Model Checking and Evaluating QoS of Batteries in MPSoC Dataflow Applications via Hybrid Automata

Waheed Ahmad, Marijn Jongerden, Mariëlle Stoelinga, Jaco van de Pol
University of Twente, The Netherlands
Email: {w.ahmad, m.r.jongerden, m.i.a.stoelinga, j.c.vandepol}@utwente.nl

***Abstract*—System lifetime is a major design constraint for battery-powered mobile embedded systems. The increasing gap between the energy demand of portable devices and their battery capacities is further limiting durability of mobile devices. Thus, the guarantees over Quality of Service (QoS) of battery-constrained devices under strict battery capacities are of primary interest for mobile embedded systems' manufacturers and stakeholders.**

This paper presents a novel approach for deriving QoS of applications modelled as synchronous dataflow (SDF) graphs. We map these applications on heterogeneous multiprocessor platforms that are partitioned into Voltage and Frequency Islands, together with multiple kinetic battery models (KiBaMs). By modelling the whole system as hybrid automata, and applying model-checking, we evaluate, (1) system lifetime; and (2) minimum required initial battery capacities to achieve the desired application performance. We demonstrate that our approach shows a significant improvement in terms of scalability, as compared to a priced timed automata based KiBaM model. This approach also allows early detection of design errors via model checking.

I. Introduction

Mobile computing has experienced a major upswing over the last two decades. As a result, applications with increasing functionality and complexity are continuously implemented on mobile embedded devices such as smart phones, allowing these systems to operate independently. However, this trend has increased the energy consumption of mobile devices manifold. On the other hand, battery energy densities have not grown at the same rate over the years, thus leading to system lifetime as a major design constraint. In this paper, we define the lifetime as the time one can use the battery before it is empty.

Mobile embedded systems are often powered only by batteries that may or may not be recharged regularly by an external power source. For example, in a military Software Defined Radio that is being operated in a desert or on a mountain where energy supplies are unreliable, the primary Quality of Service (QoS) concern is to determine the system lifetime. Similarly, a geostationary satellite with solar panels to charge on-board batteries, is recharged at a regular intervals of 12 hours when facing the sun. However, the satellites have strict limitations regarding mass and volume. In this case, the main QoS interest is to assess the battery sizes and weight that yield the performance criteria. In these cases, the evaluation of the QoS of battery-constrained mobile embedded systems has emerged as one of the most critical, challenging and essential concern for manufacturers, investors and users.

One can identify three QoS factors, and their relation with three different design choices, as given in Table I. First, the *throughput* of a system, defined as a measure of how many units of information a system can process in a given amount of time, has a direct impact on energy consumption which in turn influences the system lifetime. Secondly, the number of processors affects both the system lifetime, and manufacturing cost of the overall system. Lastly, the number of batteries relates not only to system lifetime and cost, but also to the mass and volume of a system. Therefore, this paper takes in account aforementioned design alternatives, with respect to system lifetime and minimum battery capacities.

This paper considers a very intuitive battery model termed Kinetic Battery Model (KiBaM) [16] as a representation of dynamic behaviour of a conventional rechargeable battery, see Figure 1. A KiBaM models the total charge in a battery as two separate tanks. One tank holds the charge which is immediately available to be consumed by the load. The other tank holds the charge which is chemically bound. It is because a chemical kinetics process is used as its basis, that the model is termed kinetic. Experimental studies show that the KiBaM provides a good approximation of the system lifetime across various battery types [13].

To reduce the power consumption, different system-level power management methods like Dynamic Power Management (switching to low power state) (DPM) [7] and Dynamic Voltage and Frequency Scaling (throttling processor frequency) (DVFS) [19] has gained significant value and success. The concept of voltage-frequency islands (VFIs) [11] further allows us to cluster a group of processors in such a way that each VFI runs on a common clock frequency/voltage. This achieves fine-grained system-level power management. In CMOS based processors, voltage/frequency scaling by a factor of s causes the battery current to scale by s^3 [8]. Thus, the system lifetime depends hugely on the battery capacity and the level of the load current applied to it.

If we have multiple batteries in the system, another important factor contributing to overall lifetime is the usage pattern of the batteries, i.e., how batteries are scheduled. This leads to an important research problem of devising a battery-aware scheduling mechanism, where given a set of tasks, a set of resources to execute the tasks, and a given number of batteries, we are able to derive a battery-optimal schedule of tasks.

Design Choices \ QoS Factors	System Lifetime	Cost	Volume and Mass
Required Throughput	✓		
Number of Processors	✓	✓	
Number of Batteries	✓	✓	✓

TABLE I: Relation between Design Choices and QoS factors

The charge stored in a battery is represented by a finite set of continuous variables in the KiBaM, making the behaviour of KiBaM hybrid. Synthesising optimal battery schedules for multiple batteries using existing analysis techniques for hybrid systems, is very expensive. Therefore, the state-of-the-art method in [14] discretises the KiBaM, and models it as priced timed automata (PTA) [6]. Furthermore, for a fixed load, this approach deploys the model-checker UPPAAL Cora to search the whole state-space, and generating the optimal battery schedule. However, this method also does not solve the scalability problem. As increasing the initial battery capacities leads to searching the bigger state-space, this approach only allows to model limited battery capacities. Furthermore, this approach discretises the temporal dimension, which limits its accuracy.

We propose an alternative, novel approach based on Hybrid Automata (HA) [10]. These extend timed automata [4] (for the modelling of time-critical systems and time constraints) by continuous variables. HA can be analysed using UPPAAL [5], that supports both model-checking and highly scalable Monte Carlo simulations. In contrast to discretisation, as done in [14], we take into account the continuous variables of the KiBaM by modelling it as a hybrid automaton, which obviously makes it a more accurate model than PTA-based KiBaM. This approach enables us to utilise UPPAAL to employ highly scalable technique of Monte Carlo simulations to assess various QoS parameters, such as, system lifetime and adequate battery capacities. In this paper, we show that our approach scales better than the one presented in [14]. Furthermore, we also utilise UPPAAL for applying model checking to verify various user-defined properties. Thus, as opposed to other simulation based tools for hybrid systems, modelling as HA and using UPPAAL provides an additional benefit of model checking against state-based properties.

We use Synchronous Dataflow (SDF) [15] as a computational model. SDF provides a natural representation of real-time streaming and digital signal processing applications. In this paper, SDF graphs are used to represent software applications which are partitioned into tasks, with inter-task dependencies and their synchronisation properties.

Our approach takes four ingredients: (1) a platform model that describes the specifics of the hardware, such as, VFI partitions, frequency levels and power usage per processor; (2) an SDF graph scheduler that maps the application tasks on the platform model in a static-order manner; (3) given number of batteries; and (4) a battery scheduler that defines the scheduling scheme of batteries. For given battery capacities and timing constraints, we compute system lifetime (SDF graph iterations). Similarly, for given application performance criteria, we determine the adequate battery capacities. This method facilitates system designers to evaluate aforementioned QoS factors for different design choices, such as, varying number of VFIs, processors, and batteries. Furthermore, this method also allows system designers to detect subtle battery design errors in early phases via model checking.

In particular, our main contributions are as follows. (1) We utilise hybrid automata to model check and assess QoS of multiple KiBaMs for different design alternatives, without discretising time. (2) We consider realistic hardware platforms equipped with the novel energy management techniques, compared to the state-of-the-art [20]; (3) We analyse SDF graphs as input which are more versatile and allow more realistic data-dependencies than acyclic applications [9] [14] [20]; (4) We show that our approach allows better scalability than PTA-based discretised KiBaM [14]; (5) Our approach allows early detection of design errors via model checking.

