FoREST-mn: Runtime DVFS Beyond Communication Slack

Jean-Philippe Halimi, Benoît Pradelle, Amina Guermouche, William Jalby
Université de Saint-Quentin-en-Yvelines, France
first.last@uvsq.fr

***Abstract*—Dynamic Voltage and Frequency Scaling (DVFS) is commonly used to save energy in computing systems. However, when it comes to parallel programs, existing DVFS controllers only reduce frequency while or before waiting in blocking communications. As a consequence, energy savings are only possible for the program tasks out of the critical path and when the workload is imbalanced.**

We propose a new runtime DVFS controller, FoREST-mn. It allows to take advantage of both the low CPU usage of some program phases as well as communication slack to save more energy with parallel programs. The DVFS control then becomes more complex, but energy savings are even obtained when the workload is balanced. The resulting slowdown on programs is carefully controlled and constrained by a user-defined threshold. We implemented the presented strategies and evaluated it on 4 compute nodes totaling 64 cores. FoREST-mn is able to perform significant CPU energy savings on the NAS programs, up to 34 % on MG, while efficiently bounding the resulting slowdown.

***Keywords*-DVFS, communication slack, energy saving.**

I. Introduction

Energy consumption of supercomputers regularly increased in the past to now reach more than 17 MW for the most powerful computers. In that context, energy savings become crucial and even the smallest amount of energy saved on a computer can imply a cost reduction of several thousands of dollars per year when considering a whole datacenter.

Dynamic Voltage and Frequency Scaling (DVFS) was shown to be efficient at reducing processors energy consumption. It enables processors to work at a lower frequency and voltage therefore decreasing their power consumption. However, a lower frequency of the processor frequency may increase the execution time of the workloads in most cases. Moreover, when a distributed program runs, a slowdown may propagate to other nodes due to communications and synchronizations. Such slowdown propagation leads to uncontrolled consequences on both execution time and energy consumption [4].

Because slowdown propagation is hard to overcome, existing inter-node controllers do not allow performance degradation. They rather decrease the frequency during communications or during the computations preceding idleness in blocking communications. Unlike existing work, we consider that the actual problem to solve is to maximize energy savings while allowing a controlled slowdown. Indeed, a small and controlled slowdown is tolerable in HPC systems when large amounts of money can be saved. The problem was already addressed in intra-node DVFS controllers [6], [8], [9] but remains open when considering several DVFS-capable units. Thus, we propose to extend and to adapt the principles used for intra-node DVFS with programs running over multiple DVFS-capable units. Moreover, our system is specifically designed as a runtime system and explicitly supports processors which require a unique frequency for all cores.

In the following paper, we present a novel inter-node DVFS control mechanism named *FoREST-mn* where: 1) Parallel programs can be delayed up to a user-defined threshold, enabling energy savings with well-balanced programs. Slowdown propagation is precisely controlled to avoid large energy overheads, 2) Decisions are taken at runtime. The program phases are discovered and predicted as the program runs with no prior knowledge and 3) Processors which require a unique frequency for all cores.

After a brief overview of the technique in Section II, the algorithm steps are presented in Sections III, IV and V. We show how our system enhances the existing DVFS controllers in Section VI and how it deals with realistic constraints in Section VII. Finally, FoREST-mn was evaluated through experiments presented in Section VIII, before comparing it to the state of the art in Section IX, and concluding in Section X.

II. General Overview

A. Definitions and Assumptions

The presented system is based on assumptions commonly made by runtime DVFS controllers. We target Single Program Multiple Data (SPMD) programs running on a distributed memory system. Within programs, communications are performed using MPI or any other similar message passing middleware. terminology, every

As in previous related work, we consider iterative programs that are common in HPC. Typically, such iterative programs are running a similar computation for several time steps. During one iteration, a process executes several *tasks* defined as a computation phase between two communications. In the case of a blocking reception, the task is actually made of a computation phase, followed by a slack period that precedes the communication, the slack being the time spent awaiting a message. As the *slack* does not participate in the program progression, computations can be slowed down to overlap the slack time with no consequence on the program execution time.

The tasks running during an iteration form the vertices of a DAG that we call *task graph*. It is assumed that a task graph representing an iteration remains valid for other iterations.

