D3.1 Report on analysis techniques for adaptive systems

Project acronym: ADMORPH
Project full title: Towards Adaptively Morphing Embedded Systems
Grant agreement no.: 871259

Due Date:	September 30th
Delivery:	Month 9
Lead Partner:	Lund University
Editor:	Martina Maggio
Dissemination Level:	Public
Status:	Submitted
Approved:	
Version:	1.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH project).

DOCUMENT INFO – Revision History

Date and version number	Author	Comments
12/09/2020, v1.0	Martina Maggio	Initialization
25/09/2020, v1.0	Martina Maggio	Internal Review

List of Contributors

Date and version number	Author	Comments
12/09/2020, v1.0	Sebastian Altmeyer	Section 4
12/09/2020, v1.0	Christoph Kühbacher	Section 4
12/09/2020, v1.0	Florian Haas	Section 4
12/09/2020, v1.0	Sobhan Niknam	Section 2
12/09/2020, v1.0	Andy Pimentel	Section 2
12/09/2020, v1.0	Martina Maggio	Section 1
25/09/2020, v1.0	Martina Maggio	Internal Review

Contents

Executive summary

This deliverable is a report on the consortium's work in Task 3.1 *System-level simulation of dynamically evolving embedded systems*, in Task 3.3 *Adaptivity-aware real-time scheduling policies* and in Task 3.4 *Models of Computation and derived architectures to allow seamless reconfiguration.*

1 Introduction

In this report we provide an overview of the tasks connected to the analysis of self-adative system. In particular, this work package focuses on how to evaluate the behaviour provided by these systems at runtime.

Morphing systems are special, in that it is very hard to define invariants to their behaviour – i.e., something that is always true regardless of when, where, and how these systems execute. The same morphing system may have been executing in two different environments for a given amount of time. In the first environment, it learned to behave in a certain way (for example to optimise its functionality and at the same time satisfy some battery constraints). In the second environment, due to different conditions, the same optimisation strategy would not work and the system then learned to perform different actions to achieve the same exact objective.

One way to analyze the behavior of systems that morph is to define invariants that are true in every situation regardless of the dynamic behavior of the system (and its past knowledge). Knowing that a predicate is an invariant allows us to guarnatee that the corresponding statement is true in every possible execution and with every possible history. Clearly, it is quite difficult to find these invariants, but this represents a viable way to analyze complex systems. The aim of this work package is to define techniques and implement tools that allows us to carry out this analysis, either with invariants or with other techniques. We aim at evaluating when the system morphing has been successful and how to improve it.

In particular, this report focuses on preliminary work conducted to:

- Design a simulation tool that allows us to compare different scenarios and their expected performance characteristics. This simulation should be conducted at the system-level and should include the system - hardware and software - and its evolution. From the theoretical perspective, we use Monte Carlo (MC) simulation as the main tool to perform many different simulations in variable environments and with varying characteristics.

- In parallel with the modeling and simulation of the system level characteristics of our embedded systems, we need to derive models of computation. We started to work on the derivation of models that include tasks with precedences (for example: we need to aquire a new image from a camera before being able to understand if something has happened with respect to previous frames). This is per se not new, but the novelty with respect to existing work lays in this precedence list to be dynamically updated at runtime. For example, due to an attack, we may determine that we need to aquire frames from different cameras in order to be certain that nobody has tampered with the image acquisition process. At runtime our morphing system will be able to recognise this, so we need to be able to handle the computation in different scenarios.

- Once we understood the varying characteristics of our computation model, we need to be able to handle these characteristics in terms of resource distribution. For example, two cores may be needed to perform the same analysis on two different images simultaneously, such that our answer can still be produced in real-time. Maybe, we would find that two cores are

not enough, and we need to use dedicated hardware like FPGAs or a graphic card for the image processing in order to meet the original application requirements.

The next sections will enter into details on what has been done for each of these three objectives during the first nine months of the project.

2 System-level simulation of dynamically evolving embedded systems

With technology advances and the steady increasing of the transistor density of chips, the embedded computer systems in Cyber Physical Systems are increasingly built around a so-called Multi-Processor System-on-Chip (MPSoC) component that integrates multiple processors, memories, interconnects, and other (peripheral) modules into a single chip. Pushing the boundaries of transistor miniaturization further enables the integration of more and more computational resources, i.e., processor cores and memory, onto MPSoCs. These advancements derived by transistor miniaturization result in a positive influence on the performance of the embedded systems. However, an increase of on-chip power density and consequently operating temperature due to aggressive transistors downscaling accelerate the wear-out mechanisms of semiconductor devices, such as electromigration, resulting in premature aging and permanent hardware defects that have negative impact on the lifetime reliability (measured in terms of mean time to failure (MTTF)) of the embedded systems. This reliability concern is especially crucial for mission- and/or safety-critical systems that require long operational times. To this end, reliability is becoming increasingly another important design constraint besides the traditional performance, power/energy consumption, and cost for embedded systems that should be taken into account within the earliest stages of the system design.

To ensure reliability and/or prolong the lifetime of a system, it is necessary to cope with disruptive hardware failures. To do so, an effective system-level approach is system reconfiguration by the means of a task-to-processor re-allocation policy[1]. In this way, the system remains operational as long as the applications running on the failed processors can be re-allocated to spare processors or processors with spare capacity, i.e., slack, at run-time. As a result, the system can cope with a sequence of disruptive hardware failures and postpone the moment at which it is no longer able to provide the required timeliness guarantees, for high-criticality applications, and stops working, i.e., the time to failure (TTF) will increase. For non-critical applications, however, the system can achieve graceful degradation in which the system lifetime is further prolonged at the cost of system efficiency, energy consumption, etc. We refer to such a system as an *adaptive system*.

To design and effectively deploy an adaptive embedded system, one should be able to efficiently model and simulate these systems to explore the large number of design options to determine the optimal design instances with respect to the underlying platform architecture (e.g., the number and types of processors to use in the platform) and the adaptivity policy used to cope with disruptive hardware failures. To capture the system's evolving behavior over time, we need to analyze the

[1] We refer to it as *adaptivity policy* in this report

system (executing its application workload) over a relatively long period of time while the system is exposed to a, possibly large, sequence of disruptive hardware failures according to a given fault model. Then, since hardware failures are introduced randomly [1], each design instance has to be evaluated multiple times (once for each random sequence of processor failures) in order to determine its MTTF (i.e. how long the system on average is operational) with a high degree of certainty. This method is known as Monte Carlo simulation (MCS) [8].