Paper organisation. Section II reviews related work. Section III formalises the problem, and translation to HA using UPPAAL is illustrated in Section IV. Section V experimentally evaluates the QoS analysis. Finally, Section VII draws conclusions and outlines possible future research.

II. RELATED WORK

An extensive survey paper [13] outlines the broad research work on various battery models. The work in [3] applies the combination of DVFS and DPM, on VFI-partitioned heterogeneous processors. However, this paper assumes energy source to be ideal. We solve this limitation by considering a realistic model of batteries.

The state-of-the-art methods in the realm of battery-aware scheduling for multiple batteries, are presented in [14] and [9]. The approach in [14], in comparison to ours, discretises time. This approach helps to find optimal battery schedules, but do not scale well because of the discretisation. The technique in [9] models KiBaMs as hybrid like us, and discretises time to search the state-space, leading to the better results than the work in [14]. But, due to the fact that the state-space grows larger with the number of batteries, the scalability of this approach also suffers. We, on the other hand, run Monte Carlo simulations, that allows us to avoid the state-space explosion. The analysis shows that the scalability of our approach is better than the technique in [14].

A more advanced technique that also utilises hybrid automata like us and [9], is presented in [20]. In this paper, the KiBaM provides energy to an uniprocessor. Unlike our method, this approach discusses a single battery case only. Another novel work in [12] extends KiBaMs with random initial SoC and load, without discretising time. In this way, probabilistic guarantees about the system lifetime can be provided. In comparison to our work, this technique is also confined to a single KiBaM only. Table II summarises different aforementioned KiBaM analysis methods.

To the best of our knowledge, there are no papers that

Method	Without Discretisation	Multiple KiBaMs
[9] [14]	✗	✓
[20]	✓	✗
[12]	✓	✗
Our Method	✓	✓

TABLE II: Comparison among different battery analysis methods

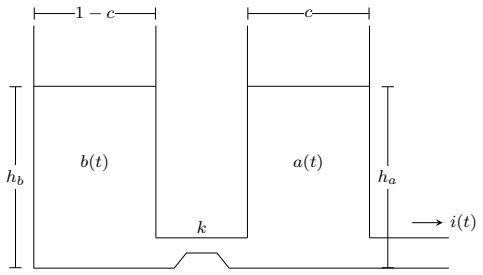


Fig. 1: Model of a KiBaM

analyse *multiple* KiBaMs *without* discretising time.

III. System Model Definition

We formalise battery model, SDF graphs, and hardware platform model in subsection III-A, III-B and III-C respectively.

A. Kinetic Battery Model

The kinetic battery model (KiBaM) [16] is a mathematical characterisation of state of charge of a battery. Key feature is that not all energy stored in a battery can be utilised at all times. To model this phenomenon, the total charge stored in a battery is divided into two "tanks", the *available charge* and the *bound charge*, see Figure 1. Only the available charge can be consumed immediately by a *load* at the time-dependent rate i, and thereby behaves similar to an ideal energy source. The bound charge is converted to available charge at a rate proportional to the height difference with the proportionality factor being the rate constant k, and is available to be consumed. Thus, *bound charge* replenishes *available charge*, and this effect is termed *recovery effect*.

If the widths of the available and bound charge tanks are c and $1-c$ respectively, then the tanks are filled to heights h_a and h_b, and the charges in both tanks are $a = ch_a$ and $b = (1-c)h_b$ respectively. Formally, the KiBaM is characterised by the following system of differential equations.

$$\dot{a}(t) = -i(t) + k(h_b - h_a) \tag{1}$$

$$\dot{b}(t) = -k(h_b - h_a) \tag{2}$$

The system starts in equilibrium, i.e. $h_a = h_b$. With an initial capacity of C, the initial conditions are $a(0) = cC$ and $b(0) = (1-c)C$. The battery is considered empty when $a = h_a = 0$, as it cannot supply charge any more at the given moment even though it may still contain bound charge. The system lifetime ends when all batteries are emptied.

Definition 1. A *KiBaM system* $\mathcal{KS} = (B, Cap)$ consists of a finite set of KiBaMs $B = \{bat_1, \ldots, bat_m\}$, and a function $Cap : B \to \mathbb{R}_{\geq 0}$ denoting the initial capacity of each $bat \in B$.

In our case-studies, we consider batteries having the capacity of 1300 mAh, as used in the Samsung Galaxy Fame smartphones [1].

B. SDF Graphs

Typically, real-time streaming applications execute a set of periodic tasks, which consume and produce a fixed amount of data. Such applications are naturally modelled as SDF graphs [15]: a directed, connected graph in which tasks are represented by *actors*. Actors communicate with each other via streams of data elements, represented by *tokens*. The tokens are transported between the actors via *edges*. The execution of an actor is known as an (*actor*) *firing*.

Definition 2. An *SDF graph* is a tuple $G = (A, D, \mathsf{Tok}_0, \tau)$ where: A is a finite set of actors, $D \subseteq A^2 \times \mathbb{N}^2$ is a finite set of dependency edges, $\mathsf{Tok}_0 : D \to \mathbb{N}$ denotes distribution of initial tokens in each edge, and the execution time of each actor is given by $\tau : A \to \mathbb{N}_{\geq 1}$.

Definition 3. Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$, the sets of *input* and *output edges* of an actor $a \in A$ are defined respectively as $In(a) = \{(a', a, p, q) \in D | a' \in A, p, q \in \mathbb{N}\}$ and $Out(a) = \{(a, b, p, q) \in D | b \in A, p, q \in \mathbb{N}\}$. The *consumption* and *production rate* of an edge $e = (a, b, p, q) \in D$ are defined respectively as $CR(e) = q$ and $PR(e) = p$.

Informally, actor a can fire if each input edge $(a', a, p, q) \in In(a)$ of a contains at least q tokens; firing actor a removes q tokens from the input edge (a', a, p, q). Firing lasts for $\tau(a)$ time units and ends by producing p' tokens on each output edges $(a, b, p', q') \in Out(a)$.

Example 1. Figure 2 shows the SDF graph of an MPEG-4 decoder [18]. The SDF graph contains five actors A={FD, VLD, IDC, RC, MC}, representing the tasks performed in MPEG-4 decoding. For example, the frame detector (FD) determines the number of macro blocks to decode. To decode a single frame, FD must process between 0 and 99 macroblocks, i.e., $x \in \{0, 1, \ldots, 99\}$ in Figure 2.

Arrows between the actors depict the edges which hold tokens (dots) representing macroblocks. The execution time (ms) of the actors is represented by a number inside the actor nodes. The numbers near the source and destination of each edge are the production and consumption rates respectively.

To avoid unbounded accumulation of tokens in a certain edge, we require SDF graphs to be *consistent*.

Definition 4. A *repetition vector* of an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ is a function $\gamma : A \to \mathbb{N}_0$ such that for every edge $(a, b, p, q) \in D$ from $a \in A$ to $b \in A$, the relation $p.\gamma(a) = q.\gamma(b)$ exists. An SDF graph is *consistent* iff $\gamma(a) > 0$ for all $a \in A$.

Definition 5. Let us consider an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ with a repetition vector γ. An *iteration* of G is defined as a set of actor firings such that for each $a \in A$, the set contains exactly $\gamma(a)$ firings of actor a. Thus, each actor fires according to γ in an iteration.

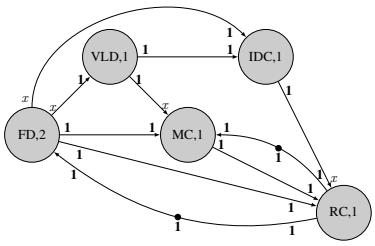


Fig. 2: SDF Graph of an MPEG-4 Decoder

C. Platform Application Model

A Platform Application Model (PAM) models a multi-processor platform where the application, modelled as SDF graph, is mapped on. Our PAM models supports several features, including (1) *heterogeneity*, i.e., actors can run on certain type of processors only; (2) a partitioning of the processors in *voltage and frequency islands*; (3) different *frequency levels* each processor can run on; (4) power consumed by a processor in a certain frequency, both when in use and when idle; (5) *power and time-overhead* required to switch between frequency levels.

