can be further extended in different ways: by applying the proposed approach to different applications or generalising the model for the cases where is possible to individually change the frequencies of the single core. Indeed, we always considered the average utilisation factors of the entire set of task-farm workers. This is a good measurement in all the cases in which the workload is not very unbalanced. However, for highly unbalanced task-farm computations, we could check the individual utilisation factors ρ_i of each worker and take actions locally, for example by increasing or decreasing only their frequency. Furthermore, it would be interesting to analyse how *hyperthreading* could interact with our approach. For example, before removing a worker, we could move it on the same core of another underutilised worker. Alternatively, we could use hyperthreading to benefit from the effects of DVFS also for computations that are not CPU intensive by running more workers on the same core.

Acknowledgment

This work has been partially supported by the EU FP7 project REPARA (ICT-609666).

References

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

[2] P. Kurp, "Green computing," *Commun. ACM*, vol. 51, no. 10, pp. 11–13, Oct. 2008.

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

[4] L. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec 2007.

[5] N. M. Asghari, M. Mandjes, and A. Walid, "Energy-efficient scheduling in multi-core servers," *Comput. Netw.*, vol. 59, pp. 33–43, Feb. 2014.

[6] 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.

[7] Intel, "Enhanced intel speedstep technology for the intel pentium m processor," 2004, ftp://download.intel.com/design/network/papers/30117401.pdf.

[8] M. Danelutto, L. Deri, D. D. Sensi, and M. Torquati, "Deep packet inspection on commodity hardware using fastflow," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE) (Proc. of PARCO 2013, Munich, Germany)*, ser. Advances in Parallel Computing, M. B. et al., Ed., vol. 25. Munich, Germany: IOS Press, 2014, pp. 92 – 99.

[9] *FastFlow website*, 20014, http://mc-fastflow.sourceforge.net/.

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, ser. LNCS, vol. 7484. Rhodes Island, Greece: Springer, Aug. 2012, pp. 662–673.

[11] T. C. UCSD, " Anonymized Internet Traces 2012 Dataset - 15/11/2012 - 13.00 to 14.00 ," 2012, http://www.caida.org/data/passive/passive_2012_dataset.xml.

[12] IBM, "Performance Optimization and Tuning Techniques for IBM Processors, including IBM Power8," 2014, http://www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf.

[13] 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. [Online]. Available: http://doi.acm.org/10.1145/2425248.2425252

[14] W. C. Athas, L. Youngs, and A. Reinhart, "Compact models for estimating microprocessor frequency and power," in *Proceedings of the 2002 International Symposium on Low Power Electronics and Design, 2002, Monterey, California, USA, August 12-14, 2002*, 2002, pp. 313–318.

[15] J. Li and J. F. Martínez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 11-15, 2006*, 2006, pp. 77–87.

[16] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 347–358.

[17] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 363–374.

[18] M. Vanneschi and L. Veraldi, "Dynamicity in distributed applications: issues, problems and the ASSIST approach," *Parallel Computing*, vol. 33, no. 12, pp. 822–845, 2007.

[19] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo, "Dynamic reconfiguration of grid-aware applications in ASSIST," in *Proc. of 11th Intl. Euro-Par 2005 Parallel Processing*, ser. LNCS, J. C. Cunha and P. D. Medeiros, Eds., vol. 3648. Springer, Aug. 2005, pp. 771–781.

[20] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto, "Behavioural skeletons for component autonomic management on grids," in *Making Grids Work*, ser. CoreGRID, M. Danelutto, P. Frangopoulou, and V. Getov, Eds. Springer, Aug. 2008, ch. Component Programming Models, pp. 3–16.

[21] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick, "Behavioural skeletons in GCM: autonomic management of grid components," in *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, D. E. Baz, J. Bourgeois, and F. Spies, Eds. Toulouse, France: IEEE, Feb. 2008, pp. 54–63.