B. Algorithm

Based on the earlier concepts introduced, each process is individually controlled to determine the characteristics of the running program, select a suitable frequency for each task, and finally apply the frequency schedule. Successive steps are realized as follows:

1) The task graph is built during the first iteration. Simultaneously, the execution time at the highest frequency is measured for every task.
2) Before each following iteration, the frequency is decreased in order to measure the execution time of the tasks at each frequency. Execution times are then combined to power information to determine the energy saved at each frequency for every task.
3) Considering the estimated energy savings, a *locally optimal* frequency is selected for every task. Such frequency minimizes the energy consumed by the task itself, ignoring the consequences on the other tasks. The next iteration is executed while setting the locally optimal frequency of every task.
4) When all the tasks run with their locally optimal frequency, their execution time may increase, creating additional slack after other tasks. That slack is compensated by speeding up the tasks emitting the messages which generate slack.
5) The remaining iterations run with the resulting frequency schedule, as long as the loop behavior remains stable.

Section III details how the program is profiled under various frequencies, Section IV explains how the locally optimal frequency is chosen, and finally, Section V presents how the remaining slack is corrected.

III. Profiling

A. Task Grapho Construction

A frequency must be set at the beginning of the tasks and, for that reason, the DVFS controller must be able to predict the characteristics of the next task before it starts. As in other runtime DVFS controllers with similar goals [12], [17], we chose to exploit the iterative nature of typical HPC programs. With such assumption, the tasks observed at one iteration can be used to predict the tasks during the next iterations.

Programs can be made of several main loops. To identify them more easily, users must insert a function call to our backend at the main loops head and exit. Note that the requirement is not inherent to our approach as loop detection can be achieved automatically [17], but it simplifies the implementation. The first iteration of the instrumented loops are run while observing the communications. The task graph constructed at the end of the first iteration is based on the hijacked communications. In the next iterations, the task characteristics are regularly checked and the whole process is restarted if the tasks do not match their predicted behavior anymore.

The graph is built while the maximal frequency is set on all the processes. In this particular configuration, the measured slack time is called *initial slack*. The task graph is then used in the following iterations to better control voltages and frequencies.

B. Energy Profiling

Once the task graph is known, we need to determine how the various frequencies impact the energy consumption of each task. However, the available energy probes are insufficiently accurate to allow energy measurements at the task granularity [1]. Thus, we decompose the energy gains into a speedup and a power gain between two frequencies $f1$ and $f2$. Energy gains can then be expressed by multiplying both ratios : $\frac{e_{f1}}{e_{f2}} = \frac{P_{f1}}{P_{f2}} \times \frac{t_{f1}}{t_{f2}}$

Tasks execution times t_f are measured while running several consecutive profiling iterations with decreasing frequencies. Profiling process is repeated one iteration per frequency.

Power consumption P_f can often not be directly measured as the tasks execution time is insufficient for the existing power probes. However, as in earlier work [8], ratios of power consumption at different frequencies P_{f1}/P_{f2} are assumed to be program-independent. Thus, an offline power measurement is performed once, typically when installing FoREST-mn, while running an arbitrary benchmark program with every frequency setting. Then, the ratios P_f/P_{fmax} of measured power consumption at the maximal frequency $fmax$ and at any frequency f are considered to hold for any program task.

The energy ratios of a given task, obtained from time and power ratios, estimate the percentage of energy that can be saved with every frequency compared to the maximal one.

IV. Locally optimal frequency

Any task, considered in isolation, can be run with a frequency setting f_{comp} that minimizes the energy consumed during its computations, ignoring the slack. Such frequency can be immediately deduced from the energy ratios described above. However, we allow the user to specify a maximal tolerated slowdown. Note that the user-defined slowdown is considered as an upper bound to the slowdown; not as a target. Thus, we restrict the set of considered frequencies to those provoking less slowdown than the user-chosen threshold. We use for that purpose the execution time ratios measured earlier.

When a task ends with a message reception, some time may be spent waiting for messages. When such slack time can be avoided by slowing down the computation, it is always preferable to do so [11]. Thus, typical inter-node DVFS aims at selecting the frequency $f_{noslack}$ that leads the task computations to end exactly when the message is received. If we consider that the complete task execution time should not be extended, $f_{noslack}$ is then optimal [17]. FoREST-mn rather selects a locally optimal frequency $f_{locOpt} = min(f_{comp}, f_{noslack})$. Indeed, the slack at the end of the tasks must always be overlapped by computations. However, if f_{comp} is low enough to allow slack to be overlapped, it must be preferred as it also minimizes the energy consumed by the computation. Thus, f_{locOpt} is guaranteed to minimize the energy consumption of the considered task.

