System-Level Modeling and Simulation of MPSoC Run-Time Management using Execution Traces Analysis

Simei Yang, Sébastien Le Nours, Maria Mendez Real, Sébastien Pillement

▶ **To cite this version:**

Simei Yang, Sébastien Le Nours, Maria Mendez Real, Sébastien Pillement. System-Level Modeling and Simulation of MPSoC Run-Time Management using Execution Traces Analysis. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIX), Jul 2019, Samos, Greece. paper #49. hal-02114092

HAL Id: hal-02114092
https://hal.science/hal-02114092

Submitted on 16 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

System-Level Modeling and Simulation of MPSoC Run-Time Management using Execution Traces Analysis

S. Yang[0000−0002−0130−8176], S. Le Nours[0000−0002−1562−7282], M. Méndez Real[0000−0002−7071−7192] and S. Pillement[0000−0002−9160−2896]

University of Nantes, CNRS, IETR UMR 6164, F-44000 Nantes, France
{simei.yang, sebastien.le-nours, maria.mendez, sebastien.pillement}@univ-nantes.fr

Abstract. Dynamic management of modern Multi-Processors System on Chip (MPSoC) become mandatory for optimization purpose. Evaluation of these run-time resource management strategies in MPSoCs is essential early in the design process to guarantee a reduced design cycle. However, most of the existing system-level simulation-based evaluation frameworks consider static application mapping and do not allow run-time management effects to be evaluated. In this paper, we present a modeling and simulation approach that allows integration of run-time management strategies in multicore system simulation. We have integrated the proposed approach in an industrial modeling and simulation framework. A case-study with seven applications (85 running tasks in total) on a heterogeneous multicore platform is considered and different management strategies are evaluated according to latency and power consumption criteria.

Keywords: System-level simulation, execution trace, run-time management strategies, heterogeneous multicore systems

1 Introduction

Modern multicore platforms contain an increasing number of heterogeneous resources, *i.e.*, processing elements, memories, and communication resources. Such platforms allow more and more functionalities to be supported while satisfying still multiple non-functional requirements such as real-time and power consumption. Due to dynamism between and within applications, the behavior of application workloads can dramatically vary over time. Hybrid application mapping methods [11] have emerged as convenient approaches to cope with applications dynamism and favor the achievement of non-functional requirements such as timing and power constraints in multicore platforms.

Hybrid application mapping methods combine design-time analysis and run-time management of platform resources. In such approaches, the design-time stage performs design space exploration to prepare a set of mappings of the supported applications. The run-time management has then the purpose of dynamically mapping the running applications on platform resources in such a way that real-time and energy consumption objectives are optimized. In this context, extensive evaluation of the run-time management strategies is essential to guarantee that the non-functional requirements will be respected.

System-level modeling and simulation approaches favor early detection of potential issues and prevent costly design cycles. In existing system-level simulation-based approaches such as ones presented in [5], a system model is formed by a combination of an application model and a platform model. The captured models are generated as executable descriptions. Then these models can be simulated under different situations to estimate system performance and optimize system design. However, in most of the existing frameworks, the allocation of applications on platform resources is statically defined and it cannot be modified during system simulation. Extending system-level simulation-based approaches is thus mandatory to allow early evaluation of run-time management strategies.

In this paper, we present a system-level modeling and simulation approach of run-time management in multicore platforms. The proposed model allows modification of applications allocation and scheduling on platform resources during the system simulation. It uses dynamic computation of instants when platform resources are used according to running applications. Using dynamically computed simulation instants, the simulation model of the run-time manager controls the order of task execution and the advancement of simulation time. The dynamic computation is based on the design-time prepared application mappings. We implemented and validated the proposed approach using Intel Cofluent Studio modeling framework [2] and SystemC simulation language [6]. In this paper, the benefits of this approach are demonstrated through a case-study that considers seven applications (85 tasks in total) running on a heterogeneous multicore platform. Different management strategies are evaluated and compared according to application latency and power consumption criteria.

The remainder of this paper is as follows. In Section 2, we present relevant related work. The application and platform models are presented in Section 3. The principles of the proposed modeling and simulation approach are explained in Section 4. We present the implementation of the approach and its application through a case-study in Section 5. Finally, we conclude this paper in Section 6.

2 Background and Related Work

As presented in [11], many run-time management strategies have been proposed to optimize applications running on multicore platforms under real-time and energy consumption constraints. The evaluation of run-time management strategies aims at estimating the achieved resource usage and time properties such as system latency. As illustrated in [12–14], early evaluation of run-time management strategies are mostly done using analytical formal approaches. Analytical formal approaches are well adapted to predict system properties under worst-case situations but it can lead to pessimistic predictions. To the best of our knowledge only two related works support dynamism in the simulation framework.

