Variation-Aware Evaluation of MPSoC Task Allocation and Scheduling Strategies using Statistical Model Checking

Mingsong Chen[†], Daian Yue[†], Xiaoke Qin[‡], Xin Fu[§], and Prabhat Mishra[‡]
[†]Shanghai Key Lab of Trustworthy Computing, East China Normal University, China
[‡]Department of Computer & Information Science & Engineering, University of Florida, USA
[§]Department of Electrical & Computer Engineering, University of Houston, USA
Email: {mschen,dayue}@sei.ecnu.edu.cn; {xqin,prabhat}@cise.ufl.edu; xfu8@central.uh.edu

***Abstract*—To maximize the overall performance yield, variation-aware analysis is becoming a key step in Multiprocessor System-on-Chip (MPSoC) Task Allocation and Scheduling (TAS). Although various approaches have been investigated to improve performance yields, most of them cannot perform quantitative comparison among existing TAS heuristics, which is important for MPSoC designers to make decisions. Based on the statistical model checker UPPAAL-SMC, we propose a framework that can automatically evaluate the performance yield of TAS strategies under time and power constraints with variations. Experimental results show that our approach can not only filter inferior strategies efficiently, but also support the automated tuning of architecture and constraint parameters to achieve the required performance yield.**

I. Introduction

For MPSoC designs, efficient TAS strategies can maximize the utilization of available Processing Elements (PEs) while satisfying various design constraints (e.g., response time, power). However, due to variations across identically designed PEs and chips, the required constraints of MPSoC designs cannot be easily guaranteed [1]. Although traditional MPSoC TAS approaches adopt worst-case timing analysis to achieve feasible TAS solutions, without considering variations, it can lead to an overly pessimistic performance estimation. To measure TAS strategies under variations, *performance yield* was proposed to define the probability of an assigned TAS instance meeting required MPSoC constraints [3].

Various heuristic strategies [3], [5] have been proposed to maximize performance yields. However, due to the complexity of accumulated variations, it is hard for designers to determine which TAS strategy works best for a given MPSoC TAS problem. Therefore, quantitative evaluation of TAS strategies is becoming an important issue to guarantee the performance yield in MPSoC design. Although simplified statistical graph analysis-based approaches can calculate execution variation, few of them can accurately model parallel task executions. Moreover, constraint solving-based approaches can only answer *yes* or *no* for a given MpSoC TAS problem. None of them can quantitatively reason why a given performance yield cannot be achieved and answer how to improve the performance yield. Clearly, the bottleneck is the lack of powerful evaluation approaches which can help MPSoC designers to make choices among TAS strategies.

In this paper, we propose a novel TAS strategy evaluation framework based on the statistical model checker UPPAAL-SMC [8]. By simulating systems using underlying statistical methods (i.e., sequential hypothesis testing and Monte Carlo simulation), UPPAAL-SMC can estimate the satisfaction probability of a specified query (i.e., property). Compared to formal model checking approaches, UPPAAL-SMC requires far less memory and time, which allows scalable validation approximation. Our approach makes two major contributions: i) we propose a novel framework that can evaluate TAS strategies by automatically converting their solutions with variation information into a network of priced timed automata and conducting quantitative analysis using property-based queries; and ii) we present an automated approach that facilitates the tuning of design architecture and constraint parameters to improve the performance yield.

This work was partially supported by NSFC grants (Nos. 91418203 and 61202103) and Innovation Program of Shanghai Municipal Education Commission 14ZZ047. This work was also partially supported by NSF grants CNS-1441667, CCF-1351054 (CAREER) and CCF-1218629.

The remainder of this paper is organized as follows. Section II presents related works on variation-aware TAS optimization and statistical model checking (SMC) based evaluation approaches. After the introduction to the preliminary knowledge of UPPAAL-SMC in Section III, Section IV presents the details of our evaluation framework. Section V presents comprehensive evaluation results of a hypothetical application. Finally, Section VI concludes the paper.

II. Related Works

The aggressive technology scaling results in increasing performance and power consumption variations in microelectronic circuits [1]. Based on the assumption that the execution time of MPSoC tasks can be approximated with Gaussian distribution [2], various TAS approaches were proposed to maximize the MPSoC performance under process variation. In [3], Wang et al. first proposed the concept of performance yield for MPSoC designs. Based on statistical task graph analysis, they presented an efficient TAS algorithm to maximize performance yield. To stochastically minimize the total execution time of an application on MPSoCs under process variation, Singhal and Bozorgzadeh [4] proposed a non-stochastic approach to reduce computational complexity of task allocation computation. In [6], Chon and Kim developed a new task allocation and scheduling method that takes the impact of resource sharing into consideration. In [5], Huang and Xu presented a novel quasi-static scheduling algorithm, which can select a right schedule during the run time based on the actual variation for each chip to satisfy the given timing constraint. Although above approaches can achieve high performance yield, most of them focus on optimization rather than quantitative evaluation. Moreover, existing approaches support only one kind of distribution for task execution time, while our approach allows programming for the modeling of various complex distributions.