Definition 6. A *platform application model* (PAM) is a tuple $\mathcal{P} = (\Pi, \zeta, F, I_{occ}, I_{idle}, I_{tr}, T_{tr}, \tau_{act})$ consisting of

- a finite set of *processors* Π assuming that $\Pi = \{\pi_1, \ldots, \pi_n\}$ is partitioned into disjoint blocks $\Pi_1, \ldots, \Pi_k$ of voltage/frequency islands (VFIs),
- a function $\zeta : \Pi \to 2^A$ indicating which processors can handle which actors,
- a finite set of discrete *frequency levels* available to all processors denoted by $F = \{f_1, \ldots, f_m\}$,
- a function $I_{occ} : \Pi \times F \to \mathbb{N}$ denoting the *operating load current*, if the processor $\pi \in \Pi$ is running at a certain frequency level $f \in F$ in the working state,
- a function $I_{idle} : \Pi \times F \to \mathbb{N}$ denoting the *idle load current*, if the processor $\pi \in \Pi$ is running at a certain frequency level $f \in F$ in the idle state,
- a function $I_{tr} : \Pi \times F^2 \to \mathbb{N}$ expressing the *transition load current*, if processor $\pi \in \Pi$ switches the running frequency from $f \in F$ to another frequency $f \in F$,
- a function $T_{tr} : \Pi \times F^2 \to \mathbb{N}$ expressing the *time overhead* of switching from one frequency level $f \in F$ to another $f \in F$ for each processor $\pi \in \Pi$, and
- the valuation $\tau_{act} : A \times F \to \mathbb{N}_{\geq 1}$ defining the execution time τ_{act} of each actor $a \in A$ mapped on a processor at a certain frequency level $f \in F$.

Example 2. Exynos 4210 is a state-of-the-art processor used in high-end platforms such as Samsung Galaxy Note, SII etc, and has 5 DVFS levels [17]. Later in Section V, we consider Exynos 4210 as an use-case for experimental evaluation.

Definition 7. Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$, a *static-order* (SO) schedule is a function $\sigma : \Pi \times \mathbb{R} \to (A \times F) \cup (\perp \times F)$ that assigns to each processor $\pi \in \Pi$ over time, an ordered list of actors or idle slots to be executed at some frequency, where $\perp$ represents the idle slots.

Definition 8. The *throughput* for a static-order schedule of an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ is the average number of graph iterations that are executed per time unit, measured over a sufficiently long period.

As discussed earlier, in case of more than one battery in the system, the batteries are chosen according to some schedule or scheduling policy. In most systems, the batteries are used sequentially, i.e., only when one battery is empty, the other is used [14]. However, as shown in [14], a specific scheduling scheme termed *best-of-all* achieves better system lifetime than other schemes. For the same reason, we consider best-of-all in this paper. In this scheduling scheme, after an SDF graph iteration finishes, (i.e., not during the execution of the iteration) the battery having the highest available charge is selected to provide energy for the next iteration.

IV. Translation to Hybrid Automata

A. Hybrid Automata

Timed automata are a popular and powerful formalism to model and analyse real-time systems [4]. TA are state-transition diagrams augmented with real-valued clocks, which can be used in enabling conditions for transitions and in state invariants that enforce deadlines.

Hybrid automata extend timed automata by continuous variables, which we use to model hybrid behaviour of the batteries. Let X be a finite set of continuous variables. A variable valuation over X is a mapping $v : X \to \mathbb{R}$, where $\mathbb{R}$ is the set of reals. We write $\mathbb{R}^X$ for the set of valuations over X. Valuations over X evolve over time according to delay functions $F : \mathbb{R}_{\geq 0} \times \mathbb{R}^X \to \mathbb{R}^X$, where for a delay d and valuation v, $F(d, v)$ provides the new valuation after a delay of d.

Definition 9. A hybrid automaton $\mathcal{H}$ is a tuple $(L, Act, X, E, F, Inv, l^0)$, where L is a finite set of *locations*; Act is a finite set of actions, co-actions and internal λ-actions; X is a finite set of continuous variables; E is a finite set of edges of the form (l, g, a, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$, action label $a \in Act$ and φ is a binary relation on $\mathbb{R}^X$; Inv assigns an invariant predicate $Inv(l)$ to any location l; for each location $l \in L$, $F(l)$ is a delay function; and $l^0 \in L$ is the *initial location*.

In particular, HA can be analysed by the tool Uppaal, where each component of the system is described by an automaton whose clocks can evolve with various rates. These rates can be specified with, e.g., ODEs. We utilise Uppaal engine to perform Monte Carlo simulations to estimate QoS, and to perform model checking.

B. Translation to Hybrid Automata

Our framework consists of separate models of KiBaMs, a KiBaM scheduler, an SDF graph scheduler, and the processor application model. In this way, we divide the problem of evaluating the QoS in terms of power source, tasks and resources. In this section, we describe the translation of an SDF graph scheduler along with a processor application model and KiBaMs to HA using Uppaal.

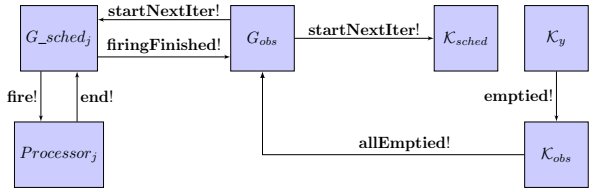


Fig. 3: Interactions between HA of different components

Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ mapped on a processor application model $(\Pi, \zeta, F, T_{tr}, \tau_{act})$ powered by a KiBaM system $\mathcal{KS} = (B, Cap, I_{occ}, I_{idle}, I_{tr})$, we generate a parallel composition of HA:

$$\mathcal{K}_{sched} \| \mathcal{K}_1 \| ,..., \| \mathcal{K}_m \| \mathcal{K}_{obs} \| G\_sched_1 \| ,..., \| G\_sched_n \| Processor_1 \| ,..., \| Processor_n \| G_{obs}$$

Here, the automaton $\mathcal{K}_{sched}$ models the scheduling scheme of batteries. This paper considers the best-of-all scheduling scheme, i.e, after every iteration, the battery with the highest available charge is chosen to serve for the next iteration. The HA $\mathcal{K}_1, \ldots, \mathcal{K}_m$ model the batteries $B = \{bat_1, \ldots, bat_m\}$. The automaton $\mathcal{K}_{obs}$ keeps account of the end of system lifetime by counting the empty batteries. Similarly, the automaton G_sched implements the static-order firing of SDF actors on the processors. The HA $Processor_1, \ldots, Processor_n$ model the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$ The SDF graph observer G_{obs} counts if each processor has fired all its mapped actors, according to its static-order schedule. Hence, this automaton determines when an iteration is finished. Note that the resulting hybrid automata is trivially extensible in the number of processors and batteries. Thus, the translation is, at least, composable with regards to the KiBaM system and PAM.

Figure 3 shows the interactions between the HA of different components. The interested readers can see some of the UPPAAL models in Appendix. Due to lack of space, we refer to [2] for more details on translation and UPPAAL models.

