Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System

Antonio Paolillo
Université libre de Bruxelles (ULB)
Faculté des Sciences
HIPPEROS S.A.

Paul Rodriguez
Université libre de Bruxelles (ULB)
Faculté des Sciences
HIPPEROS S.A.

Nikita Veshchikov
Université libre de Bruxelles (ULB)
Faculté des Sciences

Joël Goossens
Université libre de Bruxelles (ULB)
Faculté des Sciences

Ben Rodriguez
HIPPEROS S.A.

ABSTRACT

In this work, we present the experimental assessment of a parallel framework that allows to reduce the energy consumption of MPSoC platforms running hard real-time systems. We use a power-aware Fork-Join task model based on primitives of the OpenMP library, a scheduling algorithm to execute such model and a schedulability test from the literature to ensure that all timing requirements are met while the energy consumption of the whole system is reduced. Practical experiments involving the deployment of OpenMP applications on a parallel embedded real-time operating system and power measurements on a MPSoC board through an oscilloscope probe show that intra-task parallelism helps to reduce the total energy consumption of the system in a realistic setting.

Keywords

Real-time; RTOS; embedded systems; energy efficiency; low power; multi-core; parallel; OpenMP; fork-join; scheduling.

1. INTRODUCTION

Platform designers left the uni-core era for more than a decade nowadays mostly because they reached the limit of frequency scaling in terms of chip power consumption and heat dissipation [13]. In order to continue delivering devices of increasing performance while ensuring stable power consumption, they now try to exploit circuit parallelism as much as possible by reproducing several symmetric processing units (or *cores*) on the same processor platform. Even embedded system vendors provide multi-core chips, also called Multi-Processor System-on-Chips (MPSoC).

RTNS '16, October 19-21, 2016, Brest, France

DOI: http://dx.doi.org/10.1145/2997465.2997473

Meanwhile, the software community, at both system and application layers, still struggles to deliver system performance scaling with the increasing number of computational units available on a single platform. At software level, data structures shared across cores must be protected from concurrent accesses to ensure coherent execution of the system. Mutual exclusion used in excess implies the serialisation of execution which leads to poor scalability and poor predictability.

The research community invests a lot of effort in improving system performance and predictability in terms of timing analysis. In the meantime, work on energy consumption for multi-core real-time workloads is not explored in depth, even though it is a critical resource for most embedded systems.

A common technique that is used to reduce energy consumption consists in dynamically decreasing the operating voltage and frequency of the target platform when its CPU load is low. Reducing voltage results in a decrease of the dynamic power consumption by a polynomial factor. However, reducing the frequency inevitably leads the system to a higher load of the computing platform. Therefore, the system designer must ensure that the application jobs are capable of meeting their deadlines for any operating frequency selected at run-time by the operating system.

Platform designers introduced multi-core systems mainly for power consumption reasons. Meanwhile, most power-aware real-time researchers as well as embedded software vendors have focused on techniques working either on uni-core systems or on multi-core systems with a *sequential* job model of execution (i.e., at any time a job can only be executed on a single core), thus limiting the potential energy savings [11]. As stated by Baruah and Anderson [1], *"[...] in the job model typically used in real-time scheduling, individual jobs are executed* ***sequentially****. Hence, there is a certain minimum speed that the processors must have if individual jobs are to complete by their deadlines [...]"*. On the other hand, a *parallel* job model of execution, where each task of the system could express *intra-task parallelism* and then could run on multiple cores simultaneously, would imply a better distribution of the workload across cores. It would decrease the minimum viable frequency and then lead to lower power consumption and better energy efficiency.

While previous work has proven the *theoretical* benefits of intra-task parallelism on energy consumption [11], the aim of this paper is to *empirically* evaluate these potential energy gains on a *real* experimental testbed.

We present a *practical* run-time framework to run parallel real-time applications on embedded systems and optimize their energy consumption. To express parallelism in user code, these applications are written in C with OpenMP pragmas. Therefore, the underlying run-time system supports the OpenMP primitives. This framework is composed of:

- an embedded MPSoC board based on an ARM Cortex-A9 processor with 4 cores;
- *HIPPEROS*, a parallel real-time operating system kernel [10];
- a port of the OpenMP run-time library for that kernel;
- application tasks written in C with OpenMP.

To support the theoretical foundations of our run-time framework, we provide a model matching the constraints of the framework, together with an existing schedulability test of the literature [2] and a heuristic capable of optimizing the power consumption of the system by selecting the appropriate processor frequency for each task.

The goal of this research is to show with practical experiments that real-time parallel systems can save more energy than their sequential equivalents for the same workload. More specifically, we want to show that with appropriate mechanisms (a power-aware scheduling algorithm, DVFS capabilities and an offline energy optimization process), the energy savings depend on the *degree of parallelism* of the user tasks.

The contributions of this research can be summarized as follows:

- We propose a practical, system-level framework to define parallel hard real-time systems and schedule them with a power-aware policy.
- We provide a realistic experimental evaluation of our framework by implementing a set of use case programs, mapping them to randomly generated (but schedulable) hard real-time systems and running them on top of a real-time operating system deployed on a MPSoC board.
- Power measurements are made with an oscilloscope to evaluate the potential benefit of exploiting parallelism to improve energy efficiency on real running systems.

As our results show that increasing the degree of parallelism (i.e. adding threads to the execution of a task) tends to lead to better energy savings, this work suggests a wider adoption of power-aware intra-task parallelism techniques for the design of embedded real-time applications.

The paper is organized as follows: Section 2 introduces our parallel run-time framework by describing intra-task parallelism with the use of OpenMP, the chosen RTOS and the embedded platform used in the experiments. Section 3 presents the experimental setup used to measure energy savings and the use cases that we used for the evaluations. Section 4 explains the intuition behind the techniques by running some measurements on individual parallel programs. Sections 5, 6 and 7 present the methodology, the algorithms, the conducted experiments and the results obtained in terms of energy consumption on multi-tasked and multi-core real-time systems. Sections 8, 9 and 10 respectively present related and future work and concludes the paper.

2. PARALLEL RUN-TIME FRAMEWORK

In this section, we explain the system-level details of the run-time framework that we propose. We also explain its implementation for the experiments of this paper.

2.1 Intra-task parallelism and OpenMP

To design hard real-time workloads, we assume application developers write programs in the C language with the OpenMP library. OpenMP is a portable API for shared-memory multi-core processors that allows to write parallel programs. The parallel regions of these programs are defined by using the #pragma omp parallel construct of the OpenMP library. This mechanism allows to distribute workload among different threads running on different cores. An example of the structure of such a parallel program is given in Listing 1.

```
1  int main()
2  {
3    const int num_steps = 1000;
4    omp_set_num_threads(4);
5
6  #pragma omp parallel
7    {
8      int nbt = omp_get_num_threads();
9      int tid = omp_get_thread_num();
10     int i_start = (tid * num_steps) / nbt;
11     int i_end = ((tid + 1) * num_steps) / nbt;
12
13     for (int i = i_start; i < i_end; ++i) {
14       /* workload executed in parallel */
15     }
16   }
17
18   return 0;
19 }
```

Listing 1: Example of structure of an OpenMP program. The for loop is executed in parallel.

