Accurate Energy Modeling for Many-core Static Schedules with Streaming Applications

Simon Holmbacka*, Jörg Keller†, Patrick Eitschberger† and Johan Lilius*
*Faculty of Science and Engineering, Åbo Akademi University, Turku, Finland
Email: firstname.lastname@abo.fi
†Faculty of Mathematics and Computer Science, FernUniversität in Hagen, Hagen, Germany
Email: firstname.lastname@fernuni-hagen.de

***Abstract*—Many-core systems provide a great performance potential with the massively parallel hardware structure. Yet, these systems are facing increasing challenges such as high operating temperatures, high electrical bills, unpleasant noise levels due to active cooling and high battery drainage in mobile devices; factors caused directly by poor energy efficiency. Furthermore by pushing the power beyond the limits of the power envelope, parts of the chip cannot be used simultaneously – a phenomenon referred to as "dark silicon". Power management is therefore needed to distribute the resources to the applications on demand. Traditional power management systems have usually been agnostic to the underlying hardware, and voltage and frequency control is mostly driven by the workload. Static schedules, on the other hand, can be a preferable alternative for applications with timing requirements and predictable behavior since the processing resources can be more precisely allocated for the given workload. In order to efficiently implement power management in such systems, an accurate model is important in order to make the appropriate power management decisions at the right time. For making correct decisions, practical issues such as latency for controlling the power saving techniques should be considered when deriving the system model, especially for fine timing granularity. In this paper we present an accurate energy model for many-core systems which includes switching latency of modern power saving techniques. The model is used when calculating an optimal static schedule for many-core task execution on systems with dynamic frequency levels and sleep state mechanisms. We derive the model parameters for an embedded processor with the help of benchmarks, and we validate the model on real hardware with synthetic applications that model streaming applications. We demonstrate that the model accurately forecasts the behaviour on an ARM multicore platform, and we also demonstrate that the model is not significantly influenced by variances in common type workloads.**

I. Introduction

Computer systems often face a trade-off decision between performance and power dissipation. High power dissipation and fast execution usually lead to high energy consumption in modern many-core systems [13], [12]. Execution speed is usually optimized by the programmer and compiler while minimizing energy is often left to the operating system which employs Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) using sleep states. However, an operating system with a dynamic scheduler has no knowledge about the application, its behavior and its timeline. In practice, the power management for dynamic schedules is performed with respect only to the workload level, which does not describe performance requirements. This means that the application is normally executed faster than what is actually required, and energy is being wasted because of the unnecessarily high power dissipation.

For applications consisting of a set of tasks with a predictable behavior and a known execution deadline, a schedule with the information when to execute which task at which speed can be devised at compile time (i.e. a static schedule). With hints from the application, the power management techniques can more precisely scale the hardware according to the software performance demands, and energy is minimized by eliminating unnecessary resource allocation. However, Power management is a practical interplay between software algorithms and physical hardware actions. This means that accessing power management techniques in general purpose operating systems introduces practical shortcomings such as access latency. Two separate mechanisms – DVFS and DPM – are currently used for minimizing the CPU power dissipation. As DVFS regulates voltage and frequency to minimize the dynamic power, DPM is used to switch off parts of the CPU to minimize the rapidly growing static power [16]. The techniques are therefore complementing each other and a minimal energy consumption is achieved by proper coordination of both techniques [1], [24]. While both mechanisms have been evaluated in the literature [9], [17], no work has been done to determine the practical latency of both DVFS and DPM on a single platform, and its impact on power management.

In this work we present an accurate energy model for static schedules in many-core systems using DVFS and DPM. The model is based on measurements of a single synthetic application on real hardware to conform with complete platform details and a realistic view of the static and dynamic power balance. In practical real-world systems, there is always a certain latency for utilizing DVFS and DPM both due to hardware and software implementations (in this paper Linux). Instead of focusing on eliminating or minimizing this latency, we chose to acknowledge this short coming in current computing systems, and learn how to integrate this detail in the system model. We account for the latency of using DVFS and DPM on a statically scheduled many-core system by including the timings in the decision making process of power management techniques. We validate the results by implementing a framework for synthetic workloads on real hardware (ARM multicore) running an unmodified Linux OS. The evaluation demonstrates that the model is able to accurately forecast the energy consumption of a selected static schedule under different workload configurations and different deadlines. We

also demonstrate by experiments that the forecasts of the model remain stable if the benchmarks used to generate the model and the applications scheduled by the model differ in their characteristics.

The repeated execution of our synthetic applications (set of tasks with predictable workload and common deadline) models streaming applications with throughput requirements, which form a large class of applications for many-core processors. Considering several executions (rounds) together can allow further optimizations, which we include into our model. While our experiments only address a single platform, we note that the approach can be applied to other hardware platforms without re-engineering the core algorithm, as long as the platforms allow measurement for creating the model parameters.

The remainder of this article is structured as follows. In Section II, we discuss related work. In Section III, we investigate the latency and energy overhead of DVFS and DPM mechanisms as experienced by applications running on typical platform (ARM multicore) with a typical operating system (Linux). From these measurements we derive an energy model for task-based applications in Section IV. In Section V, we evaluate the forecasting capabilities of this energy model with respect to execution on a real hardware. We extend the model towards streaming applications in Section VI, and present additional energy optimizations possible by scheduling several rounds of computation together. In Section VII, we give conclusions and an outlook onto future work.

II. Related Work

DVFS and its efficiency for multi-cores has been studied in the past [9], [17], but mostly the focus has been put directly on measuring the overhead of physically switching hardware states [17], [27] including PLL locking, voltage level switching etc. Mazouz et al. present in [25] a frequency transition latency estimator called FTaLaT, which chooses a frequency depending on the current phase of a program. They argue that programs mostly have either CPU intensive phases in which the CPU is running on a high clock frequency or memory intensive phases in which the clock frequency can be decreased to save power. For very small memory intensive regions, it is favorable to ignore the frequency scaling because the switching delay would be higher than the length of the memory phase. They evaluate their estimator with a few micro-benchmarks (based on OpenMP) on different Intel machines, and they show that the transition latency varies between 20 and 70 microseconds depending on the machine. As the total switching latency is the sum of both hardware and software mechanisms, we study in this paper the practical aspects of switching latency in both DVFS and DPM for off-the-shelf operating systems running on real hardware. Influences of user space interaction and the kernel threads which control the power saving mechanisms are studied, and related to the effects on the energy consumption.

The paper of Schöne et al. [29] describes the implementation of the low-power states in current x86 processors. The wake-up latencies of various low-power states are measured and the results are compared with the vendor's specifications that are exposed to the operating system. The results show fluctuations e.g. depending on the location of the callee processor. Their work complements ours, but rather than using the x86 architecture we focus on mobile ARM processors with less support for hardware power management.

Algorithms for minimizing energy based on power and execution time have been presented in previous work such as [3], [9], [10]. Cho et al. define an analytical algorithm for expressing dynamic and static power in a multi-core system with multiple frequency levels. The minimum-energy-operation point is then calculated by determining the first order derivative of the system energy with respect to time. The mathematical expression defined in [3] exploits the task parallelism in the system to determine the amount of processing elements required, and hence influencing the static power dissipation. In our work, we define the system power model based on experiments on real hardware rather than analytical expressions in order to tailor the model closer to real-world devices.

The work in [10] uses similar reasoning to determine an *energy efficient frequency* based on the timing guarantees and available resources for real-time tasks. Their mechanism was able to map a certain number of replica tasks in a multi-core system in order to parallelize the work to an optimal number of cores running on an optimal frequency. Similar to [3], the authors used a bottom-up model to define the power as an analytical expression without taking temperature into account.

The work in [20] defines an algorithm for calculating the minimum energy consumption for a microprocessor when taking both dynamic power and static power into account in a system with DVFS and DPM. The authors define an analytical expression for the power dissipation and divides the workload for a given timeline into active time slots and sleep slots, and calculates the total energy over a given time with a given power dissipation for each slot.