In the literature, several simulators for estimating the lifetime reliability of adaptive embedded systems using MCS have already been proposed [6, 11, 3, 2, 9, 5]. These simulators, however, have the following limitations:

- Most of them are limited towards the kind of systems they can simulate. For example, the frameworks in [6, 11, 3, 2, 9] only consider homogeneous MPSoC platforms where all processors are identical. Moreover, the framework in [2] considers the simplistic assumption that the workload of a failed processor is evenly distributed to the alive processors.

- Although an adaptive system may tolerate a sequence of hardware failures by the means of task-to-processor re-allocation, the frameworks in [11, 5] consider that a single hardware failure results in the failure of the entire system, thereby resulting in an inaccurate MTTF computation.

- Some frameworks, such as in [11, 3, 9, 5], adopt a cycle-accurate system simulator, such as gem5 or Sniper, in order to estimate the MTTF of an adaptive system accurately by monitoring the power consumption and temperature of the system when running a set of applications. As a result, these are heavyweight frameworks and therefore not suitable for fast system-level exploration of the varying underlying platform architectures and the adaptivity policies of an adaptive system, needed in the context of Task 3.2.

- The frameworks in [6, 11, 3, 5] are not publicly available.

In the context of the ADMORPH project, we are mainly interested in an adaptive heterogeneous embedded system, which alters its own system configuration over time via a provided adaptivity policy. We are also interested in a lightweight simulation framework, suitable for Task 3.2, for analyzing the extra-functional properties of the system. Thus, as the aforementioned existing frameworks are not suitable for our needs, our main contribution in the context of Task 3.1 is the development of a novel open-source system-level simulator, so-called *Simuflage*[2], for analyzing the MTTF and power/energy consumption of an adaptive system, that features a heterogeneous MPSoC platform and runs a set of real-time applications. For understanding the contributions of this work, we first introduce the preliminary materials in Section 2.1. Then, we present a first version of our simulator in detail in Section 2.2. Finally, in Section 2.3, we talk about our plan as future work to improve the accuracy of the simulator.

[2] `https://github.com/sea-art/Simuflage`

2.1 Lifetime Reliability Modelling of a Processor

As mentioned above, the permanent hardware failures are mainly introduced due to the device aging caused by wear-out mechanisms that are exacerbated with aggressive transistors downscaling. The probability that a failure of this type occurs, is influenced by the time that a processor has been utilized and its operating temperature. Similar to earlier studies [6, 10, 2], we will only focus on common wear-out mechanisms that are provided by [1] such as electromigration (EM), time-dependent dielectric breakdown (TDDB), negative bias temperature instability (NBTI), and thermal cycling (TC). Under each of these wear-out mechanisms, the lifetime of a single digital component, such as a processor, in terms of the MTTF or cycles to failure can be estimated using an empirically-derived model [10]. For instance, the MTTF due to electromigration is given by the following equation [1]

$$MTTF_{EM}(T) = \frac{A_0}{(J - J_{\text{crit}})^n} \cdot e^{\frac{E_a}{kT}}, \tag{1}$$

where A_0 is a process-dependent constant, J is the current density, J_{crit} is the critical current density for the EM effect, E_a is the activation energy for EM, k is the Boltzmann's constant, n is the material-dependent constant, and T is the operating temperature. It is required for this model that J is greater than J_{crit} to produce a failure [1]. These empirical MTTFs for the aforementioned wear-out mechanisms, however, cannot be directly used in the system-level analysis. This is mainly because, the failures caused by these mechanisms are not occurring deterministically and therefore their empirical MTTF estimation only provides the expected value, but not how the failures are distributed around this mean value. To this end, Weibull and lognormal distributions are normally used to characterize the temporal failure probability (i.e., *reliability* that is the probability of surviving until a particular time) of a processor. For instance, the reliability of a processor with Weibull distribution can be represented using the following equation:

$$R(t, T) = e^{-\left(\frac{t}{\lambda(T)}\right)^{\beta}} \tag{2}$$

where t is the current time instant (generally measured in hours), T is the steady-state processor temperature (in Kelvin), $\lambda(T)$ is the scale parameter that its formulation depends on the wear-out mechanism to be considered, and β is the Weibull shape parameter[3] (considered to be independent of the temperature). Then, the lifetime (MTTF) of the processor is simply determined by the area underlying the reliability function $R(t, T)$ as follows:

$$MTTF = \int_0^{\infty} R(t, T)dt = \lambda(T) \times \Gamma(1 + \frac{1}{\beta}), \tag{3}$$

where $\Gamma()$ is the gamma function. Thus, the formulation of $\lambda(T)$ in Eq. (3) for each of the wear-out mechanisms can be derived using their corresponding empirical MTTF model. As an example, the

[3]The shape parameter enables the modeling of any phase in a processor's lifetime. More precisely, the infant mortality phase with decreasing failure rate is characterized by $\beta < 1$, the normal life period with a constant failure rate by $\beta = 1$, and the wear-out phase with an increasing failure rate by $\beta > 1$.