V. EXPERIMENTAL EVALUATION VIA MPEG-4 DECODER

We evaluate QoS factors by means of an example of the MPEG-4 decoder in Figure 2. We assume that MPEG-4 decoder is mapped on Exynos 4210 processors. The processors are provided energy by Samsung batteries. In our experiments, we use $c = 1/6$ and $k = 2.2324 \times 10^{-4} s^{-1}$, as in [14]. The processors are available with two frequency levels (MHz) $f_2 = 1400$ and $f_1 = 1032.7$ [17]. Table III shows the idle and operating load currents of all processors at both frequencies. We evaluate the system lifetime in terms of completed number of video frames, with respect to various design choices discussed earlier, i.e., varying (1) frames per second (throughput); (2) number of processors; and (3) batteries. For these choices, we assess adequate battery capacities. Due to space limitation, we present adequate battery capacities only for one battery. The SO schedules considered in this section are given in [2].

A. Varying Frames per Second

Let us consider that we have 6 Exynos 4120 processors $\Pi = \{\pi_1, \ldots, \pi_6\}$ served by two batteries $B = \{bat_1, bat_2\}$. We consider different SO schedules with varying frames per second (fps) constraints. For those SO schedules, Figure 4 shows the total number of video frames per second completed. As we can see from Figure 4, at tighter performance constraints (throughput) when the idle time of processors is not sufficient to move to low power state, the batteries are drained more rapidly. Thus, we achieve fewer iterations. If we require fewer frames per second from an MPEG-4 decoder, then more iterations are acquired.

Voltage(V)	Frequency(MHz)	I_{idle}(mA)	I_{occ}(mA)
1.2	1400	20	500
1.00	1032.7	8	190
1.2	1400	20	500
1.00	1032.7	8	190
1.2	1400	20	500
1.00	1032.7	8	190
1.2	1400	20	500
1.00	1032.7	8	190

TABLE III: Description of Exynos 4210 Processors

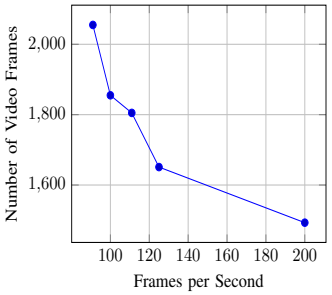


Fig. 4: System lifetime against varying fps

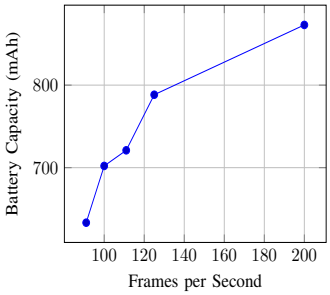


Fig. 5: Min. required capacity for bat_1

For the same SO schedules considered earlier, we compute the minimum battery capacities to complete 1000 video frames. The minimum required initial capacity $Cap(bat_1)$ for the battery $bat_1 \in B$ is shown in Figure 5. It can be seen from Figure 5 that if we relax the frames per second constraint, the minimum required capacity also decreases.

Nevertheless, if the video quality is enhanced from 125 to 200 fps, then the increase in required battery capacity is relatively small, equal to 84 mAh. However, the improvement in the video quality is considerable. Thus, higher performance can also be achieved at the expense of a small increase in the battery capacities, leading to high-performance systems with less mass and volume. Hence, this method allows us to obtain a Pareto front by sweeping throughput constraints, for a fixed number of processors and batteries.

B. Varying Number of Processors

In this case, we consider two batteries, and different SO schedules all of which yield 71 fps. Figure 6 shows the total number of video frames completed for varying number of processors. As we can see from Figure 6, for the same battery capacities, more processors achieve more or equal number of frames. The reason is that, if we reduce the number of processors, the same amount of work is done on fewer processors to attain same throughput, resulting in running on higher frequencies most of the time. Therefore, battery charge is consumed more rapidly, if the number of processors are

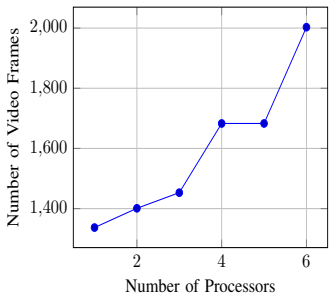


Fig. 6: System lifetime against varying processors

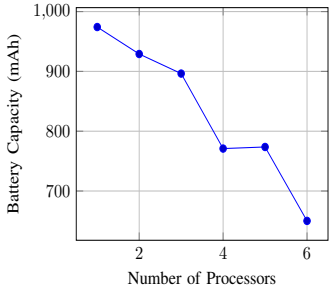


Fig. 7: Minimum required capacity for bat_1

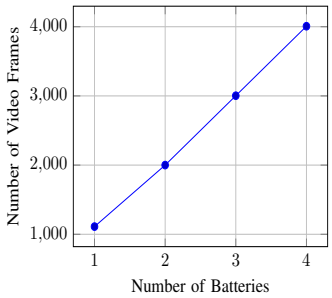


Fig. 8: System lifetime against varying batteries

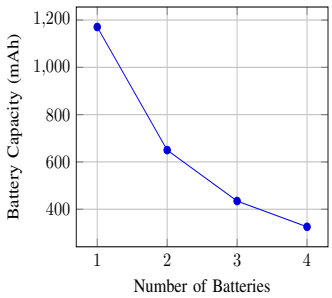


Fig. 9: Minimum required capacity for bat_1

reduced.

For the same SO schedules, Figure 7 shows the minimum required capacity $Cap(bat_1)$ to complete 1000 video frames. The results reiterate the earlier conclusions in Figure 6 that, to achieve the same throughput, fewer processors carrying out the work of same magnitude requires larger battery capacities.

Hence, using this method, a system designer can estimate QoS for different design alternatives. For instance, in our running example, one can clearly see that we can achieve same throughput for 4 processors, as 5, without requiring extra capacities for batteries. Therefore, we may not need more processors in our platform, and reach a certain throughput with fewer number of processors, and same battery capacities, contributing to low-cost embedded systems with reduced mass and volume.

C. Varying Number of Batteries

Let us assume that we have 6 Exynos 4120 processors. We further consider a SO schedule producing 71 fps. For varying batteries, Figure 8 and 9 shows the total number of video frames completed, and minimum required initial capacity for $bat_1 \in B$ to complete 1000 video frames respectively. As it can be seen from Figure 8, increasing the number of batteries improves the attainable number of frames linearly.

However, if we analyse the Figure 9, we can see that increasing the number of batteries does not reduce the battery capacities at a linear rate. Therefore, we can conclude that, having fewer batteries with larger capacities is more beneficial than higher number of batteries with smaller capacities. This achieves low-cost and high-performance systems.

$Cap(bat_1)$	$Cap(bat_2)$	S1 (computation time)		S2 (computation time)		S3 (computation time)	
		PTA-KiBaM	HA-KiBaM	PTA-KiBaM	HA-KiBaM	PTA-KiBaM	HA-KiBaM
1.25$\times10^{-4}$	**1.25**$\times10^{-4}$	**0 (520)**	**0 (28)**	**0 (200)**	**0 (46)**	**0 (2130)**	**0 (51.4)**
2.5$\times10^{-4}$	**2.5**$\times10^{-4}$	**0 (510)**	**0 (55)**	**1 (41060)**	**0 (48)**	*Out of Memory*	**0 (52.7)**
3.75$\times10^{-4}$	**3.75**$\times10^{-4}$	*Out of Memory*	**2 (62)**	**1 (14810)**	**0 (49)**	*Out of Memory*	**2 (52.8)**
5$\times10^{-4}$	**5**$\times10^{-4}$	*Out of Memory*	**4 (64)**	*Out of Memory*	**2 (49)**	*Out of Memory*	**4 (54.1)**

TABLE IV: Comparison of two approaches wrt varying battery capacities.

Batteries	HA-KiBaM	PTA-KiBaM
1	1	N/A
2	4	*Out of Memory*
3	6	*Out of Memory*
4	8	*Out of Memory*
5	9	*Out of Memory*
6	12	*Out of Memory*
7	14	*Out of Memory*
8	16	*Out of Memory*
9	17	*Out of Memory*
10	20	*Out of Memory*

TABLE V: Comparison of two approaches wrt number of batteries.

