Evaluating DVFS scheduling algorithms on real hardware

Andrew Zaliwski
Department of Electrical and Computer Engineering
University of Auckland
Private Bag 92019
Auckland 1142
New Zealand
Email: a.zaliwskia@auckland.ac.nz

Stefan Lankes
Institute of Automation of Complex Power Systems
E.ON Energy Research Center,
RWTH Aachen University
Mathieustrasse 10
52074 Aachen
Germany
Email: slankes@eonerc.rwth-aachen.de

Oliver Sinnen
Department of Electrical and Computer Engineering
University of Auckland
Private Bag 92019
Auckland 1142
New Zealand
Email: o.sinnen@auckland.ac.nz

***Abstract*—In modern processors, energy savings are achieved using dynamic voltage and frequency scaling (DVFS). For task scheduling, where a task graph representing a program is allocated and ordered on multiple processors, DVFS has been employed to reduce the energy consumption of the generated schedules, hence running the processors at heterogeneous speeds. A prominent class of energy-efficient scheduling algorithms is slack reclamation algorithms, which try to use idle times (slack) to slow down processor speed to save energy. Several algorithms have been proposed and under the assumed system model they can achieve considerable energy savings. However, the question arises, how realistic and accurate these algorithms and models are when implemented and executed on real hardware. Can one achieve the promised energy savings? This paper proposes a methodology to investigate these questions and performs a first experimental evaluation of selected slack reclamation algorithms. Using schedules created by three scheduling algorithms for a set of task graphs, we generate code and execute it on a small parallel system. We measure the power consumption and compare the results between the algorithms and relate them to the expected values.**

***Index Terms*—task scheduling, energy savings, slack reclamation, DVFS**

I. Introduction

A large proportion of papers on task scheduling algorithms contain the analysis of the algorithm's effectiveness based on simulation under a certain model [2], [8], [9], [6]. These simulation models frequently assume idealized characteristics of the target environment. This approach makes it possible to compare the effectiveness of various algorithms among them, but if these algorithms are implemented on a real hardware system, the results may be dramatically different than those obtained from the idealized model [15], [16]. The model's simplifying assumptions are often necessary to make the design of and reasoning about scheduling algorithms possible. However, if these simplifications are too strong, simulation results might not be transferable to real hardware. [14].

In this paper, we intend to investigate how accurately the energy models of scheduling algorithms using DVFS can predict the actual power efficiency. While simplified models will almost always result in some discrepancy between prediction and reality, the expectation is that prediction is *qualitatively* correct, i.e. a superior algorithm in the simulation is also a superior algorithm on a real system. We focus on slack reclamation algorithms in this first study.

In modern CMOS processors, the major component deciding on power consumption is proportional to V^2f where V is voltage and f is frequency. Thus reducing the voltage reduces *energy* consumption. Due to processor safety the frequency must be adjusted with the voltage change. Available pairs of safe frequency/voltage combinations for a given processor are called as P-States. Many works have shown in simulations [3], [7], [11], [19], [10] that energy consumption can be reduced using slack time - empty spaces between the tasks on the same processors appearing as a result of synchronization between communicating tasks. Energy consumption may be reduced by applying less energy requiring P-State to neighboring tasks, hence executing tasks at heterogeneous speeds. Thus, these tasks will be executed longer which means that slack time will be filled up by these tasks without impacting an overall schedule makespan.

This paper proposes an evaluation methodology and a software testbed to examine the algorithm effectiveness on real hardware. The methodology starts by using the selected scheduling algorithms to create schedules for a set of task graphs. From the schedules and task graphs, the code is then generated to be executed on a multiprocessor system. For the communication between processors, we use simple MPI message passing [4]. In the actual evaluation, the code is then executed on an Intel Sandy Bridge 16-core system. We measure the execution time and the energy consumption for the execution and compare with expected values.

The rest of the paper is organized as follows. Section II describes three slack reclamation algorithms selected for comparison. Section III proposes the evaluation methodology. Section IV describes the code generation process and Section V contains the experimental evaluation. Finally, Section VI closes the paper with conclusions.

II. Slack reclamation algorithms

A. Scheduling model

The scheduling objective is to minimize the schedule length or makespan of a parallel application represented as a DAG (Directed Acyclic Graph) $G = (\overline{V}, \overline{E})$, where $\overline{V}$ is a set of nodes (or tasks) and $\overline{E}$ is a set of edges. Tasks are mapped onto processors for their execution, while the precedence relationships between tasks are respected. A schedule S of the task graph G on a finite set $\overline{P}$ of processors is the function pair $(t_s, proc)$, where $t_s : \overline{V} \rightarrow Q_0^+$ is the start time function of the nodes of G and $proc : \overline{V} \rightarrow \overline{P}$ is the processor allocation function of the tasks.