In [21] an Energy-Aware Modeling and Optimization Methodology (E-AMOM) framework is presented. It is used to develop models of runtime and power consumption with the help of performance counters. These models are used to reduce the energy by optimizing the execution time and power consumption with focus on HPC systems and scientific applications. Our approach follows the same principle, but instead we use a top-down power model based on real experiments rather than analytical expressions. We also account for the latency of both DVFS and DPM which, as explained, becomes important when the time scale is shrinking.

Gerards and Kuper [6] describe various possibilities and techniques to reduce the energy consumption of a device (e.g. a processor core) under real-time constraints. They describe the problems and present optimal solutions for DPM-based and also combined DPM and DVFS approaches when both the energy and time for scaling the frequency and shutdown or wakeup a core are considered. The theoretical part is close to ours but they only focus on single devices with a fixed deadline and on real-time systems. In our approach we focus on many-core processors and static schedules and we consider different deadlines for each schedule.

Zuhravlev et al. [32] give a survey of energy-cognizant scheduling techniques. Many scheduling algorithms are presented and different techniques are explained. They distinguish between DVFS and DPM-based solutions, thermal management solutions and asymmetry-aware scheduling. None of

the described scheduling algorithms take the switching or shutdown/wakeup overhead into account like our approach.

While acknowledging that DVFS and DPM are possible energy savers in data centers [5], [19], [14], our work focus on core level granularity with a smaller time scale and our measurements are based on the per-core sleep state mechanism rather than suspension to RAM or CPU hibernation. Aside from the mentioned differences, none of the previous work deals with latency overhead for both DVFS and DPM on a practical level from the operating system's point of view. Without this information, it is difficult to determine the additional penalty regarding energy and performance for using power management on modern multi-core hardware using an off-the-shelf OS such as Linux.

III. Power Distribution and Latency of Power-Saving Mechanisms

Power saving techniques in microprocessors are hardware-software coordinated mechanisms used to scale up or down parts of the CPU dynamically during runtime. We outline the functionalities and current implementation in the Linux kernel to display the obstacles of practically using power management.

A. Dynamic Voltage and Frequency Scaling (DVFS)

The DVFS functionality was integrated into microprocessors to lower the dynamic power dissipation of a CPU by scaling the clock frequency and the chip voltage. Equation 1 shows the simple relation of these characteristics and the dynamic power

$$P_{dynamic} = C \cdot f \cdot V^2 \tag{1}$$

where C is the effective charging capacitance of the CPU, f is the clock frequency and V is the CPU supply voltage. Since DVFS reduces both the frequency and voltage of the chip (which is squared), the power savings are more significant when used on *high* clock frequencies [28], [31].

The relation between frequency and voltage is usually stored in a hardware specific look-up table from which the OS retrieves the values as the DVFS functionality is utilized. Since the clock frequency switching involves both hardware and software actions, we investigated the procedure in more detail to pinpoint the source of the latency. In a Linux based system the following core procedure describes how the clock frequency is scaled:

1) A change in frequency is requested by the user
2) A mutex is taken to prevent other threads from changing the frequency
3) Platform-specific routines are called from the generic interface
4) The PLL is switched out to a temporary MPLL source
5) A safe voltage level for the new clock frequency is selected
6) New values for clock divider and PLL are written to registers
7) The mutex is given back and the system returns to normal operation

1) DVFS implementations: To adjust the DVFS settings, the Linux kernel uses a frequency governor [15] to select, during run-time, the most appropriate frequency based on a set of policies. In order to not be affected by the governors, we selected the `userspace` governor for application-controlled DVFS. The DVFS functionality can be accessed either by directly writing to the `sysfs` interface or by using the system calls. By using the `sysfs`, the DVFS procedure includes file management which is expected to introduce more overhead than calling the kernel headers directly from the application. We studied, however, both options in order to validate the latency differences between the user space interface and the system call.

a) System call interface: The system call interface for DVFS under Linux is accessible directly in the Linux kernel. We measured the elapsed time between issuing the DVFS system call and the return of the call which indicates a change in clock frequency. Listing 1 outlines the pseudo code for accessing the DVFS functionality from the system call interface.

```
#include <cpufreq.h>
#include <sys/time.h>
latency_syscall(){
  gettimeofday(&time1);
  cpufreq_set_frequency(Core0, FREQ1);
  gettimeofday(&time2);
  cpufreq_set_frequency(Core0, FREQ2);
  gettimeofday(&time3);
}
```

Listing 1. Pseudo code for measuring DVFS latency using system calls

b) User space interface: The second option is to use the `sysfs` interface for accessing the DVFS functionality from user space. The CPU clock frequency is altered by writing the frequency to be used from now on into a `sysfs` file, which is read and consequently used to change the frequency. The kernel functionality is not directly called from the c-program, but file system I/O is required for both reads and writes to the `sysfs` filesystem. Listing 2 outlines an example for the DVFS call via the `sysfs` interface.

```
#include <sys/time.h>
latency_sysfs(){
  gettimeofday(&time1);
  system("echo FREQ1 > /sys/devices/system/cpu/cpu0/cpufreq/
       scaling_setspeed");
  gettimeofday(&time2);
  system("echo FREQ2 > /sys/devices/system/cpu/cpu0/cpufreq/
       scaling_setspeed");
  gettimeofday(&time3);
}
```

Listing 2. Pseudo code for measuring DVFS latency using `sysfs`

2) DVFS Measurement results: The user space and the kernel space mechanisms were evaluated, and the results are presented in this section. Since the DVFS mechanism is ultimately executed on kernel- and user space threads, the system should be stressed using different load levels to evaluate the impact on the response time. For this purpose, we used `spurg-bench` [23]. `Spurg-bench` is a benchmark

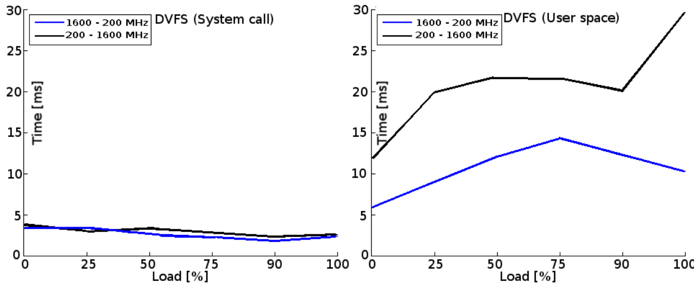

Fig. 1. Average latency for changing clock frequency under different load conditions using system call and `sysfs` mechanisms

capable of generating a defined set of load levels on a set of threads executing for example floating point multiplications. We generated load levels in the range [0;90]% using `spurg-Bench`. Furthermore we generated a load level of 100% using `stress`[1] since this benchmark is designed to represent the maximum case CPU utilization. All experiments were iterated 100 times with different frequency hops, and with a timing granularity of microseconds. We used an Exynos 4412 SoC with an ARM core capable of clock speeds in the range [200;1600] MHz.

Figure 1 shows the average latency for all load levels and with frequency hops from 1600 to 200 MHz and from 200 to 1600 MHz. When using the system call interface, the average latency decreases slightly when increasing the load (left part of Figure 1). On the other hand, the switching latency has a strong correlation to current frequency and target frequency in the `sysfs` implementation. The measurements of the `sysfs` interface show a latency increase until the load is roughly 60% after which it slightly declines and finally increases when stressing the CPU to 100%. As expected, the latency is shorter as the CPU frequency jumps from 1600 to 200 MHz because most of the DVFS procedure (including the file system call) is executed on the higher frequency. Table I shows the standard deviation from samples in the same experiments. The `sysfs` experiments show a much higher standard deviation because of filesystem I/O when accessing the `sysfs` filesystem.