D. Comparison with PTA-KiBaM

In this subsection, we compare the approach presented in this paper (HA-KiBaM) with the PTA-based approach (PTA-KiBaM) [14]. Let us take the example of an MPEG-4 decoder in Figure 2. We assume that there are two batteries $B = \{bat_1, bat_2\}$ in the system. We consider three arbitrary SO schedules. i.e., S1, S2 and S3. It is worth mentioning that the work in [14] evaluates the completed number of tasks, instead of iterations. However, as iterations are a key metric in SDF graphs, we compare both techniques in terms of completed number of iterations (video frames per second).

Columns 3-8 in Table IV show the completed number of iterations and computation time (ms), calculated using both methods, against different battery capacities (mAh) in Columns 1-2. The experiments were run on a dual-core 2.8 GHz machine with 8 GB RAM. Table IV shows that HA-KiBaM achieves the same results as PTA-KiBaM except S2. The reason of not producing the same results in S2 is that PTA-KiBaM allows to change the active battery during the iteration. Whereas, we consider a specific scheduling scheme, where we change the battery after an iteration is finished.

However, the biggest advantage of HA-KiBaM is the scale of capacities it can handle. As Table IV shows, PTA-KiBaM can only handle very small battery capacities that are able to finish not more than one video frame. This makes PTA-KiBaM impracticable for modern-day systems, as opposed to our method that scales to much larger capacities (see Section V). Furthermore, PTA-KiBaM requires considerably longer computation time than HA-KiBaM. Please note that zero in Table IV means that the battery capacities are not enough, even to finish one iteration (frame per second).

In addition to the battery capacities, our method also scales better to the number of batteries. Table V compares the iterations completed for varying number of batteries for both methods. For this experiment, we consider SO schedule S3, and $Cap(bat) = 5\times10^{-4}$ mAh for all $bat \in B$.

VI. Model Checking via MPEG-4 Decoder

In this section, we demonstrate the analysis of functional and temporal properties, using the Uppaal model checker and

its query language. We consider the case-study of an MPEG-4 decoder mapped on Exynos 4210 processors, and powered by a KiBaM system.

Deadlock

Checking deadlock freedom is achieved via the UPPAAL query (A[] not deadlock). This query allows us to check if a certain static-order schedule is deadlock free or not.

Parallel firings of actors

We can check whether any actors can fire in parallel. For example, actors p and q mapped on the processors π_0 and π_1 respectively, can fire in parallel if the query $\texttt{E <> SDFScheduler}_0\texttt{.activeActor == p and SDFScheduler}_1\texttt{.activeActor == q}$ evaluates to true. In our experiment, $p = MC$ and $q = RC$. As these two actors cannot fire in parallel, the answer to this query turns out to be false.

Same running frequency in a VFI

We can also check safety properties such as, at a given time, all processors belonging to the same VFI should not run at the different frequency. For this purpose, we create a variable named "curr_freq"available to all processors, that keeps account of current running frequency of each processor. If we have two processors π_0 and π_1 in a same VFI, then we check the query $\texttt{A[]Processor}_0\texttt{.curr\_freq == Processor}_1\texttt{.curr\_freq}$ to verify this property.

In the same way, functional correctness of properties related to KiBaMs can also be verified. However, to verify hybrid properties, UPPAAL offers statistical model checking instead of classical model checking, even though we do not have stochastic properties in our system. In the following, we demonstrate model checking of functional requirements of KiBaMs.

Fair Scheduling

We can also verify if only the best battery out of all batteries is selected after each iteration. Let us assume that we have two batteries, and an integer $empty\_count$ to count empty number of batteries. We run the query $\texttt{Pr[<= 1500](<> bound\_1} - \texttt{bound\_0 > n and empty\_count} < 2)$ that determines if difference between the bound charge of two batteries does not exceed more than a certain amount, and each battery gets a fair chance to recover its bound charge. UPPAAL answers that the probability for this query to hold is [0,0.0973938] with 0.95 confidence, which means that this property is not satisfied. In our experiments, n is 4.

Active Number of Batteries

Similarly, we can also check that no more than one battery should be active at any given time. Let us assume that we have two batteries, and boolean variables $b0\_active$ and $b1\_active$ is assigned to each battery respectively, to check if that battery is active. To verify this property, we use the query $\texttt{Pr[<= 1500](<> b0\_active == true and b1\_active == true})$. The probability for this property to hold is [0,0.0973938] with 0.95 confidence, which means that this property is not satisfied.

VII. CONCLUSIONS

With the growing gap between the energy demand and battery densities, yet compact methods for guaranteeing QoS of multiple KiBaMs are needed. We have presented a novel technique to predict system lifetime for SDF-modelled streaming applications, mapped on the processors equipped with energy reduction techniques and powered by multiple batteries. This provides us with a best trade-off between the throughput, the number of processors and batteries. The batteries are modelled as a hybrid system, which has the advantage of being accurate.

Future research direction is to explore the possibilities of battery-aware scheduling. We also plan to analyse preemptive scheduling, by having an observer automaton to record the elapsed execution time during the execution of an actor.

ACKNOWLEDGEMENT

This research is supported by the EU FP7 project SENSATION (318490). The authors would like to thank reviewers for their valuable comments.

REFERENCES

[1] Samsung Galaxy Fame Description. http://www.samsung.com/uk/consumer/mobile-devices/smartphones/others/GT-S6810PWNBTU.

[2] W. Ahmad, M. Jongerden, M. Stoelinga, and J. van de Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata (extended version). Technical Report TR-CTIT-16-03, University of Twente, 2016.

[3] W. Ahmad, P.K.F. Hölzenspies, M.I.A. Stoelinga, and J. van de Pol. Green computing: Power optimisation of VFI-based real-time multiprocessor dataflow applications. In *DSD'15*, pages 271–275, Aug 2015.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on SFM-RT '04*, LNCS, pages 200–236. Springer, 2004.

[6] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, Mar. 2005.

[7] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, June 2000.

[8] P. Chowdhury and C. Chakrabarti. Battery aware task scheduling for a system-on-a-chip using voltage/clock scaling. In *SIPS'02*, pages 201–206, Oct 2002.

[9] M. Fox, D. Long, and D. Magazzeni. Automatic construction of efficient multiple battery usage policies. In *IJCAI'11*, pages 2620–2625, 2011.

[10] T. Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292, Jul 1996.

[11] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISLPED'07*, pages 38–43, Aug 2007.

[12] H. Hermanns, J. Krcál, and G. Nies. Recharging probably keeps batteries alive. *CoRR*, abs/1502.07120, 2015.

[13] M. Jongerden and B. Haverkort. Which battery model to use? *Software, IET*, 3(6):445–457, December 2009.

[14] M. Jongerden, B. Haverkort, H. Bohnenkamp, and J. Katoen. Maximizing system lifetime by battery scheduling. In *DSN'09*, June 2009.

[15] E. A. Lee and D. G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON '87*, pages 310–315, 1987.

[16] J. F. Manwell and J. G. McGowan. Lead acid battery storage model for hybrid energy systems. *Solar Energy*, 50(5):399 – 405, 1993.

[17] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *TCAD*, May 2013.
[18] B. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE'06*, pages 185–194, July 2006.
[19] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *OSDI'94*. USENIX Association, 1994.
[20] E. R. Wognsen, R. R. Hansen, and K. G. Larsen. Battery-aware scheduling of mixed criticality systems. In *ISoLA*, pages 208–222, 2014.