We also take into account the time cost of computation and communication expressed as weights of the tasks and edges, respectively. The computation cost function $w : \overline{V} \rightarrow Q^+$ of the tasks assumes the processor is running at maximum speed (i.e. highest frequency). The computation cost $w(n)$ is the time the task n occupies a processor $\overline{P}$ for its execution. Hence, the task finish time is $t_f = t_s + w(n)$. We write $t_s(n, P)$ and $t_f(n, P)$ as short for $proc(n) = P$ and $t_s(n)$ and $t_f(n)$. The communication cost function $c : \overline{E} \rightarrow Q_0^+$ of the edge $e \in \overline{E}$ is the time $c(e)$ the communication associated with edge e takes from the origin processor until it completely arrives at a different destination processor. Local communication between tasks on the same processor is assumed to be negligible, with the cost set to 0.

B. Slack reclamation

The input for all three examined slack reclamation algorithms in this paper is obtained from a non-DVFS scheduling algorithm. Any algorithm that creates a valid schedule can be employed and in this paper we chose a simple list scheduling algorithm [17]. As input, all the slack reclamation algorithms thus receive a pre-schedule that was created by this list scheduling algorithm. The further differences among compared algorithms are related to the way how they deal with the slack time (Figure 1).

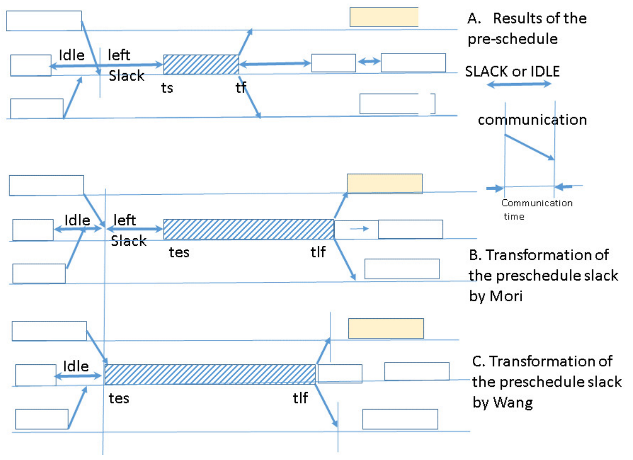

A. Results of the pre-schedule
Idle
left
Slack
ts
tf
SLACK or IDLE
communication
Communication time
Idle
left
Slack
tes
tlf
B. Transformation of the preschedule slack by Mori
Idle
tes
tlf
C. Transformation of the preschedule slack by Wang

Fig. 1. Differences in slack usage among compared algorithms.

Figure 1A shows a part of a sample schedule obtained by the use of a pre-scheduling algorithm (later refereed to as the baseline). The horizontal blue arrows mark the slack. The angled blue arrows represent communication and time necessary for the communication. Figure 1B shows how the initial pre-schedule is changed by the first examined algorithm (here called *Mori*) [12]. Slack belonging to the task n under consideration (blue shadowed) is joined with the slack of the task following task n. Slack extended that way is used next to prolong task n to fill up a whole gap by slowing down task n's execution (lowering the frequency - applying DVFS). The *Mori* algorithm is trying to make slack chains as long as possible by compensating slacks (wherever possible) belonging to the following tasks running on the same processor. Next, DVFS is applied (detailed in the following) to the slack compensated in that way.

The other two algorithms considered in this paper, called *WangA* and *WangB* [18], do not compensate slack. Instead the algorithms use both left and right slack belonging to a given task n. Usually, task n is moved left as much as possible to make bigger slack on the right side of the task n. Next, DVFS is applied to task n according to the rules presented below. Algorithm *WangB* works the same way as *WangA*, but injects additional slack time assuming that it is acceptable to extend the schedule length by an agreed ratio.

1) Mori- Joining slack time of several tasks on same processor: Algorithm $Mori$ first creates a candidate list for DVFS applications on the basis of their slack time [12]. All tasks with positive slack are added to the candidate list for each processor, in ascending start time order. The slack time of task n may be joined with slack time of its successor (Figure 1B). Next, a task is selected from the list to which the lowest frequency P-state can be applied. That means the task which gives the biggest power reduction. The selected task is removed from the list. Next, start time, slack time, and assumed pre-calculated power consumption are updated for each task in the list as the application of DVFS possibly changed these parameters. The selection is repeated until the list is empty or when there are no more tasks to apply DVFS to.

In case of $Mori$, slack-time is counted as follows: slack time between the tasks n and n_{s1} assigned to different processors p and p_{s1}, respectively, is defined as (Figure 2A):

$$slack(n, n_{s1}) = t_s(n_{s1}, p_{s1}) - t_f(n, p) - c(n, n_{s1}) \quad (1)$$

If both, task n and its successive task n_{s0} are allocated to the same processor p, slack-time is defined as (Figure 2B):

$$slack(n, n_{s0}) = t_s(n_{s0}, p) - t_f(n, p) + slack(n_{s0}) \quad (2)$$

From (1) and (2) slack-time of task n is calculated as follows:

$$slack(n) = min_{n_s \in succ(n)} slack(n, n_s) \quad (3)$$

Mori adds to the slack-time of task n slack time (if it exists) of its successor n_{s0} (Figure 2C), $slack(n)$ calculation requires