In [10], an extension of the Sesame system-level modeling and simulation framework [9] is presented. Especially, a Run-time Resource Scheduler (RRS) is introduced to control mapping of applications for each simulated use-case. Based on trace-driven simulation approach [8], each application model records its action by a set of event traces (*i.e.* computation and communication events). RRS dispatches the event traces to an architecture model during system simulation. Our proposed approach differs in the way system simulation is performed. In our case, at the beginning of each use-case, the design-time prepared database is processed to compute the instants when platform resources are used. With the knowledge of the computed instants, our proposed run-time manager controls when application tasks are run on platform resources during system simulation.

Compared to the related works, the novelty of the proposed approach concerns the use of run-time combined execution traces to control the simulation of dynamic behaviors of applications. This approach can be adapted to different run-time management strategies and to different environments.

3 System Models

3.1 Application and Use-case Models

In this work, we consider periodic applications. An application, as illustrated in Fig.1, is characterized by a directed task graph $G_{App_i} = (T_{App_i}, E_{App_i})$, where T_{App_i} is the set of tasks of the application and E_{App_i} is the set of directed edges representing dependencies among the tasks. Tasks and edges in App_i are respectively denoted by $t_{i,j}$ and $e_{i,h}$, where j is the number of the task and h the number of the edge. A task represents an atomic, non-preemptive, code which execution time can vary over time according to processed data. In this paper, we restrict to periodically executed applications and each application App_i shall be executed within its period time $Period_{App_i}$.

In the scope of this work, applications follow the synchronous data flow (SDF) semantics that was proposed in [7]. In Fig. 1 input tokens define the number of tokens that are read from the edge before executing a task and the output token defines the number of tokens that are written through the edge after executing the task.

The set of simultaneously active applications defines a use-case $uc_i = \{App_1, App_2 \cdots, App_n\}$. Let $UC = \{uc_1, uc_2, \ldots, uc_l\}$ be the set of all possible use-cases. As we consider a dynamic execution scenario, different use-cases are active over time. This information is defined in the *Use-case Definition* in Fig.1.

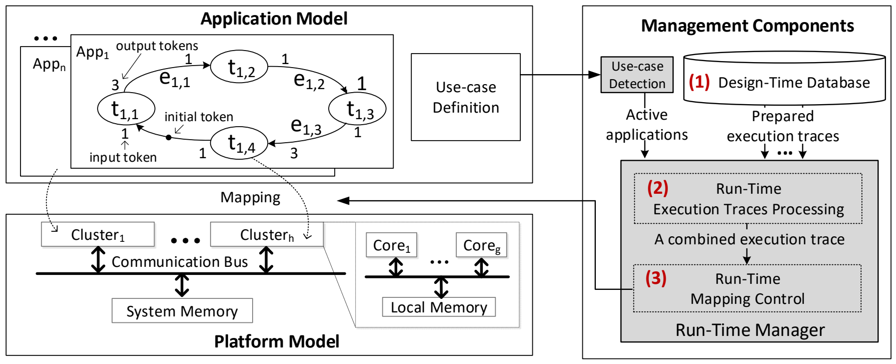


Fig. 1. System models with application, platform and management components descriptions.

3.2 Platform Model

This work targets heterogeneous cluster-based platforms, where each cluster consists of a set of homogeneous processing elements associated with a shared memory (see Fig.1). The cores within a cluster have the same voltage/frequency (v/f), and each cluster supports its own ranges of discrete v/f levels. One example of such platforms is the Samsung Exynos 5422 [1] with an ARM big.Little multicore architecture.

According to the platform model, we can define $CommTime_{e_{i,h}}$ as the communication time between dependent tasks via the edge $e_{i,h}$. Similarly, the computation time of a task $t_{i,j}$ executed on a specific cluster C at a given v/f level is defined as $CompTime_{t_{i,j}}(C, v/f)$. Additional power model can also be integrated in this approach (see the approach evaluation in Section 5).

3.3 Management Components

Application mapping defines the binding of application tasks to the architecture resources. As illustrated in Fig. 1, we consider run-time management in three steps: **(1)** a design-time preparation, **(2)** run-time mapping processing and **(3)** run-time mapping control.

