Model Checking and Evaluating QoS of Batteries in MPSoC Dataflow Applications via Hybrid Automata (extended version)*

Waheed Ahmad, Marijn Jongerden, Mariëlle Stoelinga, and Jaco van de Pol

University of Twente, The Netherlands,
{u.ahmad, m.r.jongerden, m.i.a.stoelinga, j.c.vandepol}@utwente.nl

Abstract. System lifetime is always a major design impediment for battery-powered mobile embedded systems such as, cell phones and satellites. The increasing gap between energy demand of portable devices and their battery capacities is further limiting durability of mobile devices. For example, energy-hungry applications like video streaming pose serious limitations on the system lifetime. Thus, guarantees over Quality of Service (QoS) of battery constrained devices under strict battery capacities is a primary interest for mobile embedded systems’ manufacturers and other stakeholders.

This paper presents a novel approach for deriving QoS, for applications modelled as synchronous dataflow (SDF) graphs. These applications are mapped on heterogeneous multiprocessor platforms that are partitioned into Voltage and Frequency Islands, together with multiple kinetic battery models (KiBaMs). By modelling whole system as hybrid automata, and applying model-checking, we evaluate QoS in terms of, (1) achievable application performance within the given batteries’ capacities; and (2) minimum required batteries’ capacities to achieve desired application performance. We demonstrate that our approach shows a significant improvement in terms of scalability, as compared to priced timed automata based KiBaM model [15]. This approach also allows early detection of design errors via model checking.

1 Introduction

Mobile computing has experienced a major upswing over last two decades. As a result, applications with increasing functionality and complexity are continuously implemented on mobile embedded devices such as smart phones and satellites, allowing these systems to operate independently. For example, modern-day satellites are capable of transmitting videos, communicating with aeroplanes, providing navigation to automobiles etc., compared to the first-generation satellites which could only transmit radio signals. However, this trend also 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 [7]. In this paper, we define the lifetime as the time one can use the battery before it is empty.

* This research is supported by the EU FP7 project SENSATION (318490).
Table 1: Relation between design choices and QoS factors

<table>
<thead>
<tr>
<th>Design Choices</th>
<th>Throughput</th>
<th>Number of Processors</th>
<th>Number of Batteries</th>
<th>Cost</th>
<th>Volume and Mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS Factors</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Kinetic Battery Model. 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. Also, a geostationary satellite with solar panels to charge on-board batteries, is recharged at 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 relevant 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 of such systems.

Once can identify three QoS factors, and their relation with different design choices, as given in Table 1. 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 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 the 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 batteries’ capacities.

We consider a very intuitive battery model termed Kinetic Battery Model (KiBaM) [17] as a representation of dynamic behaviour of a conventional rechargeable battery, see Figure 2. A KiBaM models the total charge in a battery as, two separate tanks separated by conductance. 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. For a given load current, KiBaM describes the charge stored in a battery by two coupled differential equations. Experimental studies show that KiBaM provides a good approximation of the system lifetime across various battery types [14].

Power Optimisation Techniques in modern HW platforms. To reduce the power consumption, modern hardware platforms such as Intel Core i7, and
NVIDIA Tegra 2, deploy a number of sophisticated power management methods [11]. Techniques like Dynamic Power Management (switching to low power state) (DPM) [6] and Dynamic Voltage and Frequency Scaling (throttling processor frequency) (DVFS) [20] help modern systems to reduce their power consumption while adhering to the performance requirements. The concept of voltage-frequency islands (VFIs) [12] further allows us to cluster a group of processors in such a way that each VFI runs on a common clock frequency/voltage. Furthermore, different VFI partitions represent DVFS policies of different granularity. Thus, with the help of VFIs, we can combine DPM, and DVFS policy with any granularity, generalising local and global DVFS. This achieves fine-grained system-level power management. To further illustrate the relation of power management in the processors, and the system lifetime, let us consider an example below.

**System Configuration of a Battery-powered Processor.** A typical system configuration for connecting a battery to a voltage/frequency scalable processor is shown in Figure 1. The battery’s voltage and current is represented by $V_{bat}$ and $I_{bat}$, and the processor’s voltage and current is represented by $V_{proc}$ and $I_{proc}$. Portable electronic devices, such as, cellular phones, satellites, and laptop computers often contain several sub-circuits, each with its own voltage level requirement, that is different from the voltage supplied by the battery. Hence, a DC-DC converter is utilised to convert DC (direct current) power provided by the battery from one voltage level to another. If we represent the efficiency of the DC-DC converter by $\eta$, the voltage/frequency scaling is governed by the following equation.

$$\eta \times V_{bat} \times I_{bat} = V_{proc} \times I_{proc} \quad (1)$$

Modern day microprocessors are designed using a specific circuitry design technology, termed as complementary metal-oxide-semiconductor (CMOS). In CMOS based processors, voltage/frequency scaling by a factor of $s$ causes the processor current $I_{proc}$ and the battery current $I_{bat}$ to scale by a factor of $s^2$ and $s^3$ respectively [8]. Therefore, slack utilisation by DVFS and DPM can greatly affect the load current, which in turn can impact the overall system lifetime. Moreover, partitioning the processors into VFIs provide even better control over system lifetime. Without VFIs, the systems are left with two options only, i.e., $I_{bat}$ with respect to either local or global frequency, resulting in unoptimised system lifetime. However, with the help of VFIs, it is possible to prolong the system lifetime, by modifying $I_{bat}$ with respect to any frequency, ranging from local to global. Earlier research shows that only the combination of DPM and
DVFS, and partitioning of processors into VFIs guarantees power optimisation [2].

As explained earlier, the system lifetime depends mostly on its capacity and the level of the load current (throttled using DVFS and DPM) applied to it. Nevertheless, if we have multiple batteries in the system, another important factor contributing to the overall lifetime is the usage pattern of 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 multiple batteries, we are able to derive a battery-optimal schedule of tasks.

The charge stored in the battery is represented by a finite set of continuous variables in the KiBaM, making the behaviour of KiBaM hybrid. Evaluating the performance of various (battery-) scheduling strategies using existing analysis techniques for hybrid systems, is very expensive [21]. Therefore, the state-of-the-art method in [15] discretises the KiBaM, and models it as priced timed automata (PTA) [5]. Furthermore, for a fixed execution order of the tasks, this approach deploys the model-checker Uppaal Cora that searches the whole state-space and generates the optimal battery schedule, using the well-developed model-checking techniques for PTA. 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 total battery capacities.