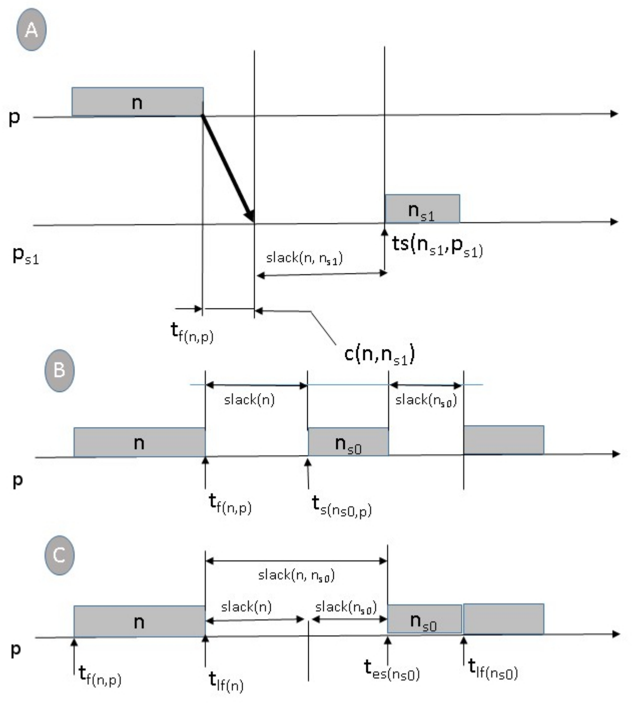


Fig. 2. Slack-time definition and usage in *Mori*.

the knowledge of slack-time of the task's successor (when both are on the same processor), slack-time is calculated in ascending task start time order of the tasks on the list of each processor. For a detailed description of *Mori* please refer to [12].

2) Wang algorithms - Energy performance trade-off scheduling: Wang et. al. [18] propose two algorithms. *WangA*: Best Effort Scheduling with objective of minimizing energy consumption without damaging the performance of a schedule, and *WangB*: energy performance trade-off scheduling with the objective of (more aggressively) reducing energy consumption with the tolerable loss of performance.

Wang et. al. interpret the result produced by the pre-scheduling algorithm as a Gantt Chart with time slots (arbitrary size unit of processor time). Each task or communication may use one or more slots. The algorithm analyzes the content of each time slot on each processor if the time slot is idle or contains communication, the algorithm scales down frequency to the lowest possible. If one or more time slots contain a non-critical task, the slack time is calculated using equation (4). The necessary frequency to apply to the task to fill up the slack is computed using equation (5). Frequencies for critical tasks are not changed. Slack time for both algorithms is calculated in the following way:

$$slack(n) = t_{lf}(n) - t_{es}(n) \tag{4}$$

This is the difference between task's "latest finish time" and "earliest start time" where task n's *earliest start time* is the earliest possible time when n may start without affecting the schedule: $t_{es}(n) = \max_{m \in pred(n)}\{t_{lf}(m) + c(m,n)\}$ and task n's *latest finish time* is the latest possible time when n can finish without affecting the schedule: $t_{lf}(n) = min_{l \in succ(n)}\{t_{es}(l) - c(l,n)\}$.

When the non-critical task n is executed on processor p_k then n's execution time can be extended to $slack(n)$ without affecting the schedule. This is done by scaling p_k's frequency:

$$f_{op}^{p_k} = f_{max} * \frac{t_0(n)}{t_o(n) + slack(n)} \tag{5}$$

where $t_0(n)$ is n's execution time when p_k is running with the maximum frequency f_{max} and $f_{op}^{p_k}$ is operating frequency of p_k.

Both *WangA* and *WangB* can use slack time from the left and right side of a task (Figure 1C), because the way of counting slack-time is the same for both cases. For more details on both algorithms please refer to [18].

III. Evaluation methodology

The general schema of our methodology is presented in Figure 3 and a software testbed has been created. The testbed employs as a pre-schedule algorithm a greedy list scheduling algorithm using bottom levels (including computation and communication costs) of the tasks for ordering [17]. The obtained final schedule, after slack reclamation, is used to generate target source code, which was compiled and run along with power measurement on our target system, a Sandy Bridge[1] 16-core Intel system consisting of two Xeon E5-2680 8c 2.7GHz 8-cores processors, with Linux RHEL 6.3 (kernel 2.6). For the communication among processors, as detailed in the next section, we employ MPI directives and used the MPICH2 1.9a2 implementation. To change the frequency of the cores the *userspace* governor from the operating system was used. Power measurement was done using LIKWID tools [5]. We selected the approach of generating code from graphs as this allows us to test many different graph types and structures. The opposite approach, i.e. selecting sample programs and generating corresponding graphs, does not allow us to cover such a wide spectrum of scenarios and to test the algorithms systematically. Furthermore, by generating code corresponding to a graph, we can fully control the execution times of the tasks and the communication volumes for each edge, removing the typical uncertainty related to execution time estimates.

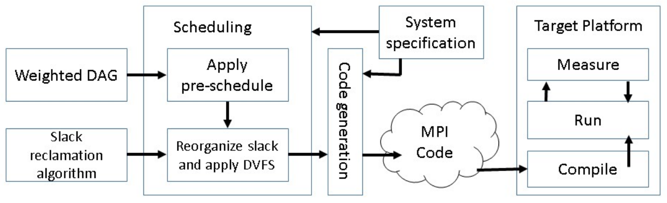


Fig. 3. Steps of research methodology.