B. Dynamic Power Management (DPM)

In older generation microprocessors, most of the power was dissipated by switching activities in the chip (dynamic power). In recent years, the static power has, on the other hand, become more dominant [16], and is even expected to dominate the total power dissipation in next generation microprocessors [22]. The static power is dissipated due to leakage currents through transistors, which is mostly a result of subthreshold and gate-oxide leakage [18]. Equation 2 shows the subthreshold leakage current

$$I_{sub} = K_1 \cdot W \cdot e^{-V_{th}/n \cdot V_\theta}(1 - e^{-V/V_\theta}) \tag{2}$$

where K_1 and n are architecture specific constants, W is the gate width and V_θ is the thermal voltage. Hence, the silicon temperature causes an exponential increase in leakage currents [8]. Moreover, when lowering the supply voltage of integrated circuits, the subthreshold leakage current increases which also increases the dissipated static power [2], [30]. The leakage current is present as long as the chip (or parts of the chip) is connected to a power source. This means that in order to reduce the leakage current, parts of the chip must be disconnected from the power source and re-connected as the functionality is required again.

CPU sleep states (or DPM) are used to disable parts of the CPU on demand to decrease the static power consumption. The functionality is accessed in Linux by the CPU hotplug facility, which was originally implemented to replace physical CPUs during run-time. On our test platform the hotplug functionality places the core in a Wait For Interrupt (`wfi`) state in which the core clock is shut down, and re-activated as soon as the core receives an interrupt from another core. The functionality of hotplugging a core differs depending on the state of the core designated to be turned off. In case the core is executing workload, the mechanism re-allocates the workload to another core in order to make it idle. In case the core is idle this action is not required. The hotplug functionality can be accessed in Linux either as a kernel space module or directly from user space using the `sysfs` interface.

The hotplug implementation consists of a platform-independent part and a platform-specific part, which lastly calls the CPU specific assembly routines for accessing the hotplug. The following procedure describes how the hotplug mechanism is used to shut down a core:

1) A core shutdown command is issued in the system
2) The system locks the core with a mutex in order to block tasks from being scheduled to this core
3) A *notification* is sent to the kernel:
CPU_DOWN_PREPARE
4) A kernel thread executes a callback function and receives the notification
5) Tasks are migrated away from the core being shut down
6) A *notification* is sent to the kernel: CPU_DEAD
7) A kernel thread executes a callback function and receives the notification
8) Interrupts to the core are disabled, the cache is flushed and the cache coherency is turned off
9) Power source is removed and core is physically shut down

As seen from the procedure, the shutdown of a core is reliant on callback functionalities in the Linux kernel, which means that the system performance and current utilization will affect the response time of the scheduled kernel thread issuing this functionality. As suggested in [7], improvements can be made to decrease the hotplug latency but the current main stream kernels still rely on the aforementioned callback facilities. The wake-up procedure is, similarly to the shutdown procedure, dependent on callbacks but with an inter-core interrupt to trigger the core startup.

[1] http://people.seas.harvard.edu/ apw/stress/

TABLE I. STANDARD DEVIATION OF DVFS LATENCY USING SYSTEM CALL AND `SYSFS`

Load	0%	25%	50%	75%	90%	100%
1600-200MHz						
System call	2%	5%	5%	6%	6%	8%
sysfs	27%	28%	40%	26%	20%	30%
200-1600MHz						
System call	3%	6%	6%	8%	6%	6%
sysfs	25%	30%	34%	39%	29%	25%

1) CPU hotplug implementations: Two separate experiments were conducted to determine the latency of CPU hotplug from kernel space and user space. Similarly to the DVFS measurements, we measured the elapsed time between issuing the shutdown/wake-up call and the return of the call.

a) Kernel space module: In the first implementation we accessed the CPU hotplug functionality directly in a kernel module which Linux executes in kernel space with closer access to the hardware. Listing 3 outlines the functionality of accessing the CPU hotplug in kernel space.

```
#include <linux/cpu.h>
#include <linux/time.h>
latency_kernel(){
  mutex_lock(&lock);
  do_gettimeofday(&time1);
  cpu_down(1); //Core 1 is shut down
  do_gettimeofday(&time2);
  mutex_unlock(&lock);
  mutex_lock(&lock);
  do_gettimeofday(&time3);
  cpu_up(1); //Core 1 is waken up
  do_gettimeofday(&time4);
  mutex_unlock(&lock);
}
```

Listing 3. Pseudo code for measuring hotplug latency in kernel module

b) User space interface: The second mechanism for accessing the CPU hotplug functionality was implemented as a normal user space application accessing `sysfs` files. The benefit of using the user space functionality rather than the kernel space is a significantly simpler implementation and misbehavior in user space will be intercepted safely by the kernel rather than causing system crashes. The downside is an expected higher latency for accessing the hardware due to file system I/O and kernel space switches. Listing 4 outlines the functionality of accessing the CPU hotplug in user space.

```
#include <sys/time.h>
latency_user(){
  gettimeofday(&time1);
  system("echo 0 > /sys/devices/system/cpu/cpu1/online");
  gettimeofday(&time2);
  system("echo 1 > /sys/devices/system/cpu/cpu1/online");
  gettimeofday(&time3);
}
```

Listing 4. Pseudo code for measuring hotplug latency in user space

2) CPU hotplug results: Similarly to the DVFS experiments, we stressed the system with different load levels using `spurg-bench`. The system was running on a selected range of clock frequencies and the timings were measured on microsecond granularity.

Figure 2 shows the average latency for shutting down a core in kernel- and user space respectively. The axes of the figures have been fixed in order to easily compare the different configurations and implementations. From Figure 2 it is clear that the average latency for shutting down a core is rather constant in kernel space and not significantly dependent on clock frequency. The user space implementation is more

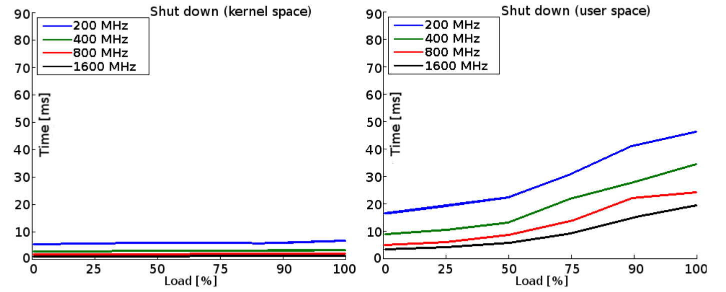

Fig. 2. Average latency for shutting down a core under different load conditions using kernel and userspace mechanisms

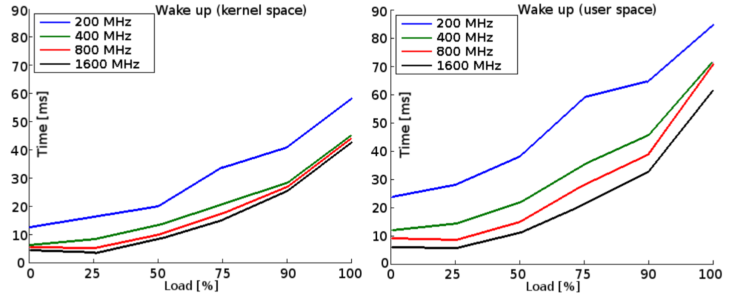

Fig. 3. Average latency for waking up a core under different load conditions using kernel and userspace mechanisms

dependent on the load level as the latency doubles between 0% load and 100% load.

On the contrary, the wake-up time is dependent on the load level in both the kernel space and the user space case. As seen in Figure 3 (left), the kernel space implementation measures up to 6x higher latency for the 100% load case compared to the 0% load case. A similar ratio is seen in the user space implementation, but the latency is on average roughly 2x higher than the kernel space implementation. Similarly to the shutdown procedure, the wake-up is also dependent on several kernel notifications followed by kernel callbacks. An additional factor in waking up a core is the long process of creation and initialization of kernel threads (kthreads), which are required to start the idle loop on a newly woken-up core. Only after the kthread is running, the CPU mask for setting a core available can be set.