The locally optimal frequency f_{locOpt} is determined for every task in the task graph as soon as the energy profiling is done. Then, during a single iteration, the tasks all run using their locally optimal frequency.

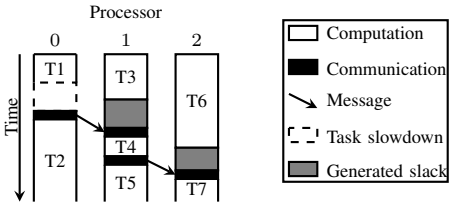

Fig. 1: When task T1 is slowed down, it creates slack before the receptions performed in T3 and T6.

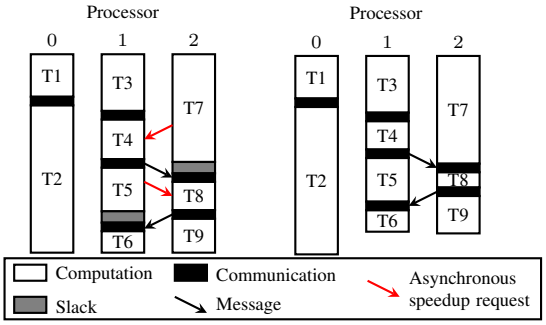

Fig. 2: The tasks running with their locally optimal frequency (left), request speedups to compensate secondary slack, resulting in globally optimal frequencies (right).

V. Iterative corrections

When the locally optimal frequency f_{locOpt} is set for a task execution, the task may be delayed. Then, all the direct and indirect successors of the delayed task in the task graph also are delayed. As a result, slowing down a single task may introduce slack after many other tasks. The created slack, that we call *secondary slack*, is distinct from the initial slack, observed while all the processes run at the maximal frequency. In Figure 1, some secondary slack is created after tasks T3 and T6 by the new frequency schedule. In such case, some energy can be lost in the slack and, in total, energy can only be saved if the energy saved by setting f_{locOpt} on T1 is greater than the one lost in secondary slack of T3 and T6. Thus, although the energy savings may seem optimal for each task, the overall energy consumption may actually increase.

As for the initial slack, the secondary slack could be overlapped by slowing down the computations preceding it. However, the secondary slack is artificially introduced by setting f_{locOpt}, thus slowing down the computations preceding the slack is not guaranteed to be optimal. Instead, we propose to accelerate the tasks sending the messages creating secondary slack. After speeding up the emitting tasks, their energy consumption increases when compared with their run using f_{locOpt}. However, the frequency is not changed for all the successor tasks after which slack was created, limiting the energy loss to a single task.

The process of accelerating tasks creating slack is illustrated in Figure 2. T4 and T8 create secondary slack on tasks T7 and T5. Then T7 and T5 send control messages to T4 and T8 which increase their frequency to achieve the requested speedup. to If the speedup cannot be achieved by increasing only the frequency of the task emitting the message, its parent tasks are also accelerated. Notice how the other tasks continue running using their local optimal frequency. Once all the requests are received and handled, the tasks run using their *globally optimal frequency* $f_{globOpt}$.

In our implementation, the speedup requests are exchanged asynchronously in a separate MPI communicator not to impact the running program. Moreover, once the slack is totally compensated, the loop iterations can run using globally optimal frequencies as long as the loop behavior remains stable. Thus, the high cost of message exchange is usually paid only once per loop execution and impacts only a short fraction of the loop execution.

VI. Basic properties

After speedup request exchanges, the locally optimal frequency f_{locOpt} may be increased to finally reach $f_{globOpt}$. However, if a task originally ended with some slack when all the tasks run at the maximal frequency, that initial slack is necessarily overlapped by computations. If an additional speedup is requested to the task, it is achieved by the parent tasks. As a result, $f_{locOpt} \leq f_{globOpt} \leq f_{noslack}$.