There are three possible outcomes that can be obtained from the experiments:

- Simulation results more or less accurately reflect results on real hardware, *or*
- Implementation of the evaluated algorithms will give worse results than simulation, however, simulation based ranking of the algorithms will be preserved on a real hardware, i.e. is qualitatively correct, *or*
- Evaluated algorithms will have different results to simulations, both quantitatively and qualitatively. In other words comparing simulation results obtained by authors of algorithms will not give us insights into how these algorithms will behave on real hardware. Simulation-based ranking of algorithms will not be preserved on real hardware.

IV. Code generation

The code generator (Figure 4) is a Java based software which receives a weighted DAG graph (Figure 4A) to generate from it simple code with MPI directives for communication (Figure 4D) ready to compile and run on a given hardware system (Figure 4E). The general principle and in particular the communication generation is similar to the one proposed in [13], [15]. MPI was used as it is closer to the task graph model with the explicit communication edges. If shared memory programming were used instead of message passing, communication would not be as explicit. Hence the results could be more difficult to interpret and reproduce. Firstly, the input graph is pre-scheduled (Figure 4B) by the non-DVFS list scheduling algorithm considering communication cost, however without energy usage optimization.

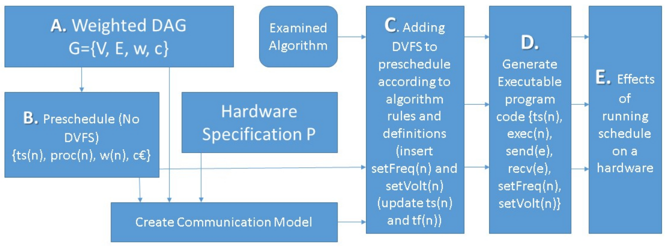

A. Weighted DAG G={V, E, w, c}
Examined Algorithm
C. Adding DVFS to preschedule according to algorithm rules and definitions (insert setFreq(n) and setVolt(n) (update ts(n) and tf(n))
D. Generate Executable program code {ts(n), exec(n), send(e), recv(e), setFreq(n), setVolt(n)}
E. Effects of running schedule on a hardware
B. Preschedule (No DVFS) {ts(n), proc(n), w(n), c€}
Hardware Specification P
Create Communication Model

Fig. 4. Code generation. The functions: exec(n) compute task n, send(e) and recv(e) implements communication for edge e, setFreq(n) and setVolt(n) implements voltage/frequency change according to selected P-state.

Inside the code generator a pre-schedule is represented as a set of tasks assigned to cores of $p_k \in P$ of the target system, containing starting $t_s(n)$ and finishing times $t_f(n)$ for each task n (Figure 5A).

Secondly, the pre-schedule obtained from the previous step is processed by the evaluated slack reclamation scheduling algorithm (either *Mori*, *WangA* or *WangB*) (Figure 4C). The evaluated algorithms can be seen as a mapping of an initial set of pairs $\forall_{n\in V}(t_s(n), t_f(n))$ obtained from pre-scheduling into a new set of pairs $\forall_{n\in V}(t_{es}(n), t_{lf}(n))$ defining new schedule. The new position of $t_{es}(n)$ on time axis for a given processor's p_k depends on the method of slack utilization defined by the evaluated algorithm, while the position of $t_{lf}(n)$ additionally may depend on voltage and frequency applied to processor p_k when running task n according to how the algorithm implements DVFS.

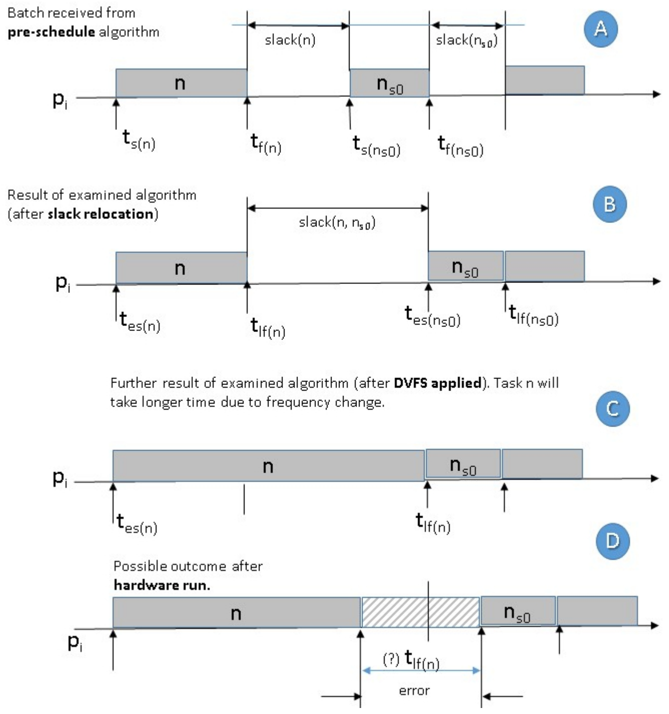

Batch received from **pre-schedule** algorithm
slack(n)
slack(n_s0)
A
n
n_s0
p_i
t_s(n)
t_f(n)
t_s(n_s0)
t_f(n_s0)
Result of examined algorithm (after **slack relocation**)
slack(n, n_s0)
B
t_es(n)
t_lf(n)
t_es(n_s0)
t_lf(n_s0)
Further result of examined algorithm (after **DVFS applied**). Task n will take longer time due to frequency change.
C
D
Possible outcome after **hardware run.**
(?) t_lf(n)
error

Fig. 5. Pre-schedule (A), schedule with slack relocation (B), DVFS-schedule (C), and hardware schedule (D).