Due to the efficacy in quantitative query of performance related metrics, SMC is widely used in evaluating system designs. In [11], David et al. extends the stochastic semantics of UPPAAL-SMC to enable modeling of hybrid system networks. Based on the proposed stochastic hybrid automata, their approach can be used to perform modeling and evaluation of different kinds of stochastic objects in various domains such as cloud workflows [9] and energy-aware buildings. However, so far, SMC-based approaches are seldomly used in TAS modeling and evaluation in MPSoC design. To the best of our knowledge, our work is the first SMC-based approach that can not only evaluate TAS strategies in the system-level MPSoC design under process variation, but also can support the parameter tuning to maximize the performance yield.

978-3-9815370-4-8/DATE15/©2015 EDAA

III. Preliminary

Our approach adopts the Priced Timed Automata (PTAs) [7] to model task execution on allocated PEs. PTA is a variant of timed automata whose clocks can evolve with different rates in different locations. A Network of Price Timed Automata (NPTA) comprises a set of correlated PTAs that communicate via broadcast channels and shared variables. Figure 1 shows an example of an NPTA with two PTAs A (id=*ida*) and B (id=*idb*), where each PTA has four locations and one local clock (i.e., c_1 and c_2) respectively. In this example, we use an array of broadcast channels $msg[2]$ for the purpose of synchronization. While using message-based synchronization, we adopt the non-deterministic *selections* to filter useless messages. For example, on the outgoing edge of location B_1 in Figure 1, the *selections e:msg_t* and the *guard condition e==id* are used to filter messages which are not sent to PTA B.

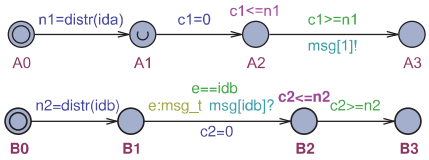


Fig. 1. An NPTA, (A | B)

Since we focus on TAS evaluation, we need to model the stochastic behaviors of PTAs. Currently, UPPAAL-SMC only supports the normal and exponential distributions explicitly, which cannot cover all complex scenarios in practical designs. To enable the modeling of different kinds of stochastic behaviors, we adopt the pattern shown in Figure 1. By proper programming using the built-in function *random*(), the self-defined function *distr*() can produce values following a large set of commonly used distributions. For example, the *Box-Muller method* can be used to generate normally distributed random delays for PTAs. In the pattern, since location A_2 sets an upper bound for clock c_1 (i.e., $c_1 <= n_1$) and its outgoing transition sets a guard condition $c_1 >= n_1$, PTA A can stay in location A_2 with a delay of t_1 (randomly generated by *distr(ida)*). The parallel execution semantics of NPTA can be found in [7]. Note that if there is a PTA process in a *commit* or *urgent* location (i.e., a location marked with the symbol "C" or "U"), the process will have a zero delay in this location, and the next transition must involve an edge from one of the *commit* or *urgent* locations (*commit* locations have higher priority).

Assume that PTAs A and B follow the Gaussian distribution $N(3,1^2)$ and $N(6,2^2)$ respectively. The following run is a possible transition sequence of the NPTA $(A|B)$.

$$((A_0,B_0),[c_1=0,c_2=0]) \xrightarrow{0} ((A_1,B_1),[c_1=0,c_2=0]) \xrightarrow{0}$$
$$((A_2,B_1),[c_1=0,c_2=0]) \xrightarrow{2.5} \xrightarrow{msg[1]!} ((A_3,B_2),[c_1=2.5,c_2=0]) \xrightarrow{5.1}$$
$$((A_3,B_3),[c_1=7.6,c_2=5.1]) \xrightarrow{\cdots} \ldots$$

The example demonstrates that the composite location (A_3,B_3) is reachable within 7.6 time units. Assuming that there is no correlation between clocks n_1 and n_2, while more runs are simulated, we can find that the time to reach the composite location (A_3,B_3) will follow the Gaussian distribution $N(9,1^2+2^2)$. Based on proper distribution modeling in *distr*() and message-based synchronization among PTAs, arbitrarily complex stochastic behavior can be described. By monitoring random simulation runs bounded by either time or other constraints, UPPAAL-SMC can report the probability range of each property with a specified confidence. To facilitate TAS strategy evaluation, our approach adopts the temporal logic based properties [8] for quantitative query. The details will be described in Section IV-D.

IV. SMC-Based TAS Evaluation

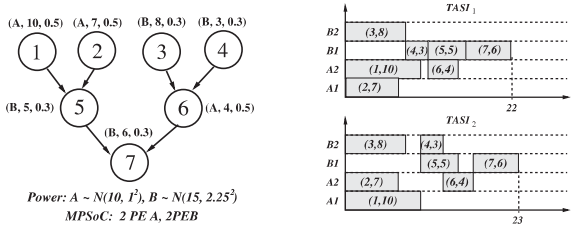
Aiming at improving performance yields under variations, this section presents the details of our SMC-based evaluation framework for system-level TAS strategies. In this paper, we focus on the time and power variations in TAS.