The code block under the #pragma omp parallel statement (lines 7–16) is the region of the program that will be executed in parallel by several threads of the underlying operating system. The main thread of the running process, called the *master thread*, starts the execution at the main function and sets the limit on the number of threads that have to be created for the execution of the parallel region. When the master thread reaches the OpenMP pragma, it creates enough threads by *forking* itself to execute the parallel region. These newly created threads are called *worker threads*. Worker threads finish their execution at the end of the pragma scope, while the master thread (after having executed its own parallel part of the program) waits for them to finish (the *join* mechanism) before carrying on the execution of a non-parallel stage of the program. The execution flow of an OpenMP program with fork and join procedures is illustrated by Figure 1. The expected benefit of this for loop parallelisation is a substantial reduction in the program execution time.

These parallel programs constitute *user tasks* of our proposed framework. As explained later in this work, they come with hard real-time requirements on their execution. These user tasks are compiled using a *C cross-compiler toolchain* to generate code compliant with the targeted embedded platform. Since version 3.7, the LLVM-based *Clang* compiler fully supports OpenMP pragmas. This compiler supports a large variety of back-end architectures, including ARM Cortex-A9 which was used in the experiments (see Section 3).

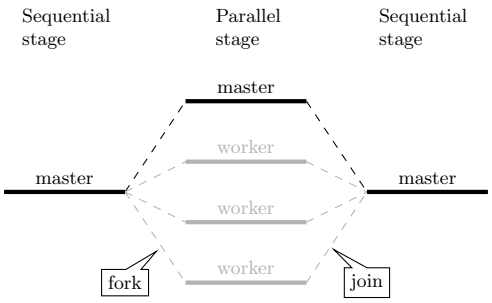


Figure 1: Scheme of an OpenMP program.

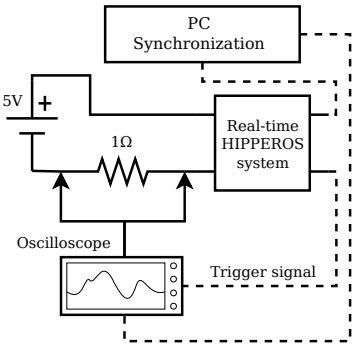


Figure 2: Hardware setup. The PC sends the RTOS and user tasks images to the embedded board and initialises the oscilloscope. At RTOS startup, a trigger is sent from the board to the oscilloscope to start the acquisitions. Voltage values are then measured on the boundaries of the leakage resistor while *HIPPEROS* schedules the user tasks with a multi-threaded power-aware policy.

It is important to notice that the OpenMP API does not constrain the scheduling policy used to execute the created threads. Different thread allocation and scheduling policies can easily be implemented at the operating system level. In this research, as explained later in the paper, we limited ourselves to Fixed Task Priority (FTP) assignment.

2.2 The HIPPEROS RTOS

The chosen embedded real-time operating system (RTOS) is *HIPPEROS*. It is a multi-task and multi-core real-time operating system targeting reliable and efficient embedded hard real-time systems [10]. The kernel has support for both power-aware scheduling and intra-task parallelism using partitioned threads. *HIPPEROS* has been developed from the ground up as a multi-core RTOS and as such natively supports multi-core scheduling at kernel level [10]. *HIPPEROS* provides a build-system written in CMake that can be used to integrate user tasks, libraries and the RTOS into binary images. These binary images can then be deployed on the supported target systems. For this research, we developed a port of the OpenMP run-time library for *HIPPEROS*. We implemented the same primitives as defined by the Intel OpenMP run-time library which is well documented and compatible with the latest *Clang* release.

The *HIPPEROS* kernel combined with the *HIPPEROS* OpenMP Run-Time Library (HOMPRTL) allows system designers to define parallel user tasks written with OpenMP pragmas and to schedule them with a power-aware policy, which is what we seek for the framework we propose.

HIPPEROS can be packaged with a variety of compile-time build options. For this research, we chose a package implementing static priority scheduling (allowing to deploy scheduling policies as rate and deadline monotonic) and a power-aware policy which associates an operating frequency to each task. How the priority, the core affinity and the speed of each task are assigned in the context of our framework is explained in Section 6.3.

2.3 Embedded platform

The embedded platform is a symmetric multi-core processor, meaning that each core of the platform can execute each application at the same speed. These cores will carry the simultaneous execution of the different OpenMP threads.

The platform has homogeneous Dynamic Voltage and Frequency Scaling capabilities (DVFS), meaning that all the cores run at the same frequency but this frequency can be changed at run-time by the operating system. In the context of our parallel OpenMP task framework, the absence of heterogeneous frequency is not a very disruptive constraint as the goal is to execute a maximum number of symmetric threads in parallel (as depicted by Figure 1). As explained later in the paper, we tried in our experiments to maximise the degree of parallelism to reduce the energy consumption.

There is a finite and relatively small set of pre-defined *operating points* supported by the operating system, which are defined as couples of voltage and frequency values. The values of the different voltage and frequency couples are determined in order to minimize the power consumption of the platform, by taking the maximum viable frequency value for each possible voltage value.

These characteristics are present in the actual hardware platform we used in our experiments which is based on the ARM Cortex-A9 multi-processor architecture.

3. EXPERIMENTAL SETUP

In this section, we describe how and with what tools we actually implemented the framework presented in Section 2.

3.1 Experimental testbed presentation

We built an experimental testbed that allows us to evaluate energy consumption gains using real embedded software and hardware. To execute embedded real-time systems, we selected an embedded development board supported by the *HIPPEROS* RTOS. The selected embedded board is the `Boundary Devices Sabrelite` (BD-SL-i.MX6) with the Freescale i.MX6q processor platform which is a MPSoC that implements a four cores Cortex-A9 ARM architecture. This board has been chosen for the following reasons:

- Multi-core with 4 ARM Cortex-A9 architecture (which is supported by *HIPPEROS*);
- DVFS features are available on this Freescale platform design with several possible operating points — notice that only homogeneous frequency is available (a single global frequency for all the cores).

These elements show that this embedded board is a perfect match for our use case.