Existing inter-node runtime systems aim at overlapping the initial slack time by computations using a frequency equivalent to $f_{noslack}$. On the other hand, our system chooses a frequency $f_{globOpt}$ such that $f_{locOpt} \leq f_{globOpt} \leq f_{noslack}$. Moreover, energy consumption is a convex function over the frequencies [20] and, by definition, the minimal energy consumption during computations is achieved by f_{locOpt}. Thus, the frequency selected by our system leads to a lower or equivalent energy consumption compared to other inter-node runtime mechanisms.

VII. Dealing with reality

A. Task groups

Nothing ensures that the tasks last significantly longer than a frequency transition. Thus, we gather consecutive short tasks into groups as long as the group execution time is lower than 2 ms, which is about 50 times the average frequency transition latency [15]. The frequency is then controlled at the group granularity, using the maximal frequency requested by the tasks inside the group for the whole group. The maximal frequency is chosen as it enforces the slowdown constraint for all the tasks in the group and avoids undesired slack creation if a task in the group sends a message. Task grouping is then a way to select an efficient frequency even for short tasks.

B. Multicore

Multicore processors are commonly found in servers and cause specific issues as all the cores must often run at the same frequency in some processors [2]. In such processors, each core can select a different frequency but the processor actually applies the maximal frequency among those requested. During the profiling step, measurements are performed for every frequency during successive iterations. As several processes can run on different cores of the same processor, it is essential that they coordinate themselves before setting the next frequency. The processes are then all synchronized at the iteration

boundary during the measurements to make sure that no core requests a new frequency before another process ends running the iteration. When the profiling ends, an additional iteration is run using the locally optimal frequency f_{locOpt} for every task. As the secondary slack time is measured, the processes must again be synchronized at the iteration boundary. However, in the remaining loop iterations, no synchronization is required anymore as no measurement is performed.

Having a frequency setting shared by all of a processor cores impacts many existing DVFS controllers, although it is rarely mentioned in the literature. The shared frequency across processor cores is an issue as soon as different operations must be performed depending on the core frequency. Thus, although synchronizing the cores may appear to be a drawback for our system, it actually is a requirement imposed on most DVFS controllers but rarely fulfilled.

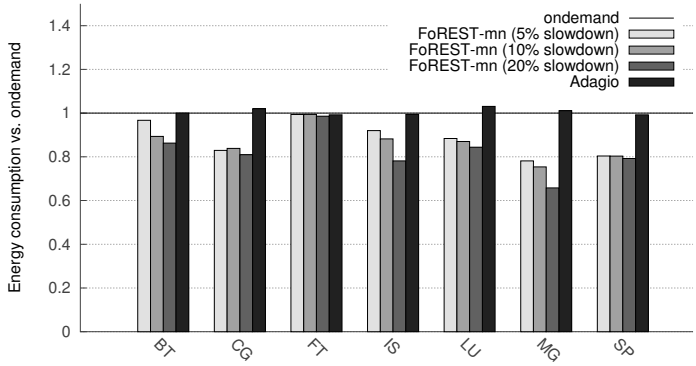

Fig. 3: Energy consumption of FoREST-mn with slowdown constaints varying from 5% to 20% compared to Adagio, expressed relatively to `ondemand`. FoREST-mn can elaborate an efficient frequency schedule even with well-balanced programs whereas Adagio only exploits slack to save energy.

VIII. Experiments

A. Experiment setup

oREST-mn has been implemented and tested with several programs. Unless specified otherwise, the experiments were run on 4 compute nodes made of 2 Intel Xeon E5-2670 (8 cores) CPUs with multi-threading deactivated. The computers run a 2.6.32-431.el6 Linux kernel, MPI is pro- vided by OpenMPI-1.6.5, and all programs are compiled with `icc` 13.1.0. The cluster is not instrumented for energy so we had to use the energy probes embedded in recent Intel processors, designated as the Intel RAPL technology. Consequently, the energy measurements were performed at the processor scale, excluding the rest of the system from the presented results. We mitigate the impact of processor energy measurements at the end of the experimental section.

We provide the implementation of the presented mechanism as open source software at https://bitbucket.org/uvsqenergy/forest-mn/src/master/, along all raw results obtained from the measurements.