**Hybrid Automata.** We propose an alternative, novel approach based on Hybrid Automata (HA) [9]. These extend timed automata [3] (for the modelling of time-critical systems and time constraints) by continuous variables. HA can be analysed using Uppaal [4], that supports both model-checking and highly scalable Monte Carlo simulations.

In contrast to discretisation, as done in [15], 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. This approach enables us to utilise Uppaal to employ the 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 [15]. Furthermore, we utilise Uppaal also 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.

**Synchronous Dataflow.** The existing literature on battery scheduling [21] considers applications modelled without data dependencies between periods. However, in real-time applications, the iterations overlap in time and we have to deal with data dependencies within and across iterations. Moreover, critical performance constraints such as throughput must be met. Hence, we cannot capture all semantics of real-time applications without inter-period data dependencies.

We use Synchronous Dataflow (SDF) [16] as a computational model. SDF provides a natural representation of real-time streaming and digital signal pro-
cessing 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.

**Methodology and Contributions.** 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. In this paper, we consider the best of all scheduling scheme only. 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 VFI, 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.

- We utilise hybrid automata to model check and assess QoS of multiple KiBaMs for different design alternatives, without discretising time.
- We consider realistic hardware platforms equipped with the novel energy management techniques, compared to the state-of-the-art [21];
- We analyse SDF graphs as input which are more versatile and allow more realistic data-dependencies than acyclic applications [10][15][21];
- We show that our approach allows better scalability than PTA-based discretised KiBaM [15];
- Our approach allows early detection of design errors via model checking.

2 Related Work

An extensive survey paper [14] outlines the broad research work on various battery models. The state-of-the-art methods in the realm of battery-aware scheduling for multiple batteries, are presented in [15, 10]. These papers, in comparison to ours, discretise time. This approach helps to find optimal battery schedules, but it does not scale well because of the discretisation.

The state-of-the-art methods in the realm of battery-aware scheduling for multiple batteries, are presented in [15] and [10]. The approach in [15], 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 [10] models KiBaMs as hybrid like us, and discretises time to search the state-space, leading to the better results than the work in [15]. 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 [15].
Table 2: Comparison among different KiBaM analysis methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Without Discretisation</th>
<th>Multiple KiBaMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15, 10]</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>[21]</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>[13]</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Our Method</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

A more advanced technique that utilises hybrid automata like us, is presented in [21]. In this paper, the KiBaM provides energy to a uniprocessor. Unlike our method, this approach discusses a single battery case only. Another novel work in [13] 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 2 summarises different aforementioned KiBaM analysis methods.

To the best of our knowledge, there are no papers that analyse multiple KiBaMs without discretising time.

3 System Model Definition

3.1 Kinetic Battery Model

For an ideal battery, the voltage stays constant over time until the moment it is completely discharged, then the voltage drops to zero. The capacity in the ideal case is the same for every load for the battery. Reality is different, though: the voltage drops during discharge and the effectively perceived capacity is lower under a higher load. The second key difference between an ideal and realistic battery is that not all energy stored can be utilised at all times.

The kinetic battery model (KiBaM) [17] is a mathematical characterisation of state of charge of a battery. To address the earlier mentioned concerns with an ideal battery, the KiBaM divides the total charge stored in a battery into two "tanks" respectively termed as, the available charge and the bound charge, see Figure 2. 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. During low or no discharge current, some of the bound charge is converted to available charge. This conversion is at a rate proportional to the height difference with the proportionality factor being the rate constant \( k \), and is available to be consumed. Thus, the available charge replenishes bound charge, and this effect is termed as recovery effect.

If the widths of the available and bound charge tanks are given by \( c \) 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.

\[
\dot{a}(t) = -i(t) + k(h_b - h_a) \quad (2)
\]

\[
\dot{b}(t) = -k(h_b - h_a) \quad (3)
\]

The system starts in an 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. In fact, due to the dynamics of the system, the bound charge cannot reach zero in finite time. The system lifetime ends when all batteries are emptied.

The differential equations can be solved using Laplace transforms, which gives:

\[
y_1 = y_{1,0}e^{-k't} + \frac{(y_0k'c - i)(1 - e^{-k't})}{k'} - \frac{ic(k't - 1 + e^{-k't})}{k'} \quad (4)
\]

\[
y_2 = y_{2,0}e^{-k't} + y_0(1 - c)(1 - e^{-k't}) - \frac{i(1 - c)(k't - 1 + e^{-k't})}{k} \quad (5)
\]

where \( k' \) is defined as:

\[
k' = \frac{k}{c(1 - c)} \quad (6)
\]

and \( y_{1,0} \) and \( y_{2,0} \) are the amount of available and bound charge, respectively, at \( t = 0 \). For \( y_0 \), we have: \( y_0 = y_{1,0} + y_{2,0} \).

**Definition 1.** A KiBaM system is a tuple \( KS = (B, Cap) \) consisting of,

- a finite set of KiBaMs \( B = \{bat_1, \ldots, bat_m\} \), and
- a function \( Cap : B \rightarrow \mathbb{R}_{\geq 0} \) denoting the initial capacity of a KiBaM \( bat \in B \).

In our case-studies, we consider batteries having the capacity of 1300 mAh, as used in the Samsung Galaxy Fame smartphones [1].
3.2 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: a directed, connected graph in which tasks are represented by actors. Actors communicate with each other via streams of data elements, represented by tokens. Each edge \((a, b, p, q)\) connects a producer \(a\) to a consumer \(b\), and transports tokens between actors. The execution of an actor is known as an (actor) firing. Moreover, the number of tokens consumed or produced onto an edge \((a, b, p, q)\) as a result of a firing is referred to as consumption and production rates respectively. An SDF graph is timed if each actor is assigned an execution time.

Definition 2. An SDF graph is a tuple \(G = (A, D, \text{Tok}_0, \tau)\) where:

- \(A\) is a finite set of actors,
- \(D \subseteq A^2 \times \mathbb{N}^2\) is a finite set of dependency edges,
- \(\text{Tok}_0 : D \rightarrow \mathbb{N}\) denotes distribution of initial tokens in each edge, and
- the execution time of each actor is given by \(\tau : A \rightarrow \mathbb{N}_{\geq 1}\).