According to the policies defined in later sections of the paper, the *HIPPEROS* RTOS sets the clock frequency as well as the operating voltage, thus effectively allowing us to reduce the total energy consumption of our system. In order to measure the total power consumption of the sys-

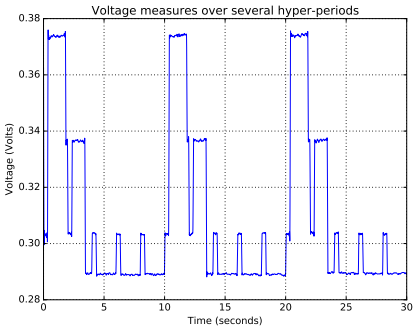


Figure 3: Voltages trace of the execution of a *HIPPEROS* system running three periodic tasks. Voltage measurements clearly show an execution pattern occurring at each hyper-period (10 seconds).

tem we inserted a $1\,\Omega$ resistor on the power line between the board and its power supply (see Figure 2). While the system was executing tasks we measured the voltage across the resistor using the digital sampling oscilloscope Infiniium MSO9254A. The board as well as the oscilloscope were connected to a Personal Computer to control the execution flow of the experiments. Specifically, the PC was responsible for deploying the *HIPPEROS* system image on the board and to initialise the oscilloscope to start and stop voltages acquisitions. Using this setup we obtained voltage traces similar to Figure 3. Power values are then computed from voltage values by knowing the circuit structure and the energy is computed by computing the numerical integration of the power values over the experiment time.

3.2 Use cases description

Use cases are application programs written in C with OpenMP pragmas (as presented in Section 2). The use cases' code structure is similar to the one of Listing 1. The use cases will be the code associated to periodic real-time tasks of our evaluation, meaning that each time a periodic task releases a job, the program that will be executed will be the one defined by the associated use case.

We selected three use cases that are representative of different typical applications of the real-time domain:

- the π (or simply "pi") use case: this program simply computes the value of π with a classic approximation of a Riemann integral of the function $f(x) = \frac{4}{1+x^2}$ between 0 and 1. It could represent any *CPU-intensive* numerical program. It is easy to implement in parallel by distributing to the different threads an equal share of the area to compute under the curve.
- the *Image* use case: this program applies an edge-detection filter on a given input image stored in memory and produces the corresponding output image in memory. Each parallel thread is responsible for applying the filter on an equal part of the image. Multiple configurations of this use case were used to show the effect of the size of the image (and therefore, memory footprint) on execution time.
- the *AES* use case: this program takes as input a plain text and a key stored in memory and applies the Advanced Encryption Standard algorithm to produce the corresponding cipher text. Each thread is assigned an equal sub-string of the plain text to cipher. As AES is a *block cipher* encryption algorithm, each block is encrypted independently from the others; therefore there is no data dependency and the program is easily parallelised.

Voltage (mV)	Frequency (MHz)
1100	876
1125	948
1150	1008
1175	1068
1200	1116
1225	1164
1250	1212
1275	1260
1300	1296

Table 1: Table of operating points. The voltage and frequency couples were obtained empirically.

As the π use case is CPU-intensive, the other two should have a fair utilisation of the memory (and then access to shared resources like memory buses, etc.). We expect then with these three use cases to evaluate realistic scenarios of real-time applications that can have a good parallelisation profile. Actual execution time profiles and energy profiles of these use cases are shown and discussed in Section 4.2.

4. EXPECTED POWER SAVINGS

In this section we explain the basic rationale behind this work. We confront the CPU power consumption model with preliminary measures on systems running a single task at different processor frequency and with different degrees of parallelism thus varying the number of simultaneous active cores of the platform.

4.1 Power consumption model

Li *et al.* [9] provide detailed models of power and energy consumption for multi-core real-time systems. Generally, the instantaneous power consumption of a multi-core processor with homogeneous voltage and frequency can be modeled as follow:

$$P(f, V_{dd}, k) = k(\alpha V_{dd}^2 f + \beta V_{dd}) + \gamma \qquad (1)$$

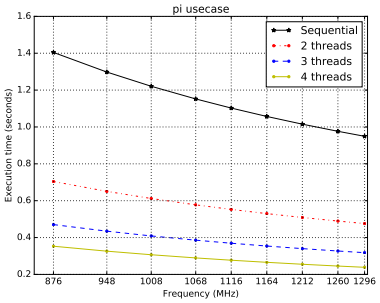
where f is the operating frequency, V_{dd} is the supplied voltage (the couple (V_{dd}, f) forms the current operating point of the platform) and k is the number of active cores. α, β and γ are inherent constants depending on the processor manufacturing technologies, such as switching capacitance, current leakage, etc.

As chip designers understood it while entering into the multi-core era, Equation 1 shows that adding cores to a platform only contributes linearly to the power consumption while increasing the frequency and voltage can result in quadratic or cubic contribution to the power consumption.

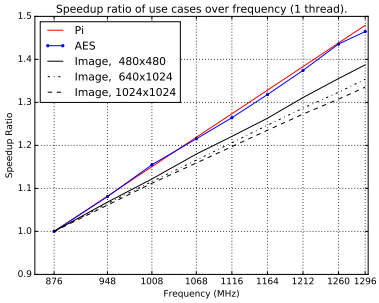
This suggests therefore that a computational model capable of distributing work on several cores rather than increasing the voltage and frequency would result in better energy savings while increasing the execution capabilities of the platform.

4.2 Measuring a running OpenMP program

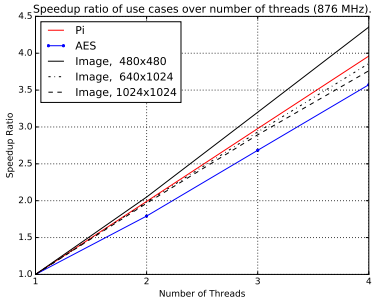
The following simple experiments using the setup described in Section 3 confirm the intuition given by the power model.



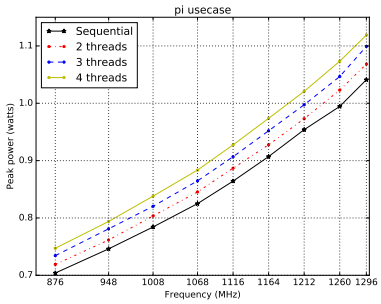
a: Execution time of the *pi* use case in all configurations.



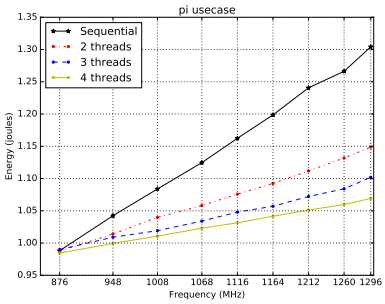
b: Speedups obtained in execution time for the different use cases by changing voltage and frequency.



c: Speedups obtained in execution time for the different use cases by changing the degree of parallelism.



d: Peak power of the *pi* use case in all configurations.



e: Energy consumption of the *pi* use case over a fixed time in all configurations.

Figure 4: Single task execution measurements.

These measurements are based on the individual execution of each of our use case programs as described in Section 3.2.

The operating points selected for our experiments are shown in Table 1.

We deployed *HIPPEROS* systems with a single task executing one of our use case program on our experimental testbed (in the figures the *pi* use case is represented). We executed this use case a certain number of times and each time we varied (1) the number of threads used by the task in its parallel stage (from 1 to 4) and (2) the operating point of the processor (from 876 to 1296 MHz).