APPENDIX

Here we show the HA models with respect to the MPEG-4 decoder example in Figure 2 that is mapped on Exynos 4210 processors and powered by KiBaMs, in Figure 10.

a) Hybrid Automaton $\mathcal{K}_{sched}$.: The hybrid automaton $\mathcal{K}_{sched}$ models the best-of-all scheduling scheme for KiBaMs, as shown in Figure 10a.

The automaton $\mathcal{K}_{sched}$ is defined as, $\mathcal{K}_{sched} = (L, Act, X, E, F, Inv, l^0)$. For each battery $bat_y \in B$, we include a location $L = \{\mathsf{avail\_baty}\}$ to indicate which the battery is currently active. For $B = \{bat_1, bat_2, \ldots, bat_m\}$, the initial location is, $l^0 = \mathsf{avail\_bat1}$, indicating that the battery bat_1 serves first. We do not have any clocks and invariants in $\mathcal{K}_{sched}$. The HA $\mathcal{K}_{sched}$ has one urgent broadcast action, i.e., $Act = \{\mathsf{startNextIter?}\}$ to synchronise with G_{obs} when the current iteration finishes, so that $\mathcal{K}_{sched}$ can choose the best battery for the next iteration. There is no delay function in $\mathcal{K}_{sched}$. The HA $\mathcal{K}_{sched}$ contains one continuous variable $X = \{\mathsf{avail\_y}\}$ to denote the available in each $bat_y \in B$, respectively. $\mathcal{K}_{sched}$ has a variable: active_KiBaM_id that determines the currently active battery. For $B = \{bat_1, bat_2, \ldots, bat_m\}$, the initial value of active_KiBaM_id=1, indicating that the battery bat_1 is the first to serve. For each battery $bat_i \in B$ and $bat_k \in B$, the transition set E have following transitions.

- avail_bati $\xrightarrow{\text{avail\_k>=avail\_i, startNextIter?, active\_KiBaM\_id:=k}}$ avail_batk
- avail_bati $\xrightarrow{\text{avail\_i>avail\_k, startNextIter?, } \varnothing}$ avail_bati
- avail_batk $\xrightarrow{\text{a\_bati>=a\_batk, startNextIter?, active\_KiBaM\_id:=i}}$ avail_bati
- avail_batk $\xrightarrow{\text{avail\_k>avail\_i, startNextIter?, } \varnothing}$ avail_batk

After each iteration finishes, the action StartNextIter synchronises with G_{obs} to start the new iteration. But, before the new iteration starts, the battery $bat_y \in B$ with the highest available charge is determined using the guard conditions. This symbolises that only the battery having highest charge is going to serve for the next iteration, and all other batteries are going to stay idle. For the battery $bat_i \in B$ and $bat_k \in B$, the guard condition $avail\_k >= avail\_i$ on the first transition is checking if the available charge of $bat_k \in B$ is greater than or equal to $bat_i \in B$. If the guard condition turns out to be true, then $bat_k \in B$ provides energy for the next iteration. Otherwise, the guard condition on the second transition, i.e., $avail\_i > avail\_k$ is satisfied, and $bat_i \in B$ stays as the active battery.

b) Hybrid Automata $\mathcal{K}_y$.: The HA $\mathcal{K}_1, \ldots, \mathcal{K}_m$ model the batteries $B = \{bat_1, \ldots, bat_m\}$, according to the description in Section III-A. The model of $bat_y \in B$ is shown in Figure 10b. This automaton informs $\mathcal{K}_{obs}$, when the battery bat_y gets empty.

For each $bat_y \in B$, the HA $\mathcal{K}_y$ is defined as, $\mathcal{K}_y = (L_y, Act_y, X_y, E_y, F_y, Inv_y, l_y^0)$ where $L_y = \{\mathsf{Initial}, \mathsf{Emptied}\}$, and $l_y^0 = \{\mathsf{Initial}\}$. The automaton $\mathcal{K}_y$ contains two continuous variables $X = \{\mathsf{avail\_baty}, \mathsf{bound\_baty}\}$ to denote the available and bound charge in $bat_y \in B$, respectively. There is an urgent broadcast action in $\mathcal{K}_y$, i.e, $Act_y = \{\mathsf{emptied!}\}$ to synchronise with $\mathcal{K}_{obs}$. The automaton $\mathcal{K}_y$ contains number of variables: a boolean variable on_y to determine if the battery has available charge left or whether it has run out of it; and a variable i_y to annotate the load current being consumed from $bat_y \in B$. Initially, we have $on\_y = true$ and $i\_y = 0$. The transition set E_y has only transition, given as follows.

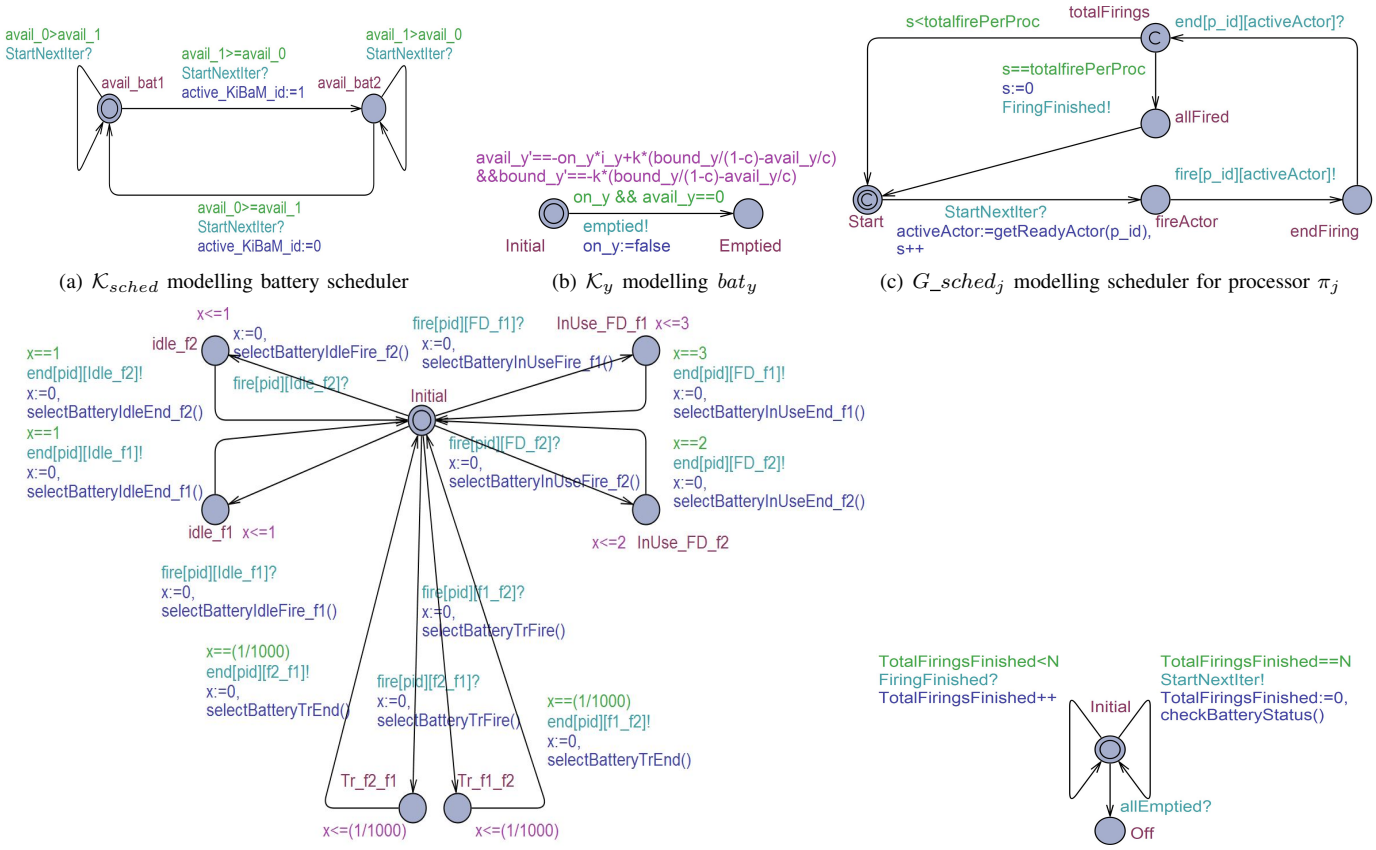
- Initial $\xrightarrow{\text{on\_y}\wedge\text{avail\_y==0, emptied!, on\_y:=false}}$ Emptied

The above transition synchronises with $\mathcal{K}_{obs}$ over the urgent channel emptied!, and is taken if the available charge avail_y reaches or falls below zero, emphasising that the battery $bat_y \in B$ is empty. As a result of this action, the value of on_y changes to false.