A. Problem Definition

The MPSoC TAS problem under variations studied in this work is formulated as follows. Given

- A directed acyclic graph (DAG) $G=(V,E)$ indicating the task graph of an MPSoC application, where each node in $V=\{v_1, \ldots, v_n\}$ represents a task. E is a set of directed edges which represent precedence relations between tasks.
- A set of PEs $PE=\{p_1, \ldots, p_m\}$ denoting all the available PEs on a given chip that can serve all the tasks in G, and a set of types of PEs $PT=\{pt_1, \ldots, pt_k\}$.
- A PE type function $T_{PE}: PE \rightarrow PT$, where $T_{PE}(p_i)$ represents the type of p_i ($p_i \in PE$). A task type function $T_{task}: V \rightarrow PT$, where $T_{task}(v_j)$ denotes the type of PEs to which task v_j ($v_j \in V$) can be allocated. A task v_j can be allocated to p_i if and only if $T_{task}(v_j)=T_{PE}(p_i)$.
- An execution time distribution function $ETD: V \times PE \rightarrow DIST$, where $ETD(v_i,pt_j)=dist$ denotes that the execution time of task v_i running on a PE of type pt_j follows the distribution $dist$.
- A power distribution function $PD: PE \rightarrow DIST$, where $PD(p_i)=dist$ indicates that of the running power of PE p_i follows the distribution $dist$. A power consumption function $Power: PE \times \mathbb{R} \rightarrow \mathbb{R}$, where $Power(p_i,t)$ denotes the power consumption of p_i at time t.
- X TAS functions representing X TAS solutions produced by different TAS strategies. They are in the form of $TAS_x: V \rightarrow PE \times 2^V$, where $TAS_x(v_i)=(p_j,Prec_{v_i})$ ($x \le X$, $v_i \in V$ and $p_j \in PE$) indicates that v_i is assigned to PE p_j with a precedence relation $Prec_{v_i}$ indicating that the task v_i can be dispatched when all the tasks in $Prec_{v_i}$ are finished. A TAS instance is a binary relation $TASI_x=\{(v_1,TAS_x(v_1)), \ldots, (v_n,TAS_x(v_n))\}$, which corresponds to function TAS_x. $BEGIN(v_i,TAS_x)$ and $END(v_i,TAS_x)$ indicate of the real begin and finish time of the allocated task v_i in a random simulation of TAS_x.
- Design constraints p, t and py, where p indicates the maximum power constraint; t is the maximum response time constraint of the MPSoC application and py is the required performance yield under the power and time constraints.

To achieve a feasible TAS instance $TASI$, it is required that the probability of successful application execution which meets the requirement $\forall t.\ \Sigma_{i=1}^{m} Power(p_i,t) \le p$ and $Max_{i=1}^{n} END(v_i,TASI) \le t$ is equal to or larger than py, i.e., $PY(p,t,TASI) \ge py$. The TAS evaluation process is to find a TAS instance $TASI_y$ ($y \le X$) such that $PY(p,t,TASI_y) \ge Max_{i=1}^{X} PY(p,t,TASI_i)$.



(a) Task graph with assigned PEs (b) TAS instances

Fig. 2. A TAS problem and its two TAS instances

Figure 2(a) shows an example of a task graph together with variation information of allocated PEs following Gaussian