To evaluate our approach, we selected the NAS Parallel Benchmark in version 3.3.1 in class D (unless specified differently). The programs represent a difficult target for inter-node runtime DVFS because most of them are well-balanced [17]. EP is excluded from our tests as it contains no communication in its main loop. Since FoREST-mn needs a few iterations for profiling, we chose to increase the iteration count of the shortest NAS programs in order to measure their energy gain potential. Thus, the number of iterations in IS, MG, and FT is set to 100. The programs were run five times on the machines and the presented results are medians of the measurements. The following results are normalized using `cpufreq` with the `ondemand` policy as a reference. `ondemand` is the default DVFS controller on most Linux systems.

B. Energy savings

The NAS benchmarks were run on our experimental platform, with three different maximal slowdown constraints of 5%, 10% and 20%. Increasing the slowdown constraint allows for greater energy saving potentials, although FoREST-mn might also apply a much lower slowdown if it predicts it to be beneficial for energy. Energy consumption results are presented in Figure 3. Our approach demonstrates significant energy savings regardless the maximal slowdown constraint used, although more energy is generally saved as the slowdown constraint grows. For instance, 34% of MG energy is saved with a 20% slowdown constraint, which represents 14% more energy savings than with a 5% slowdown constraint. Moreover, the potential for energy saving depends on the program: for example, 20% energy is saved for SP regardless the slowdown constraint. In fact, the frequency schedule of SP, computed with a 5% slowdown constraint already is maximizing energy savings. In most cases, FoREST-mn achieves energy savings higher than 10%, but saves only 1% with FT because the highest frequency minimizes energy consumption for most of the tasks. Finally, we also evaluated the gains provided by $f_{globOpt}$ compared to f_{locOpt}. During our experiments, message exchanges were initiated for most of the programs but $f_{globOpt}$ did not significantly enhanced energy consumption of the 64 cores used. Nevertheless, we expect message exchanges and slack elimination to be a requirement at larger scales, where much more energy may be wasted in slack.

Figure 3 also shows a comparison with Adagio [17]. Adagio is a state-of-the-art, inter-node DVFS controller which slows down computation to overlap slack time without impacting the overall execution time of a program. Sadly, no public implementation of Adagio is available. Thus, we re-implemented a similar system based on the authors' description. We did our best to match their description but subtle variations may remain with the original implementation. Our Adagio implementation is distributed as open source software along our own system at https://bitbucket.org/uvsqenergy/forest-mn/branch/adagio.

In our experiments, FoREST-mn saves more energy than our implementation of Adagio. The NAS programs are in fact regular and generally well-balanced in our configuration. As Adagio exploits load imbalance and communication slacks to save energy, it fails to save energy in such context. On the other hand, FoREST-mn simultaneously can reduce the frequency of several processors when energy savings are expected, even in

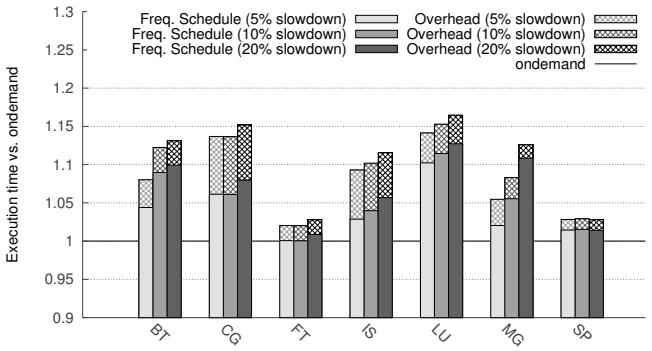

Fig. 4: Execution time of FoREST-mn with slowdown constraint varying from 5% to 20%, relatively to the maximal frequency fmax. The profiling overhead is paid once per loop execution and is independent from the number of executed iterations

absence of load imbalance. Moreover, FoREST-mn carefully controls multicore processors with unique frequency domains and takes advantage of task groups.

C. Execution time

As shown earlier, FoREST-mn saves energy with several programs. However, energy is saved at the expense of a potentially increased execution time. Thus, we evaluate the impact of FoREST-mn on program execution times. Two aspects are considered: the enforcement of the maximal slowdown constraint, and the overhead of FoREST-mn on execution times. The execution times presented in Figure 4 are decomposed into the time spent while running the frequency schedule, "Freq. Schedule" on Figure 4, and the profiling overhead as "Overhead". The execution times are normalized over the execution time of the program using the maximal frequency.