Since this work focus on user space implementations with static scheduling, we chose to access the hotplug functionality from the `sysfs` interface in our evaluation. However we acknowledge that a more optimized solution is possible by embedding parts of the functionality in kernel space. Table II shows the standard deviation for shutdown and wake-up from the experiments.

TABLE II. STANDARD DEVIATION OF DPM LATENCY IN KERNEL- AND USERSPACE

Load	**0%**	**25%**	**50%**	**75%**	**90%**	**100%**
Shut-down						
Kernelspace	3%	8%	9%	11%	5%	9%
userspace	3%	9%	11%	12%	16%	15%
Wake-up						
Kernelspace	7%	20%	26%	27%	23%	4%
userspace	8%	13%	18%	17%	18%	28%

IV. Energy Model

An energy model is used to predict the energy consumption of the system using a set of fixed parameters. The input to our model are descriptions of the workload and the target architecture. The workload is represented by the number of instructions w to be executed, and the deadline D before which the workload must be processed as illustrated in Figure 4. In case the workload is processed sufficiently prior to the deadline, the cores can be shut down or scaled down with a given overhead penalty, and woken up or scaled up again prior to reaching the deadline, so that the initial setting is restored.

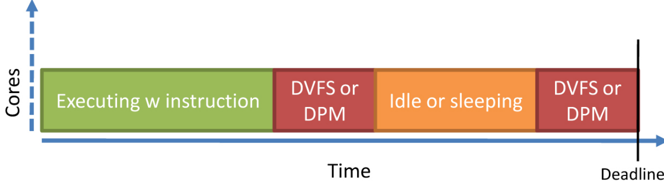

Fig. 4. Execution of workload before a given deadline

As we target compute-bound applications, we do not have to care for long I/O latencies and thus the time t to process the workload is considered inversely proportional to the core frequency. For simplicity we assume that the proportionality factor is 1, i.e. one instruction per cycle is processed on average. Thus:

$$t(w, f) = w/f$$

Let t_{min} be the time to process the workload at maximum possible speed, we then call $D/t_{min} \geq 1$ the pressure. If the pressure is close to 1, the CPU must run on maximum speed to meet the deadline. If the pressure is e.g. 2, the CPU can either run at half the maximum speed for time D, or run at maximum speed until time $D/2$, and then idle or shut down the cores until the deadline. We assume a workload consisting of a large number of small, independent tasks, so that the workload can be balanced among the cores.

The target architecture is a many-core CPU represented by p cores with frequencies $f_1, \dots, f_k$, together with the power consumption of the chip at each frequency in idle mode and under load, i.e. $P_{idle}(j, f_i)$ and $P_{load}(j, f_i)$, where $1 \leq j \leq p$ denotes the number of active cores. As we target compute-bound applications, we assume that the cores are stressed to 100% load, however, extensions for power consumptions at different load levels are possible. We further assume that all cores run at the same frequency, because that is a feature of our target architecture.

From previous sections we have obtained the latency t_{scale} of switching frequencies of the cores (the cores are assumed to be idle) from f_{i1} to f_{i2}, the power consumption during this time is assumed to be the average of $P_{idle}(f_{i1})$ and $P_{idle}(f_{i2})$. While t_{scale} might vary depending on the frequencies, the model confines it to an average value as a close enough approximation. An idle core at frequency f_i can be shut down, and later wake up again. We consider these consecutive events, as the cores must be available after the deadline. The total time for shutdown and wake-up is denoted by $t_{shut}(f_i)$, we assume that the core consumes idle power during this time[2].

If each of the j active cores, where $1 \leq j \leq p$ has to process the same workload w/j until the deadline, the cores must run at least at frequency $f_{min} = (w/j)/D$. Hence they can utilize any $f_i \geq f_{min}$ to process the workload in time $t_i = (w/j)/f_i$.

There are several possibilities to process the workload w: **1)** For any number j of active cores, the cores can run at any frequency $f_i \geq f_{min}$ for time t_i to process the workload and then idle at the same frequency for time $D - t_i$ consuming total energy:

$$E_1(j, f_i) = t_i \cdot P_{load}(j, f_i) + (D - t_i) \cdot P_{idle}(j, f_i) \tag{3}$$

2) The idle cores could also be scaled down to the lowest frequency f_1 if $D - t_i$ is larger than t_{scale}, with a resulting energy consumption of

$$\begin{aligned} E_2(j, f_i) = t_i \cdot P_{load}(j, f_i) + t_{scale} \cdot P_{idle}(j, f_i) + \\ (D - t_i - t_{scale}) \cdot P_{idle}(j, f_1) \end{aligned} \tag{4}$$

3) Finally, the cores could be shut down after processing the workload, and wake up just before the deadline, if $D - t_i \geq t_{shut}(f_i)$. In our target architecture, the cores must be shut down in sequence. This follows the implementation in the mainline Linux kernel which executes the shut down/wake up call in kernel threads scheduled by the kernel. The first core must remain active idle to wake the others up. Consider q being the cores to shut down: If $D - t_i \geq q \cdot t_{shut}(f_i)$ but $D - t_i < (q + 1) \cdot t_{shut}(f_i)$, then at most $q < p$ cores can be shut down. The consumed energy can be modeled as:

$$\begin{aligned} E_3(j, f_i) = t_i \cdot P_{load}(j, f_i) + \sum_{l=j-q+1}^{j} t_{shut} \cdot P_{idle}(l, f_i) + \\ (D - t_i - (j - q)t_{shut}) \cdot P_{idle}(1, f_1) \end{aligned} \tag{5}$$

where l is the $(j - q)$:th core to shut down out of the j active cores. The last part of the equation $(D - t_i - (j - q)t_{shut})$ is the time the remaining active cores $(j - q)$ idle in case not shut down.

Having formulated the model, and given a concrete workload, we enumerate all feasible solutions prior to execution, and choose the one with the lowest energy consumption. Hence we create a static schedule, i.e. a balanced mapping of the tasks onto the cores together with information about core speeds and necessary frequency scalings or shutdowns. If the target architecture has some other characteristics such as concurrent shutdown of all cores, the energy formula can be adapted, e.g. to $E_3(j, f_i) = t_i \cdot P_{load}(j, f_i) + t_{shut} \cdot P_{idle}(j, f_i)$. Also, the number of solutions might increase, e.g. if the cores can be run at different frequencies. However, the core algorithm design remains, as the number of feasible solutions is still small enough for enumeration. The model can also be refined to scale the frequency of the cores prior to shutdown to a frequency level with a more favorable shutdown time.

[2] Others [6] assume power under load during shutdown. We have tested this, but found that in Table V it will not lead to differences except in one corner case where best and 2nd best configurations switch their places. However, both are very close together, so that slight variation of any parameter might promote one or the other configuration

TABLE III. POWER DISSIPATION (IN WATTS) FOR THE EXYNOS 4412 UNDER FULL WORKLOAD. ROWS ARE THE NUMBER OF ACTIVE CORES AND COLUMNS ARE CLOCK FREQUENCY

	200	**400**	**600**	**800**	**1000**	**1200**	**1400**	**1600**
1	2.875	3.02	3.095	3.16	3.315	3.43	3.675	3.955
2	2.975	3.125	3.275	3.375	3.55	3.775	4.22	4.715
3	3.045	3.305	3.45	3.65	3.85	4.225	4.935	5.71
4	3.105	3.365	3.6	3.845	4.185	4.745	5.795	7.615

TABLE IV. POWER DISSIPATION (IN WATTS) FOR THE EXYNOS 4412 UNDER IDLE WORKLOAD. ROWS ARE THE NUMBER OF ACTIVE CORES AND COLUMNS ARE CLOCK FREQUENCY

	200	**400**	**600**	**800**	**1000**	**1200**	**1400**	**1600**
1	2.148	2.162	2.173	2.139	2.048	2.035	2.143	2.284
2	2.152	2.163	2.179	2.133	2.11	2.057	2.202	2.381
3	2.156	2.167	2.183	2.146	2.122	2.08	2.279	2.407
4	2.158	2.173	2.181	2.155	2.172	2.105	2.33	2.503

In contrast to an analytic model, we do not have to make assumptions of convexity and continuity of power functions etc., which are often not true in practice, as well as the distinction between idle power and power under load. Yet, the model still uses optimizations, such as a non-decreasing power function with respect to frequency and number of active cores. For example, we do not scale frequencies while processing workload.