For each setting we measured the total execution time and the instantaneous power consumption of the platform. The power consumption was measured over a fixed period of time longer than the longest execution. We noted experimentally that this longest execution corresponds to the setting where a single thread of execution was allocated to the parallel stage – the sequential case – and the smallest operating point was set. In the following, we call this setting the *reference execution time* of a task.

Figure 4a shows the execution times of the *pi* use case. The data suggests that at least when it comes to this use case, the expected behaviour is correct: increasing the number of threads (and therefore the number of cores) associated to a task and/or increasing the frequency results in a substantial reduction of the execution time. We measured the gained speedup in execution time for each of our use case run individually. For the *image* use case, we tried with different image resolution in order to vary the memory access patterns and get a realistic view of the expected speedup. Execution time speedups due to frequency scaling (resp. degree of intra-task parallelism) are shown in Figure 4b (resp. Figure 4c). We see that the more the task needs to access the memory, the worst the speedups are with frequency scaling. This does not hold for the speedups due to multi-threading. For CPU-bounded task (the π use case here) the speedups are near to perfect for both techniques.

To evaluate the costs in terms of power consumption of these speedups, Figure 4d shows the peak power being drawn by the board for the *pi* use case. The peak power is defined as the maximum power consumption of the system while executing the use case. As the power is more or less constant during the execution of the use case, it represents the average power consumption. The intuition given by Equation 1 is confirmed by this figure. We see that the power cost to increase the frequency is way larger than adding cores. Moreover, OpenMP threading techniques achieve way better execution time speedups than scaling the frequency up for cheaper power costs.

Finally, the same trade-off between time speedup and power can be observed by computing the energy consumption of the tasks for the different settings (which is the actual resource that we want to reduce the usage). Figure 4e shows the energy consumption of the board over a fixed period of time larger than the reference execution time for the *pi* use case. Notice that in the experiments the system switches in idle state when the task finishes, a state where the power consumption is minimal (and therefore has a small contribution to the total energy consumed). Two main observations can be made. Firstly that the introduction of parallelism greatly reduced energy consumption across all frequency set-

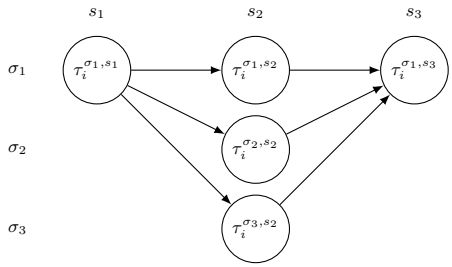


Figure 5: Task model. The nodes of each task τ_i are divided into *segments* horizontally and into *stages* vertically.

tings, secondly that the sequential scheme is affected more sharply by the frequency of the platform in terms of energy.

We obtained similar results for each use case run individually. Therefore for the purpose of brevity, we do not include the individual charts of each use case in the paper.

As a preliminary conclusion, we can already state that parallelism (i.e. adding cores to a task) seems to be an efficient way to reduce energy consumption or have better time speedups than scaling the platform frequency and voltage up. Multi-threading techniques like OpenMP can help to achieve this goal, which was one of the main rationale of the hardware designer when introducing symmetric multi-core platforms on the market. The next sections of the paper will be dedicated to show that this holds also for multi-tasked hard real-time systems running on platforms with multiple cores. In these system-wide experiments, the different use cases will be executed concurrently and the individual power contributions of each use case will be aggregated in an evaluation of the system energy savings.

5. SYSTEM MODEL

In this section we present the model used to analyse the multi-tasked and multi-thread real-time systems evaluated in the experiments.

5.1 Parallel Task Model

We represent the multi-thread processes with a fork-join task model similar to the one used by Axer *et al.* [2]. We consider a set τ of synchronous periodic fork-join tasks with implicit deadlines. Each task τ_i is released every P_i time units. Tasks are comprised of *nodes* $\tau^{\sigma,s}$, where σ is a *segment* and s is a *stage*, which means that $\tau^{\sigma,s}$ is the node of stage s in segment σ. Naturally, for task τ_i the corresponding nodes are noted $\tau_i^{\sigma,s}$. See Figure 5 for a graphical representation of this model.

There can only be at most one node corresponding to a segment and stage pair. The concept of segment denotes that a collection of nodes of different stages are to be partitioned on the *same core*. The stages of a task are ordered in sequence. A node belonging to a given stage cannot be executed before all the nodes of previous stages are completed (i.e., finished their execution).

In this paper we use a restrictive model where all tasks have 3 stages, as illustrated by the program structure in Listing 1, by the execution flow chart in Figure 1 and by the task graph in Figure 5. This restriction allows us to define a simple partition mapping and fits the use cases we study in the scope of our experimental evaluations. The first and the last stages (which are part of the execution of the master thread) are sequential (i.e., they only have one node) and the middle stage is the *parallel stage* (i.e., it might have more than one node). If the middle stage has exactly one node then the task is said to be *sequential*. The two nodes of the first and last stages, as well as exactly one node of the middle stage are parts of the same segment. Each node $\tau_i^{\sigma,s}$ has the worst-case execution time $C_i^{\sigma,s}$. The segments defined in this section conceptually correspond to *HIPPEROS* threads generated from OpenMP parallel constructs.

5.2 Platform, Power and Energy Models

We denote by m the number of available symmetric cores of the platform. Within the scope of our framework, we assume that power consumption at any time $P(V, f)$ is a non-decreasing monotonic function of the running voltage and frequency of the platform, and therefore depends on the currently selected operating point. We do not need a more detailed power model as:

- our scheduling analysis framework is based on a selection of operating points that are strictly ordered by voltage and frequency (see Table 1);
- only the offline analysis (partitioning and operating point selection) is influenced by the accuracy of this model;
- for the experiments, we directly measure the power consumption on a real embedded board to derive results on the energy efficiency of our technique.

The energy consumption is simply defined as the integral of the measured power over time.

6. ALGORITHMS AND POLICIES

To be able to run energy-efficient systems for each possible degree of parallelism, we used offline analysis techniques to optimize task sets before loading them on the hardware platform. Each task set is only run on the target platform using operating points and partition definitions found during the offline analysis described in this section. As the object of this research is not to provide new analysis techniques for low power systems nor even to demonstrate the use of such techniques but rather to demonstrate the impact of introducing intra-task parallelism on power consumption, the details of the analysis are kept to a minimum in this paper.

The analysis of a system is a process illustrated in Figure 6. The figure shows two nested cycles. The outer cycle corresponds to Section 6.1 considering the set of configurations of operating points in a greedy manner, registering solutions each time a better one is found. The inner cycle corresponds to Section 6.2 testing over the set of task partitionings for the configured system until either a feasible solution is found or the set is exhausted. This process is a practical use of various existing results in fork-join task scheduling theory. As input, it takes an upper-bound of the worst-case execution times for each use case for each configuration (chosen degree of parallelism and set operating point). For this research, we neglected the effect of shared resources such as memory buses, caches and task interferences on timings.