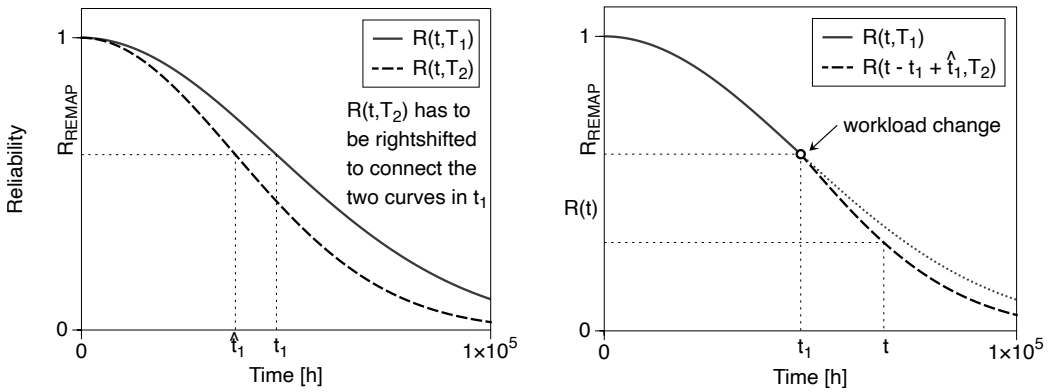

Figure 1: Reliability curve considering temperature changes (taken from [2]).

scale parameter for the EM mechanism, $\lambda_{EM}(T)$, can be expressed as:

$$MTTF_{EM}(T) = \lambda_{EM}(T) \times \Gamma(1+\frac{1}{\beta}) \Longrightarrow \lambda_{EM}(T) = \frac{MTTF_{EM}(T)}{\Gamma(1+\frac{1}{\beta})} = \frac{A_0}{(J-J_{\text{crit}})^n \times \Gamma(1+\frac{1}{\beta})} \cdot e^{\frac{E_a}{kT}} \tag{4}$$

where $MTTF_{EM}$ is calculated for temperature T using Eq. (1). Now, having the λ and β parameters, the reliability curve of a processor (for a corresponding failure mechanism) can be derived. For instance, Fig. 1(a) demonstrates the reliability curves for a processor under two steady-state temperatures T_1 and T_2, where $T_1 < T_2$. This figure clearly shows that, as expected, the operational temperature of the processor has an inverse relation to the processor's lifetime reliability. Thus, if the processor's temperature is changed at run-time, for example due to changing the workload of the processor, the reliability curve of the processor should be refined accordingly by calculating the new λ parameter corresponding to the new steady-state temperature, in order to accurately estimate MTTF. This reliability curve refinement is demonstrated in Fig. 1(b) in which the processor's temperature is changed from T_1 to T_2 at time instance t_1.

As a conclusion to the above discussion in this section, the combination of a failure mechanism of choice combined with Weibull or lognormal (failure) distribution allows us to model random permanent failures for an individual processor. Then, the TTF of the processor for a failure mechanism, for example EM, can be determined by sampling its failure distribution, as follows

$$TTF_{EM}(T) = \lambda_{EM}(T)(-\ln(X))^{\frac{1}{\beta}}, \tag{5}$$

where $X \in (0, 1]$ (i.e. drawn from the uniform distribution over $(0, 1]$). Finally, if multiple failure mechanisms are considered, the TTF of the processor is the minimum TTF among all considered failure mechanisms, i.e.,

$$TTF(T) = \min\Big(TTF_{EM}(T), TTF_{TDDB}(T), TTF_{TC}(T), TTF_{SM}(T)\Big).$$

2.2 The Proposed Simulator

This section looks at the core concepts of the simulator in a general form. As a reminder, the main goal of the simulator is to aid in evaluating different design space exploration methods, in the context of Task 3.2. Thus, the main goal of the simulator is not to be of the highest absolute accuracy, which many other simulators and models do [9, 5]. These simulators often sacrifice the execution time of the simulator in order to achieve more accurate results. Since our simulator targets efficient system-level design space exploration, it favours fidelity over absolute accuracy [7]. Moreover, our simulator is developed towards the ease of 1) utilization by the exploration strategies and 2) changeability with respect to the user needs. The former requires that the software of this simulator should have an easy interaction with a variety of exploration strategies by keeping the overhead in this regard minimal. The latter requires that the user of the simulator should be able to easily adapt the simulator to his/her own needs.

As discussed in Section 2.1, there are multiple fault models and distributions to capture different processor failures that the simulator can be armed with. It should also be possible to completely replace various models (e.g. thermal model, power model, execution model) within the main simulator cycle, depending on the priority of the user. This also means that the simulator is flexible with respect to adapting its input(s) and output(s) in order to integrate it with existing software(s). The inputs/outputs and the core part of the simulator will be discussed later on in this section. Since the software should also be maintainable, the software is available in open source[4] via the GNU General Public License version 3.0[5]

2.2.1 Inputs and Outputs of the Simulator

The simulator has two types of inputs 1) an adaptive system, and 2) parameters related to the environment and the context that the system will be evaluated in accordance with. More specifically, the adaptive system consists of a platform, application workload(s), an initial application(s) mapping on the platform, and an adaptivity policy to handle processor failures. In the following subsections, these elements of the system will be discussed individually. While the first input of the simulator describes the system to be evaluated, the contextual parameters represent the environment in which the simulator operates and can be altered within the simulator with respect to the user needs, but should be adjusted before the evaluation of the adaptive system. For instance, some contextual parameters are the environmental temperature, fault model(s), fault distribution, neighbor thermal influences, static power consumption for the processors on the platform. The exact list of contextual parameters is very dependent on which models are being used and the user needs, but it is important to note that these parameters can be altered. Then, the simulator evaluates the system with respect to the contextual parameters and reports the TTF and power/energy consumption of the system as outputs.

[4] `https://github.com/sea-art/Simuflage`
[5] `https://www.gnu.org/licenses/gpl-3.0`

2.2.1.1 Platform The platform includes several heterogeneous processors connected through a network-on-chip (NoC) infrastructure. Within the simulator, processors on the platform are abstracted to only represent the computational capacity of the system. In this way, each processor consists of three elements:

- a value representing the speed/computational capacity of the processor,
- a value representing the heat the processor will generate upon 100% usage,
- a coordinate (x, y) where the processor is placed on the NoC.