The initial location l_y^0 uses equations (1) and (2) as a delay function. This represents that, as long as $bat_y \in B$ is non-empty, the available and bound charge of $\mathcal{K}_y$ evolves according to equations (1) and (2) respectively.

c) Hybrid Automata G_sched_j.: The HA G_sched_j implement the static-order firing of SDF actors on the processors. For this purpose, after G_{obs} informs G_sched_j that an iterations has started, G_sched_j map actors on $Processor_j$ according to the SO schedule of that processor. When all actors are fired according to the SO schedule on $Processor_j$, G_sched_j inform G_{obs} back, indicating the end of current iteration. For a $\pi_j \in \Pi$, Figure 10c presents the automaton G_sched_j, with respect to our running example.

For each $\pi_j \in \Pi$, G_sched_j is defined as, $G\_sched_j = (L_j, Act_j, X_j, E_j, Inv_j, l_j^0)$. where $L_j = \{\mathsf{Start}, \mathsf{FireActor}, \mathsf{EndFiring}, \mathsf{totalFirings}, \mathsf{Off}\}$, and $l_j^0 = \{\mathsf{Start}\}$. The HA G_sched_j contain three broadcast actions, i.e., $Act_j = \{\mathsf{fire!}, \mathsf{end?}, \mathsf{startNextIter?}\}$. The actions fire and end are parametrised with processor and action ids, and are used to synchronise with $Processor_j$. The action StartNextIter synchronises with G_{obs}. The actions fire and StartNextIter are the urgent actions. There are no clocks and invariants in G_sched_j. There are no delay functions and continuous variables in G_sched_j. The HA G_sched_j have a number of local variables: $activeActor\_j$ that determines the active actor currently mapped on the processor π_j; and s_j that determines the index of the active actor in the static-ordered list. Initially, $activeActor\_j = 0$, and $s\_j = 0$. The HA G_sched_j also contain a parametrised variable $totalFirePerProc\_j$, that defines the total number of tasks in the SO schedule of the processor π_j. Since these variables are local, we can abbreviate them by $activeActor$, s and $totalFirePerProc$ respectively. The transition set E_j has following transitions.



(a) $\mathcal{K}_{sched}$ modelling battery scheduler

(b) $\mathcal{K}_y$ modelling bat_y

(c) G_sched_j modelling scheduler for processor π_j

(d) $Processor_j$ showing processor model wrt FD

(e) G_{obs} modelling battery observer

Fig. 10: HA models for KiBaM, KiBaM Scheduler, SDF Scheduler, Processor, and KiBaM Observer

- The following transition fetches the active actor according to the SO schedule for each processor π_j, using the function $getReadyActor(j)$. As a result of this transition, the value of s is incremented by 1, which means that the next actor in the SO schedule is fetched next time.

$$\mathsf{Start} \xrightarrow{\varnothing,\ \varnothing,\ \mathsf{activeActor{:=}getReadyActor(j)} \wedge \mathsf{s{+}{+}}} \mathsf{fireActor}$$

- The following transition maps the fetched (active) actor, on the processor automaton $Processor_j$, using the urgent channel fire!.

$$\mathsf{fireActor} \xrightarrow{\varnothing,\ \mathsf{fire[j][activeActor]!},\ \varnothing} \mathsf{endFiring}$$

- In the following transition, the urgent action end? synchronises with the processor automaton $Processor_j$. As a result, the processor automaton $Processor_j$ informs the automaton G_sched_j that the firing of the active actor has finished.

$$\mathsf{endFiring} \xrightarrow{\varnothing,\ \mathsf{end[j][activeActor]?},\ \varnothing} \mathsf{totalFirings}$$

- The following transition checks if the SO schedule of a processor π_j is not fully executed, using the guard condition $s < totalFirePerProc$. If this is the case, the following transition is taken, leading to the Start location where the next actor in the SO schedule is fetched.

$$\mathsf{totalFirings} \xrightarrow{\mathsf{s{<}totalFirePerProc},\ \varnothing,\ \varnothing} \mathsf{Start}$$

- If all actors in the SO schedule of a processor π_j are executed as checked by the guard condition $s == totalfirePerProc$ on the following transition, the urgent channel FiringFinished! synchronises with the observer automaton G_{obs}. In this way, G_sched_j informs G_{obs} that the processor π_j has executed all of the mapped actors in the current iteration. The variable s is also reset.

$$\mathsf{totalFirings} \xrightarrow{\substack{\mathsf{s{=}{=}totalfirePerProc},\\ \mathsf{firingFinished!},\ \mathsf{s{:=}0}}} \mathsf{allFired}$$

- The following transition synchronises with the observer automaton G_{obs} on the urgent channel StartNextIter? to start executing the static-order schedule of the next iteration.

$$\mathsf{allFired} \xrightarrow{\varnothing,\ \mathsf{startNextIter?},\ \varnothing} \mathsf{Start}$$

d) Hybrid Automata $Processor_j$.: Likewise, the HA $Processor_1, \ldots, Processor_n$ model the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$, as shown in Figure 10d. For better visibility, Figure 10d shows the HA of $Processor_j$, with respect to one actor only, i.e., $FD \in A$.

For each $\pi_j \in \Pi$, we define HA $Processor_j$ =

$(L_j, Act_j, X_j, E_j, F_j, Inv_j, l_j^0)$. The initial location is defined as $l_j^0 = \{\mathsf{Initial}\}$. For each frequency level $f_i \in F$, we include both an idle state and an active state running on that frequency level. For each $a \in \zeta(\pi_j)$ and $F = \{f_1, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_l$, let $L_{mapping} = \{\mathsf{Idle\_f1}, \ldots, \mathsf{Idle\_fl}, \mathsf{InUse\_a\_f1}, \ldots, \mathsf{InUse\_a\_fl}\}$ indicating that the processor $\pi_j \in \Pi$ is currently used by the actor $a \in A$ in the frequency level $f_i \in F$, either in idle or running state. Furthermore, for $F = \{f_1, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_l < f_m$, we have an location which defines the overhead of switching between the frequencies, such that $L_{overhead} = \{\mathsf{Tr\_f1\_f2}, \mathsf{Tr\_f2\_f1}, \ldots, \mathsf{Tr\_fl\_fm}, \mathsf{Tr\_fm\_fl}\}$. Thus, $L_j = L_{mapping} \cup L_{overhead}$. Moreover, for each actor $a \in \zeta(\pi)$ and frequency level $f_i \in F$, $Inv_j(\mathsf{Idle\_fi}) \leq 1$, and $Inv_j(\mathsf{InUse\_a\_fi}) \leq \tau_{act}(a, f_i)$ enforcing the system to stay in InUse_a_fi for at most the execution time $\tau_{act}(a, f_i)$. A processor is in the occupied state only for the time period, when an actor is mapped on it. However, the idle time spent by a processor $\pi_j \in \Pi$ is not a fixed time interval, and a processor $\pi_j \in \Pi$ can stay idle for any finite period of time. Therefore, we divide the idle time spent by a processor $\pi_j \in \Pi$ into slots of one time unit, by annotating $Inv_j(\mathsf{Idle\_fi}) \leq 1$, Similarly, for $F = \{f_1, f_2, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_m$, and $Inv_j(\mathsf{Tr\_f2\_f1}) \leq T_{tr}(\pi, f_1, f_2)$. Please note that $Processor_j$ contains exactly one clock x_j; since clocks in UPPAAL are local, we can abbreviate x_j by x. A separate clock variable $global$ observes the overall time progress.