In the design-time preparation step, one or several mappings for each application are prepared and stored in a database. Run-time mapping processing is performed when a new use-case is detected (*Use-case Detection*). In this step, a run-time mapping is established based on a particular algorithm (under evaluation) and, as defined in [11], on the analysis of design-time execution traces of every active application in the detected use-case. Afterwards, the third step introduces a new simulation approach to control the execution of active tasks during system simulation based on the mapping established in the previous step.

4 Run-Time Mapping Modeling and Simulation Method

In this section, the proposed three steps: design-time preparation, run-time mapping processing and a new simulation approach control for run-time mappings are detailed.

4.1 Design-Time Database preparation

The first step of most of dynamic resource managers is the design-time preparation. This step consists in storing into a database a set of prepared mappings, one or several for each application. The prepared mappings can be obtained by any design-time mapping algorithm.

In our approach, a mapping is characterized by its execution trace, *i.e* a set of instants defining the start (x_s) and end time (x_e) of each task when executed within one possible set of platform configurations (processing element, $v/f, \cdots$). Only the instants within a period are prepared for a design-time mapping.

To describe the execution trace of the design-time prepared mappings, lets consider that only one mapping is prepared for each application in $A = \{App_1, App_2\}$. According to the mapping strategy presented in [13], each task is mapped into one distinct core. The prepared mappings of App_1 and App_2 are illustrated in Fig.2[1].

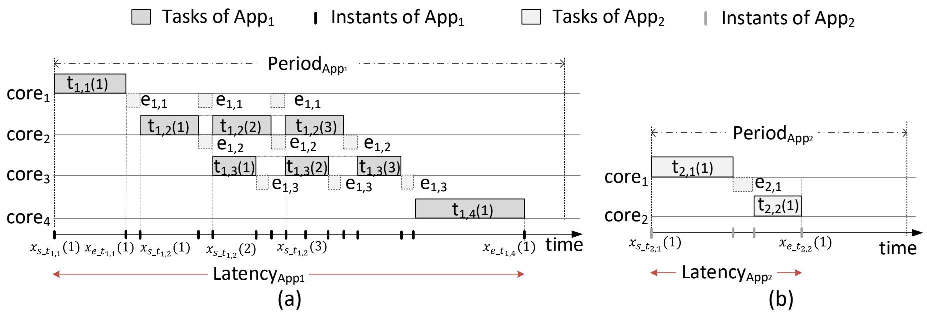


Fig. 2. A design-time prepared execution trace for the mapping of App_1 (a) and App_2 (b) on homogeneous cores, according to [13].

Let $X_{App_i} = \{x_{s\_t_{i,j}}(1), x_{e\_t_{i,j}}(1), \cdots x_{s\_t_{i,j}}(k), x_{e\_t_{i,j}}(k)\}, j \in \mathbb{N}^+, k \in \mathbb{N}^+$ be the execution trace of App_i where k refers to the k^{th} instance of a task. As an example, execution traces for applications App_1 and App_2 in Fig.2 are respectively defined by $X_{App_1} = \{x_{s\_t_{1,1}}(1), x_{e\_t_{1,1}}(1), \cdots, x_{s\_t_{1,4}}(1), x_{e\_t_{1,4}}(1)\}$, and $X_{App_2} = \{x_{s\_t_{2,1}}(1), x_{e\_t_{2,1}}(1), x_{s\_t_{2,2}}(1), x_{e\_t_{2,2}}(1)\}$.

Start and end instants x_s and x_e are expressed according to dependencies between tasks. In the example given by Fig.2, dependencies of task $t_{2,1}$ for instance, are expressed as follows: $x_{s\_t_{1,2}}(1) = x_{e\_t_{1,1}}(1) + CommTime_{e_{1,1}}(1)$ and $x_{e\_t_{1,2}}(1) = x_{s\_t_{1,2}}(1) + CompTime_{t_{1,2}}(1)$. For sake of clarity, communication time of edges are not illustrated in next figures. Finally, $Latency_{App_1}$ refers to the time duration for the execution of App_1 from the input to the last instant within one period. It has to be noticed that the instants here are relative as $CompTime_{t_{i,j}}(k)$ and $CommTime_{e_{i,j}}(k)$ will depend on the real mapping determined at run-time in the next step.

4.2 Run-Time Execution Traces Processing

The second step concerns the processing of run-time execution traces. This step is performed each time a new use-case uc_i is detected. The objective is to obtain at run-time a combined execution trace of the n active applications in the use-case uc_i defined by $X'_{Apps}(uc_i)$.