For example, the pre-schedule of Figure 5A is transformed by the evaluated slack reclamation algorithm into final DVFS-schedule in the following way: Task n_{s0} was moved to the right (its coordinates of $(t_s(n_{s0}), t_f(n_{s0}))$ from the pre-schedule (Figure 5A) were transformed into the new coordinates $(t_{es}(n_{s0}), t_{lf}(n_{s0}))$ (so, in fact, task n_{s0} was moved to the right) without changing the task size $w(n_{s0})$ (Figure 5B). With task n it is different: Start time of task n does not change, $t_{es}(n)$ remains at the same position as $t_s(n)$, but there is also a selected P-State which should be applied to the task n to use the entire slack-time gap (Figure 5C) without challenging the makespan.

The DVFS-schedule (Figure 5C) at this point is an "ideal" schedule as it does not consider the discrete frequencies and P-States of the target platform. All tasks coordinates are calculated by the examined algorithms and they fit into the place on the schedule where exactly (according to the examined algorithm) they should be.

When the "ideal" DVFS-schedule is running on the target hardware in a form of compiled MPI program, the results may be different than assumed in the schedule from Figure 5C. The produced MPI code contain start position of the task n, operating frequency, which according to the evaluated algorithm should be the lowest that still guarantees that task n will fit into the slack gap, and the computation code for task n corresponding to its weight (Figure 4D). The finish time of task t_{lf} depends on the time of execution on hardware, which is directly related to the task weight and determined operating frequency and some other hardware and software factors usually beyond our control. The real finish time of the task n may be different to the assumed by the examined algorithm, e.g. task n on hardware may finish earlier not using a whole slack or may finish later (margin of "error" in the Figure 5D) causing disturbances for other tasks. These disturbances can be propagated further by communication between tasks worsening the overall hardware results of evaluated algorithm.

A. *Generating communication code*

All examined algorithms consider communication cost. The module "Creating Communication Model" (Figure 4) creates a linked list representing all outgoing (send) and all incoming (receive) communications from a given task n. When target executable code is created, instructions responsible for communication are inserted into the code. If a task (e.g. task B) requires data from other tasks to perform its computation, all data receiving instructions are placed before that task (Figure 6top). Similarly, all data sending instructions are located at the end of the task B. Sending instruction $send(n)$ is implemented by the non-blocking function *MPI_Isend()* and receiving instruction $recv(n)$ is implemented by the non-blocking function *MPI_Irecv()*. However, *MPI_Irecv()* is non-blocking but uses *MPI_wait()* to enforce that the task B will not run until it receives all required data. A single communication unit was setup to use the same time as a single computation unit. To give the program execution flow the highest possible flexibility, the receive functions are initiated at the beginning of the generated code on each processor (Figure 6 bottom), so that the message sending can progress as soon as possible. The wait calls before tasks enforce the correct precedences.

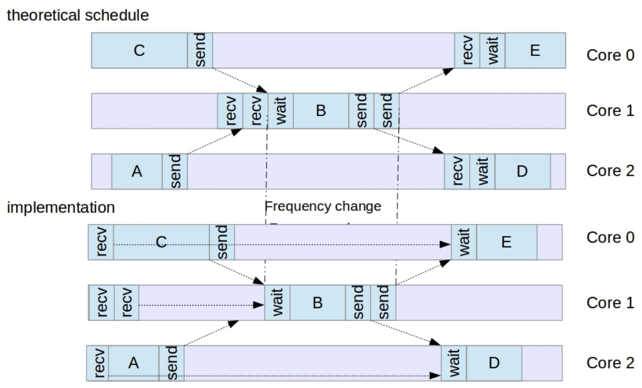


Fig. 6. Example for implementing communication, showing schedule on 3 processors; (top) theoretical schedule, (bottom) actual implementation with recv initiated at start

V. Experimental evaluation

In this evaluation we execute the code generated in the proposed testbed for many input graphs according to the considered slack reclamation scheduling algorithms and compare their power consumption.

A. *Setup*

The evaluation process uses 710 graphs generated randomly with different characteristics and properties. We use the graph structures serial-parallel, stencil, pipeline, random, fork-join and tree with sizes varying from 40 to 110 tasks. The weights for the tasks and edges are created randomly with a uniform distribution. Furthermore to evaluate the impact of communication on energy consumption, three values of the communication-to-computation ratio (CCR) were considered: 0.1 (low), 1 (medium) and 5 (high communication). CCR is defined as the total communication cost over the total computation costs $CCR(G) = \frac{\sum_{e \in E} c(e)}{\sum_{n \in V} w(n)}$. To achieve different CCR values, the weights of the tasks and edges are scaled correspondingly.