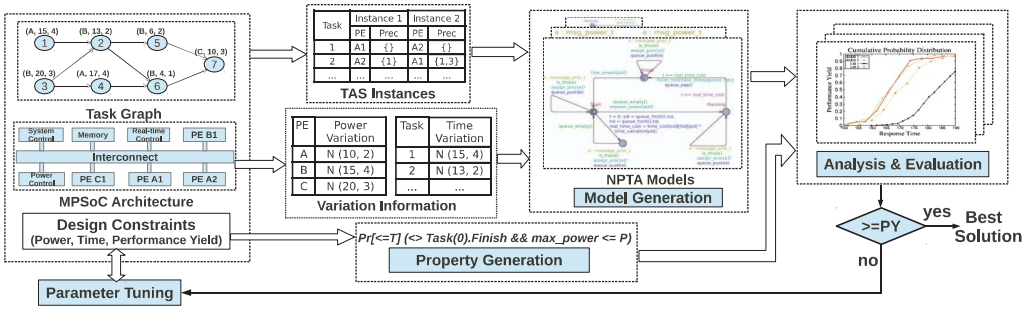


Fig. 3. Our TAS strategy evaluation framework

distribution. For example, task 1 is attached with a setting $(A, 10, 0.5)$ which means that task 1 can only be assigned to PEs of type A, and its execution time follows the Gaussian distribution $N(10, 0.5^2)$. The power variation of different PE types is also given in the figure. Assume that the target MPSoC has two A-type PEs and two B-type PEs, and it requires that the generated TAS instance TAS should satisfy $PY(35, 24, TAS) \geq 0.9$. Without considering any variations, the TAS instances $TASI_1 = \{(v_1,(A_2,\{\})),(v_2,(A_1,\{\})),(v_3,(B_2,\{\})),(v_4,(B_1,\{2,3\})),(v_5,(B_1,\{1,2,3,4\})),(v_6,(A_2,\{1,2,3,4\})),(v_7,(B_1,\{1,2,3,4,5,6\}))\}$ and $TASI_2 = \{(v_1,(A_1,\{\})),(v_2,(A_2,\{\})),(V_3,(B_2,\{\})),(v_4,(B_2,\{1,2,3\})),(v_5,(B_1,\{1,2,3\})),\ (v_6,(A_2,\{1,2,3,4\})),(v_7,(B_1,\{1,2,3,4,5,6\}))\}$ shown in Figure 2(b) can both satisfy the requirement. In Figure 2(b), each task represented by a rectangle is marked with the execution time information. For example, $TASI_1$ allocates task 3 to the PE B_2 of type B. We use $(3,8)$ to indicate that the nominal (mean) execution time of task 3 is 8 time units. We can find that, without variations, $TASI_1$ needs less time than $TASI_2$, and both of them can achieve 100% performance yield. However, while tasks are dispatched in the same order as above TAS instances considering variations, due to different task overlap scenarios, it is difficult to determine which TAS instance has better performance yield.

B. Our Framework

As shown in Figure 3, our UPPAAL-SMC-based evaluation framework supports the comparison between different TAS strategies. By using our defined mapping rules, the generated TAS instances produced by different strategies combined with variation information can be automatically converted into NPTA models, and design constraints can be translated into properties which enable queries for performance yields. Based on the capability of UPPAAL-SMC model checker, our framework can automatically analyze and evaluate the generated TAS instances. If there are multiple satisfying TAS instances, the best one will be finally reported. If none of the TAS instances meets the designer's constraints, our framework can automatically search for satisfying TAS instances by iteratively tuning design architecture and constraint parameters, regenerating TAS instances and performing reevaluation. Since the model generation, property generation and UPPAAL-SMC-based model checking can be fully automated, both the evaluation and parameter tuning processes can be conducted without human intervention. The following subsections describe the major steps of our framework in detail.

C. NPTA Model Generation

When evaluating a TAS instance, we need to first convert it into an executable NPTA model. In our approach, we adopt three kinds of PTAs: task, PE, and power monitor. The task PTA describes the execution of a single task in the given task graph. The PE PTA specifies the behavior of a single PE handling multiple allocated tasks. The power monitor PTA monitors the power usage information of the whole chip. To automate an NPTA model construction, we divide it into two parts: front-end models and back-end configuration. All the TAS problems share the same front-end models but different back-end configuration. Front-end models describe common behaviors for the PTAs, and back-end configuration defines a set of data structures (e.g., task graph DAG, PE variations, synchronization) which are used to control the front-end model execution. To ease the property generation (see Section IV-D), we use a dummy task with $tid = 0$ to merge all the tasks without any successors. The execution time of the dummy task is 0 without any variations. In our framework, we assume that a given TAS problem has *T+1* tasks (with $tid \in [0, T]$) and *P* PEs (with $pid \in [0..P\text{-}1]$).

TAS instances generated by different strategies may have different dispatching order of tasks on resources. As defined in Section IV-A (second parameter of TAS_x), in back-end configuration, tasks are sorted using a precedence relation indicating the task dispatching order of $TASI_x$. To model the concurrent task execution, we construct a precedence matrix $PM[T+1][T+1]$ in the back-end configuration, where $PM[i][j] = 1$ means that task v_i finishes before v_j starts. In our approach, a task can be executed if all its precedent tasks are complete. As an example shown in Figure 2, for TAS_1, the task v_5 can be started only when its predecessors v_1, v_2, v_3 and v_4 finish. When one task finishes, it will notify all its successors. Therefore, each task should figure out how many predecessors have been finished and how many successors need to be notified. In the back-end configuration, we define two arrays *pre_count[T+1]* and *post_count[T+1]* to indicate how many predecessors have completed their tasks and how many successors need to be notified after the completion of the current task, respectively. Both arrays are initialized using the information of the precedence matrix. During the simulation, the values of these two arrays are updated by inter-task synchronization.

In the back-end configuration, we use two multi-dimensional arrays *tvar* and *pvar* to specify the distribution parameters for execution time and power, respectively. For example, if designers select the normal distribution mode, in back-end configuration our framework will generate two arrays *tvar[T+1][2]* and *pvar[T+1][2]*, which specify the mean value and standard deviation of task execution time and PE power. Before simulation, the real execution time of each task and power information of each PE following the specified distribution are initialized and saved in the arrays *real_time[T+1]* and *real_power[P]* respectively.