6.1 Offline Power Optimization

Within our framework, we implemented a *task-level* DVFS policy, meaning that an operating point is statically assigned to each task of the system. This means that whenever a job from a given task is being executed, the frequency (or

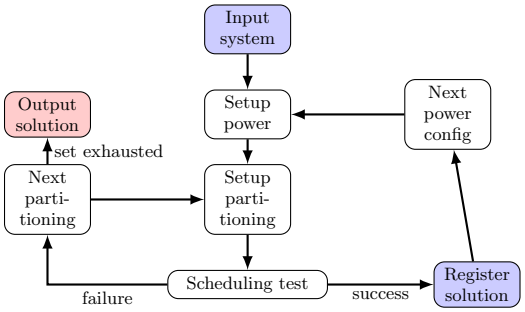


Figure 6: Flow chart representing the analysis process.

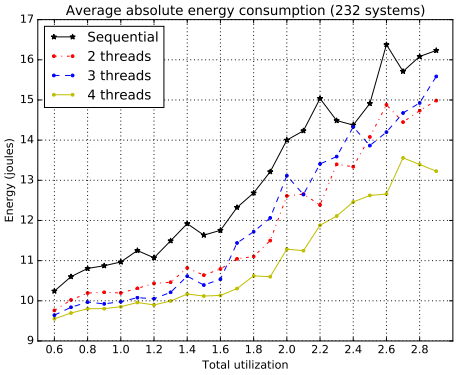


Figure 7: Absolute energy consumption of generated systems.

a higher one) that has been assigned to that task is used. Therefore, the operating system is responsible to change (if need be) the operating point of the platform (this is a global setting on the platform considered in this paper) to a value appropriate for a task when a job of this task is scheduled.

The number of configurations (i.e. choices of operating points per task) per set of tasks is potentially large. The simplified power model introduced in Section 5.2 is used to derive an offline optimization heuristic to minimise the energy consumption of a task system.

The degree of parallelism is fixed for one execution of this optimisation step. Therefore, the heuristic starts by assigning the highest operating point to each task (resulting in the highest expected energy consumption profile) and attempts to greedily decrease it by decreasing the operating points of individual tasks, one at a time. To validate that a set of tasks with these assigned operating points will be schedulable on the experimental platform, a partitioned schedulability test is used.

6.2 Partitioned Schedulability Test

We used an implementation of a schedulability test algorithm derived from the response-time bound based on the work by Axer *et al.* [2]. In that work, the authors showed that the stages of a fork-join task can be considered in sequence starting with the first one without any backtracking to maximize the response-time of the last stage, provided a conservative approximation method for the response-time of each stage is used.

The use of this schedulability test introduces significant pessimism in systems with parallel tasks compared to sequential tasks. Many such systems are schedulable using some Fixed Task Priority (FTP) scheduling algorithm but do not pass the schedulability test. Globally, this results in higher frequency and voltage requirements and therefore power consumption for parallel tasks than what could be achieved with a more accurate test.

Unfortunately, this test was found flawed by Fonseca *et al.* [7]. However, this does not impact our results as discussed in Section 7.3.

6.3 Online Scheduler

The *HIPPEROS* RTOS is capable of scheduling processes and threads according to any Fixed Task Priority (FTP) assignment. To be compliant with the schedulability testing and power optimization procedure described in the previous sections, Rate Monotonic is used. The fixed partitioning of threads to cores is also supported by the operating system (by defining thread affinities).

The offline analysis presented above produces these outputs for a given task system:

- a partition mapping for each node of each task of the system, i.e. how the core affinities of each *HIPPEROS* thread must be defined;
- a task-to-operating point mapping ensuring that the jobs of each task will never miss a deadline and reducing the power consumption contribution of this task as much as possible.

This information is sufficient to generate a run-time system which will be scheduled statically with a Partitioned Rate Monotonic policy. Each thread of every task will then be assigned to a core of the target platform. As soon as a thread is chosen to be scheduled by the system, the scheduler sets the platform operating point to the one associated with the thread's task.

It is possible that threads of different tasks having different operating points assigned are executed concurrently on different cores of the platform. However, as explained in Section 2.3, the embedded platform has only global DVFS capabilities. Therefore, only one operating point can be selected at any time for all the cores of the platform. In such scenario, the operating system scheduler always chooses the operating point having the maximum frequency, ensuring therefore that the requirements of the highest demanding task are satisfied. We assumed that there was no "speed anomalies", i.e. increasing the operating frequency does not increase the execution time of any task. This was the case in the execution time measures of the experiments' usecases in Section 4.2. We expect that this holds in most practical cases, and if not, the system designer must be warned by some static analysis tool. Therefore, any thread receiving an operating point greater than the one associated to its task will meet its timing requirements.

The operating point switching time can last up to 60 microseconds on our experimental platform. Due to the size of our use case, we chose to neglect those (as we did for the preemption time and scheduling overheads). These parameters could be integrated in the worst-case execution time analysis for a more precise study. In practice, operating point switches only affect the scheduling core.

7. SYSTEM EXPERIMENTS

7.1 Methodology overview

We implemented a complete experimental flow. The idea is to be able to evaluate the overall energy consumption of task systems under a range of degrees of parallelism. By implementing the algorithms and policies presented is Section 6, we expect to save more energy when increasing the degree of parallelism of the user tasks (i.e. the number of nodes in the parallel stage). This would confirm the intuition of the preliminary study of Section 4.

This experimental flow is composed of the following steps:

- **Randomly generating a set of task systems.** We generated 232 schedulable task systems with varying total utilisation (from 0.6 to 2.9), each composed of 5 independent tasks[1] mapped to the use cases presented in Section 3.2.
- **Determining operating points and thread partitioning.** For each possible degree of parallelism (from 1 to 4), all the systems generated at the previous step are analysed through a program implementing the flow represented by Figure 6 and running offline on a personal computer. For each task of the input system, this step chooses an appropriate operating point and produces a partition mapping for each thread of the task. This step can fail on generated systems that do not meet the schedulability conditions of the heuristic in any configuration. In this case, new systems are generated as in the previous step until schedulable systems are found.
- **Configuring the *HIPPEROS* applications.** The information produced by the previous step is used to configure *HIPPEROS* applications that faithfully reproduce the generated task systems and the result of their offline optimisation produced by the heuristic. For each generated task system, 4 *HIPPEROS* applications are produced, each corresponding to a possible degree of parallelism.
- **Deploying and measuring the *HIPPEROS* applications.** Each build generated at the previous step has been validated by the schedulability test and can then be tested on the real platform. An image is produced, containing the components of the system and the use cases code. Then it is deployed on the Sabre-Lite board and the measures are recorded by the oscilloscope. With these records, the energy consumption of each build can be computed over a fixed period of time (12 seconds, corresponding to a multiple of the hyperperiod of all the generated systems).