Every generated graph was scheduled by each of the four algorithms (described in Section II) on 16 processors: a) baseline - algorithm using only list based pre-scheduling; b) *Mori*; c) *WangA* and d) *WangB*. This gives a total of 2840 (4 algorithms x 710 graphs) generated programs and 17040 (4x710x6) executions of the generated programs on the dedicated target system. The same program was run six times to compensate fluctuations caused by background operating system activities. For each program implementing a scheduled graph the average power usage (in Watts) for the whole application was measured.

B. *Results*

Figure 7 shows box plots of average power consumption over the different graph structures. Hence each box plot is for all sizes and CCR values of graphs of that structure. The shown values are normalised to the baseline (list scheduling algorithm *without* any DVFS), which is given the value 100. As can be observed immediately, the three slack reclamation algorithms consume more power for the large majority of the cases, sometimes significantly.

It was expected that the results would significantly differ from the theoretical results, given that the used scheduling model is very idealizing. However, it is still very surprising that the best results (the lowest energy usage) were obtained for pre-schedule algorithm with no DVFS. Energy aware algorithms (like *Mori*, and *WangA*) have worse results. The most aggressive algorithm in terms of energy savings WangB, which even increases the schedule length, has the worst results. The more advanced algorithms have bigger energy usage independently on processed graph type.

To investigate the results further, we looked at the number of cases a certain algorithms was better than baseline, categorized by graph structure. Figure 8 shows the percentage of cases when a given algorithm had lower energy usage than baseline algorithm. For example, for ca. 14% of the Series-Parallel graphs *Mori*'s power usage was lower than that of the baseline algorithm. It can be observed that the performance seems to be strong graph structure dependent. The slack reclamation algorithms achieve the best results for random graphs and in/out-trees.

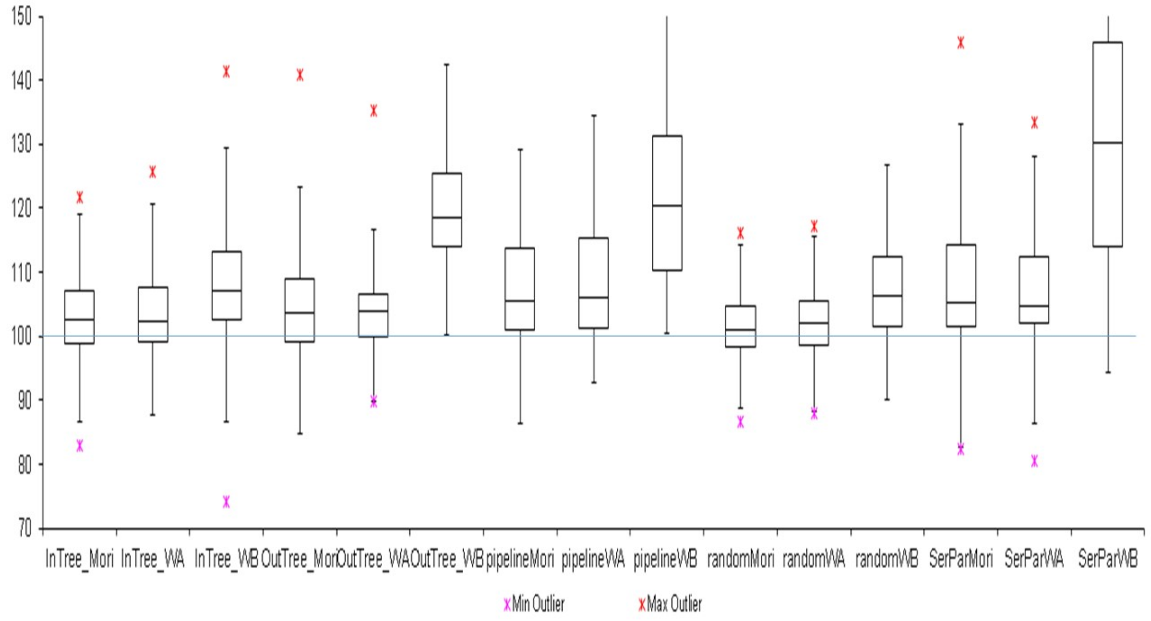
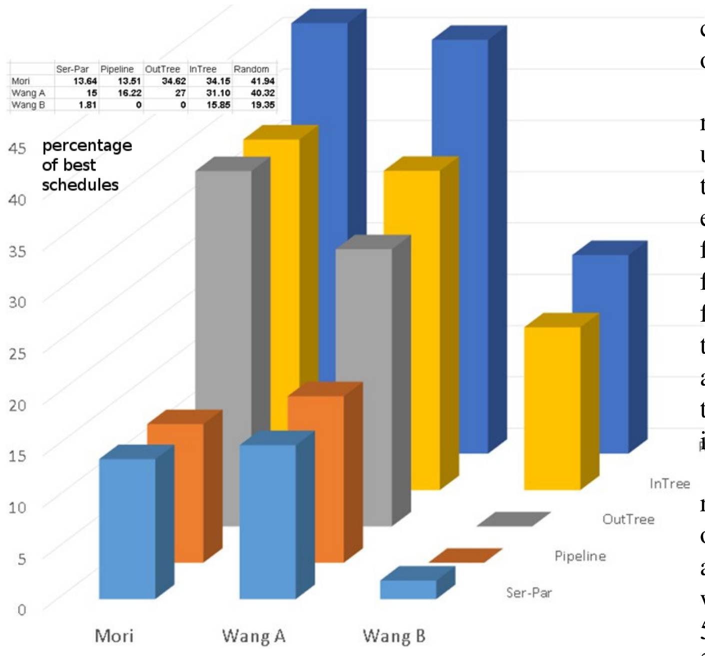