The difference in the speed/computational capacity of the processors allows us to model a subset of heterogeneous platforms known in the real-time community as *uniform heterogeneous multiprocessor platforms* [4] in which different processors may have different speeds, but on a given processor, every application/task is executed at the same speed. Further, each processor can be placed somewhere on a 2-dimensional NoC, which has to be specified during the designing phase of the system. Since processors run at different speeds, they have different power consumption and as a result, maximum temperature, which greatly influences the thermal model we consider in Section 2.2.2.3 and should be specified during the designing phase of the system. This value will indicate the temperature that a processor itself will generate at a 100% usage (excluding environmental and temperature influences from the adjacent neighboring processor(s)). More formally, a processor, with location of (i, j) on the NoC, can be specified as a vector $\vec{c}_{ij} = \begin{bmatrix}\alpha_{ij} & \tau_{ij}\end{bmatrix}$, where α_i indicates its computational capacity, and τ_i indicates its maximum temperature. Then, we can define the platform as an $m \times n$ matrix

$$\mathbf{C} = \begin{pmatrix} \vec{c}_{00} & \vec{c}_{10} & \cdots & \vec{c}_{n0} \\ \vec{c}_{01} & \vec{c}_{11} & \cdots & \vec{c}_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{c}_{0m} & \vec{c}_{1m} & \cdots & \vec{c}_{nm} \end{pmatrix}. \tag{6}$$

Note that this matrix has a direct correspondence with the floorplan of the platform, i.e., the matrix values on position (i, j) correspond with the processor located in (i, j)-coordinate on the NoC. For the sake of simplicity, $\mathbf{C}$ is separated in two distinct $m \times n$ matrices containing all α_{ij} and τ_{ij} values for the processors on the respective location, i.e.,

$$\mathbf{C}^{\alpha} = \begin{pmatrix} \alpha_{00} & \alpha_{10} & \cdots & \alpha_{n0} \\ \alpha_{01} & \alpha_{11} & \cdots & \alpha_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{0m} & \alpha_{1m} & \cdots & \alpha_{nm} \end{pmatrix}, \quad \mathbf{C}^{\tau} = \begin{pmatrix} \tau_{00} & \tau_{10} & \cdots & \tau_{n0} \\ \tau_{01} & \tau_{11} & \cdots & \tau_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ \tau_{0m} & \tau_{1m} & \cdots & \tau_{nm} \end{pmatrix}. \tag{7}$$

2.2.1.2 Application Workload Applications define the computational needs of the system and specify the tasks that have to be executed by processors on the platform in order for the system to be functional. Application specifications are abstracted away to a single value that indicates how much computational capacity is required to run this application, i.e., task utilization $u \in (0, 1]$.

We denote the application workload consisting of m independent applications in a single set as $A = \{a_0, a_1, ...a_{m-2}, a_{m-1}\}$, where $a_i \in A$ is the ith application which has utilization u_i. We intend to extend this relatively simple model in the future to include more details of the applications.

2.2.1.3 Initial Application Mapping Initial application mapping indicates which applications should be executed initially on which processor. We can denote the mapping of the application set A to the platform $\mathbf{C}$ as a function $f_{ac} : A \to C$. It is required that all applications in A are mapped on a single processor of the platform. Processors can run multiple applications, but no application can be mapped on multiple processors (for now). An application $a_k \in A$ can only be executed on a processor $c_{ij} \in \mathbf{C}$ with a capacity higher than or equal to the utilization of the application, i.e., $\alpha_{ij} \geq u_k$. It is also the case that the total utilization of all applications mapped on the processor c_{ij} should not exceed its capacity, i.e.,

$$\alpha_{ij} \geq \sum_{\substack{\forall a_k \in A, \\ f_{ac}(a_k) = c_{ij}}} u_k. \tag{8}$$

Based on the application mapping f_{ac}, we can also define an $m \times n$ matrix, to indicate the computational capacity that is being used for each processor

$$C^{req} = \begin{pmatrix} \alpha_{00}^{req} & \alpha_{10}^{req} & \cdots & \alpha_{n0}^{req} \\ \alpha_{01}^{req} & \alpha_{11}^{req} & \cdots & \alpha_{n1}^{req} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{0m}^{req} & \alpha_{1m}^{req} & \cdots & \alpha_{nm}^{req} \end{pmatrix} \tag{9}$$

where $\alpha_{ij}^{req} = \sum_{\substack{\forall a_k \in A, \\ f_{ac}(a_k) = c_{ij}}} u_k$. With the matrices from 7 and 9, we define the following properties that will be used later on within the main body of the simulator:

$$C^{slack} = C^{\alpha} - C^{req}, \text{ (10a)} \qquad C^{workload} = C^{req} \oslash C^{\alpha}, \text{ (10b)} \qquad C^{capacities} = C^{\alpha}, \text{ (10c)} \qquad C^{max\_thermals} = C^{\tau}, \text{ (10d)}$$

where $\oslash$ is the Hadamard division (i.e. element-wise matrix division).

2.2.1.4 Adaptivity Policy The adaptivity policy can be seen as a function that defines how the system will handle processor failures at run-time. When a processor is failed, the policy defines how the applications that are mapped on the failed processor should be distributed among other alive processors that have enough capacity. If the policy is not able to do so, the system is then permanently failed. The set of applications that are mapped on failed processors can formally be defined as

$$A_{fail} = \{a_i \mid a_i \in A, f_{ac}(a_i) \in C_{fail}\}, \tag{11}$$

where C_{fail} is the set of all processors that have failed. Generalised, the policy describes how the system will adjust f_{ac} to $f'_{ac} : A \to C \setminus C_{fail}$, where $C \setminus C_{fail}$ is the set of all alive processors (i.e. the elements of C that are not in the set C_{fail}). Section 2.2.2.5 will describe in more detail how the policy will be utilised to change the application mapping upon a processor failure.

2.2.2 Main Part of the Simulator

This section will look at how a given adaptive system is evaluated in terms of TTF and power/energy consumption. Similar to the inputs/outputs of the simulator, the main part of the simulator has been designed with modularity in mind, making it easy for other interested individuals to extend or alter the current implementations of the simulator.