The overhead of FoREST-mn comes from factors such as extra synchronizations, or the additional messages exchanged to request speedup across processors. In fact, the slowdown associated to those operations is close to 4% in average, which is small compared to that of frequency profiling, where entire iterations are run using low frequencies to measure task execution times. In many cases, only a few tasks benefit from lower frequencies, but as the frequency is set at the iteration boundary, the overall iteration may be drastically slowed. The profiling overhead is paid once per loop execution, and lasts for a fixed number of iterations. Thus, the overhead tends to become negligible as the program runs longer. As a conclusion, the overhead of the profiling phase of FoREST-mn may be significant for short programs, but it is amortized as the loop execute more iterations.

FoREST-mn guarantees the slowdown constraint by carefully selecting the frequencies in the schedule, therefore excluding the profiling overhead from its control. Such limitation has no concrete consequence on the most interesting targets: long running programs. Focusing on the frequency schedule, Figure 4 shows that the slowdown constraint is fulfilled in most cases. The only noticeable exception is LU where the constraint is not enforced for the 5% and 10% cases. It is mostly due to the tasks workload that evolves during the execution, progressively invalidating the predicted task characteristics.

Program	Execution time	CPU energy	System energy
BT	1.09	0.84	0.93
CG	1.05	0.85	0.93
FT	1.03	0.97	0.99
IS	1.13	0.90	0.99
LU	1.01	0.96	0.98
MG	1.02	0.60	0.77
SP	1.00	0.92	0.95

Fig. 5: Execution time, CPU energy, and whole-system energy consumption of a single computer for FoREST-mn allowed to perform at most 10% slowdowns. An execution time > 1 is a slowdown compared to `fmax`. Energy is saved compared to `ondemand` when energy is < 1. The energy savings at the CPU scale translate into whole-system energy savings.

D. CPU vs. system energy

Due to the lack of system-wide power instrumentation, we only considered CPU energy consumption so far. However, the RAPL technology is based on power modeling and is known to be slightly inaccurate in some cases [7]. Moreover, it may be possible to achieve energy savings at the processor scale while the overall system energy consumption actually increased. Thus, we increase the previous results with system-wide energy measurements.

To perform system wide measurements, we have at our disposal a single computer instrumented with a Yokogawa WT210 power meter measuring the overall system consumption. The computer is made of a single Intel Core i7 3770 CPU with 4 cores and 2 threads per core. The computer runs a 3.8.2 Linux kernel. As a single computer is used, we used the class C NAS benchmark programs. The programs use 8 processes except for BT and SP which are compiled to use 4 processes.

System energy consumption along with the corresponding CPU energy consumption and execution time are presented in Figure 5. Execution time is normalized over maximal frequency whereas energy consumption is normalized over `ondemand`. The processor roughly accounts for half of the total system consumption, thus the energy savings at the system scale are lower than those measured on the CPU. However, the experiment results clearly demonstrate that the energy savings measured at the CPU level lead to actual savings on the overall system.

In FoREST-mn, the frequencies are chosen according to the CPU energy savings they are expected to provide. However, the ultimate target is the overall system energy and, ideally, the impact of a frequency on system energy should should be considered. Nevertheless, precisely estimating or measuring system energy at the scale of a task is rarely possible in production environments. One solution to the problem is presented in prior work [8], where the system power consumption is approximated by a simple constant. It is shown to be already sufficient to avoid major energy losses when large slowdowns are allowed. Such simple technique could be implemented in FoREST-mn to increase its accuracy and ensure energy savings for the whole system, even when large slowdowns are allowed.

IX. Related work

Various solutions were proposed to control DVFS at the scale of a single processor. Some offline mechanisms target specific programs and reduce frequency during phases with high off-chip activity [10], [5], [19]. On the other hand, runtime systems usually focus on the total processor workload to allow an efficient frequency control when several programs are simultaneously running [6], [8], [9]. All such intra-node DVFS controllers work efficiently in the restricted case of a single processor. The frequency is usually decreased during memory activities or local I/O operations but it is always assumed that slowing down the execution has no consequences outside the local node.