Definition 3. Given an SDF graph \(G = (A, D, \text{Tok}_0, \tau)\), the sets of input and output edges of an actor \(a \in A\) are defined respectively as \(\text{In}(a) = \{ (a', a, p, q) \in D | a' \in A, p, q \in \mathbb{N} \}\) and \(\text{Out}(a) = \{ (a, b, p, q) \in D | b \in A, p, q \in \mathbb{N} \}\). 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 \((a', a, p, q) \in \text{In}(a)\) 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 \((a, b, p', q') \in \text{Out}(a)\).

Example 1. Figure 3 shows the SDF graph of an MPEG-4 decoder [19]. The SDF graph contains five actors \(A = \{\text{FD}, \text{VLD}, \text{IDC}, \text{RC}, \text{MC}\}\), represented as 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 3.

Arrows between the actors depict the edges which hold tokens (dots) representing macroblocks. The worst-case 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 rates.

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 \(G = (A, D, \text{Tok}_0, \tau)\) is a function \(\gamma : A \rightarrow \mathbb{N}_0\) such that for every edge \((a, b, p, q) \in D\) from \(a \in A\) to \(b \in A\), the relation \(p.\gamma(a) = q.\gamma(b)\) holds. An SDF graph is consistent iff \(\gamma(a) > 0\) for all \(a \in A\).
Definition 5. Let us consider an SDF graph \( G = (A, D, \text{Tok}_0, \tau) \) with a repetition vector \( \gamma \). An iteration of \( G \) is defined as a sequence 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 \( \gamma \) in an iteration.

3.3 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

- heterogeneity, i.e., actors can run on certain type of processors only,
- a partitioning of the processors in voltage and frequency islands,
- different frequency levels each processor can run on
- power consumed by a processor in a certain frequency, both when in use and when idle,
- power and time-overhead required to switch between frequency levels.

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

- a finite set of processors \( \Pi \) assuming that \( \Pi = \{\pi_1, \ldots, \pi_n\} \) is partitioned into disjoint blocks \( \Pi_1, \ldots, \Pi_k \) of voltage/frequency islands (VFIs) such that \( \bigcup \Pi_i = \Pi \), and \( \Pi_i \cap \Pi_j = \emptyset \) for \( i \neq j \),
- 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 \( F = \{f_1, \ldots, f_m\} \) such that \( f_1 < f_2 < \ldots < f_m \),
- a function \( I_{\text{occ}} : \Pi \times F \to \mathbb{N} \) denoting the operating load current, if the processor \( \pi \in \Pi \) is running at frequency level \( f \in F \) in the working state,
- a function \( I_{\text{idle}} : \Pi \times F \to \mathbb{N} \) 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,
Table 3: DVFS levels of Samsung Exynos 4210

<table>
<thead>
<tr>
<th>Level</th>
<th>Voltage</th>
<th>Frequency</th>
<th>Level</th>
<th>Voltage</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2</td>
<td>1400</td>
<td>4</td>
<td>1.05</td>
<td>1128.7</td>
</tr>
<tr>
<td>2</td>
<td>1.15</td>
<td>1312.2</td>
<td>5</td>
<td>1.00</td>
<td>1032.7</td>
</tr>
<tr>
<td>3</td>
<td>1.10</td>
<td>1221.8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 2. Exynos 4210 is a state-of-the-art processor used in high-end platforms such as Samsung Galaxy Note, SII etc. Table 3 shows its different DVFS levels, and corresponding CPU voltage (V) and clock frequency (MHz) [18].

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

Definition 8. The throughput for a static-order schedule of an SDF graph \( G = (A, D, T_{\text{ok}_0}, \tau) \) 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 [15]. However, by switching between the batteries, their recovery effect is utilised, which in turn extends the overall system lifetime [15]. In this paper, we consider a scheduling scheme termed best-of-all. 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.
3.4 Example

Consider the SDF graph of an MPEG-4 decoder from Figure 3, where we take \( x = 5 \), mapped on four Samsung Exynos 4210 processors. The processors \( \Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\} \) are partitioned in three VFIs such that \( \Pi_1 = \{\pi_1\}, \Pi_2 = \{\pi_2, \pi_3\} \) and \( \Pi_3 = \{\pi_4\} \). Two DVFS levels (MHz) \( \{f_1, f_2\} \in F \) taken from Table 3 i.e. \( f_2 = 1400 \) and \( f_1 = 1032.7 \), are available to all processors. The supposed transition overhead (ms) of all Exynos 4210 processors is, \( \text{Tr}(\pi, f_2, f_1) = \text{Tr}(\pi, f_1, f_2) = 1.00 \).

Table 5 shows a specific static-order (SO) schedule of our running example. Here, \( (f_i \rightarrow f_k) \) represents the frequency transition from \( f_i \in F \) to another frequency \( f_k \in F \). The execution of an actor \( a \in A \) at a frequency level \( f_i \in F \) is represented by \( (a, f_i)^{ex} \), where \( ex \) indicates the consecutive executions of the actor. Similarly, \( (Idle, f_i)^{ex} \) denotes the idle time spent by a processor \( \pi \in \Pi \) at a frequency level \( f_i \in F \), where \( ex \) represents the duration of the idle time (ms).

We assume that the execution times (ms) of all actors \( a \in A \) at frequency level \( f_1 \) are rounded to the next integer. As \( f_1 = 0.738 \times f_2 \), we obtain \( \tau_{act}(a, f_1) = \lceil \frac{\tau_{act}(a, f_2)}{0.738} \rceil \).

Figure 4 shows the Gantt chart of the SO schedule in Table 5. As seen from Figure 4, the SO schedule given in Table 5 takes 10 ms to complete an iteration.
Definition 9. A hybrid automaton \( \mathcal{H} \) is a tuple \((L, Act, X, E, Inv, \mathcal{I}^0)\), where

- \( L \) is a finite set of locations;
- \( Act \) is a finite set of actions, co-actions and internal \( \lambda \)-actions;
- \( X \) is a finite set of continuous variables;
- \( E \) is a finite set of edges of
Fig. 5: Simulation of the SO schedule in Table 5 powered by two batteries
the form \((l, g, a, \varphi, l')\), where \(l\) and \(l'\) are locations, \(g\) is a predicate on \(\mathbb{R}^X\), action label \(a \in \text{Act}\) and \(\varphi\) is a binary relation on \(\mathbb{R}^X\); \(\text{Inv}\) assigns an invariant predicate \(\text{Inv}(l)\) 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.