Algorithm 1 Simulator overview

Require: Contextual parameters (Section 2.2.1)
Input: An adaptive system (Section 2.2.1)
Initialise: TTF $\leftarrow 0$, $E \leftarrow 0$, platform model (Section 2.2.1.1), thermal model (Section 2.2.2.3), and ageing model (Section 2.2.2.4)

```
1:  repeat
2:      increment ageing model                          ▷ Section 2.2.2.4
3:      increment platform model                        ▷ Section 2.2.1.1
4:      increment thermal model                         ▷ Section 2.2.2.3
5:      if failure occurs then                          ▷ Section 2.2.2.5
6:          handle the failure via policy
7:          adjust ageing, platform and thermal model
8:      end if
9:      TTF ← TTF + timestep
10:     E ← E + timestep × (power consumption)
11: until application can not be executed
```

Output: TTF, E

2.2.2.1 Overview The simulator works in timesteps, i.e., the time between each two consecutive processor failures, where in each timestep several calculations are performed to advance the given system to the next timestep. A system that is being simulated is always in one of the three phases: *functioning correctly*, *handling a failure*, or *has failed*. The system is always in the first phase by default. Once a failure has occurred, it will move to the second phase. In the second phase, there are two possibilities. The applications on the failed processor can either be successfully remapped to some of the alive processors according to the policy, that results in the system returning to the first phase, or one or more applications can not be mapped on any alive processor. When the latter occurs, the system has permanently failed and enters the third phase when the simulator will report the desired outputs.

The essence of the simulator is the combination of three individual models that are interacting with each other in order to simulate the behavior of an adaptive system over the course of time. The three models are *platform model*, *thermal model*, and *ageing model*. Algorithm 1 shows high-level pseudocode of the simulator that highlights the use of the three aforementioned models. The algorithm of the simulator will work in timesteps where in each timestep the TTF will be

incremented and the models will advance with a single timestep. The following subsections will look at the individual elements that make up the simulator.

2.2.2.2 Power model The power consumption is a metric in Watts, which indicates how much power is required in order to run the workload on the platform. Since the power consumption is very dependent on how the system is operating, it changes over time for adaptive systems. Since the behaviour of an adaptive system is not deterministic, power consumption can be measured per simulation iteration. To receive a more accurate power consumption, the simulator has to run multiple times in order to yield the mean power consumption of the system given for a specific workload.

At this moment, the power model in the simulator is heavily abstracted and is considered to be linearly dependent on the workload of a processor. The power consumption of a processor has a direct linear correlation with the computational capacity of the processor. The current model will sum up 1) the static power consumption for all alive processors, and 2) the power consumption based on workload per alive processor (i.e. processors running one or more applications), in order to determine the power consumption per timestep of the simulator. More formally, we can determine the current power consumption of the simulator as

$$P_{(t)} = \sum_{\forall C_{i,j} \in C \setminus C_{fail}} P_{static} + P_{dynamic} \cdot \mathbf{C}_{ij}^{workload} \cdot \mathbf{C}_{ij}^{capacities}, \tag{12}$$

where P_{static} is the static power consumption per processor and $P_{dynamic}$ is the power consumption per processor capacity.

2.2.2.3 Thermal model Thermal behaviour plays a key role in determining when a processor is going to fail. This is mainly because, most of ageing models, like those introduced in Section 2.1, are functions solely based on temperature. Therefore, the temperature per processor has to be stored based on the current system state. Within the simulator, we use a simple thermal model that will determine the thermal behaviour of processors based on four parameters: workload ($\mathbf{C}^{workload}$), maximum temperature per processor ($\mathbf{C}^{max\_thermals}$), neighbor thermal contributions (Equation 16), and uniform thermal fluctuation (Equation 17). Similar to the platform model, the thermals are stored in an $m \times n$ matrix

$$\mathbf{T} = \begin{pmatrix} t_{00} & t_{10} & \cdots & t_{n0} \\ t_{01} & t_{11} & \cdots & t_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ t_{0m} & t_{1m} & \cdots & t_{nm} \end{pmatrix}. \tag{13}$$

where t_{ij} is the current temperature of processor $\vec{c}_{ij}$. Equation 18 will show how this matrix is composed. Note that thermals of empty positions, where there is no processor present, in $\mathbf{T}$ are set to 0. Each processor individually generates heat based on its maximum temperature and the current workload of that processor. Besides the thermals that each individual processor generates,

Algorithm 2 Thermal model per timestep

Require: t_{env} ▷ Environmental temperature (subsubsection 2.2.1)
t_{idle} ▷ Idle temperature (subsubsection 2.2.1)
fluc ▷ Temperature fluctuation amount
$\mathbf{K}$ ▷ Equation 15

Input: $\mathbf{C}^{\text{workload}}$ ▷ Workload of current timestep

1: $\mathbf{T}_{ij}^{\text{indiv}} \leftarrow \begin{cases} t_{\text{env}} + t_{\text{idle}}, & \text{if } \mathbf{C}_{ij}^{\text{workload}} = 0 \\ t_{\text{env}} + \mathbf{C}_{ij}^{\text{max\_thermals}} \cdot \mathbf{C}_{ij}^{\text{workload}}, & \text{if } \mathbf{C}_{ij}^{\text{workload}} > 0 \end{cases}$ ▷ Individual temperatures (14)
2: $\mathbf{T}^{\text{neighbour}} \leftarrow \mathbf{T}^{\text{indiv}} * \mathbf{K}$ ▷ Equation 16
3: $\mathbf{T}^{\text{random}} \leftarrow \mathbf{U}_{mn}(-\text{fluc}, \text{fluc}]$ ▷ Equation 17
4: $\mathbf{T} \leftarrow \mathbf{T}^{\text{indiv}} + \mathbf{T}^{\text{neighbour}} + \mathbf{T}^{\text{random}}$ ▷ Equation 18

Output: $\mathbf{T}$ ▷ Adjusted thermals

there is also the environmental temperature t_{env} that will influence each individual processor. This is an environmental constant of the simulator that can be defined in the system configuration. With these factors known, we can determine the temperature of each processor as follows

$$\mathbf{T}^{indiv} = t_{env}\mathbf{J}_{mn} + \mathbf{C}^{max\_thermals} \circ \mathbf{C}^{workload}, \tag{14}$$

where $\circ$ indicates the Hadamard product (i.e. element wise matrix multiplication) and $\mathbf{J}_{mn}$ is the unit matrix of $m \times n$ (i.e. matrix with all ones).