In order to extend the energy gains of DVFS to parallel programs, several offline DVFS control strategies were proposed [21], [3], [16]. Noticeably, in Green Queue [18], intra-node and inter-node slacks are treated separately with dedicated approaches based on a profiled run. The results from previous runs are stored in a database to apply an efficient schedule during re-executions. The presented offline approaches can only take a decision after at least one complete program execution and the resulting frequency schedule is only useful for the profiled workload. On the contrary, FoREST-mn is a runtime system able to achieve energy savings with programs that were not already profiled.

The simplest form of runtime DVFS controllers compatible with parallel program are probably those reducing the frequency during communication phases. For instance, Lim et al. propose to reduce the frequency during communication intensive phases [14]. Some more complex runtime systems, dedicated to DVFS control for distributed programs, are described in the literature. For instance, Jitter [12] reduces the frequency of the nodes proportionally to the time they spent executing tasks out of the critical path. In Adagio [17], the frequency is reduced for a task if it is out of the critical path. Thus, finer grain decisions are taken compared to Jitter. A similar approach, based on manual program annotations, was also developed for the Intel SCC processor [13].

The runtime systems introduced above carefully select the tasks that can be slowed down while maintaining the maximal performance. However, they only slow tasks until compensating the communication slack. As a result, they cannot save any energy when the program is perfectly balanced. FoREST-mn not only controls its impact on other nodes but it also allows energy savings with perfectly balanced program because even the tasks on the critical path can be slowed down. Moreover, it considers realistic hardware constraints such as long frequency transition latencies and shared frequency domains, as opposed to most of the presented work.

X. Conclusion

We presented FoREST-mn, a runtime DVFS controller aiming at reducing the energy consumption of parallel programs. Unlike existing systems, it slows programs down whenever required to reach higher energy efficiency. As a result, our approach simultaneously exploits off-chip activity and communication slack to reduce frequency.

As a future work, we plan to use the techniques presented earlier to consider the overall system energy consumption. Moreover, the source code instrumentation, limits the automation of the whole process so we plan to use automated techniques to determine the hot loops in the program.

Acknowledgments

The authors would like to thank R. David and M. Ringenbach at the HPC center of Strasbourg, France for their valuable help and support during the experiments.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. Feb 2014.

[2] *Intel Xeon Processor E5-1600/E5-2600/E5-4600 Product Families, Volume 1*. Intel, May 2012.

[3] G. Aupy, A. Benoit, F. Dufossé, and Y. Robert. Reclaiming the energy of a schedule: models and algorithms. *Concurrency and Computation: Practice and Experience*, 25:1505–1523, 2013.

[4] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 90–100, 2010.

[5] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 164–173, New York, NY, USA, 2005. ACM.

[6] R. Ge, X. Feng, W. Feng, and K. W. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 18–18, Sept 2007.

[7] D. Hackenberg, T. Ilsche, R. SchÃ¶ne, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, April 2013.

[8] J. Halimi, B. Pradelle, A. Guermouche, N. Triquenaux, A. Laurent, J. C. Beyler, and W. Jalby. Reactive dvfs control for multicore processors. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 102–109, Aug 2013.

[9] C. hsing Hsu and W. chun Feng. A power-aware run-time system for high-performance computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 1–1, Nov 2005.

[10] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 38–48, New York, NY, USA, 2003. ACM.

[11] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, pages 197–202, Aug 1998.

[12] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 33–33, Nov 2005.

[13] A. Kohler and M. Radetzki. Power management for high-performance applications on network-on-chip-based multiprocessors. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 77–85, Aug 2013.

[14] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 14–14, Nov 2006.

[15] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of cpu frequency transition latency. *Computer Science - Research and Development*, 29(3):187–195, Aug 2014.

[16] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–9, Nov 2007.

[17] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM.

[18] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely. Green queue: Customized large-scale clock frequency scaling. In *2012 Second International Conference on Cloud and Green Computing*, pages 260–267, Nov 2012.

[19] N. Triquenaux, A. Laurent, B. Pradelle, J. C. Beyler, and W. Jalby. Automatic estimation of dvfs potential. In *2013 International Green Computing Conference Proceedings*, pages 1–6, June 2013.

[20] K. D. Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. *CoRR*, abs/1401.4655, 2014.

[21] L. Wang, S. U. Khan, D. Chen, J. KolOdziej, R. Ranjan, C.-Z. Xu, and A. Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Gener. Comput. Syst.*, 29(7):1661–1670, Sept. 2013.