Example 3. Let us consider an example of a thermostat maintaining the temperature \(T\) of a room at 25 Celsius. If the thermostat is on, the temperature dynamics is given by \(T' = -T + 30\), and if it is off, the temperature dynamics is given by \(T' = -T + 20\). The hybrid automaton describing the heating of the room is shown in Figure 6. The two states \(q(t) \in \{\text{on}, \text{off}\}\) represent the two discrete modes of the system: the thermostat is either on or off. As long as the thermostat is on, the temperature \(T\) will follow the dynamics specified in the left state, i.e., \(T\) will tend to 30. When the temperature is 27, the thermostat jumps from the on to the off mode. This is indicated by the invariant in the on mode and the guard condition on the transition from the on to the off mode. In the off mode, the temperature follows the dynamics given by the differential equation specified in the right vertex, i.e., \(T\) will tend to 20. When the temperature is 23, the thermostat jumps from the off to the on mode. This is indicated by the invariant in the off mode and the guard condition on the transition from the off to the on mode.

The state evolution of this example is shown in Figure 7. Initially the temperature is \(T = 0\), and the thermostat is in the mode \(q = \text{on}\). Furthermore, the thermostat is switched on and off via the discrete jumps at \(T = 23\) and \(27\) respectively.

In particular, HA can be analysed by the tool UPPAAL, where each component of the system is described with an automaton whose clocks can evolve with various rates. Such rates can be specified with, e.g., ODEs. We utilise UPPAAL engine to perform Monte Carlo simulations, to estimate QoS. The values of expressions (evaluating to integers or clocks) can be visualised along the simulated runs in the form of a line or bar plot, by using the following query.

\[
simulate N [<= bound]\{E_1, \ldots, E_k\}
\]

where \(N\) is the natural number representing the number of simulations to be performed, \(bound\) is the time bound on the simulations, and \(E_1, \ldots, E_k\) are \(k\)
state-based expressions that are to be monitored and visualised. UPPAAL also supports the evaluation of expected values of min or max of an expression that evaluates to integers or clocks. The syntax of the queries is as follows.

\[ E[\text{bound}; N](\min : \text{expr}) \]

or

\[ E[\text{bound}; N](\max : \text{expr}) \]

where \text{bound} is the time bound on the runs, \text{N} is the explicit number of runs, and \text{expr} is the expression to be evaluated.

5 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.

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

\[ K_{\text{sched}} \parallel K_1 \parallel \cdots \parallel K_m \parallel K_{\text{obs}} \parallel G_{\text{sched}}_1 \parallel \cdots \parallel G_{\text{sched}}_n \parallel Processor_1 \parallel \cdots \parallel Processor_n \parallel G_{\text{obs}}. \]

Here, the automaton \( K_{\text{sched}} \) models the scheduling scheme of KiBaMs. This paper considers the best-of-all scheduling scheme, i.e., after every iteration, the KiBaM with the highest available charge is chosen to serve for the next iteration. The HA \( K_1, \ldots, K_m \) model the KiBaMs \( B = \{ bat_1, \ldots, bat_m \} \). Similarly, the automaton \( G_{\text{sched}} \) 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 automaton \( G_{\text{obs}} \) 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 KiBaMs. Thus, the translation is, at least, composable with regards to the KiBaM system and processor application model.

Figure 8 shows the interactions between the HA of different components. Similarly, Figure 9 shows the HA models of the example given in subsection 3.4. The detailed translation of all components to HA, with respect to Figure 9 is presented as follows.

**Hybrid Automaton** $\mathcal{K}_{\text{sched}}$. The hybrid automaton $\mathcal{K}_{\text{sched}}$ models the scheduling scheme of KiBaMs, i.e., best-of-all. After an iteration is finished, this automaton chooses the battery having the highest available charge. Figure 9a shows the automaton $\mathcal{K}_{\text{sched}}$, with respect to the example in subsection 3.4.

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

- $\text{avail}_{bat_i} \xrightarrow{\text{avail}_{bat_i} \geq \text{avail}_{bat_j}, \text{startNextIter}, \text{active}_{\text{KiBaM}}id=k} \text{avail}_{bat_k}
- \text{avail}_{bat_i} \xrightarrow{\text{avail}_{bat_i} \geq \text{avail}_{bat_j}, \text{startNextIter}, \emptyset} \text{avail}_{bat_i}$
- $\text{avail}_{bat_k} \xrightarrow{\text{avail}_{bat_k} \geq \text{avail}_{bat_j}, \text{startNextIter}, \text{active}_{\text{KiBaM}}id=i} \text{avail}_{bat_i}$
- $\text{avail}_{bat_k} \xrightarrow{\text{avail}_{bat_k} \geq \text{avail}_{bat_j}, \text{startNextIter}, \emptyset} \text{avail}_{bat_k}$

After each iteration finishes, the action $\text{StartNextIter}$ synchronises with $G_{\text{obs}}$ to start the new iteration. But, before the new iteration starts, the battery $bat_y \in B$
Fig. 9: HA models for KiBaM, KiBaM Scheduler, SDF Scheduler, Processor, SDF and KiBaM Observer
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 \( avail_k \geq avail_i \) 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.

Hybrid Automata \( K_y \). The HA \( K_1, \ldots, K_m \) model the batteries \( B = \{ bat_1, \ldots, bat_m \} \), according to the description in Section 3.1. The model of \( bat_y \in B \) is shown in Figure 9b. This automaton informs \( K_{obs} \), when the battery \( bat_y \) gets empty.