Fig. 7. Distribution of Power measurements in relationship to algorithm and structure of graphs. Values above 100% mean higher power usage than baseline algorithm. Below 100 means lower power usage than baseline algorithm. All values were normalized in relationship to baseline algorithm (100% energy usage).



	Ser-Par	Pipeline	OutTree	InTree	Random
Mori	13.64	13.51	34.62	34.15	41.94
Wang A	15	16.22	27	31.10	40.32
Wang B	1.81	0	0	15.85	19.35

Fig. 8. Percentage of graphs for which given algorithm had less power usage than baseline algorithm (data normalized).

C. Discussion

While many might have expected that scheduling algorithms behave in practice differently than in theory, the clear negative result is astonishing. The slack reclamation algorithms did not only have no positive effect (on average) on the power consumption; it was clearly negative. Reasons for this result can come from our proposed methodology and the approach of the slack reclamation algorithms.

a) Non-continuous frequency spectrum: Many algorithms assume that slack time gaps will be completely filled up. However, in general there will remain a small gap between tasks because the algorithms assume an ideal hardware environment with a continuous spectrum of the voltage and frequency changes. In practice we usually have some discrete frequency steps. The closest frequency to desired theoretical frequency will leave a gap. In the case of certain processor types, some voltage/frequency combinations are not allowed as they may damage the processor. This could explain less than ideal performance, however it should not have a negative impact.

b) Slack time: By their nature, slack reclamation algorithms can only perform well if there is a significant amount of slack. Hence, we analysed the available slack time across all scheduled graphs and it is usually a small percentage of the whole schedule. Average slack time for used set of graphs was 5.13%, with a standard deviation of 5.91%, and with median 3.24% (expressed as a percentage in relation of computational time). In other words, the slack reclamation algorithms had little to work on and the overhead of DVFS negated any benefit. This observation is in correspondence with the results in Figure 8. The slack reclamation algorithms performed better (but still not good) for graph structures where the likelihood of slack is high, namely random graphs and trees. The more structured graphs like pipeline and SP-graphs have little or no slack in their initial pre-schedules.

c) Idle time: Slack time is the time between tasks that can be used to stretch the execution of tasks. But even if all slack time has been reclaimed, there is still idle time in a schedule, namely before the execution of the first task on a processor (the task is waiting for communication from another processor) and after the completion of the last task (which has sent a communication to another processor). We analysed this idle time across all scheduled graphs and it is an order of magnitude more significant than slack time with average idle time 51.3%, standard deviation 116%, and with median idle equal to 25.2%. This is related to the fact that for some graphs not all cores were used by the scheduling algorithm. In theory, a core may be turned off or switched to the lowest frequency during these idle times. However, this is independent of slack reclamation algorithms and can be applied to start and end idle periods in any schedule. An additional problem is knowing when the first task should start if it is waiting for results of other tasks running earlier on another core. A processor waiting for communication needs to be awake. It must issue a data receiving instruction waiting for data from other tasks (required by the MPI non-blocking communication), what in fact enforces each core to work from the beginning of the schedule without leading idle time (Figure 6bottom). The same holds at the end, when a core can only go to sleep after all MPI communications it is involved in have been fully completed (i.e. received by the last tasks).

d) Processor utilisation : Under the classic communication delay model, scheduling algorithms put many more tasks on the first processors when communication is significant, leaving more distant processors idle. Also slack in that case is not evenly distributed among the cores. We analysed the average number of unused processors in all created schedules and found that it is equal to 3. However, some types of graphs have a high number of unused processors, which causes large idle time. For example, In-Tree - all 16 cores were used, Out-Tree - almost all cores were used (only an average 0.54 core were not used), pipeline graphs - an average of 10 cores were *not* used, random graphs - all cores were fully used, series-parallel - average of 4.54 cores were *not* used. Again, this observation aligns very well with the performance observed in Figure 8, where graphs with more idle time perform worse. Naturally, the utilisation of processors depends on the CCR. In-Tree and Random graphs used all processors independently of CCR, but SP graphs did not use 10-12 processors for CCR=5, but only 2-4 were unused for CCR 0.1.

e) High Communication-Computation Ratio (CCR): Using communication time comparable with a time of computation to achieve high CCR values requires the transfer of large volumes of data. This consumes memory for buffers, and it is limited technically by accessible memory. This enforced us in our framework to use short computational units which makes reliable energy measurement more difficult. The cost of frequency switching (and especially voltage switching) has a more significant impact in this scenario.