A. *Model based simulation*

The model based simulation was, as mentioned, based on real-world measurements using different numbers of cores and different clock frequencies. The system parameters were obtained from the quad-core Exynos 4412 platform by measuring the power under full load P_{load} using the `stress` benchmark shown in Table III. Similarly, the idle power P_{idle} was measured for each configuration in a completely idle system shown in Table IV. These power values were used to create the power model used in the simulator.

We scheduled task sets with 10k, 100k and 1M synthetic jobs with pressure levels 1.1, 1.3, 1.5 and 4.0 in order to determine the *best*, *2nd best* and *worst* energy efficient configuration parameters. The number of instructions of each job were randomly chosen to obtain a [0;500ms] runtime normalized to the highest clock frequency.

The scheduler used four threads for execution, which models a scalability up to four cores. We executed the scheduler with different parameters, and the output shows the possible scheduling configurations which meet the deadline. Table V shows the configuration settings for three chosen outputs: best, 2nd best and worst energy efficiency with the aforementioned power values and scheduling parameters. The output is a working frequency combined with a power management feature: *DPM*, *DVFS* or *idling* the whole slack time.

Since a pressure of 1.1 poses a very tight deadline for the jobs, the only feasible clock frequency setting is 1600 MHz. The best case with this parameter uses rather DVFS than DPM because of a faster switching time which costs less energy. For pressure levels > 1.1 a more relaxed deadline allows slower execution speed, which impacts significantly on the dynamic power dissipation. Hence, the best case uses DPM rather than DVFS since the more relaxed deadline allows a longer sleep time, which reduces the energy consumption more than the cost of activating the DPM mechanism. Finally the pressure level 4.0 – with a very relaxed deadline – allows the system to execute on 1000 MHz, which is the most energy efficient clock frequency. Because of the execution model illustrated in Figure 4, the number of jobs does not affect the usage of the power management techniques, since the clock frequency scaling or core shutdown is always executed only once after the workload finishes.

Finally, in Table V we only list configurations that use all four cores since this is be most energy efficient configuration for all settings in this particular platform. The model predicts an identical usage of power saving techniques independently of the number of jobs. This is predicted since the number of job only affects the total execution time but not the relation to the pressure. When demanding a low pressure, the clock frequency is forced to a high value in order to keep the deadline, and DPM introduces a too large overhead to be energy efficient. When gradually relaxing the pressure with higher values, DPM is favored over DVFS and the clock frequencies can be lowered to save energy.

TABLE V. CONFIGURATION PARAMETERS FOR DIFFERENT NUMBER OF JOBS AND DIFFERENT PRESSURE LEVELS

			Number of jobs		
		Config	10k	100k	1M
Pressure	1.1	Best	1600MHz+DVFS	1600MHz+DVFS	1600MHz+DVFS
		2nd Best	1600MHz+DPM	1600MHz+DPM	1600MHz+DPM
		Worst	1600MHz+IDLE	1600MHz+IDLE	1600MHz+IDLE
	1.3	Best	1400MHz+DPM	1400MHz+DPM	1400MHz+DPM
		2nd Best	1400MHz+DVFS	1400MHz+DVFS	1400MHz+DVFS
		Worst	1600MHz+IDLE	1600MHz+IDLE	1600MHz+IDLE
	1.5	Best	1200MHz+DPM	1200MHz+DPM	1200MHz+DPM
		2nd Best	1200MHz+IDLE	1200MHz+IDLE	1200MHz+IDLE
		Worst	1600MHz+IDLE	1600MHz+IDLE	1600MHz+IDLE
	4.0	Best	1000MHz+DPM	1000MHz+DPM	1000MHz+DPM
		2nd Best	1000MHz+DVFS	1000MHz+DVFS	1000MHz+DVFS
		Worst	1600MHz+IDLE	1600MHz+IDLE	1600MHz+IDLE

V. REAL-WORLD VALIDATION

To validate the model presented in Section IV we executed real-world experiments to compare the final energy consumption with the mathematical representation.

A. *Experimental setup*

We replicated the scenarios described in Section IV by constructing experiments with: **a)** a set of configuration parameters used to time trigger the hardware power saving features according to the scheduler results **b)** one or more benchmark threads executing a selected job for a given time. In one example configuration, the benchmark executes on four threads on 1600 MHz for n milliseconds after which the clock frequency is reduced to 200 MHz for m milliseconds or the cores are shut down until the deadline. We chose `stress` as the benchmark since it was used to train the mathematical model and its behavior is easily repeatable. `stress` is also a good candidate for representing CPU intensive workloads.

We used the quad-core Exynos 4412 platform on which the experiments were earlier conducted. In order to measure the power of the Exynos board and to not disturb its performance, we added an external Raspberry Pi board to monitor the board power consumption similar to the work in [11]. Figure 5 shows the complete workflow for time synchronization and power

measurements:

1) Raspberry Pi sends a start signal over UDP to Exynos
2) Raspberry Pi starts to log the power measured by an INA226 A/D converter connected to its i2c bus
3) Exynos starts the benchmark threads
4) Exynos scales frequency or shuts down cores if requested
5) Exynos finishes benchmark and sends stop signal over UDP
6) Raspberry Pi ends power monitor

Figure 6 shows the real implementation of the power measurement device consisting of the INA226 A/D converter allowing the Raspberry Pi to sample the power values of the odroid board.

In order to get an average power log as accurate as possible, the additional overhead including 2 ms ping latency between the Raspberry Pi and the Exynos was accounted for and excluded in the power monitor.

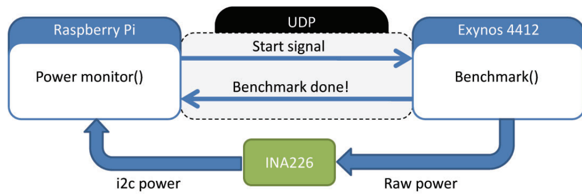

Fig. 5. Raspberry Pi connected to Exynos board with A/D converter to measure power and send data to Raspberry-Pi

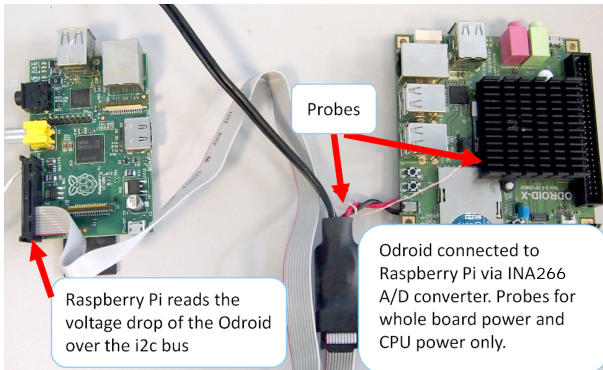

Fig. 6. Real-world implementation of power measurement device. The power feed of the odroid board (right hand side) is connected via the INA226 A/D converter (middle) to the Raspberry Pi board (left hand side).

B. Experimental Results

We used the task sets with 10k, 100k and 1M jobs from the previous section. The respective execution times for executing the jobs were measured and used in the benchmark framework. Furthermore, we also evaluated pressure levels: 1.1, 1.3, 1.5 and 4.0 for all task sets. For each combination of task set and pressure levels the *best case*, *2nd best case* and *worst case* energy scenarios were compared against the mathematical model.