For each \( bat_y \in B \), the HA \( K_y \) is defined as, \( K_y = (L_y, Act_y, X_y, E_y, F_y, Inv_y, \ell^0_y) \) where \( L_y = \{ \text{Initial}, \text{Emptyed} \} \), and \( \ell^0_y = \{ \text{Initial} \} \). The automaton \( K_y \) contains two continuous variables \( X = \{ \text{avail}_y, \text{bound}_y \} \) to denote the available and bound charge in \( bat_y \in B \), respectively. There is an urgent broadcast action in \( K_y \), i.e., \( Act_y = \{ \text{emptied!} \} \) to synchronise with \( K_{obs} \). The automaton \( K_y \) 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 \( on_y = \text{true} \) and \( i_y = \text{0} \). The transition set \( E_y \) has only transition, given as follows.

\[
\text{Initial} \xrightarrow{on_y \land avail_y = 0, \text{emptied!}, on_y = \text{false}} \text{Emptyed}
\]

The above transition synchronises with \( K_{obs} \) over the urgent channel \( \text{emptied!} \), and is taken if the available charge \( \text{avail}_y \) 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 \( \text{false} \).

The initial location \( \ell^0_y \) uses equations (2) and (3) as a delay function. This represents that, as long as \( bat_y \in B \) is non-empty, the available and bound charge of \( K_y \) evolves according to equations (2) and (3) respectively.

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 9c presents the automaton \( G_{sched_j} \), with respect to our running example.

For each \( \pi_j \in \Pi \), \( G_{sched_j} \) is defined as, \( G_{sched_j} = (L_j, Act_j, X_j, E_j, Inv_j, \ell^0_j) \), where \( L_j = \{ \text{Start}, \text{FireActor}, \text{EndFiring}, \text{totalFirings}, \text{Off} \} \), and \( \ell^0_j = \{ \text{Start} \} \). The HA \( G_{sched_j} \) contain three broadcast actions, i.e., \( Act_j = \{ \text{fire!}, \text{end?}, \text{startNextIter?} \} \). 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
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 $\pi_j$; 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 $\pi_j$. Since these variables are local, we can abbreviate them by $activeActor$, $s$ and $totalFirePerProc$ respectively. The transition set $E_j$ has following transitions.

- The following transition fetches the active actor according to the SO schedule for each processor $\pi_j$, 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.

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

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

$$\text{fireActor} \xrightarrow{\emptyset, fire\{j\}[activeActor]!, \emptyset} \text{endFiring}$$

- 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.

$$\text{endFiring} \xrightarrow{\emptyset, end\{j\}[activeActor]?, \emptyset} \text{totalFirings}$$

- The following transition checks if the SO schedule of a processor $\pi_j$ 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.

$$\text{totalFirings} \xrightarrow{s<totalFirePerProc, \emptyset, \emptyset} \text{Start}$$

- If all actors in the SO schedule of a processor $\pi_j$ 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 $\pi_j$ has executed all of the mapped actors in the current iteration. The variable $s$ is also reset.

$$\text{totalFirings} \xrightarrow{s=totalFirePerProc, firingFinished!, s:=0} \text{allFired}$$

- 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.

\[ \text{allFired} \rightarrow \emptyset, \text{startNextIter} \rightarrow \emptyset \rightarrow \text{Start} \]

**Hybrid Automata Processor**. Likewise, the HA Processor\(_{j}\), Processor\(_{1}\), \ldots, Processor\(_{n}\) model the processors \( \Pi = \{ \pi_1, \ldots, \pi_n \} \), as shown in Figure 9d. For better visibility, Figure 9d shows the HA of Processor\(_{j}\), with respect to one actor only, i.e., \( FD \in A \). The actors in the SO schedule of a processor \( \pi_j \) are mapped on the HA Processor\(_{j}\) by the HA \( G_{sched} \), using the actions fire and end.

For each \( \pi_j \in \Pi \), we define HA Processor\(_{j}\) = \( (L_j, Act_j, X_j, E_j, Inv_j, I_0^j) \). The initial location is defined as \( I_0^j = \{ \text{Initial} \} \). 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 \( f_1 < f_2 < \ldots < f_m \), let \( L_{mapping} = \{ \text{Idle}_f1, \ldots, \text{Idle}_fm, \text{InUse}_a_f1, \ldots, \text{InUse}_a_fm \} \) 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 \( f_1 < f_2 < \ldots < f_m \), we have an location which defines the overhead of switching between the frequencies, such that \( L_{overhead} = \{ \text{Tr}_f1_f2, \text{Tr}_f2_f1, \ldots, \text{Tr}_f1_fm, \text{Tr}_fm_f1 \} \). Thus, \( L_j = L_{mapping} \cup L_{overhead} \).

For each location \( \text{InUse}_a_fi \in L_j \), we have an invariant \( \text{Inv}_j(\text{InUse}_a_fi) \leq \tau_{act}(a, f_i) \) enforcing the system to stay in \( \text{InUse}_a_fi \) 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 \( \text{Inv}_j(\text{Idle}_f) \leq 1 \). Similarly, for \( F = \{ f_1, f_2, \ldots, f_n \} \) such that \( f_1 < f_2 < \ldots < f_m \), and \( \text{Inv}_j(\text{Tr}_f1_f2) \leq \tau_{tr}(\pi, f_1, f_2) \). 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 \( Act_j = \{ \text{fire?}, \text{end!} \} \) contains two broadcast actions fire?, end!. The actions fire? and end! in \( Act_j \) are parametrised with the processor and actor 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:

- \( \text{Initial} \rightarrow \emptyset, \text{fire}([a]?, \{ x: 0 \} \land \text{selectBatteryInUseFire}_f(a)) \rightarrow \text{InUse}_a_fi \), and
- \( \text{InUse}_a_fi \rightarrow x = \tau_{act}(a, f_i), \text{end}[a]! \land \text{selectBatteryInUseEnd}_f(a), \rightarrow \text{Initial} \).

For \( bat_j \in B \) and \( bat_k \in B \), the functions \( \text{selectBatteryInUseFire}_f(a) \) and \( \text{selectBatteryInUseEnd}_f(a) \) are defined in Listings 1.1 and 1.2 respectively.