To enable the communication between different PTAs, we adopt the broadcast-based synchronization. In our approach, task PTAs, PE PTAs, and the power monitor PTA can only accept the messages of type *msg_task_t*, *msg_proc_t*, and *msg_power_t* respectively. In back-end configuration, each task is assigned with a private channel to receive notifications from predecessor tasks. Such channels are combined to form a channel array *t_notify[T+1]*. For task allocations, we use an array *assignp[T+1]* to provide assignment mapping

between PEs and tasks. To enable the dispatching of tasks to PEs, we utilize an array of channels *assign_proc*$[(T+1)\times(P)]$, where *assign_proc[e]* is used to assign a task with $tid = e/P$ to a PE with $pid = e\%P$. After a PE finishes executing a task v_j, it will use a channel array $pt\_notify[T+1]$ to notify the PTA of task v_j, where $pt\_notify[j]$ is used as a private channel from all PEs to the task v_j. Similarly, to enable the power monitoring, we adopt two channel arrays *requestP[P]* and *freeP[P]* which can dynamically update the total power (saved in *total_power*) of all current running PEs and the maximum overall power (saved in *max_power*) so far.

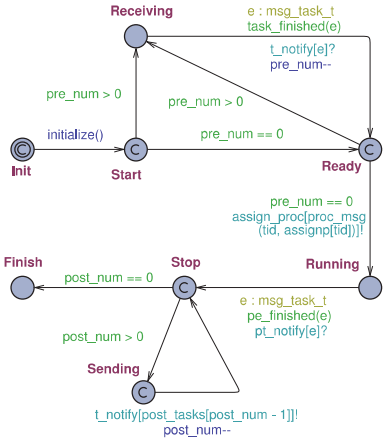


Fig. 4. Front-end model of a task

Based on the global data structures and functions defined in the back-end configuration, the front-end models describe the behaviors of tasks, PEs, and the power monitor. Since tasks share the same behavior template, we only need one model for all tasks. The execution and interaction of all tasks are specified and synchronized using the information provided in the back-end configuration. Figure 4 presents the model of a task. It contains 8 states: i) *Init state* initializes data structures in both the front-end task model and back-end configuration. Based on the *PM* information, we can figure out all the IDs of successor tasks (saved in a local array *post_tasks[T+1]*) and the number of predecessor and successor tasks of the current task (saved in local variables *pre_num* and *post_num*). Initialization also randomly generates initial "real" execution time for all the tasks (saved in *real_time[T+1]*) and power for each PE (saved in *real_power[P]*) following specified distributions. ii) *Start state* indicates that the task is active. iii) *Receiving state* tries to obtain notifications from all the predecessor tasks. iv) *Ready state* means that the task is ready to be dispatched. v) *Running state* indicates that the task has been assigned to the designated PE with $pid = assignp[tid]$. vi) *Stop state* means that the task execution is done, when it assures that a notification is sent from the designated PE (i.e, *pe_finished(e)==true*). vii) *Sending state* notifies all the successor tasks saved in *post_tasks* about the ending of current task. viii) *Finish state* represents the end of the task.

Similar to tasks, PEs share the same front-end model as shown in Figure 5. In our approach, each PE is associated with a task queue which allows accepting new incoming tasks no matter whether the PE is idle or busy. When a PE starts to execute, it needs to notify the power monitor via channel *request_p[pid]* to update the *max_power*. When the task finishes, the PE will notify the corresponding task ($tid = finish\_msg()$) about the completion via channel *pt_notify[tid]*, and try to load a new task from its task queue. A PE model contains 4 states: i) *Start state* tries to receive new incoming tasks. It uses *t_assign(e)* to check whether a task is sent to itself or not. ii) *Waiting state* waits for new tasks when its task queue is empty. iii) *Running state* indicates the execution of a task whose execution time is saved in *real_time[e/P]* and manipulated by *real_exec_time()*, which corresponds to the function $distr()$ in Figure 1. Before running, the PE notifies the power monitor about the execution via channel $request\_p[pid]$. iv) *Finish state* terminates the running task.

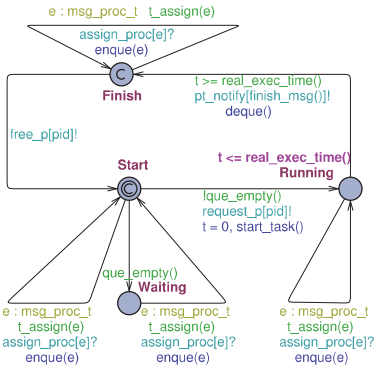


Fig. 5. Front-end model of a PE

For each TAS instance, one power monitor model as shown in Figure 6 is created to monitor the overall power of PEs. It has two states: *handling state* deals with power switch for running tasks, and *waiting state* listens to task events from PE models. The power monitor keeps a global variable *max_power* which indicates the maximum power so far. Via *request_p[e]* and *free_p[e]* channels, the overall power can be updated using the functions *power_alloc(e)* and *power_free(e)*.