Figure 7 shows results for 10k jobs. Both the best case model and data show a high energy consumption for low pressure and for high pressure; the lowest energy consumption is achieved at a pressure level $P = 1.5$ for both data and model. This is a result of low pressure levels pushing the

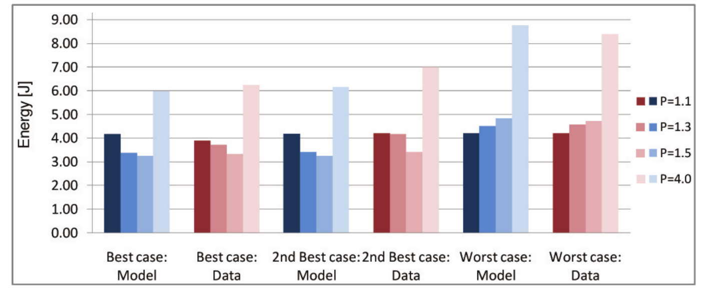

Fig. 7. Energy consumption for model and data running 10k jobs with different pressure settings

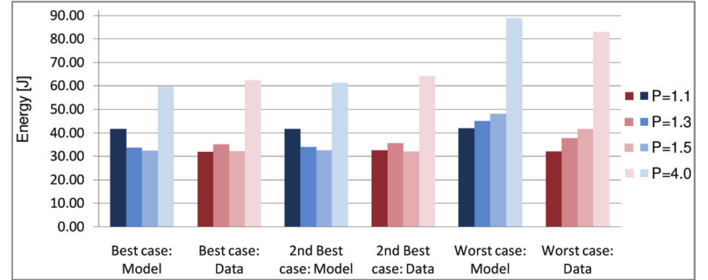

Fig. 8. Energy consumption for model and data running 100k jobs with different pressure settings

deadline very close and the CPU is hence forced to execute on a high clock frequency to meet the deadline. Even though the execution time is short, the dynamic power dissipated when executing on the highest frequency results in high energy consumption. Large values for pressure also result in high energy consumption since deadline and the execution time becomes very long and the ever present static power significantly increases the energy consumed even though the clock frequency is low.

Figure 8 shows the results of the benchmarks with 100k jobs. The relation between data and model are rather similar to Figure 7 with the exception of pressure level $P = 1.1$. This case has a higher prediction than the actual measured data. As previously explained, a low pressure level forces a high clock frequency – and running the CPU on the maximum frequency for a long time activates the thermal throttling of the CPU as it reaches a critical temperature. The CPU is then temporarily stalled resulting in lower power dissipation, which leads to low energy consumption. Naturally by using CPU throttling, fewer operations are executed in the benchmark which causes poor performance. The situation is avoided by adding active cooling, but we chose to acknowledge this anomaly in the mathematical power model of a thermally limited chip.

Figure 9 shows the results from the longest running experiments, i.e. 1M jobs. Similarly to Figure 8 the thermal throttling of the CPU causes a misprediction of the model.

The mean squared error between data and model is finally shown in Figure 10 for all previously mentioned experiments. The figure shows the largest misprediction in cases with P=1.1 and for long running jobs (1M and 100k). As previously concluded, the misprediction is mostly caused by the CPU thermal throttling activated when running the CPU on maximum frequency for a long time (in range 10s of seconds). Furthermore, the thermal throttling is occasionally

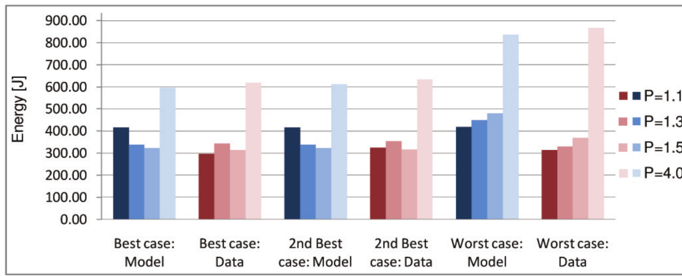

Fig. 9. Energy consumption for model and data running 1M jobs with different pressure settings

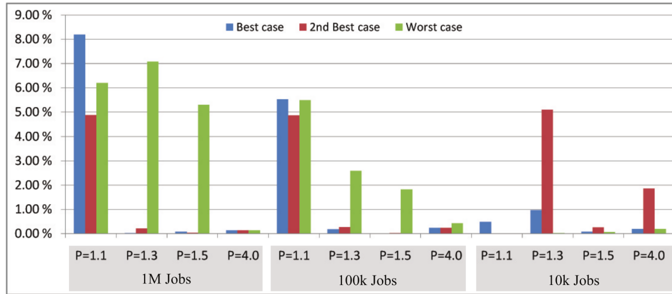

Fig. 10. Error squared between model and data for all task sets and pressures

also activated when running the CPU on the second highest frequency for a very long time (several minutes) as can be seen in 1M case with P=1.3. Hence, the model remains accurate as long as the CPU remains within its thermal bounds.

C. Dependence of Model on Benchmark

In order to evaluate the level of dependency between different types of workload, a completely separate benchmark was used to compare the power dissipation to the `stress` benchmark used to train the model. All configurations similar in Table V was used together with a second benchmark i.e. the best, second best and worst case execution given by the simulator. `Whetstone` [4], which primarily measures floating-point arithmetic performance, was used in the second round of experiments. The benchmark was chosen since this has been a traditional metric of determining floating-point performance in computer systems.

The results in form of squared error from the second benchmark is shown in Figure 11. Similarly to the previous cases shown in Figure 10, the scenario with many jobs and low pressure values forces the CPU to execute at a high clock frequency for a long time. This increases the temperature on the chip beyond the safety limit, and the chip is finally stalled as a safety precaution. For medium size work (100k jobs) the data aligns even better with the model than using the `stress` benchmark, and the short size work (10k jobs) are within a few percentages margin to the model.

The influence of changing the workload was hence not considered to significant strive from the model except for in the extreme cases with many jobs and a low pressure value. The model was hence accepted as a common use-case representation of workload execution. We conclude that the forecasting capability of the model is only weakly dependent

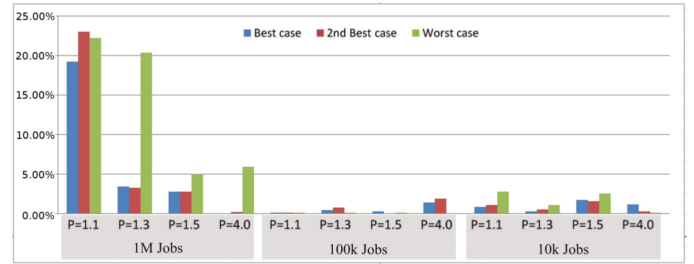

Fig. 11. Error squared between model and data for all task sets and pressures using the `whetstone` benchmark

on the concrete workload used to compute the best scenarios.

VI. Optimizing for Multiple Rounds

We have so far investigated applications comprised of a set of tasks with a common, single deadline. We call the execution of all tasks, together with possible power management, a (schedule) round. Streaming applications consist of a set of tasks communicating with each other, where each task can be assumed to be invoked once during one scheduling round, and the communication between tasks is assumed to occur between the run of the sending task in one round, and the start of the receiving task in the next round. Hence, within one round the tasks may be considered to be independent, and a static scheduling with deadline per round be applied to achieve (soft) real-time behaviour of the streaming application. Streaming applications model a large class of parallel applications, e.g. in image processing, and static scheduling is regularly used to schedule them, cf. e.g. [26]. In a simple example, a first task performs preprocessing on an image, two further tasks detect horizontal and vertical edges, respectively, and a final task completes the analysis or transformation of the image. In each round, the first task will receive a new image to process, and between successive rounds the tasks will forward their results to the follow-up tasks. Thus, a streaming application will execute a long sequence of similar rounds with similar deadlines, and for a given mapping of the tasks to cores, the energy model from Section IV can be applied to minimize the energy consumption of each round.