In summary, it seems that avoiding or reducing idle times in schedules, i.e. before the first task and after the last on each processor, or having completely idle processors, is significantly more important than to use slack time to decrease energy consumption. Slack reclamation algorithms by their nature do not change the processor allocation nor the order of the tasks. The presented results here are early results and require confirmation, but they indicate that energy efficient algorithms might be more successful when reducing the number of used processors at the same time they try to reduce the schedule length.

VI. Conclusions

This paper proposed a methodology to investigate the question whether energy-efficient scheduling algorithms really save energy. A software testbed was proposed that generates code from input DAGs for a specific hardware platform. Then the code generated by the testbed was compiled and run with energy measurement on a hardware platform to perform a first evaluation of the question stated above. The obtained results were surprising as slack reclamation algorithms actually consumed more energy than pre-schedule algorithm without DVFS. Future work will be focused on further investigating the reasons for obtaining results like presented above and it will be necessary to perform more experiments for confirmation. Current results suggest that reducing the number of processors and reducing idle time (that cannot be reclaimed) is more important than slack reclamation.

VII. Acknowledgments

The author(s) wish to acknowledge the contribution of NeSI high-performance computing facilities to the results of this research. NZ's national facilities are provided by the NZ eScience Infrastructure and funded jointly by NeSI's collaborator institutions and through the Ministry of Business, Innovation & Employment's Research Infrastructure programme. URL https://www.nesi.org.nz.

References

[1] J. Crop A. Varma, B. Bowhill. Power management in the intel xeon e5 v3. *Low Power Electronics and Design (ISPLED), 2015 IEEE/ACM International Symposium on*, pages 371–376, 2015.

[2] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proc. Design, Automation and Test in Europe Conf. and Exhibition*, pages 168–175, 2002.

[3] Beena and C. S. R. Prashanth. Power cognizant algorithms using slack reclamation method. In *Contemporary Computing and Informatics (IC3I), 2014 International Conference on*, pages 1101–1106, 2014.

[4] W. Gropp, E. Lusk, and A. Skjellum. MPI resources online. In *Using MPI:Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 2014.

[5] G. Hager and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216, 2010.

[6] Stoimenov S. Thiele L. Chen J. Feasibility analysis of on-line dvs algorithms for scheduling arbitrary event streams. volume RTSS 2009 of *Real-Time Systems Symposium, 30th IEEE*, pages 261–270, 2009.

[7] Nat. Taiwan Univ. Jian-Jia Che, Chuan-Yue Yang, and Tei-Wei Kuo. Slack reclamation for real-time task scheduling over dynamic voltage scaling multiprocessors. In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 1, 2006.

[8] Jaeyeon Kang and S. Ranka. Dynamic algorithms for energy minimization on parallel machines. In *Proc. 16th Euromicro Conf. Parallel, Distributed and Network-Based Processing PDP 2008*, pages 399–406, 2008.

[9] Jaeyeon Kang and S. Ranka. Assignment algorithm for energy minimization on parallel machines. In *Proc. Int. Conf. Parallel Processing Workshops ICPPW '09*, pages 484–491, 2009.

[10] Young Choon Lee. and Y. Zomaya. Energy conscious scheduling for distributed computing systems under different operating conditions. *Parallel and Distributed Systems, IEEE Transactions on*, 22(8):1374–1381, Aug. 2011.

[11] A. Manzak and C. Chakrabarti. Variable voltage task scheduling algorithms for minimizing energy. In *Proc. Low Power Electronics and Design, Int. Symp*, pages 279–282, 2001.

[12] Y. Mori, K. Asakura, and T. Watanabe. A task selection based power-aware scheduling algorithm for applying dvs. In *Proc. Int Parallel and Distributed Computing, Applications and Technologies Conf*, pages 518–523, 2009.

[13] O. Sinnen and L. Sousa. Comparison of contention aware list scheduling heuristics for cluster computing. In *Proc. Int Parallel Processing Workshops Conf*, pages 382–387, 2001.

[14] O. Sinnen and L. Sousa. Task scheduling: considering the processor involvement in communication. In *Proc. Third Int Parallel and Distributed Computing Third Int. Symp. /Algorithms, Models and Tools for Parallel Computing Heterogeneous Networks Workshop*, pages 328–335, 2004.

[15] O. Sinnen and L. A. Sousa. Communication contention in task scheduling. 16(6):503–515, 2005.

[16] O. Sinnen, L. A. Sousa, and F. E. Sandnes. Toward a realistic task scheduling model. 17(3):263–275, 2006.

[17] Oliver Sinnen. *Task Scheduling for parallel systems*. Willey Series on Parallel and Distributed Computing., 2007.

[18] Lizhe Wang, Gregor von Laszewski, Jay Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proc. 10th IEEE/ACM Int Cluster, Cloud and Grid Computing (CCGrid) Conf*, pages 368–377, 2010.

[19] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. 14(7):686–700, 2003.