To obtain $X'_{Apps}(uc_i)$, the design-time prepared execution traces of each active application (X_{App_i}) are combined according to a given algorithm (different mapping combination strategies can be used). In the following, we denote the process of combining execution traces by $/processing$. Fig.3 gives an example of one possible combined execution trace of applications on homogeneous cores of use-case $uc_1 = \{App_1, App_2\}$. In this example the $LASP$ (Longest Available Slot Packing) strategy presented in [13] has been used. According to this example, $X'_{Apps}(uc_1) = /processing\{X_{App_1}, X_{App_2}\}$, and it includes all the execution instants, from x_s of the first task, to x_e of the last task in a least common multiple LCM of periods, for the active applications in uc_1.

[1] As can be seen in Fig. 2, $t_{1,2}$ and $t_{1,3}$ are executed three times for each iteration of App_1. App_2 is a 2-task application

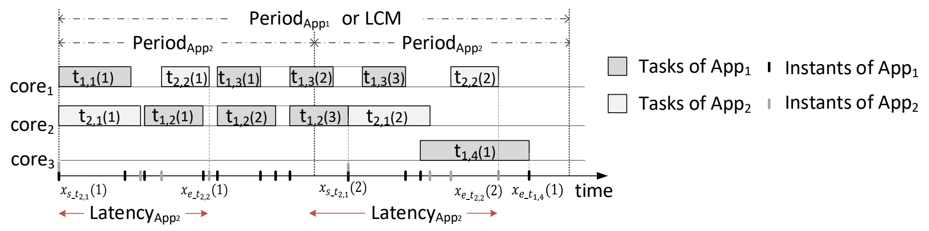


Fig. 3. A run-time combined execution trace $X'_{Apps}(uc_1)$ using strategy in [13].

In LASP, the instances of a task are always mapped into the same core through periods (task's instances $t_{2,1}(1)$ and $t_{2,1}(2)$, allocated on $core_2$, are an example). Due to this, once the execution traces are combined, the start time of $t_{2,1}(2)$ (i.e. $x_{s\_t_{2,1}}(2)$) is adjusted and delayed in order to start after the previous task allocated into $core_2$ (*i.e.*, starting instant dependency on $x_{e\_t_{1,2}}(3)$). The adjusted instants increase the $Latency_{App_2}$ of the second period. In $X'_{Apps}(uc_1)$ the instants are now absolute and computed according to the active mapping.

4.3 Run-Time Mapping Control

In our approach, the run-time mapping simulation, handled by the *Run-Time Manager* (RTM), aims to control the execution of tasks according to the information provided by the run-time execution traces processing (see in Subsection 4.2). The proposed simulation approach is depicted in Fig.4 through the previously used example of $uc_1 = \{App_1, App_2\}$.

The RTM is activated each time a new use-case is detected. As illustrated in Fig.4, when uc_1 is detected, the */processing* step is performed to determine the run-time mapping and dependencies of each task instance ($X'_{Apps}(uc_1)$). The execution of the */processing* action is done in zero simulation time with no call to the simulation kernel. The simulated effort of the RTM to perform this step depends on the used mapping strategy. The RTM then controls the states of each task according to the processed results.

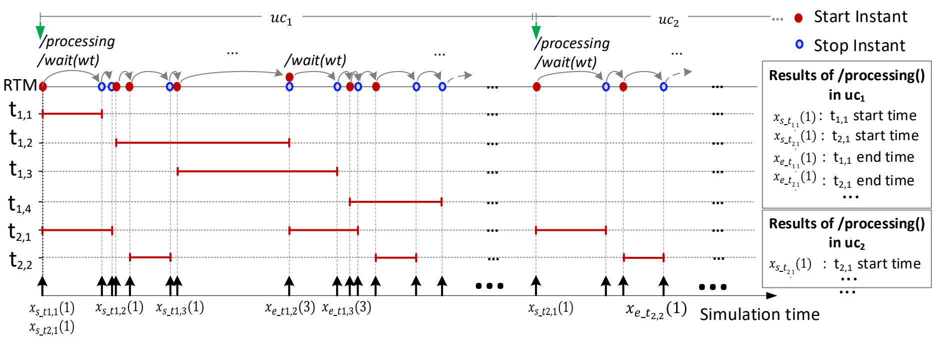


Fig. 4. System-level approach for the simulation of run-time mapping strategies through the dynamic control of the execution of tasks for different use-cases.