The action \( \text{fire}[\pi][a] \) is enabled in the initial state Initial and leads to the location \( \text{InUse}_a_fi \). Thus, the action \( \text{fire}[\pi][a] \) 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. As each location \( \text{InUse}_a_fi \) has an invariant \( \text{Inv}_j(\text{InUse}_a_fi) \leq ...
Listing 1.1: selectBatteryInUseFire fi() Function

double selectBatteryInUseFire fi() {
    if (active.KiBaM_id==i) {
        return I_i=I_i+Iocc.fi;
    } else
        return I_k=I_k+Iocc.fi;
}

Listing 1.2: selectBatteryInUseEnd fi() Function

double selectBatteryInUseEnd fi() {
    if (active.KiBaM_id==i) {
        return I_i=I_i−Iocc.fi;
    } else
        return I_k=I_k−Iocc.fi;
}

\[ \tau_{act}(a, f_i), \] the automaton can stay in \text{lnUse}_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 \text{lnUse}_a.fi at exactly the execution time of actor \( a \in A \) at frequency level \( f_i \in F \), by taking the \text{end}[\pi][a] action.

For each \( \pi \in \Pi \), and \( f_i \in F \), the transition set \( E_j \) contains two transitions for handling broadcast such that:

- Initial \( \emptyset \), fire[\pi][idle_\pi], \{x=0\} \( \land \) selectBatteryIdleFire fi() \( \rightarrow \) Idle fi, and
- Idle fi \( x=1 \), end[\pi][idle_\pi], selectBatteryIdleEnd fi() \( \rightarrow \) Initial.

For \( bat_i \in B \) and \( bat_k \in B \), the functions selectBatteryIdleFire fi() and selectBatteryIdleEnd fi() are defined in Listings 1.3 and 1.4 respectively.

The action fire[\pi][idle_\pi] is enabled in the initial state Initial and leads to the location Idle fi. Thus, fire[\pi][idle_\pi] 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 lnUse a.fi has an invariant \( Inv_j(lnUse_a.fi) \leq 1 \), the automaton can stay in lnUse a.fi for at most 1 time unit. If \( x = 1 \), the system has to leave Idle fi at exactly one time unit, by taking the end[\pi][idle_\pi] 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:
Listing 1.3: selectBatteryIdleFire \texttt{fi}() Function

```c
double selectBatteryIdleFire\_\texttt{fi}()
{
    if (active.KiBaM\_id==i)
    {
        return I_i=I_i+I_{idle}\_i;
    }
    else
    return I_k=I_k+I_{idle}\_i;
}
```

Listing 1.4: selectBatteryIdleEnd\_\texttt{fi}() Function

```c
double selectBatteryIdleEnd\_\texttt{fi}()
{
    if (active.KiBaM\_id==i)
    {
        return I_i=I_i-I_{idle}\_i;
    }
    else
    return I_k=I_k-I_{idle}\_i;
}
```

---

The action \texttt{fire}[\pi][f_l f_m] 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, and so on.

**Hybrid Automaton** \(G_{obs}\). The SDF graph observer automaton \(G_{obs}\) observes if each processor has fired its all mapped actors in an static-order schedule. The automaton \(G_{obs}\) also counts the number of finished iterations. Figure 9e shows the HA model of \(G_{obs}\).
The automaton $G_{obs}$ is defined as, $G_{obs} = (L, Act, X, E, F, Inv, l^0)$, where $L = l^0 = \{\text{Initial}\}$. The set of urgent broadcast actions is defined as, $Act = \{\text{FiringFinished}?, \text{StartNextIter}!\}$. There are no clocks, invariants, delay functions and continuous variables in $G_{obs}$. The automaton $G_{obs}$ has number of variables: an integer variable $N$ to determine the total number of variables, i.e, $N = n(\Pi)$; an integer variable $\text{Tot}_\text{Iter}$ to count the number of finished iterations; and an integer variable $\text{TotalFiringsFinished}$ to count the number of finished firings in an iteration. Initially, $\text{Tot}_\text{Iter} = 0$ and $\text{TotalFiringsFinished} = 0$. The transition set $E$ has following transitions.

- In the following transtion, the guard condition $\text{TotalFiringsFinished} < N - 1$ checks if less than $N$ number of processors have finished the static-order mappings assigned to them. If this is the case, the transition is synchronised with $G_{sched_j}$ over the urgent channel $\text{FiringFinished}?$. As a result, $\text{TotalFiringsFinished}$ is incremented by one.

  $\text{Initial} \xrightarrow{\text{TotalFiringsFinished}<N, \text{FiringFinished}?, \text{TotalFiringsFinished}++} \text{Initial}$

- If $N$ number of processors have executed all mappings assigned to them in an iteration, the following transition is taken. This means that all processors $\pi_j \in \Pi$ are done with executing the static-order mappings assigned to them, and an iteration is finished. The automaton $G_{obs}$ also informs all instances of the automaton $G_{sched_j}$ to start next iteration, by synchronising over urgent broadcast channel $\text{StartNextIter}$. The function $\text{checkBatteryStatus}()$ checks whether the active battery has not got emptied during the iteration. If this is the case, the value of variable $\text{Tot}_\text{Iter}$ is increased by one.

  $\text{Initial} \xrightarrow{\text{FiringsFinished}=N, \text{StartNextIter}!, \text{checkBatteryStatus}() \land \text{FiringsFinished}=0} \text{Initial}$

- If all batteries are emptied, the automaton $K_{obs}$ informs the $G_{obs}$ via the following transition over the urgent channel $\text{allEmptied}?$. This signifies that the system lifetime has ended, and $G_{obs}$ needs to stop counting the number of finished iterations.

  $\text{Initial} \xrightarrow{\text{empty\_count}=\text{totBat}, \text{allEmptied}!, \emptyset} \text{Initial}$

For $bat_i \in B$ and $bat_k \in B$, the function $\text{checkBatteryStatus}()$ is defined in Listing 1.5.

Hybrid Automaton $K_{obs}$. The KiBaM observer automaton $K_{obs}$ observes if any battery gets empty. When all batteries get emptied, $K_{obs}$ synchronises with $G_{sched_j}$ to inform the end of the system lifetime.

The automaton $K_{obs}$ is defined as, $K_{obs} = (L, Act, X, E, F, Inv, l^0)$, where $L = l^0 = \{\text{Initial}\}$. The set of urgent broadcast actions is defined as, $Act = \{\text{emptied}?, \text{allEmptied}!\}$. There are no clocks, invariants, delay functions and continuous variables in $K_{obs}$. The automaton $K_{obs}$ has two variables: an integer
variable `totBat` to determine the total number of batteries in the system, i.e, \(\text{totBat} = n(B)\) where \(B = \{\text{bat}_1, \ldots, \text{bat}_m\}\); and an integer variable `empty_count` to count the number of emptied batteries. Initially, `empty_count = 0`. The transition set \(E\) is explained as follows.