A. Theoretical framework

The information regarding a large number of similar rounds can be used to further optimize the energy consumption per round. The most obvious part is to consider two successive rounds, see Figure 12 top. In case the idle time or shutdown time in the second round is moved from the end to the beginning of the round, we consider the two rounds as a “back to back” configuration as indicated in Figure 12 middle. In this case, the overhead of scaling up and down or shutting down and waking up is only necessary once per two rounds, instead of twice as before. Note that despite this change, the deadline D per round is still respected, i.e. there is no change in the real-time aspects.

Even more can be gained if we allow to relax the real-time behaviour within a certain limit, e.g. if two rounds must be processed until deadline $2D$, and the first round is allowed to use more than time D. In this case we can combine

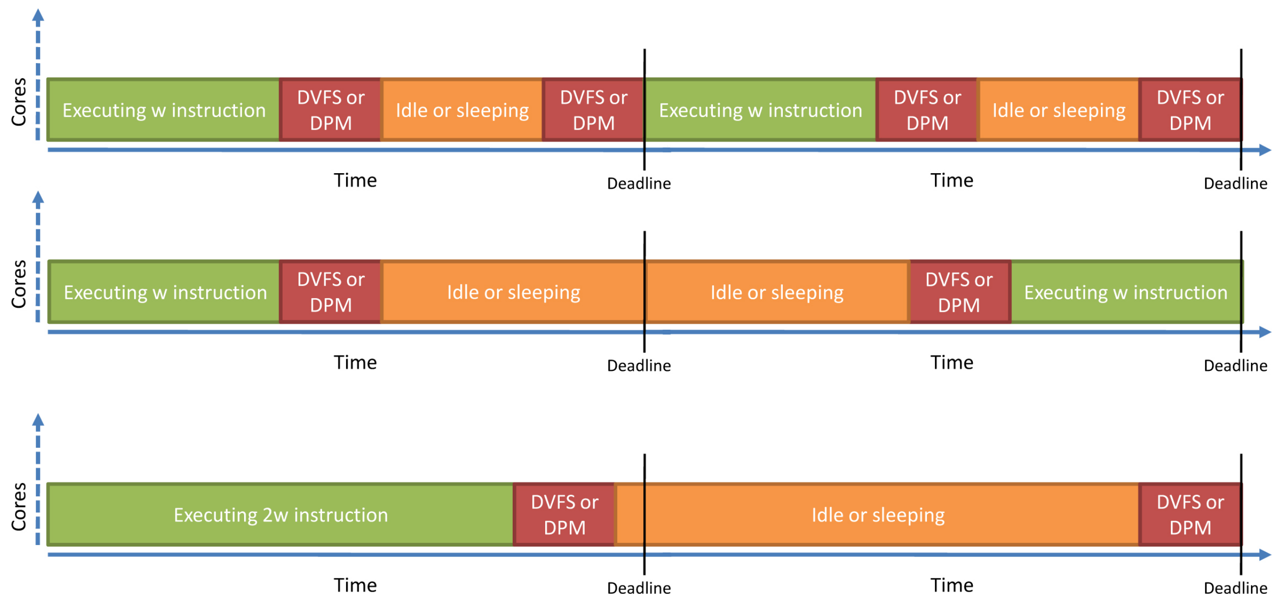

Fig. 12. Schedule for two successive rounds of computation. Top: without further optimization. Middle: with combined idle times. Bottom: combined workload processing.

the workloads of the two rounds and process them in one execution, as illustrated in Figure 12 bottom. This optimization is more general than the previous one, as it can be used also for $k > 2$ rounds, in case the application has enough flexibility in the deadline. It can also be combined with the previous optimization, i.e. the second k rounds are executed with idle or sleeping time first, and the idle and sleeping times of the two sets of k rounds are combined into one. But even for $k = 2$, and without combination with the back-to-back optimization, the latter optimization might bring advantages over the former. For example, if we imagine an example in which the idle time in one round is not long enough to shutdown a core, but where the combined idle times of two rounds are long enough to do so. The optimization can then increase the number of possible solutions and thus be used for saving more energy. As the solutions of the back-to-back optimization are always included in the solution space of the $k = 2$-round combination, applying the latter can never, in theory, lead to a disadvantage.

We have integrated the k-round optimization into our energy model, and illustrated its potential and limits with a number of examples. If we process 10 tasks (each task requiring 250 ms at highest frequency) and a pressure of 5.0, then for a single round, the lowest energy per round is obtained by processing the tasks at 1000 MHz with 4 cores. After this, the cores are scaled down to 200 MHz when idling until they are scaled back again to 1000 MHz prior to the deadline. Although the idle time is longer than the processing time, the round, in this case, is so short that it is not possible to shut down the cores. If we consider $k = 3$ rounds together, the lowest energy per round is obtained by:

1) Processing the tasks from all rounds at 1000 MHz using 4 cores
2) Shutting down three of the cores
3) Making the remaining core idle
4) Waking up the three other cores again prior to the deadline

The forecasted reduction of energy per round using our power model is then about 0.85%. While this reduction seems small, we have to consider that streaming applications are often long-running, and in energy-constrained environments (e.g. lack of active cooling), so that even small improvements help. Furthermore, the combination with the former optimization (consider two sets of rounds "back to back") would increase the energy reduction per round to roughly 2.05%, and combining more rounds (if possible) decreases the energy consumption more according to the number of rounds combined.

The limitation of this method can be seen in an example with 1000 tasks and a pressure of 1.36. In one round, the lowest energy consumption is achieved by executing 4 cores at 1200 MHz until the workload is processed. After this, scale the cores down to 200 MHz and idle until they are scaled up again to 1200 MHz prior to the deadline. When we consider $k = 2$ rounds together, it is possible to shutdown one of the 4 cores after processing the workload at 1200 MHz, but not enough time is left to scale the remaining 3 cores down, let alone shutting off another core before the 4th core must be woken up again prior to the deadline. In this case, the idle time is very short compared to the processing time, so that energy savings from optimizing this phase only comprise a tiny fraction of the total energy consumption: 0.02% in the reported case.

B. Hardware limitations

The previously mentioned benchmark setup can unfortunately not be replicated on the current platform in practice. The reason is because the very fine grained timings cannot be realized on a Linux platform without including scheduling overhead influencing the reliability of the results.

Firstly, the external INA266 A/D converter used for sampling the power dissipation cannot register enough values in the very short time span. A higher sampling rate could,

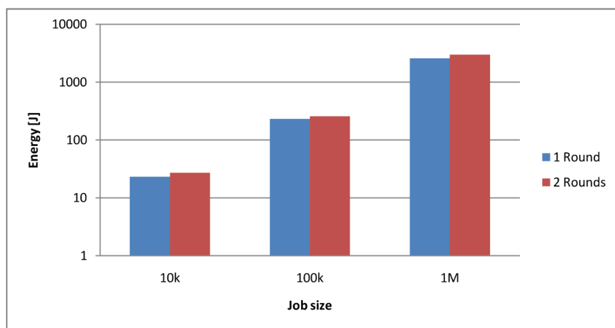

Fig. 13. Energy consumption of executing one two-round execution compared to executing two one-round execution

however, be selected, but with a consequence of a decrease in measurement accuracy due to physical limitations. Another suggestion would be to iterate one benchmark run several times to increase the measurement time and thus increase the accuracy of the outcome. The problem with this approach is the added overhead of creating and killing pthreads under Linux. Killing the benchmark thread is necessary in cases the pressure is larger than 1.0 in order for the system to remain idle until the deadline. With a very fine grained timing, the overhead with regard to the operating system becomes the same order of magnitude as the benchmark itself, and the outcome looses its accuracy. The overhead in thread creation is so high that for a workload much smaller than 10,000 tasks, it dominates the time of the round.

We evaluated, however, a system using job sizes of 10k, 100k and 1M and the pressure of 1.5 using the two-round approach illustrated in Figure 12 (bottom). Even though the granularity becomes larger, the theoretical framework suggests that the energy consumption would decrease due to the smaller amount of frequency scalings and usage of sleep states. The results shown in Figure 13 (logarithmic scale) shows, on the contrary, a higher energy consumption for the two-round approach in each case. The reason for this is by running two long rounds back to back, the core temperature increases more compared to running them one by one with idle or sleep phases in between. The increase in temperature increases the static power consumption (for the same core frequency), and the energy savings are diminished.