In Fig. 4, at the simulated instant $x_{s\_t_{1,1}}(1)$, $t_{1,1}(1)$ and $t_{1,2}(1)$ are started. The RTM inserts some simulation delays through action $/wait(wt)$ to wait for the next instant. Simulation time $SimTime$ moves forward $SimTime =$

$SimTime + wt$. As the next instant is $x_{e\_t_{1,1}}(1)$, the waiting duration wt is expressed as $wt = x_{e\_t_{1,1}}(1) - x_{s\_t_{1,1}}(1)$. After this time, $t_{1,1}(1)$ is stopped. As for task instances , $x_{s\_t_{1,2}}(1)$, $x_{s\_t_{1,2}}(2)$ and $x_{s\_t_{1,2}}(3)$, the RTM detects when several instances of the same task execute successively on the same processing core. In this case, only the first instance ($x_{s\_t_{1,2}}(1)$) is started and last instance $x_{s\_t_{1,2}}(3)$ is stopped. This further reduces the activity of the simulation effort. This process is repeated for all the task instances in every LCM period. When a new use-case is detected, the RTM performs the dynamic control for the tasks in the new use-case (*e.g.* $X'_{Apps}(uc_2)$).

It is worth noting that this approach allows as well the evaluation and simulation of run-time mapping strategies in a heterogeneous platform (different processing cores, different v/f levels, $\cdots$). In this case, the waiting time wt is dynamically adapted to the varying values of the computation and the communication time ($CompTime_t$ and $CommTime_e$), according to the heterogeneous resources configuration. An evaluation of the proposed run-time mapping control within a heterogeneous platform is presented in Section 5.

5 Evaluation of the modeling and simulation approach

5.1 Simulation Environment

We use the industrial modeling and simulation framework Intel CoFluent Studio [2] to validate the proposed approach. Once again, our proposal does not need any modification in the used framework and thus can be used in other environments.

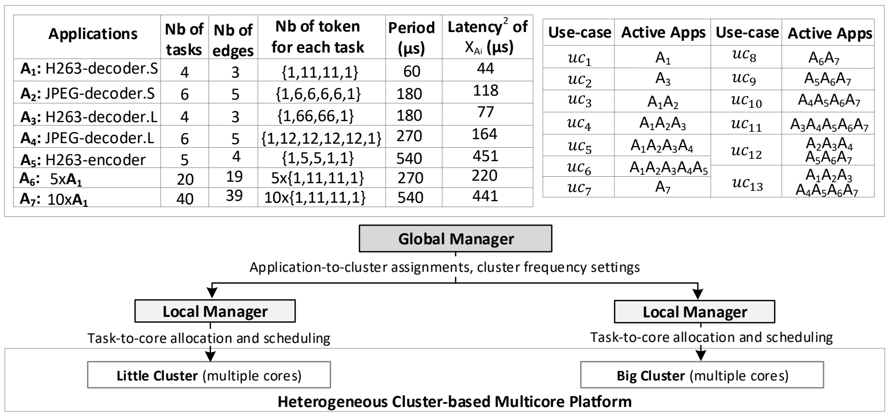
In the CoFluent framework, each application is modeled graphically with several functions (*i.e.*, tasks) and communications (*i.e.*, edges). For each function and communication, the computation/communication time and power consumption can be set by considering the influence of the platform. The built system model is then generated as a SystemC description for further execution analysis.

In our implementation, the run-time manager model is captured graphically and can be considered as a specific function of the system. The implementation of the action `processing` corresponds to the call to a C++ code that is developed to manipulate the previously defined data structures X_{App_i} and X'_{Apps}.

During the simulation, the run-time manager controls the states of each function and the advancement of simulation time according to the combined execution trace. Elementary procedures available in the used framework (`start`, `stop`, `resume`, `wait`) are used by the run-time manager to control the state of the functions. In the following, we evaluate the influence of this run-time manager model on the effort required for the system simulation.

5.2 Simulation Setup

In the case study, we aim to illustrate how the proposed modeling and simulation approach is applied to a heterogeneous architecture. The organization of the evaluated system is presented in Fig.5.



Applications	Nb of tasks	Nb of edges	Nb of token for each task	Period (µs)	Latency[2] of X_{A_i} (µs)
A_1: H263-decoder.S	4	3	{1,11,11,1}	60	44
A_2: JPEG-decoder.S	6	5	{1,6,6,6,6,1}	180	118
A_3: H263-decoder.L	4	3	{1,66,66,1}	180	77
A_4: JPEG-decoder.L	6	5	{1,12,12,12,12,1}	270	164
A_5: H263-encoder	5	4	{1,5,5,1,1}	540	451
A_6: 5x**A_1**	20	19	5x{1,11,11,1}	270	220
A_7: 10x**A_1**	40	39	10x{1,11,11,1}	540	441