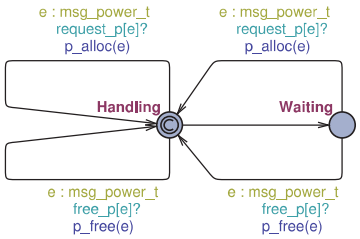


Fig. 6. Front-end model of the power monitor

D. Property Generation

When NPTA models are generated from different TAS instances, we need to compare their performance yields. In our framework, we assume that MPSoC designers would like to figure out *"what is the performance yield if the task graph can be completed under the power constraint of x and time constraint of y"*. In UPPAAL-SMC, we conduct the query using the following property in the form of temporal logic:

$$Pr[<=x](<> Task(0).Finish \;\&\&\; max\_power <= y),$$

where $Task(0).Finish$ denotes the completion of the whole task graph, and $max\_power <= y$ indicates that the overall power will never exceed y. In UPPAAL-SMC, the property works as a simulation monitor to check whether a run can satisfy the property finally (denoted by $<>$) within time x (denoted by $<= x$). When the checking finishes, the distribution of the probability of successful simulation runs will be reported to enable the quantitative analysis.

E. TAS Instance Evaluation

Since response time and power in MPSoC design requirements are often conflicting with each other, it is very difficult to guarantee the optimality of the two parameters at the same time, especially under variations. To achieve such a satisfying TAS instance, different MPSoC designers may prefer to use different TAS strategies, since they do have different demands. For example, if MPSoC designers

focus on real-time applications, they will be inclined to adopt TAS strategies with optimal response time. Unfortunately, so far, no existing approach can guarantee to find TAS instances with best performance yield under both power and response time constraints. Moreover, even for the satisfying TAS instances achieved using the same strategy, the performance yield under variations may be significantly different. Therefore, it is practical for MPSoC designer to conduct the comparison among multiple TAS strategies.

To support MPSoC designers' decision making on TAS, our framework provides two power-constrained time minimization heuristics by default: i) list scheduling which can quickly achieve near-optimal schedules, and ii) BULB approach [10] which explores all the search space to obtain an optimal result in a branch-and-bound manner. It is important to note that, due to the independence between TAS instance generation and evaluation, other TAS strategies can be easily integrated into our framework. To obtain an approximate solution for evaluations, all these strategies produce TAS instances using nominal values of PE power and task execution time without considering variations. Based on the quantitative analysis results of generated TAS instances, our framework can automatically filter inferior instances and select the best TAS instance with respect to the performance yield or response time. Moreover, our framework also supports the MPSoC architecture exploration, since TAS instances generated from different MPSoC architectures can be evaluated in the same way.

F. Parameter Tuning

In TAS strategy evaluation, if none of the generated TAS instances satisfy the required performance yield, our framework supports the tuning of constraint or architecture parameters to find new satisfying TAS instances based on designers' demand. For example, to design real-time applications, power constraint parameters will be modified such that shorter application response time can be achieved. To explore TAS instances with higher performance yields, our evaluation framework supports the tuning of multiple kinds of design and constraint parameters. Now, we discuss two typical parameter tuning strategies. Due to space limit, we do not list all kinds of tuning strategies in our framework.

Power Constraint Tuning: Since our framework contains TAS strategies based on power-constrained time minimization heuristics, our framework supports the tuning of power constraint. At the beginning of tuning, the power constraint is the same as the required power constraints. If the performance yield is smaller than the requirement, our approach will reduce the power constraint by a fixed number, regenerate a new TAS instance, and conduct the evaluation again. The process will be iterated until: i) the performance yield is achieved, or ii) the time constraint is violated with the settings of nominal values.

MPSoC Architectural Configuration Tuning: For specific real-time applications, we may customize MPSoC with different number and type of PEs to achieve a better performance yield. Our framework can figure out all possible architectural configurations which meet both time and power constraints under a given TAS strategy. By comparing the performance yields of TAS instances under different configurations, our framework can select the best architecture for the given MPSoC application and TAS strategy.

V. Case Study

To evaluate the effectiveness of our framework, this section presents the evaluation results of a synthetic example generated by TGFF [12], which is an open source tool designed to generate pseudo-random task-graphs. In the experiment, we adopt UPPAAL-SMC (version 4.1.18 with parameters $\varepsilon = 0.05$, $\alpha = 0.05$) as the engine of our framework. All the other components of our framework including TAS generation heuristics (e.g., list scheduling, parallel BULB [10]), model/property generation, and parameter tuning are implemented using C programming language. By modifying the task enumeration ordering of list scheduling and parallel BULB approach, we can get different TAS instances. In the experiment, we use *List x* and *BULB y* to indicate different versions of list scheduling and BULB approaches with different task enumeration orderings. All the experiments were conducted on a Ubuntu Linux desktop with 4.0GHz AMD FX CPU and 8 GB RAM.