7.2 Results & Discussion

The first thing to notice is that the high utilisation systems are under-represented in the set of generated schedulable systems. We choose to generate systems of total utilisation up to 2.9 to maximise the scheduling opportunities. In a partitioned schedule, systems with utilisation greater than 0.7 per core are unlikely to be schedulable. Recall that we ran our experiments on a platform with 4 cores. The scarcity of schedulable systems at high utilisation is to be expected considering the limitations of the scheduling technique we used in the paper (which is based on a partitioned rate-monotonic analysis).

For the discussion about energy savings, we only consid-

[1]We chose the number of tasks for this paper arbitrarily. Indeed, we observed similar results in energy savings for larger task sets.

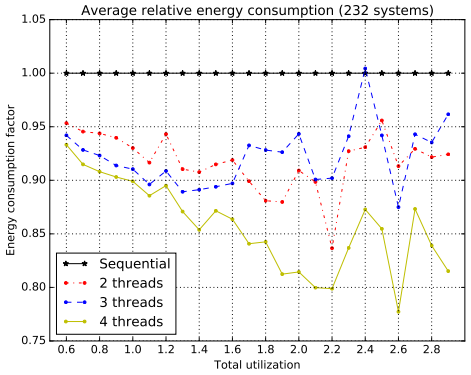


Figure 8: Relative energy savings of parallel systems w.r.t. their sequential equivalent.

ered systems that present a schedulable solution for every degree of parallelism. Figure 7 shows the average absolute energy consumption (in Joules) and Figure 8 shows the average energy savings of the multi-threaded solutions relative to their sequential equivalent.

Regardless of the selected degree of parallelism, Figure 7 shows that the higher the system utilisation is, the higher energy it will drain from the power supply. This can be explained for two reasons: first, an under-utilized system will not tend to require a high frequency, and therefore the dynamic part of the program will drain less power; secondly, low utilisation systems will remain in idle mode most of time, which is a state handled by the kernel and the underlying platform and therefore easy to optimize and setup for low-power.

Figure 8 shows that parallelism is clearly a good option to reduce the energy consumption of low utilisation systems. In the best cases, the average relative factor can be as low as 77%. The maximum standard deviation in computed energy savings for each system utilisation value was no greater than 0.13. We see that the maximum degree of parallelism (the one that uses all the cores) is also the one which dominates in terms of energy savings. This is expected because using all the cores with all the threads leads to a high speedup (near to 4 for our use cases) and to a better distribution of the workload among cores. More generally, until a certain threshold of utilisation value, we can see the following tendency: the higher the degree of parallelism, the better the energy savings.

However, we see a limit in this tendency when the utilisation increases. This can be caused by two reasons. The first reason is that in this data, the only systems that are represented are those that are schedulable for all degrees of parallelism, including the sequential ones. However, a schedulable mono-thread system with a high system utilisation is already well distributed across the cores of the platform, and there is not much that parallelism can do to improve this situation. Moreover, these charts do not show the fact that a system which is not schedulable in his sequential version could be schedulable in its parallel version. Energy savings apart, multi-threading potentially bring also better schedulability ratio (with the application of right policies). The second reason is that the scheduling analysis used in this work introduces a higher pessimism when increasing the degree of parallelism. This forces the optimization procedure

to adopt higher operating points for the parallel systems, resulting in less energy gains. Still, good energy savings for low-to-middle total utilisation gives good opportunity to reduce the energy consumption of industry systems that are usually over-provisioned (and result in low system load).

More precisely, we see that configurations with 3 threads seem to degenerate for some utilisation values. This can be explained because of the number of cores of the platform. As there are 4 cores, a setting with tasks with 3 threads leads to an unbalanced use of the platform and a loss of synchronicity for OpenMP threads. This is not energy efficient.

Notice that all the systems that are part of the data plotted on the charts have been passed as schedulable by the offline analysis and that this was confirmed at run-time as the *HIPPEROS* RTOS (which natively supports real-time workloads) didn't detect any deadline miss (for at least an execution lasting more than one hyperperiod). Therefore in the context of our work hypotheses made on timings appeared to hold and the system did not suffer too much from task interferences and implicit resource sharing.

Finally, note that the techniques used in this paper only focus on the power consumption of the processor (by changing its voltage and frequency), but the measurements were done directly on the power supply of the whole system. There is a large (and constant) contribution to the system power consumption that has nothing to do with the processor. Therefore, the relative energy savings on the processor alone are likely to be somewhat greater than those shown in our data.

7.3 Flaw in schedulability test

As mentioned in Section 6.2, the schedulability test of Axer *et al.* [2] used to discard solutions (partitioning and operating point assignments to the generated systems) before running the experiments was found to be flawed.

After careful analysis, we found that this flaw causes the analysis to be optimistic, accepting certain systems that are actually not schedulable. We consider that this flaw did not impact the results presented in Section 7.2.

Indeed, our paper discusses the energy savings that can be achieved through the use of parallel tasks on a real platform, not tied to any specific schedulability test. The (flawed) schedulability test was primarily used to make large scale experiments by discarding unschedulable systems quickly.

It is important to notice the following:

- only (randomly generated) systems schedulable according to the flawed analysis were executed at run-time on *HIPPEROS*;
- we executed all theses systems on the RTOS with execution budget and deadline checking activated for a period of time at least longer than their hyper-period;
- all these systems met their deadlines at run-time.

Considering this, we only ran systems that both returned schedulable by the (flawed) analysis and didn't miss any deadline at run-time. To impact our energy saving results, the test had to mark systems as schedulable and those systems would need to miss deadlines when executed in practice. This was not the case in our experiments.

If we hadn't used a schedulability test and ran all generated systems on the RTOS directly, discarding systems with detected deadline misses, the results and contributions of our submission would have been the same (while the experiment would have taken more time by several orders of magnitude).

8. RELATED WORK

This paper introduces the usage of power-aware parallelism in a practical and implementable scenario, using an existing real-time operating system and an embedded multi-core platform. To the best of our knowledge, currently no other research has simultaneously addressed realistic and practical energy-aware and low-power parallel real-time computing with operating system support. However, these topics have been treated separately in the literature.

Practical Fork-join parallelism. Wang and Parmer addressed the problem of real-time fork-join parallelism in practice [14], with a focus on the timing overheads of their implementation in the Composite real-time kernel. Ferry *et al.* [5] designed and implemented *RT-OpenMP*, a platform for supporting OpenMP in real-time systems based on these results, and made an experimental assessment of their implementation on hardware.

Energy-aware real-time computing. Numerous theoretical works have been produced on energy-aware real-time systems. Most of these works involve uni-core or multi-core systems with a sequential model of execution. However, some consider a parallel model of execution, e.g.: Zhu *et al.* designed graph-task power-aware scheduling techniques [15]; Kong *et al.* considered the scheduling of parallel tasks with a single, global deadline (frame-based task systems) [8]; Chen *et al.* provided techniques that combine DVFS and Dynamic Power Management (DPM, which consists in individually switching on and off the cores of the platform) for dependent tasks (a model close to OpenMP) [3]; Paolillo *et al.* evaluated the potential benefits of parallelisation over sequential systems through theoretical simulations of a malleable task model [11]. However, most of these works lack an experimental evaluation of the energy savings of the proposed techniques on real-world hardware. Moreover, often very few concerns are raised about the implementation of operating system support for these techniques.