Use-case	Active Apps	Use-case	Active Apps
uc_1	A_1	uc_8	A_6A_7
uc_2	A_3	uc_9	$A_5A_6A_7$
uc_3	A_1A_2	uc_{10}	$A_4A_5A_6A_7$
uc_4	$A_1A_2A_3$	uc_{11}	$A_3A_4A_5A_6A_7$
uc_5	$A_1A_2A_3A_4$	uc_{12}	$A_2A_3A_4A_5A_6A_7$
uc_6	$A_1A_2A_3A_4A_5$	uc_{13}	$A_1A_2A_3A_4A_5A_6A_7$
uc_7	A_7		

Global Manager
Application-to-cluster assignments, cluster frequency settings
Local Manager
Local Manager
Task-to-core allocation and scheduling
Task-to-core allocation and scheduling
Little Cluster (multiple cores)
Big Cluster (multiple cores)
Heterogeneous Cluster-based Multicore Platform

Fig. 5. Evaluated hierarchical run-time management of multiple applications executed on a heterogeneous cluster-based platform.

We considered H263 decoder, JPEG decoder and H263 encoder multimedia applications and two synthetic applications. Each application A_i, has been captured as an SDF model, based on the descriptions provided in SDF3 [3]. A_1 and A_3 (respectively A_2 and A_4) are set to consume different tokens sizes for processing at different data exchanging speed. The first five applications are representatives and require different computation time and power. To evaluate the scalability of the proposed simulation approach, A_6 and A_7 are arbitrarily created to significantly increase the number of tasks. They were created by duplicating A_1 5 and 10 times respectively, while the iterations execute successively in one period. Each application is constrained by a predefined period. For further evaluation, in the following, 13 possible use-cases are defined by different active applications (seen on top right part of Fig.5). The duration of each use-case is not depicted in the figure for the sake of clarity.

We choose the Samsung Exynos 5422 [1] platform as hardware target. As summarized in [4], the computation time of a task presents a ratio of $1 : 0.5$ when executed on the little (Cortex-A7) or the big (Cortex-A15) cluster. Besides, the ratio of power consumption of a task executed on the little cluster and the big cluster is set to $1 : 4$. This platform allows frequency scaling of each cluster, while the operating voltage is adapted to the frequency setting. The supported frequencies range from $0.2GHz$ to $1.4GHz$ for the little cluster, and from $0.2GHz$ to $2.0GHz$ for the big cluster. The frequency step is $0.1GHz$. We use the models of Exynos 5422 in [15] to model how computation time and dynamic power consumption of tasks change with frequency.

The hierarchical managers are built to implement run-time management strategies of the system. The two local managers are individually used for each cluster to optimize task-to-core allocation and scheduling. In order to coordinate the local managers, the global manager determines application-to-cluster allocations and sets cluster frequencies. The management strategies are based on design-time prepared execution traces. We establish X_{A_i} for each application by using the strategy in [13]. For each X_{A_i}, information provided in SDF3 [3] is used to compute the values of instants and the application latency (see Fig.5).

[2] The latency of X_{A_i} is assumed to be obtained in the little cluster at $1.4GHz$.

5.3 Validation of the Simulation Approach on Latency Criteria

In this part, the proposed simulation approach is applied to a homogeneous architecture, where a local manager determines the task-to-core mapping inside a cluster. LASP [13] is applied to get a combined mapping of active applications in a use-case and then the latency of each application can be obtained.

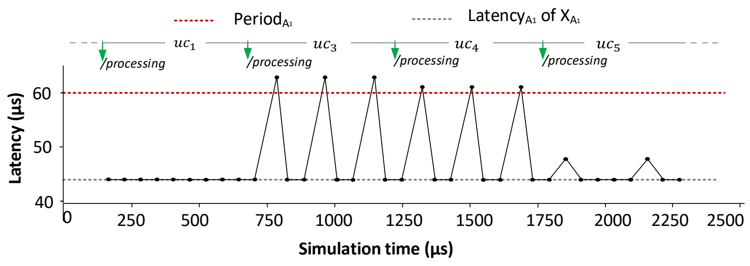


Fig. 6. Evolution of simulated A_1 latency, captured for four different use-cases. Results are given for the LASP strategy [13].