- The following transition synchronises with the KiBaM automaton \(K\) on the urgent channel `emptied?`, if the battery `bat_y` is emptied. The guard condition checks if not all batteries are emptied. The variable `empty_count` is incremented by one as a result of taking this transition.

\[
\text{Initial} \quad \text{empty_count} < \text{totBat}, \text{emptied?}, \text{empty_count}++ \rightarrow \text{Initial}
\]

- If all batteries are emptied, the following transition synchronises with \(G_{\text{obs}}\) to inform about the end of the system lifetime.

\[
\text{Initial} \quad \text{empty_count} = \text{totBat}, \text{allEmptied!}, \emptyset \rightarrow \text{Initial}
\]

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 adequate batteries’ capacities.

\[
\text{simulate} \{1[\leq \text{bound}]\} \{\text{Tot_Iter}\}
\]

# 6 Experimental Evaluation via MPEG-4 Decoder

We evaluate QoS factors of an MPEG-4 decoder in Figure 3 [19] capable of 5 macroblocks. The experimental set-up consists of an MPEG-4 decoder mapped on Samsung Exynos 4210 processors \(\Pi = \{\pi_1, \ldots, \pi_n\}\). The processors \(\Pi = \{\pi_1, \ldots, \pi_n\}\) are provided with energy by Samsung batteries \(B = \{\text{bat}_1, \ldots, \text{bat}_n\}\).
used in Samsung Galaxy Fame smartphones. The capacity of all $bat \in B$ is, $Cap(bat) = 1300 \text{ mAh}$. The processors $\Pi = \{\pi_1, \ldots, \pi_n\}$ are available with two frequency levels (MHz) $f_2 = 1400$ and $f_1 = 1032.7$. Table 4 shows the idle and operating load currents of both KiBaMs $B = \{bat_1, bat_2\}$ at both frequencies. The supposed transition overhead (ms) of all Exynos 4210 processors is, $T_{tr}(\pi, f_2, f_1) = T_{tr}(\pi, f_1, f_2) = 1$. Recall that $I_{tr}(\pi, f_2, f_1) = I_{tr}(\pi, f_1, f_2) = 0.5 \text{ mA}$ for all $\pi \in \Pi$. We evaluate the completed number of video frames with respect to various QoS aspects varying (1) frames per second (throughput); (2) number of processors; and (3) batteries. Similarly, for the same factors, we assess adequate battery capacities. Please see Figures 10 - 15 for results.

6.1 Varying Frames per Second

For 6 Exynos 4120 processors $\Pi = \{\pi_1, \ldots, \pi_6\}$ served by two batteries $B = \{bat_1, bat_2\}$, we consider different SO schedules, as given in Table 6. For varying frames per second (fps) constraint, Figure 10 shows the total number of video frames completed as a function of the throughput. At tighter performance constraints (fps), the idle time of processors is not sufficient to move to low power state. As a result, the batteries are drained more rapidly. Thus, we achieve less number of frames. Alternatively, if we require fewer fps from an MPEG-4 decoder, then the battery lifetime increases.

For the same SO schedules, Figure 11 shows the minimum initial required capacity $Cap(bat_1)$ for KiBaM $bat_1 \in B$ to complete 1000 video frames. It can be seen from Figure 11 that if we relax the fps constraint, the minimum required capacity also decreases.

Nevertheless, if the video quality is enhanced from 125 to 200 fps, then the increase in required initial 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.

6.2 Varying Number of Processors

We consider different SO schedules, all yielding 71 fps, as given in Table 7. Figure 12 shows the total number of video frames completed for varying number of processors. As we can see from Figure 12, for the same batteries’ capacities, higher number of processors achieve more or equal number of frames. The reason is that, if we reduce the number of processors, then the same amount of work is done on fewer processors to attain same throughput, resulting in shorter idle times. Therefore, battery charge is consumed more rapidly, if the number of processors are reduced.

For the same SO schedules considered earlier in Table, Figure 12 shows the minimum required capacity $Cap(bat_1)$ for KiBaM $bat_1 \in B$ to complete 1000 video frames. The results reiterate the earlier conclusions in Figure 12 that, to
Consider a SO schedule producing 71 fps, as given in Table 8. For varying bat-

<table>
<thead>
<tr>
<th>S0 Schedule</th>
<th>Fps</th>
<th>( \pi_1 )</th>
<th>( \pi_2 )</th>
<th>( \pi_3 )</th>
<th>( \pi_4 )</th>
<th>( \pi_5 )</th>
<th>( \pi_6 )</th>
</tr>
</thead>
</table>

Table 6: Static-Order Schedules for varying number of video frames

![Fig. 10: System lifetime against varying fps](image)

![Fig. 11: Minimum required capacity for bat_1](image)

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 batteries' capacities, contributing to low-cost embedded systems with reduced mass and volume.

### 6.3 Varying Number of Batteries

Let us consider that we have 6 Exynos 4120 processors \( \Pi = \{ \pi_1, \ldots, \pi_6 \} \). We consider a SO schedule producing 71 fps, as given in Table 8. For varying bat-
As it can be seen from Figure 14, increasing the minimum required capacity \( \text{Cap} \) for battery \( \text{bat}_1 \) improves the attainable number of video frames linearly.
Table 8: Static-Order Schedules for varying batteries

<table>
<thead>
<tr>
<th>π1</th>
<th>π2</th>
<th>π3</th>
<th>π4</th>
<th>π5</th>
<th>π6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f₁ → f₂)(PD-f₁)</td>
<td>(f₁ → f₂)(PD-f₂)</td>
<td>(f₁ → f₂)(PD-f₁)</td>
<td>(f₁ → f₂)(PD-f₂)</td>
<td>(f₁ → f₂)(PD-f₁)</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 14: System lifetime against varying number of KiBaMs

Fig. 15: Minimum required capacity for $bat_1$

However, if we analyse the Figure 15, we can see that increasing the batteries does not reduce the minimum required 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 the low-cost and high-performance systems.