Processors that are positioned adjacently, influence the thermals of each other since the heat sources can increase temperatures when they are close enough together. We model this thermal influence of adjacent processors on each other using a discrete convolution. The kernel for the convolution can easily be adjusted within the software, but a straightforward kernel K is

$$\mathbf{K} = \begin{pmatrix} 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01, \end{pmatrix} \tag{15}$$

which will add the temperature for each processor with one percent of its neighbors' temperature, as follows

$$\mathbf{T}_{ij}^{neighbour} = \mathbf{T}^{indiv} * \mathbf{K} = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} \mathbf{T}^{indiv}[i-k, j-l]\mathbf{K}[k,l]. \tag{16}$$

Random fluctuation of the thermals based on some distribution (e.g. Uniform distribution) is also considered in the simulator and stored as an $m \times n$ matrix of random values

$$\mathbf{T}^{random} = \begin{pmatrix} x_{00} & x_{10} & \cdots & x_{n0} \\ x_{01} & x_{11} & \cdots & x_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{0m} & x_{1m} & \cdots & x_{nm} \end{pmatrix}, \tag{17}$$

Algorithm 3 Aging model at timestep t

Require: $\mathbf{A}^{\lambda}$ ▷ Current wear-out rate (Equation 20)
$\mathbf{A}^{\text{wear}}_{(t-1)}$ ▷ Current wear of processors (Equation 21)
$\mathbf{C}^{\text{workload}}_{(t-1)}$ ▷ Workload of previous timestep (Equation 10b)

Input: $\mathbf{C}^{\text{workload}}_{(t)}$ ▷ Workload of current timestep

1: **if** $\mathbf{C}^{\text{workload}}_{(t)} \neq \mathbf{C}^{\text{workload}}_{(t-1)}$ **then** ▷ If workload has changed
2: $\quad \mathbf{A}^{\lambda} \leftarrow \mathbf{J}_{mn} \oslash \mathbf{A}^{\text{TTF}}$ ▷ Resample wear-out rates with new thermals (2)
3: **end if**
4: $\mathbf{A}^{\text{wear}}_{(t)} \leftarrow \mathbf{A}^{\text{wear}}_{(t-1)} + \mathbf{A}^{\lambda}$ ▷ Equation 21

Output: $\mathbf{A}^{\text{wear}}_{(t)}$ ▷ Adjusted wear-out

where $x_{ij} \in X \sim U(-p, p)$ and p is the maximum fluctuation amount.

By using the previous equations, we can now determine the thermals based on the current system state as:

$$\mathbf{T} = \mathbf{T}^{indiv} + \mathbf{T}^{neighbour} + \mathbf{T}^{random}. \tag{18}$$

Algorithm 2 shows the calculations that are being executed in order to advance the thermal model a single time step given the system configuration and the current workload $\mathbf{C}^{workload}$.

2.2.2.4 Ageing Model The ageing aspects of the simulator will indicate when processors are failed, thus the ageing model will introduce the processor failures. As explained in Section 2.1, the core of this process is determined by the chosen fault model and distribution. Similar to CALIPER [2], we chose to use the Weibull distribution while introducing faults via electromigration. The scale parameter for the Weibull distribution is determined using the Eq. (4). Similar to [10], we have chosen the shape parameter of $\beta = 5.0$ for this distribution. Now, using Eq. (5) and the thermal matrix $\mathbf{T}$, we can define a matrix including the TTF of all processors as follows

$$\mathbf{A}^{TTF} = \begin{pmatrix} TTF_c(\mathbf{T}_{00}) & TTF_c(\mathbf{T}_{10}) & \cdots & TTF_c(\mathbf{T}_{n0}) \\ TTF_c(\mathbf{T}_{01}) & TTF_c(\mathbf{T}_{11}) & \cdots & TTF_c(\mathbf{T}_{n1}) \\ \vdots & \vdots & \ddots & \vdots \\ TTF_c(\mathbf{T}_{0m}) & TTF_c(\mathbf{T}_{1m}) & \cdots & TTF_c(\mathbf{T}_{nm}) \end{pmatrix}. \tag{19}$$

Note that the steady-state temperature of the processors might change over time as a result of application remapping at run-time based on the policy due to processor failures. For this reason, we utilize the TTF of processors to the reciprocal, which will indicate the wear-out per timestep:

$$\mathbf{A}^{\lambda} = \mathbf{J}_{mn} \oslash \mathbf{A}^{TTF} \tag{20}$$

were $\oslash$ is the Hadamard division operator and $\mathbf{J}_{mn}$ is the unit matrix of $m \times n$ (i.e. matrix of ones). The values of $\mathbf{A}^{\lambda}$ will be incremented in each iteration of the Algorithm 1, for this reason

Algorithm 4 Adaptive policy

Require: f_{policy}
f_{ac}
C

1: **procedure** POLICY(C_{fail}, A_{fail})
2: Remove C_{fail} from f_{ac}
3: **for** $a \in A_{\text{fail}}$ **do**
4: $C_{\text{candidates}} \leftarrow \{\vec{c} \mid \vec{c} \in C \setminus C_{\text{fail}}, \vec{c}_{\text{slack}} \geq a\}$ ▷ Set of processors where a can be mapped to
5: $\vec{c} \leftarrow f_{\text{policy}}(C_{\text{candidates}}, a)$ ▷ Policy selects candidate out of the set $C_{\text{candidates}}$
6: Extend f_{ac} with $(a, \vec{c})$ ▷ Add the mapping of a to $\vec{c}$ to the application mapping
7: **end for**
8: **end procedure**

we need to keep track of the current wear that has occurred on the processors, which is defined recursively as

$$\mathbf{A}^{wear}_{(0)} = \mathbf{0}_{mn},\ \mathbf{A}^{wear}_{(t)} = \mathbf{A}^{wear}_{(t-1)} + \mathbf{A}^{\lambda}, \tag{21}$$

where $\mathbf{0}_{mn}$ is the zero matrix of $m \times n$ and t represents the timestep of the simulation. The wear values of the current timestep will be referred to as $\mathbf{A}^{wear}$. On the event that workload (and thus temperatures) have changed, $\mathbf{A}^{\lambda}$ has to be adjusted to the new temperatures which are calculated by the thermal model. When the workload and thus temperatures have changed, the history of the current wear-out of the processors is stored. The rate of wear-out will however change when $\mathbf{A}^{TTF}$ (Equation 5) changes based on the thermal matrix $\mathbf{T}$ (Equation 18) changes.