Parallel real-time scheduling theory. For this work, we chose to restrict ourselves to partitioned and periodic fixed priority scheduling policies. Other works considered the partitioned fixed priority scheduling of more generic parallel tasks [12], based on a task decomposition technique and a sporadic schedulability test [6], however this approach resulted in excessive pessimism when coupled with the restrictive task model and technical limitations of our experimental setup. Courbin *et al.* [4] investigated a number of scheduling and partitioning techniques for parallel tasks, also based on task decomposition.

Model precision. With regards to existing works, this paper also uses a more general model of power consumption and execution time than what is generally considered in the literature [1, 2, 4, 5, 9, 12]. Usually it is assumed that the relation between frequency and execution time is simply inverse, but also that introducing intra-task parallelism and multi-core execution results in perfect speedup.

9. FUTURE WORK

Potential extensions to this work on energy-aware real-time computing would use more advanced software techniques like global multi-core scheduling, EDF policies and combine DPM with DVFS. In particular, to extend the ex-

periments to the explicit use of DPM and DVFS in a dynamic manner while a parallel task is executing. There is no obvious domination of DVFS versus DPM in terms of energy consumption and it would be worthwhile to explore their effectiveness depending on platform characteristics.

We would also like to try and combine different optimisation heuristics and schedulability tests that would improve the energy savings. Indeed the schedulability test used in this paper carries significant pessimism that artificially penalised parallel tasks.

Parallelisation techniques can also be at *extra-cores* level, by using reconfigurable hardware (i.e. FPGA) and model it as a resource in a real-time framework.

The experiments could be extended to more varied use cases, using more complex features present in real embedded applications such as I/O, inter-process communications and synchronisation primitives. CPU frequency has a lesser impact on a I/O-heavy process and therefore could be lowered further at little cost in performance. We could also add complexity in the parallel task model, by allowing several parallel stages in the task, both *in sequence* or *nested*.

Finally, we could also evaluate the impacts and benefits of parallelisation and multi-threaded scheduling on the system temperature, which would be an additional metric to consider alongside energy consumption.

All these techniques would be implementable and evaluated with an existing RTOS.

10. CONCLUSIONS

In this paper we presented a complete experimental framework able to measure the energy consumption of a physical board with a multi-core processor when running randomly generated real-time task sets from a set of use case programs. We used this framework to demonstrate the impact of introducing intra-task parallelism on the energy consumption of the system for a given workload and timing constraints. Our results suggests that with the specific techniques used, energy savings up to 30% might be realised, depending on the load of the system and the profile of the programs.

Another contribution of this paper is to confront practical evaluations on a real embedded platform with techniques and idea usually exploited in the theoretical literature with such as the actual effects of frequency/parallelism scaling on the execution time of cpu-/memory-bounded programs.

The practicality of the approach must be questioned with the feasibility of implementing multi-threaded applications on real world real-time systems with good speedup. This can be proved to be challenging as usually industry designers prefer to avoid complexity by implementing very simple (and sequential) programs. However, standard API like OpenMP bring an abstraction layer to simplify the design of parallel applications. Therefore by exploiting parallelism, our work shows practical benefits in terms of energy savings with the use of easy-to-apply low-power operating system techniques.

However, our results also show the limitations in terms of energy savings of the use of pessimistic schedulability test. In order to leverage the full capabilities of multi-core hardware platforms and reduce their energy consumption as much as possible, real-time operating systems needs to implement more precise and validated policies. This requires the development of novel theoretical techniques.

Acknowledgments. This work is supported by the H2020-ICT-2015 Innovation Action 688403 TULIPP project. We also thank the anonymous reviewer that pointed out the flaw in the schedulability test we used.

11. REFERENCES

[1] J. Anderson and S. Baruah. Energy-efficient synthesis of EDF-scheduled multiprocessor real-time systems. *International Journal of Embedded Systems*, 2008.

[2] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *25th Euromicro Conf. on Real-Time Systems*, 2013.

[3] G. Chen, K. Huang, and A. Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination. *ACM Trans. Embed. Comput. Syst.*, 13(3s):111:1–111:21, Mar. 2014.

[4] P. Courbin, I. Lupu, and J. Goossens. Scheduling of hard real-time multi-phase multi-thread (mpmt) periodic tasks. *Real-time systems*, 49(2):239–266, 2013.

[5] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. A real-time scheduling service for parallel tasks. In *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[6] N. Fisher, S. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10–pp. IEEE, 2006.

[7] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

[8] F. Kong, N. Guan, Q. Deng, and W. Yi. Energy-efficient scheduling for parallel real-time tasks based on level-packing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

[9] D. Li and J. Wu. *Energy-aware Scheduling on Multiprocessor Platforms*. Springer, 2013.

[10] A. Paolillo, O. Desenfans, V. Svoboda, J. Goossens, and B. Rodriguez. A new configurable and parallel embedded real-time micro-kernel for multi-core platforms. In *Proceedings of the ECRTS Workshop on Operating Systems Platforms for Embedded Real-Time applications (ECRTS-OSPERT '15)*, July 2015.

[11] A. Paolillo, J. Goossens, P. Hettiarachchi, and N. Fisher. Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies. In *20th International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2014.

[12] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[13] M. M. Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[14] Q. Wang and G. Parmer. Fjos: Practical, predictable, and efficient system support for fork/join parallelism. In *20th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[15] D. Zhu, N. AbouGhazaleh, D. Mosse, and R. Melhem. Power aware scheduling for and/or graphs in multiprocessor real-time systems. In *International Conference on Parallel Processing*, 2002.

APPENDIX

A. METHODOLOGY DETAILS

In this appendix we explain the details of our experiment testbed and how we implemented the whole flow of our methodology.

A.1 Platform operating points determination

As the documentation of the i.MX6q processor platform is really succinct about what are the supported operating points, we decided to find them out by running some benchmarks. We executed in parallel on the 4 cores of the platform a CPU-intensive, fault-sensitive task. For each possible voltage value, we executed this workload at higher and higher frequencies until the system became unstable. Then, we lowered the frequency and attributed it to this voltage value. This empirical method allows to generate the operating points (V_{dd}, f) with the highest viable frequency value for each voltage value (minimising then the individual power consumption of each operating point as depicted by Equation 1). The operating points selected for our experiments are shown in Table 1.

A.2 Random task sets generation

To imitate a wide range of applications, a random generation technique was used.

The utilisation of each task was generated using the efficient technique of Emberson, Stafford and Davis [16, 17] to generate multi-core systems that are uniformly-distributed. This technique allows to generate individual task utilisations with a given number of tasks and a fixed total utilisation (the sum of the utilisation of all the tasks of the system).