Fig.6 shows the latency evolution of A_1 in four different simulated use-cases. The simulations are performed in the little cluster at a fixed $1.4GHz$ cluster frequency. In this figure, the green arrows indicate the instants when a new combined execution trace is computed by $/processing$. For a clear illustration, the latency of A_1 is captured nine iterations for each use-case. In uc_1 where only A_1 is active, $Latency_{A_1}$ equals to the latency of its design-time prepared mapping. However, $Latency_{A_1}$ can be larger in uc_3, uc_4 and uc_5. In particular, the maximum $Latency_{A_1}$ in uc_3 and uc_4 even violate the timing constraint $Period_{A_1}$. As discussed in Section 4.2, the increase of latency comes from the possible delay of tasks re-allocation using the LASP combination strategy.

The latency of A_1 observed in the simulation is consistent with the latency obtained from the combined execution trace, while the combined execution trace is obtained by a run-time mapping strategy. Therefore, we can see that our simulation approach is able to correctly capture the behavior of an application under a dynamic management.

5.4 Validation of the Simulation Approach on Power Criteria

We then applied the proposed simulation approach to a heterogeneous architecture. In the simulated model, the global manager determines the application-to-cluster allocation and set the cluster frequencies. Different platform configurations lead to different computation time and different dynamic power consumption of a task.

Fig. 7 shows the dynamic power consumption of A_1 (in uc_1) under the control of the global manager. The green arrows indicate the instants when an execution trace is adapted according to different platform configurations. In the first configuration (little cluster, $1.4GHz$), index (1) corresponds to the active state of $t_{1,1}$. Index (2) indicates the activities of $t_{1,2}$ and $t_{1,3}$ that are active in parallel. Index (3) shows the activity of task $t_{1,4}$. Lets take $t_{1,1}$ as an example for further discussions. For this task, the power consumption with different platform configurations are represented in indexes (1), (4), (5) and (6), while the task

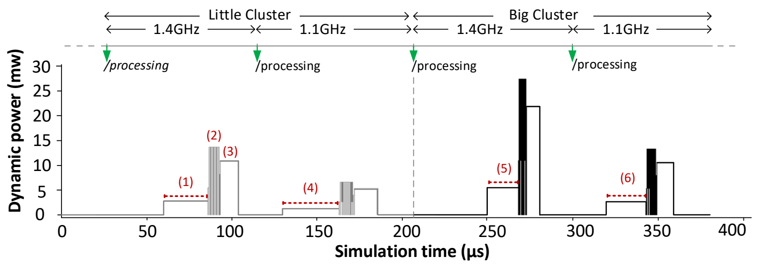


Fig. 7. Simulated dynamic power of A_1 is captured with the advancement of simulation time. Results are given for uc_1 according to different platform configurations.

computation time is reflected by the length of the red dotted lines. From (1) and (4), when the operating frequency decreases from $1.4GHz$ to $1.1GHz$, the dynamic power consumption of $t_{1,1}$ decreases and the task computation time increases. In the case of (1) and (5), $t_{1,1}$ is executed at $1.4GHz$ on the little cluster and big cluster respectively. The power consumption of the task observed on the big cluster is higher, while the computation time is smaller.

From the observed results in Fig.7, we can see that our approach is able to capture the behavior of an application with different platform configurations (*i.e.* different allocations, various v/f settings).

5.5 Evaluation of the Simulation Approach

Comparison of run-time mapping strategies: The proposed simulation approach allows the evaluation of different run-time management strategies. We compare two Local Management Strategies (LMS). LASP [13] is the first local strategy (LMS-1), which allows the task of different applications to be mapped on the same core. The second strategy (LMS-2) is the strategy introduced in [12], where only the tasks from one application can be mapped on the same core. The simulations are performed in the little cluster at $1.4GHz$. In Table 1, we summarize the estimated application latency for different use-cases. As previously observed, LMS-1 leads to some increase in application latency in some use-cases.

Three Global Management Strategies (GMS) are also compared. They differ in how they allocate applications to the clusters. GMS-1 and GMS-2 denote the strategies that allocate all the active applications to the little cluster and to the big cluster respectively. GMS-3 refers to the strategy that assigns applications

Table 1. Evaluation of run-time management strategies based on latency and power

Compared Critera	Strategy	uc_1	uc_2	uc_3	uc_4	uc_5	uc_6
Latency[3]	LMS-1	1	1	1.43	1.39	1.18	1.64
	LMS-2	1	1	1	1	1	1
System Power[4]	GMS-1	1.68	2	1.68	1.89	1.94	2.14
	GMS-2	1	1	1	1.12	1.15	1.07
	GMS-3	1	1	1	1	1	1

[3] Depicts the latency of the application that has the highest variation in a use-case. Each value is normalized by the latency obtained by LMS-2.