In order to benefit from multiple-round executions, the timing granularity should be finer for example by using a real-time OS with much lower scheduling overhead. The idea should also be explored on other platforms which employ active cooling to reduce the influence of increased thermal dissipation.

VII. Conclusions

As the hardware becomes more complex and the manufacturing techniques shrink, accurate power consumption details for multi-core systems are difficult to derive from an analytical mathematical approximation. An alternative is to model the system top-down based on real experiments and include practical aspects such as such as power management overhead, which cannot be ignored for applications with deadlines in the millisecond range. We have presented an energy model derived from real-world power measurements of benchmarks including power management latencies from a general purpose operating system. The model is used to calculate an energy-optimal static schedule for applications with a given deadline. It was validated with experiments on real hardware and we demonstrated its accuracy and its independence from the particular benchmark chosen to derive the model. We obtained the practical timing granularity for DVFS and DPM after which the latency of power saving techniques cannot longer be neglected.

Furthermore, we have extended the model towards streaming applications and presented further opportunities for energy reduction by scheduling several scheduling rounds together. Optimizing static schedules by merging execution phases can result in theoretical energy savings for very fine grained timing granularities. Merging long running executions, on the other hand, increases the energy consumption because of the significant increase in working temperature on the chip. We can conclude that high performance software should preferably not execute uninterrupted for a long time on passively cooled systems such as mobile phones. Instead, by dividing the execution into shorter phases, static power is decreased by the lower working temperature.

In future work, we would like to extend our model to Intel-based multi-core platforms with independent core frequencies, and to heterogeneous platforms such as the big.LITTLE systems. By using core independent frequency levels the model must coordinate both the location of the running task and the clock frequency of the core possibly by defining a dynamic power model. The heterogeneous platform must further also define the type of core which leads to both a dynamic power model and a dynamic performance model of the currently used core type. Moreover, we would like to extend the model to multiple applications, and to applications with partly stochastic behavior.

References

[1] K. Bhatti, C. Belleudy, and M. Auguin. Power management in real time embedded systems through online and adaptive interplay of dpm and dvfs policies. In Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on, pages 184–191, 2010.

[2] S. Borkar. Design challenges of technology scaling. Micro, IEEE, 19(4):23 –29, jul-aug 1999.

[3] S. Cho and R. Melhem. On the interplay of parallelization, program performance, and energy consumption. Parallel and Distributed Systems, IEEE Transactions on, 21(3):342–353, 2010.

[4] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. Computer Journal, 19:43 – 49, February 1976.

[5] A. Gandhi, M. Harchol-Balter, and M. Kozuch. Are sleep states effective in data centers? In Green Computing Conference (IGCC), 2012 International, pages 1–10, June 2012.

[6] M. E. T. Gerards and J. Kuper. Optimal DPM and DVFS for frame-based real-time systems. ACM Transactions on Architecture and Code Optimization, 9(4), 2013.

[7] T. Gleixner, P. E. McKenney, and V. Guittot. Cleaning up linux's cpu hotplug for real time and energy management. SIGBED Rev., 9(4):49–52, Nov. 2012.

[8] F. Hällis, S. Holmbacka, W. Lund, R. Slotte, S. Lafond, and J. Lilius. Thermal influence on the energy efficiency of workload consolidation in many-core architecture. In R. Bolla, F. Davoli, P. Tran-Gia, and T. T. Anh, editors, Proceedings of the 24th Tyrrhenian International Workshop on Digital Communications, pages 1–6. IEEE, 2013.

[9] C. Hankendi and A. K. Coskun. Adaptive power and resource management techniques for multi-threaded workloads. In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '13, pages 2302–2305, Washington, DC, USA, 2013. IEEE Computer Society.

[10] M. Haque, H. Aydin, and D. Zhu. Energy-aware task replication to manage reliability for periodic real-time applications on multicore platforms. In Green Computing Conference (IGCC), 2013 International, pages 1–11, June 2013.

[11] S. Holmbacka, F. Hällis, W. Lund, S. Lafond, and J. Lilius. Energy and power management, measurement and analysis for multi-core processors. Technical Report 1117, 2014.

[12] S. Holmbacka, S. Lafond, and J. Lilius. Performance monitor based power management for big.little platforms. In D. Nikolopoulos and J.-L. Nunez-Yanez, editors, Workshop on Energy Efficiency with Heterogeneous Computing, pages 1 – 6. HiPEAC, 2015.

[13] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, and J. Lilius. Energy efficiency and performance management of parallel dataflow applications. In A. Pinzari and A. Morawiec, editors, The 2014 Conference on Design & Architectures for Signal & Image Processing, pages 133 – 141. ECDI Electronic Chips & Systems design initiative, 2014.

[14] T. Horvath and K. Skadron. Multi-mode energy management for multi-tier server clusters. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, pages 270–279, New York, NY, USA, 2008. ACM.

[15] IBM Corporation. Blueprints: Using the linux cpufreq subsystem for energy management. Technical report, June 2009.

[16] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In Proceedings of the 41st annual Design Automation Conference, DAC '04, pages 275–280, New York, NY, USA, 2004. ACM.

[17] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Enhancing the efficiency of energy-constrained dvfs designs. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 21(10):1769–1782, Oct 2013.

[18] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. Computer, 36(12):68–75, 2003.

[19] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. H. Katz. Napsac: Design and implementation of a power-proportional web cluster. In Proceedings of the First ACM SIGCOMM Workshop on Green Networking, Green Networking '10, pages 15–22, New York, NY, USA, 2010. ACM.

[20] C.-M. Kyung. Energy-Aware System Design. Springer, 2011.

[21] C. Lively, V. Taylor, X. Wu, H.-C. Chang, C.-Y. Su, K. Cameron, S. Moore, and D. Terpstra. E-amom: an energy-aware modeling and optimization methodology for scientific applications. Computer Science - Research and Development, 29(3-4):197–210, 2014.

[22] W. Lockhart. How low can you go? http://chipdesignmag.com/display.php?articleId=3310, 2014.

[23] W. Lund. Spurg-bench: Q&d microbenchmark software. https://github.com/ESLab/spurg-bench, May 2013.

[24] M. Marinoni, M. Bambagini, F. Prosperi, F. Esposito, G. Franchino, L. Santinelli, and G. Buttazzo. Platform-aware bandwidth-oriented energy management algorithm for real-time embedded systems. In ETFA, 2011 IEEE 16th Conference on, pages 1–8, 2011.

[25] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of cpu frequency transition latency. Comput. Sci., 29(3-4), Aug. 2014.

[26] N. Melot, C. Kessler, P. Eitschberger, and J. Keller. Fast crown scheduling heuristics for energy-efficient mapping and scaling of moldable streaming tasks on many-core systems. ACM Transactions on Architecture and Code Optimization, 11(4), 2015.

[27] J. Park, D. Shin, N. Chang, and M. Pedram. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on, pages 419–424, Aug 2010.

[28] T. Rauber and G. Rünger. Energy-aware execution of fork-join-based task parallelism. In Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on, pages 231–240, 2012.

[29] R. Schöne, D. Molka, and M. Werner. Wake-up latencies for processor idle states on current x86 processors. Computer Science - Research and Development, pages 1–9, 2014.

[30] H. Singh, K. Agarwal, D. Sylvester, and K. Nowka. Enhanced leakage reduction techniques using intermediate strength power gating. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 15(11):1215 –1224, nov. 2007.

[31] D. Zhi-bo, C. Yun, and C. Ai-dong. The impact of the clock frequency on the power analysis attacks. In Internet Technology and Applications (iTAP), 2011 International Conference on, pages 1–4, 2011.

[32] S. Zuhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of energy-cognizant scheduling techniques. IEEE Transactions on Parallel and Distributed Systems, 2012.