The action set $Act_j = \{\mathsf{fire?}, \mathsf{end!}\}$ contains two broadcast actions fire?, end!. The actions fire? and end! in Act_j are parametrised with the processor and action ids, and synchronise with G_{sched}.

For each $\pi \in \Pi$, $a \in \zeta(\pi)$ and $f_i \in F$, the transition set E_j contains two transitions such that:

- Initial $\xrightarrow{\varnothing,\ \mathsf{fire}[\pi][\mathsf{a}]?,\ \{\mathsf{x}:=0\} \wedge \mathsf{selectBatteryInUseFire\_fi}()}$ InUse_a_fi, and
- InUse_a_fi $\xrightarrow[\mathsf{selectBatteryInUseEnd\_fi}()]{\mathsf{x}=\tau_{act}(\mathsf{a},\mathsf{f_i}),\ \mathsf{end}[\pi][\mathsf{a}]!,}$ Initial.

The action fire$[\pi][$a$]$ is enabled in the initial state Initial and leads to the location InUse_a_fi. Thus, the action fire$[\pi][$a$]$ is taken, if the actor $a \in A$ is supposed to "claim" the processor $\pi \in \Pi$ at frequency level $f_i \in F$ in the static-order schedule. The function $selectBatteryInUseFire\_fi()$ consumes the charge from the active battery, i.e., $I_{occ}(\pi, f_i)$. As each location InUse_a_fi has an invariant $Inv_j(\mathsf{InUse\_a\_fi}) \leq \tau_{act}(\mathsf{a}, \mathsf{f_i})$, the automaton can stay in InUse_a_fi for at most the execution time of actor $a \in A$ at frequency level $f_i \in F$, i.e., $\tau_{act}(a, f_i)$. If $x = \tau_{act}(a, f_i)$, the system has to leave InUse_a_fi at exactly the execution time of actor $a \in A$ at frequency level $f_i \in F$, by taking the end$[\pi][$a$]$ action. For each $\pi \in \Pi$, and $f_i \in F$, the transition set E contains two transitions for handling broadcast such that:

- Initial $\xrightarrow{\varnothing,\ \mathsf{fire}[\pi][\mathsf{idle\_f_i}]?,\ \{\mathsf{x}:=0\} \wedge \mathsf{selectBatteryIdleFire\_fi}()}$ Idle_fi, and
- Idle_fi $\xrightarrow{\mathsf{x}=1,\ \mathsf{end}[\pi][\mathsf{idle\_f_i}]!,\ \mathsf{selectBatteryIdleEnd\_fi}()}$ Initial.

The action fire$[\pi][$idle_fi$]$ is enabled in the initial state Initial and leads to the location Idle_fi. Thus, fire$[\pi][$idle_fi$]$ causes the processor $\pi \in \Pi$ to go to Idle_fi at frequency level $f_i \in F$, whenever the processor $\pi \in \Pi$ is supposed to stay idle at $f_i \in F$ in the static-order schedule. As the idle slots are divided into time slots of one time unit, each location InUse_a_fi has an invariant $Inv_j(\mathsf{InUse\_a\_fi}) \leq 1$, the automaton can stay in InUse_a_fi for at most 1 time unit. The function $selectBatteryIdleFire\_fi()$ consumes the charge from the active battery, i.e., $I_{idle}(\pi, f_i)$. If $x = 1$, the system has to leave Idle_fi at exactly one time unit, by taking the end$[\pi][$idle_fi$]$ action.

For $F = \{f_1, \ldots, f_l, f_m\}$ such that $f_1 < f_2 < \ldots < f_l < f_m$, and $\pi_j \in \Pi$, the transition set E_j has following transitions such that:

- Initial $\xrightarrow{\varnothing,\ \mathsf{fire}[\pi][\mathsf{f1\_f2}]?,\ \{\mathsf{x}:=0\} \wedge \mathsf{selectBatteryTrFire\_f1\_f2}()}$ Tr_f1_f2,
- Tr_f1_f2 $\xrightarrow[\mathsf{selectBatteryTrEnd\_f1\_f2}()]{\mathsf{x}=T_{tr}(\pi,\mathsf{f_1},\mathsf{f_2}),\ \mathsf{end}[\pi][\mathsf{f1\_f2}]!,}$ Initial,
- Initial $\xrightarrow{\varnothing,\ \mathsf{fire}[\pi][\mathsf{f2\_f1}]?,\ \{\mathsf{x}:=0\} \wedge \mathsf{selectBatteryTrFire\_f2\_f1}()}$ Tr_f2_f1,
- Tr_f2_f1 $\xrightarrow[\mathsf{selectBatteryTrEnd\_f2\_f1}()]{\mathsf{x}=T_{tr}(\pi,\mathsf{f_2},\mathsf{f_1}),\ \mathsf{end}[\pi][\mathsf{f2\_f1}]!,}$ Initial,
- $\vdots$
- Initial $\xrightarrow{\varnothing,\ \mathsf{fire}[\pi][\mathsf{fl\_fm}]?,\ \{\mathsf{x}:=0\} \wedge \mathsf{selectBatteryTrFire\_fl\_fm}()}$ Tr_fl_fm,
- Tr_fl_fm $\xrightarrow[\mathsf{selectBatteryTrEnd\_fl\_fm}()]{\mathsf{x}=T_{tr}(\pi,\mathsf{f_l},\mathsf{f_m}),\ \mathsf{end}[\pi][\mathsf{fl\_fm}]!,}$ Initial,
- Initial $\xrightarrow{\varnothing,\ \mathsf{fire}[\pi][\mathsf{fm\_fl}]?,\ \{\mathsf{x}:=0\} \wedge \mathsf{selectBatteryTrFire\_fm\_fl}()}$ Tr_fm_fl,
- Tr_fm_fl $\xrightarrow[\mathsf{selectBatteryTrEnd\_fm\_fl}()]{\mathsf{x}=T_{tr}(\pi,\mathsf{f_m},\mathsf{f_l}),\ \mathsf{end}[\pi][\mathsf{fm\_fl}]!,}$ Initial

The action fire$[\pi][$fl_fm$]$ causes the processor $\pi \in \Pi$ to incur the transition overhead, whenever the processor $\pi \in \Pi$ is supposed to change the frequency $f_l \in F$ to $f_m \in F$ in the static-order schedule, using the function $selectBatteryTrFire\_fl\_fm()$, and so on.

e) Hybrid Automaton G_{obs}.: The automaton of SDF graph observer G_{obs} counts if each processor has fired its all mapped actors in an static-order schedule, observing the number of iterations finished. Figure 10e shows the HA model of G_{obs}. The automaton G_{obs} has an integer variable Tot_Iter to count the number of finished iterations. Initially, $\mathsf{Tot\_Iter} = 0$.

After modelling the whole system, we run the following query, where $bound$ is the time bound on running the simulation, and Tot_Iter is the variable representing the completed number of iterations. As a result, we get a plot, by which we determine the total number of iterations completed within $bound$ time units. We use the same models and query to determine the adequate batteries' capacities.

$$\texttt{simulate 1[<= bound]\{Tot\_Iter\}}$$