Algorithm 3 shows the calculations that are being performed in order to advance the ageing model a single timestep based on the calculated values of the previous timestep. By iteratively refining the aging model, the processors $\vec{c}_{ij}$ is failed when $\mathbf{A}^{wear}_{ij} \geq 1$.

2.2.2.5 Adapting to Processor Failures As mentioned in Section 2.2.2.4, during timestep t of the simulator, we can determine if any processor has failed by verifying $\exists i, j : \mathbf{A}^{wear}_{ij} \geq 1$. On the event that the processor $\vec{c}_{ij}$ fails, the simulator has to

1. change the state of $\vec{c}_{ij}$ from active to failed,
2. remap the applications that were mapped on the failed processor to the alive processors, which have sufficient capacity, based on the policy,
3. refine the temperature of the alive processors based on the new application mapping,
4. refine the wear-out rate (Equation 20) based on the new temperatures.

These individual steps will be explained consecutively in this section. At first, the failed processor $\vec{c}_{ij}$ should be removed from the platform. This can be done by $\mathbf{C}^{capacities}_{ij} = 0$, $\mathbf{C}^{req}_{ij} = 0$, which

Algorithm 5 Adapting to failures

Require: C ▷ Set of all processors
A ▷ Set of all applications

Input: $\mathbf{A}^{\text{wear}}$ ▷ Current aging wear of processors

1: **if** $\exists \mathbf{A}_{ij}^{\text{wear}} \geq 1.0$ **then** ▷ Verify if any processor has failed
2: $A^{\text{fail}} \leftarrow \{\}$ ▷ Set of all applications that have to be remapped
3: $C_{\text{fail}} \leftarrow \{\vec{c}_{ij} \mid \vec{c}_{ij} \in C, \mathbf{A}_{ij}^{\text{wear}} \geq 1.0\}$ ▷ All failed processors
4: **for** $\vec{c}_{ij} \in C_{\text{fail}}$ **do**
5: $\mathbf{C}_{ij}^{\text{capacities}} \leftarrow 0$ ▷ Remove processor by setting its capacity to 0
6: $A^{\text{fail}} \leftarrow A^{\text{fail}} \cup \{a \mid a \in A, f_{ac}(a) = \vec{c}_{ij}\}$ ▷ Add applications mapped towards this failed processor to set
7: **end for**
8: POLICY(C_{fail}, A_{fail}) ▷ Algorithm 4
9: update platform model ▷ Update matrices 10a - 10d
10: **end if**

Output: Status of system ▷ Is the policy able to handle processor failure?

indirectly changes the state of the processor $\vec{c}_{ij}$ to failed. Since the workload has changed, the $\mathbf{T}$ matrix (Equation 18) will also be updated to the current application mapping allowing in turn for $\mathbf{A}^{\lambda}$ to be updated via re-sampling with adjusted thermals as described in Algorithm 3. The set of applications that require remapping are obtained via Equation 11. The possible processors a failed application a^{fail} can be mapped to are

$$\{\vec{c}_{ij} \mid \vec{c}_{ij} \in C, \mathbf{C}_{ij}^{slack} \geq a^{fail}\}.$$

The processor(s) from this list where the applications are mapped to is determined by the policy. How the policy achieves this is illustrated in Algorithm 4.

Utilising the policy, the complete algorithm to adapt to failures is defined in Algorithm 5, which in essence finds all applications that have to be remapped and utilises the policy (Algorithm 4) in order to remap the applications and thus adapting the system. Since the policy will alter the workload, a lot of matrices have to be recalculated after a processor failure. If the policy is not able to map a single or more applications to any alive processors, the system has failed. System failure is always introduced by processor failures.

2.3 Future Work

As the initial implementation of our simulator still uses various simplified models as well as simplifying assumptions, affecting its absolute accuracy, our plan as future work is to improve this aspect by exploiting more accurate models in the simulator. Moreover, and related to the previous, in the coming time, we will also perform validation experiments for our simulator.

3 Models of computation and derived architectures to allow seamless reconfiguration

Our work on models of computation started with a study of the state of the art. Inspired by the research that has been done on the topic, we plan to propose a set of features for models of computation and their derived constraints for hardware and software architectures. The idea is to target multi-core and distributed heterogeneous architectures that need seamless reconfiguration. The ability to provide seamless reconfiguration is the key to achieve dynamic adaptation.

In connection with work package 5, we have identified the physical constraints and the architecture definition. This task will explore these elements from the computational perspective, determining and analyzing what we need to seamlessly reconfigure. We target in particular the update of task-graphs and/or Synchronous Dataflow (SDF) Graphs, including reconfiguration tasks, on-demand redundancies and the remapping of communication links. This is also in accordance with the use-case specification (see deliverable 5.1) and the coordination language of ADMORPH.

4 Adaptivity-aware real-time scheduling policies

The Objective of the task has been defined as follows:

> *In this task, we will develop real-time scheduling techniques and analyses for dynamically evolving systems. The immense state-space of system configurations prohibits an analysis and optimization of the individual system configurations. We will thus develop techniques to analyse the timing behaviour of multiple system configurations at once. To this end, we will extend the concept of sustainable scheduling analysis to hardware components. Starting from a feasible system, where any additional component failure will result in the violation of the timing behavior, we can provision additional resources while ensuring timing correctness of the resulting system configuration. SYS will work with the partners on using the scheduler plug-in framework developed in Task 4.2 for implementing adaptivity-aware scheduling algorithms on the PikeOS separation kernel.*

Real-time scheduling traditionally assumes immutable systems. Not only the hardware is assumed to be fixed, but also the application composed of a set of tasks (where task is the common terminology for an individual software component scheduled on a system). Changing the set of tasks and especially changing the available processing resources violates the assumptions under which the real-time behavior has been analyzed. Consequently, the prediction of the timing behavior is invalidated.

In a first step, we have evaluated how we can execute tasks on an adaptive system, where either the set of tasks or the underlying hardware changes during runtime. There are the following main questions:

- At which point in time can we allow the system to morph and to adapt to faults, failures or attacks?

- How can we analyze the timing behavior, when the system is allowed to change during runtime?

From a real-time perspective, static systems are particularly desirable. The timing correctness can be analyzed entirely offline and the variation during runtime is heavily limited. In the context of ADMORPH, static solutions are difficult to achieve.