[4] Represents the average dynamic power of the system. Each value is normalized by the system power obtained by GMS-3.

to the two clusters by searching the best power efficiency. Once the application allocation is done, cluster frequency is decreased as much as possible under the timing constraints. Then LMS-2 is used in each local manager to determine task-to-core mapping. From Table 1, we can observe the poor power efficiency of using only one cluster.

Evaluation of simulation efficiency: We analyze the scalability of the proposed simulation method by comparing it with the CoFluent default simulation method. The proposed approach simulates the execution of applications under the control of the Run-Time Manager model (RTM), and different mappings can be provided for each application in different use-cases. On the other hand, without the RTM model, the default simulation approach only provides one static mapping of the applications in every use-case. Fig.8 shows the differences in the simulation effort between the two approaches. Simulation effort is characterized by the average time needed to complete one simulation run. The results include the execution traces processing and mapping control overheads.

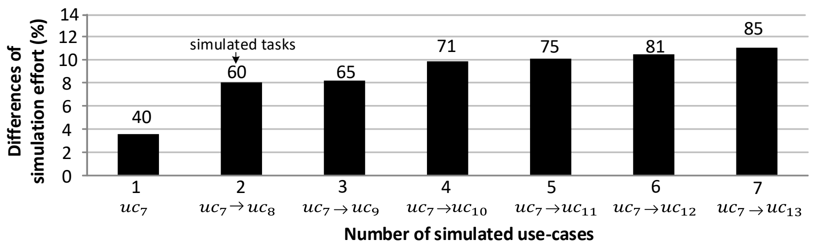


Fig. 8. The differences of simulation effort between the proposed approach and the default approach. Results are given for an increasing number of simulated use-cases and running tasks.

We define an increasing number of running use-cases within a fixed duration of simulation time, allowing each application to execute 100 to 240 periods. When the number of simulated use-cases increases from 1 to 7, the number of considered tasks increases from 40 to 85, while the difference of the simulation effort increases only from 3.8% to 10.8%. Since the proposed approach dynamically starts or stops the execution of tasks during simulation, it is reasonable to use more time to finish a simulation. But this overhead is also due to the fact that our approach takes into account the run-time manager in simulation while the default approach considers a static mapping (requiring eventually more corner-cases study). The improvement of the proposed simulation approach could be considered to reduce the dynamic activity of the run-time manager model.

6 Conclusion

In this paper, we present an approach to allow system-level simulation of run-time management strategies in multicore systems. This approach could be used to consider different numbers of applications executed on heterogeneous architectures at varied v/f configurations. It has been observed that the influence of the proposed approach on the simulation effort is reasonable (less than 10.8%

compared to the default Cofluent framework for 85 running tasks). In the future, we plan to work on reducing the simulation effort of the proposed approach.

References

1. Exynos 5 octa (5422). `Available:http://www.samsung.com/exynos`.
2. Intel cofluent studio. `Available:http://www.intel.com/`.
3. Sdf3. `Available:http://www.es.ele.tue.nl/sdf3`.
4. A. Butko, F. Bruguier, D. Novo, A. Gamatié, and G. Sassatelli. Exploration of performance and energy trade-offs for heterogeneous multicore architectures. *arXiv preprint arXiv:1902.02343*, 2019.
5. Andreas Gerstlauer, Christian Haubelt, Andy D Pimentel, Todor P Stefanov, Daniel D Gajski, and Jürgen Teich. Electronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.
6. IEEE computer society. IEEE standard SystemC language reference manual. IEEE Std. 1666–2011, 9 2011.
7. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.
8. Paul Lieverse, Pieter Van Der Wolf, Kees Vissers, and Ed Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI signal processing systems for signal, image and video technology*, 29(3):197–207, 2001.
9. A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.
10. W. Quan and A. Pimentel. A hybrid task mapping algorithm for heterogeneous mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(1):14, 2015.
11. A.K. Singh, P. Dziurzanski, H.R. Mendis, and L.S. Indrusiak. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Computing Surveys (CSUR)*, 50(2):24, 2017.
12. Amit Kumar Singh, Akash Kumar, and Thambipillai Srikanthan. A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 175–184. ACM, 2011.
13. Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Resource and throughput aware execution trace analysis for efficient run-time mapping on mpsocs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):72–85, 2016.
14. Andreas Weichslgartner, Stefan Wildermann, Deepak Gangadharan, Michael Glaß, and Jürgen Teich. A design-time/run-time application mapping methodology for predictable execution time in mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(5):89, 2018.
15. H. Zahaf, A. Benyamina, R. Olejnik, and G. Lipari. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture*, 74:46–60, 2017.