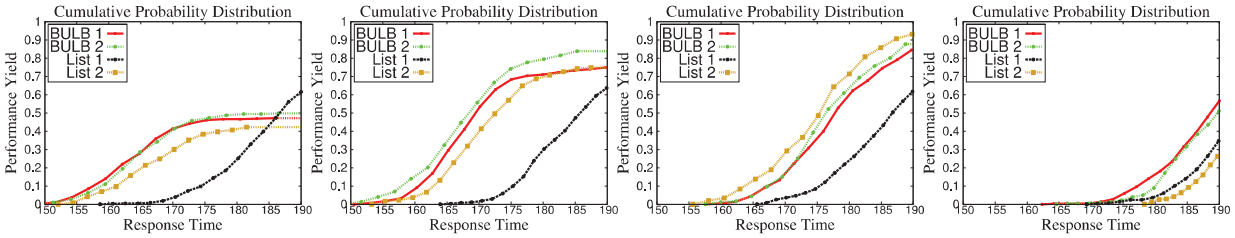
In this case study, we assume that the task execution time and power follow Gaussian distribution without considering correlation between PEs. Note that our framework can be easily extended to support the correlation modeling by adding extra data structures in the back-end configuration and modifying the front-end functions *initialize*() in the task model and *real_exec_time()* in the PE model. By using TGFF, we generate a 22-node task graph with a maximum input degree of 3 and a maximum output degree of 2. Beside ID information, each task node is labeled with the information of corresponding PE type and mean execution time. Table I presents both execution time and power variations for different PE types. For example, for the PE of type A, its power distribution follows the Gaussian distribution $N(10, 1^2)$, and its execution time follows the Gaussian distribution $N(x,\ (0.05x)^2)$, where x represents the mean execution time of a task on PEs of type A. For example, if the mean execution time of a task allocated to some PE of type A is 10, its execution time will follow the Gaussian distribution $N(10, 0.5^2)$.

TABLE I
Power Variation for MPSoC PEs

Type	Power Variation	Exec. Time Variation
A	N(10, 1.00^2)	N(x, $(0.05x)^2$)
B	N(15, 2.25^2)	N(x, $(0.10x)^2$)
C	N(20, 2.60^2)	N(x, $(0.07)^2$)

Assume that the MPSoC design is fixed before the TAS of MPSoC application, and the given MPSoC design consists of 8 PEs, including 3 PEs of type A, 3 PEs of type B, and 2 PEs of type C. Assume that the design constraint is that "*the performance yield cannot be smaller than* 90% *within 190 time units and 65 power units*". In other words, the evaluation tries to find a TAS instance such that $Pr[<=190](<> Task(0).Finish \ \&\& \ max\_power <= 65) >= 0.90$. Initially, to obtain feasible TAS instances, we apply four TAS strategies: two list scheduling variants (i.e., *List 1* and *List 2*) and two BULB variants (i.e., *BULB 1* and *BULB 2*) as described in Section IV-E.

We generate four TAS instances using the provided strategies *List 1-2* and *BULB 1-2* under the constraints of 190 time units and 65 power units. Figure 7(a) shows the evaluation results of these four TAS instances. Interestingly, we can find that the TAS instance generated by *List 1* achieves the best performance yield under design constraints, though list scheduling is not an optimal searching approach for shortest schedules without considering variations. At time 190, all these four TAS instances show different performance yields. For example, the TAS instance generated by *List 1* can achieve the performance yield of 60% on average (i.e., the performance yield range is $[0.55, 0.65]$ with a confidence of 95% reported by UPPAAL-SMC), and the performance yield evaluation requires simulation time of 2067 seconds. In addition, this figure shows that TAS instances generated using *BULB 1* and *BULB 2* have better response time than the ones generated by *List 1* and *List 2* strategies. For example, *BULB 2* instance can achieve a performance yield of 50% at time 175, while *List 1* needs 187 time units to achieve the same performance yield. When the power constraint equals to 65, no strategy can achieve the specified performance yield. For the two TAS instances generated by *BULB* approaches, we can find that the reason of low performance



(a) Power constraint = 65 (b) Power constraint = 60 (c) Power constraint = 55 (d) Power constraint = 50

Fig. 7. Evaluation results under different power constraints

yield is mainly due to the power variation, because the increase of execution time (response time) cannot effectively improve the performance yield after time 175.

Since power and response time are two conflicting constraints, lowering the power constraint will increase the MPSoC application response time. To reduce the impact of the power variation without drastically affecting the response time, in this experiment, we reduce the power constraint by 5 time units in each tuning iteration. Note that the reduced power constraints are applied for the TAS instance generation only. We do not change the power constraint in the query property for the quantitative analysis. Therefore, if there exists enough slack time, due to the increasing tolerance of power variation, tuning down the power constraint for TAS instance generation can improve the overall performance yield.