All the systems we evaluated in our experiments were composed of 5 periodic tasks. We generated systems with total utilisation varying from 0.6 to 2.9 (we deployed our applications on a platform with 4 cores) by step of 0.1. For each of these total utilisation values we generated 10 random systems, resulting in a total of 240 generated task systems to analyse. Each individual task utilisation was comprised between 0.1 and 1.0.

Each individual utilisation value of a generated system was randomly matched to one of our three actual use cases (introduced in Section 3.2) to be run on the RTOS.

The period of each task was generated by picking a value uniformly from the set $\{2, 3, 4, 6, 12\}$ (these values are expressed in seconds). This set of period values ensures that the maximum possible hyper-period for any generated task set would be 12 seconds and therefore limits the amount of necessary voltage measures and bound the time to experimentally evaluate a system on the actual platform.

Notice that the generated utilisation are defined as reference utilisation values for a fixed operating point (the lowest one) and a sequential model of execution (i.e. the number of nodes of the parallel stage is fixed to 1). Any other choice for these parameters impacts the actual utilisation as this choice modifies the execution time of the executed task.

Randomly generating individual utilisation and period values for every task of a system constraint the execution time to a fixed value, as $u_i = \frac{C_i}{P_i}$. We explained how we managed to fix the execution time of our use cases in the next sub-section.

A.3 Execution time measurements

A worst case execution time bound for each use case presented in Section 3.2 is evaluated empirically.

The optimisation procedure that fixes the degree of freedom of our analysis (introduced in Section 6.1) requires, for each use case, a matrix of Worst Case Execution Times (WCETs), as represented by the plot in Figure 4a. This matrix must provide a WCET bound for a given operating point and a chosen number of nodes for the parallel stage (i.e. the chosen parallelism degree). More specifically, a bound must be provided for each stage of the use case (sequential, parallel, sequential; see Section 5.1).

For the purpose of our experiments, for each use case and for each possible combination of operating point (there are 9, see Table 1) and number of nodes in the parallel stage (from 1 to 4), we ran an *HIPPEROS* system with a single task to execute and we measured the execution time of each stage (as required by the schedulability evaluation procedure defined in Section 6.2).

In order to avoid to introduce non-determinism in the execution time of the use cases, the data processed by those are fixed at compile-time. However, we ran this experiment several times to aggregate the results and avoid non-determinist timing effects (e.g. cache effects, kernel overheads) and these WCET bounds are over-provisioned by a 10% factor to avoid any time budget overflow in the execution of the real task systems.

This simple empirical approach to WCET evaluations does not intend to replace a complex WCET analysis framework with the appropriate tools but is sufficient for the purpose of the experiments of this paper (as our goal is not to provide techniques nor tools to determine the WCET of parallel programs). In the context of our experiments, this was sufficient as the running RTOS did not detect any WCET overflow of the jobs at run-time.

As the periods and the reference utilisation values of the tasks are generated randomly, the reference WCET must be fixed (WCET when the selected operating point is the lowest and when no degree of parallelism is allowed). To this aim, we introduced in the code of all the use cases an execution time factor in order to be able to artificially increase linearly the workload to execute and therefore to (almost) linearly increase its WCET bound. This factor allows that $u_i = \frac{C_i}{P_i}$ holds (approximately) for any generated utilisation and period values. Therefore, the WCET measures are executed for the least possible WCET factor (= 1).

A.4 Power optimisation step

At this step of the evaluations, we have a set of task systems composed of tasks, each having an associated use case, period, a utilisation value and a WCET matrix. We implemented the optimisation heuristic, the partitioner and the schedulability test (explained in Section 6 and illustrated by Figure 6) as a Python program. The program takes as input a task system and a chosen degree of parallelism (a number between 1 and 4) and selects an operating point for each task of the system. This selection is made in order to guarantee system schedulability while trying to minimize the overall energy consumption (by selecting the lowest possible operating points).

If the program found a schedulable solution, it produces as output a *system configuration* that defines how the *HIPPEROS* tasks and threads must be configured to represent the generated system. For each task, this configuration defines the

selected operating point and to which core to assign every thread of the task. This information is used to produce the actual *HIPPEROS* build.

If the program did not found any schedulable solution for any selection of the operating points, it produces a *null* system. Then no *HIPPEROS* build is generated.

On each generated task system, we run the heuristic for all possible parallelism degree. Therefore, for each system, there are 4 *HIPPEROS* builds generated, one for each allowed parallelism degree. Systems having a degree of parallelism proven to be unschedulable are simply discarded. Notice that when the degree of parallelism is set to 1, the system is equivalent to the sequential case.

A.5 Systems deployment on the RTOS

The system configurations generated at the previous step are supposed to be schedulable from the analysis. The output of the heuristic is then used to generate a *HIPPEROS* application package. The application containing the code of the use cases together with the tasks' configuration, the HOMPRTL software and the *HIPPEROS* kernel are then deployed on the SabreLite MPSoC platform to validate our theoretical framework in a practical setting.

The application is executed for a period of time equal to a multiple of the system hyper-period (12 seconds). The kernel releases the periodic parallel jobs corresponding to the configured tasks while the oscilloscope records voltage measurements on the hardware platform.

This allows us to compare the measurements done for the same system executed with different parallelism degree. An external script checks that no WCET overflow or no deadline miss occurs (to validate the schedulability analysis).

A.6 Power measurements

As explained earlier in this section, voltage values are measured and recorded with the oscilloscope on the boundaries of a $1\,\Omega$ resistor. The power supply provides a $5V$ voltage potential. The circuit is illustrated by Figure 2. If V_h is the voltage drained by the SabreLite platform (running the *HIPPEROS* system), V_r the voltage measured at the resistor boundaries by the oscilloscope, R is the resistor value, I the current in the circuit and P_h the power consumed by the *HIPPEROS* system, then simple electricity laws imply:

$$V_h = 5V - V_r, V_r = RI \Rightarrow P_h = V_h I = (5V - V_r)\frac{V_r}{R},$$

and we know $R = 1\,\Omega$, then we obtain the power consumed by the board at any time by applying $P(v) = (5-v)v$ on any voltage measure v (i.e. as shown in Figure 3). The overall energy consumption is computed over a time of a multiple of one hyper-period (12 seconds) and is obtained by approximating the integral of the power over time, i.e. by summing all power measures (taken between system boot and the multiple of the first hyper-period) and multiplying them by the constant time interval between two measures. Let H be the system hyper-period, E_H is the energy consumed over this hyper-period of time. v_i is the set of all voltage measures of the oscilloscope around the resistor boundaries separated by a small constant interval of time Δt, then

$$E_H = \int_H P(v(t))dt \simeq \Delta t \sum_i (5 - v_i)v_i.$$

B. REFERENCES

[16] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.

[17] R. Stafford. Random vectors with fixed sum, 2006.