In the first period of the ADMORPH project, we have concentrated on the practical aspects of real-time scheduling techniques for adaptively morphing systems. We have experimented with a run-time environment (RTE) which permits adaptivity while enabling timing predictability. This RTE will allow us to run experiments early on in the course of the project and we will be able to ground the scheduling analysis, which we will develop in the coming months, on realistic measurements.

For the development of the adaptivity-aware runtime environment, we borrow from a previous but ongoing project at the University of Augsburg funded by the German Research Foundation. The project is called Adaptive Redundancy for Manycore Architectures (ARoMA) and focuses on the fault-tolerance of manycore architectures and the use of different redundancy levels to establish it. Within the ARoMA project, tasks are scheduled with the aim to minimize the make-span, i.e., to finish the application as early as possible. There exists no notion of deadlines or periods. Although timing and scheduling analysis has not been part of ARoMA, the aim was to design a predictable fault-tolerant runtime environment for homogeneous systems. Lessons learned from ARoMA, and part of the code-base of the runtime environment have been used for the adaptivity-aware runtime environment.

The main features of the adaptivity-aware runtime environment are:

- Applications are modeled as directed acyclic graphs (DAG), where vertices represent executable code, i.e., tasks, and edges represent data dependencies. Although these graphs are acyclic, periodic behavior can be modeled as a simple re-execution of the entire DAG. The application model will be connected to the coordination language. We believe that using DAGs as underlying programming model is sufficiently generic for the coordination language and the use-cases.

- The DAGs are divided using barriers into sections. The barriers are used to detect faults and to initiate changes in the system configuration and schedule. The sections are scheduled in highly predictable fashion using Heterogeneous Earliest Finish Time (HEFT) scheduling. Instead of allowing reconfiguration at any time, barriers represent a kind of checkpoints to identify errors and to correct them. Different schedules within a section are pre-computed statically and will be selected at the barriers depending on the current system configuration.

  The division into barriers and sections enables the adaptivity at the barriers and the predictability during the execution of the sections. The tradeoff between predictability and adaptivity can be tuning by modifying the number of barriers and the length of the sections.

For now, changes in the system configuration and schedule are limited to a coarse-grained correction of the redundancy. Also, the system is currently assumed to be homogeneous. Both limitations will be addressed in the remainder of the project.

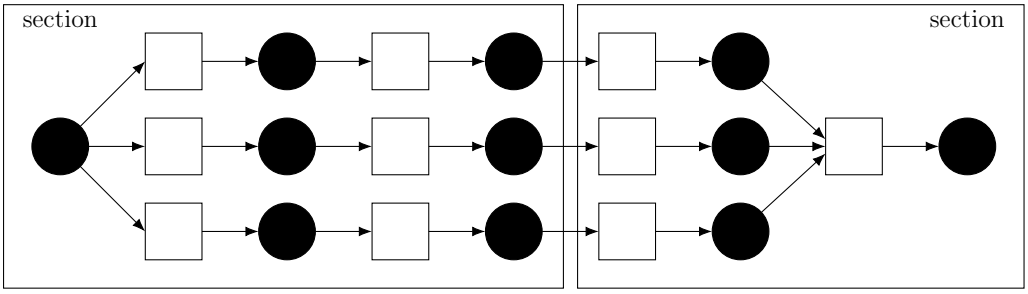

Figure 2: A directed acyclic graph (DAG) with data and task nodes divided into sections allows for a parallel execution and enables the adaptive transition between different redundancy levels at the section borders.

Since we borrowed from the ARoMA project, we were able to develop a prototype of the adaptivity-aware runtime environment for x86 architectures and the Kalray Bostan MPPA. For the x86 architectures, the prototype is implemented on top of Linux using POSIX threads. This will ease experimentation with the case studies and also the integration of the scheduling algorithms with PikeOS.

We now know how applications can be executed on a real system in a predictable, yet adaptive fashion, which allows the systems to react to faults or attacks, while preserving timing correctness.

5 Conclusion

In conclusion, our work on analysis techniques for adaptive and morphing systems just started. However, the ability to simulate the behaviour of these system is a signficant step towards understanding and providing guarantees on the reconfiguration and adaptation procedure. Our simulator is also the key element that will allow us to fast-prototype our analysis techniques and test them on realistic data and scenarios before the application to the real hardware and software. We believe the currently work-in-progress research is on track towards the project objectives.

6 References

[1] JEDEC Solid State Technology Association et al. Failure mechanisms and models for semiconductor devices. *JEDEC Publication JEP122-B*, 2003.

[2] Cristiana Bolchini, Matteo Carminati, Marco Gribaudo, and Antonio Miele. A lightweight and open-source framework for the lifetime estimation of multicore systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 166–172. IEEE, 2014.

[3] Simone Corbetta, Davide Zoni, and William Fornaciari. A temperature and reliability oriented simulation framework for multi-core architectures. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 51–56. IEEE, 2012.

[4] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[5] Nikos Foutris, Christos Kotselidis, and Mikel Luján. Simulating wear-out effects of asymmetric multicores at the architecture level. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2019.

[6] Lin Huang and Qiang Xu. Agesim: A simulation framework for evaluating the lifetime reliability of processor-based socs. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 51–56. IEEE, 2010.

[7] H. Javaid, A. Ignjatovic, and S. Parameswaran. Fidelity metrics for estimation models. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2010.

[8] Dirk P Kroese, Tim Brereton, Thomas Taimre, and Zdravko I Botev. Why the monte carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(6):386–392, 2014.

[9] R Rohith, Vijeta Rathore, Vivek Chaturvedi, Amit Kumar Singh, Srikanthan Thambipillai, and Siew-Kei Lam. Lifesim: A lifetime reliability simulator for manycore systems. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 375–381. IEEE, 2018.

[10] Yun Xiang, Thidapat Chantem, Robert P Dick, X Sharon Hu, and Li Shang. System-level reliability modeling for mpsocs. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 297–306, 2010.

[11] Alexandre Yasuo Yamamoto and Cristinel Ababei. Unified system level reliability evaluation methodology for multiprocessor systems-on-chip. In *2012 International Green Computing Conference (IGCC)*, pages 1–6. IEEE, 2012.