In the second evaluation iteration, the power constraint used for TAS instance generation is set to 60. Figure 7(b) shows the evaluation results of the four newly generated TAS instances. Compared to the results presented in Figure 7(a), the performance yields of both *BULB 1-2* and *List 2* increase drastically. The reason of such improvement is mainly due to the increasing tolerance of power variation for the newly generated TAS instances. Since no TAS instance in Figure 7(b) satisfies the specified performance yield, the power constraint tuning process continues and Figure 7(c) shows the evaluation results when the power constraint is reduced to 55. Due to the larger tolerance of power variation and sufficient slack time, we can achieve a satisfying TAS instance (i.e., instance generated by *List 2* approach) with 93% performance yield. From Figure 7(a)-Figure 7(c), we can see the trend of an increasing average response time for the TAS instances. This is because that the reduction of power constraint will increase the overall scheduling time of the task graph. When the tuning iteration sets the power constraint to 50, all the four strategies still can generate TAS instances within 190 time units regardless of variations. Figure 7(d) shows the evaluation results. We can find that though the power constraint (i.e., 50 power units) can generate satisfying TAS instances using nominal values, while considering variations, the performance yield is far lower than the requirement. When the power constraint is set to 45, none of the four TAS strategies can produce feasible schedules within 190 time units based on nominal constraints. Therefore, the whole evaluation process terminates. Finally, the TAS instance generated using the *List 2* strategy shown in Figure 7(c) will be reported as the best choice.

Besides design constraints, architectural configuration plays an important role in determining the performance yield. Assume that we can customize MPSoCs with different architectural configurations to execute the given task graph generated by TGFF, and assume that each MPSoC chip can have at most 8 PEs. Our framework can also conduct the evaluation for such different configurations. Figure 8 shows the evaluation results of four architectural configurations using the TAS strategy *BULB* 1. In order to have proper tolerance of power variation, the power constraint for the TAS instance generation is set to 60 (original power limit in design constraint is 65). In this figure, $Proc(1,4,3)$ means the MPSoC has 1 PE of type A, 4 PEs of type B, and 3 PEs of type C. From this figure, we can find that the configuration $Proc(2,3,3)$ has the best performance yield and average response time.

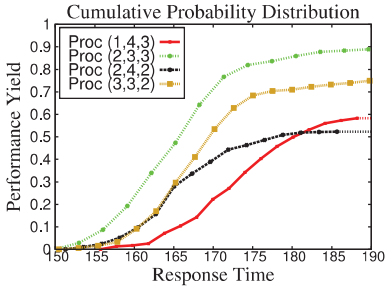


Fig. 8. Evaluation of different architectural configurations

VI. Conclusions

To reduce the decision making efforts of MPSoC designers, we propose a UPPAL-SMC-based evaluation framework, which can automatically compare the performance yields of different task allocation and scheduling (TAS) strategies under variations. Our framework can accurately model TAS strategies under time and power variations as well as support performance yield queries on the generated models. Moreover, our framework enables design constraint parameter tuning to explore TAS instances with better performance yields. Comprehensive experimental results demonstrated the efficacy of our framework.

References

[1] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. Rosing, M. Srivastava, S. Swanson and D. Sylvester. Underdesigned and Opportunistic Computing in Presence of Hardware Variability. *IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD)*, 32(1): 8-23, 2013.

[2] S. Hervert and D. Marculescu. Characterizing Chip-Multiprocessor Variability-Tolerance. *Proc. of DAC*, 313-318, 2008.

[3] F. Wang, C. Nicopoulos, X. Wu, Y. Xie and N. Vijaykrishnan. Variation-Aware Task Allocation and Scheduling for MPSoC. *Proc. of ICCAD*, 598-603, 2007.

[4] L. Singhal and E. Bozorgzadeh. Process Variation Aware System-Level Task Allocation using Stochastic Ordering of Delay Distributions. *Proc. of ICCAD*, 570-574, 2008.

[5] L. Huang and Q. Xu. Performance Yield-Driven Task Allocation and Scheduling for MPSoCs under Process Variation. *Proc. of DAC*, 326-331, 2010.

[6] H. Chon and T. Kim. Resource Sharing Problem of Timing Variation-Aware Task Scheduling and Binding in MPSoC. *The Computer Journal*, 53(7): 883-894, 2010.

[7] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. Poulsen, J. Vliet and Z. Wang. Statistical Model Checking for Networks of Priced Timed Automata. *Proc. of FORMATS*, 80-96, 2011.

[8] K. G. Larsen. Priced Timed Automata and Statistical Model Checking. *Proc. of IFM*, 154-161, 2013.

[9] S. Huang, M. Chen, X. Liu, D. Du and X. Chen. Variation-Aware Resource Allocation Evaluation for Cloud Workflows using Statistical Model Checking. *Proc. of BDCloud*, 2014, accepted.

[10] M. Chen, L. Zhou, G. Pu and J. He. Bound-Oriented Parallel Pruning Approaches for Efficient Resource Constrained Scheduling of High-Level Synthesis. *Proc. of CODES+ISSS*, 1-10, 2013.

[11] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikucionis, D. Poulsen and S. Sedwards. Statistical Model Checking for Stochastic Hybrid Systems. *Proc. of Int. Workshop on Hybrid Systems and Biology*, 122-136, 2012.

[12] R. P. Dick, D. L. Rhodes and W. Wolf. TGFF: Task Graphs for Free. *Proc. of CODES*, 53(7): 97-101, 1998.