6.4 Comparison with PTA-KiBaM

In this subsection, we compare the approach presented in this paper (HA-KiBaMs) with the PTA-based approach (PTA-KiBaM) [15]. In the PTA-KiBaM, the behaviour of batteries is based on a discretised version of the KiBaM, and is modelled as priced timed automata (PTA). For a given load, the model-checker UPPAAL Cora is utilised to search the whole state-space and to generate optimal battery schedules. However, this approach suffers serious scalability issues. As increasing the initial batteries’ capacities leads to searching the bigger state-space, this approach only allows to model limited batteries’ capacities. Furthermore, this approach requires to discretise the temporal dimension, which limits the accuracy of this approach. In contrast, we use hybrid automata to model the continuous behaviour of batteries. This leads us to analyse the behaviour of KiBaMs without discretising time. Furthermore, following this approach, we can make use of highly scalable Monte Carlo simulations, over hybrid automata. It is worth mentioning that the PTA-KiBaM [15] analyses the completed num-
 unfavorable number of tasks, instead of iterations. However, as iterations are the key metric in SDF graphs, we also compare both techniques in terms of completed number of iterations. Let us consider the example of an MPEG-4 decoder in Figure 3. We further assume that we have two batteries, i.e., $B = \{bat_1, bat_2\}$. Table 10 shows the completed number of video frames for the SO schedules in Table 9, calculated using both methods. The experiments were run on a dual-core 2.8 GHz machine with 8 GB RAM.

Columns 3-8 in Table IV show the system lifetime, 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. Table 10 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.

<table>
<thead>
<tr>
<th>SO Schedule</th>
<th>$\pi_1$</th>
<th>$\pi_2$</th>
<th>$\pi_3$</th>
<th>$\pi_4$</th>
<th>$\pi_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$(f_1 \rightarrow f_2)(FE-f_3)$</td>
<td>$(f_2 \rightarrow f_3)(FE-f_4)$</td>
<td>$(f_2 \rightarrow f_3)(IDE-f_3)^2$</td>
<td>$(f_2 \rightarrow f_3)(IDE-f_3)^2$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_3)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td>S2</td>
<td>$(f_j \rightarrow f_j)(FE-f_j)$</td>
<td>$(f_j \rightarrow f_j)(FE-f_j)$</td>
<td>$(f_j \rightarrow f_j)(IDE-f_j)$</td>
<td>$(f_j \rightarrow f_j)(IDE-f_j)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td>S3</td>
<td>$(f_j \rightarrow f_j)(FE-f_j)$</td>
<td>$(f_j \rightarrow f_j)(FE-f_j)$</td>
<td>$(f_j \rightarrow f_j)(IDE-f_j)$</td>
<td>$(f_j \rightarrow f_j)(IDE-f_j)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
<tr>
<td></td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$(VLD-f_j)(f_1 \rightarrow f_2)$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Table 9: Static-Order Schedules for Comparison

<table>
<thead>
<tr>
<th>Cap(bat_1)</th>
<th>Cap(bat_2)</th>
<th>S1 (computation time)</th>
<th>S2 (computation time)</th>
<th>S3 (computation time)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PTA-KiBaM</td>
<td>HA-KiBaM</td>
<td>PTA-KiBaM</td>
</tr>
<tr>
<td>1.25x10^{-4}</td>
<td>1.25x10^{-4}</td>
<td>0 (520)</td>
<td>0 (28)</td>
<td>0 (200)</td>
</tr>
<tr>
<td>2.5x10^{-4}</td>
<td>2.5x10^{-4}</td>
<td>0 (510)</td>
<td>0 (55)</td>
<td>1 (41060)</td>
</tr>
<tr>
<td>3.75x10^{-4}</td>
<td>3.75x10^{-4}</td>
<td>Out of Memory</td>
<td>2 (62)</td>
<td>1 (14810)</td>
</tr>
<tr>
<td>5x10^{-4}</td>
<td>5x10^{-4}</td>
<td>Out of Memory</td>
<td>4 (64)</td>
<td>Out of Memory</td>
</tr>
</tbody>
</table>

Table 10: Comparison of two approaches wrt varying battery capacities.
<table>
<thead>
<tr>
<th>Batteries HA-KiBaM PTA-KiBaM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 N/A</td>
</tr>
<tr>
<td>2 4 Out of Memory</td>
</tr>
<tr>
<td>3 6 Out of Memory</td>
</tr>
<tr>
<td>4 8 Out of Memory</td>
</tr>
<tr>
<td>5 9 Out of Memory</td>
</tr>
<tr>
<td>6 12 Out of Memory</td>
</tr>
<tr>
<td>7 14 Out of Memory</td>
</tr>
<tr>
<td>8 16 Out of Memory</td>
</tr>
<tr>
<td>9 17 Out of Memory</td>
</tr>
<tr>
<td>10 20 Out of Memory</td>
</tr>
</tbody>
</table>

Table 11: Comparison of two approaches wrt number of batteries.

However, the biggest advantage of HA-KiBaM is the scale of capacities it can handle. As Table 10 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 6). Furthermore, PTA-KiBaM requires considerably longer computation time than HA-KiBaM. Please note that zero in Table 10 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 11 compares the iterations completed for varying number of batteries for both methods. For this experiment, we consider SO schedule S3, and $\text{Cap}(bat) = 5 \times 10^{-4}$ mAh for all $bat \in B$.

7 Model Checking via MPEG-4 Decoder

In this section, we demonstrate 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 \( \pi_0 \) and \( \pi_1 \) respectively, can fire in parallel if the query \( E \not< SDFScheduler_0.activeActor == p \text{ and } SDFScheduler_1.activeActor == q \) 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 \( \pi_0 \) and \( \pi_1 \) in a same VFI, then we check the query \( A[\text{Processor}_0.curr_freq == \text{Processor}_1.curr_freq] \) to verify this property.

Similarly, we can verify reachability properties such as, “does RC eventually fire?” and “after five consecutive VLD firings, MC must fire at least once”, and liveness properties such as, “after a processor is occupied, it is eventually released”. 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 Battery 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 \( \text{Pr}[<= 1500] (< bound_1 \not< \text{bound}_0 > n \text{ and empty\_count } < 2) \) 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, \( n = 4 \).

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 \( \text{Pr}[<= 1500] (< b0.active == true \text{ and } b1.active == true) \). The probability for this property to hold is \([0,0.0973938]\) with 0.95 confidence, which means that this property is not satisfied.
8 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.

References