Resource and Throughput Aware Execution Trace Analysis for Efficient Run-time Mapping on MPSoCs

Amit Kumar Singh, *Member, IEEE,* Muhammad Shafique, *Member, IEEE,* Akash Kumar, *Senior Member, IEEE,* Jörg Henkel, *Fellow, IEEE*

***Abstract*—There have been several efforts on run-time mapping of applications on Multiprocessor-Systems-on-Chip (MPSoCs). These traditional efforts perform either on-the-fly processing or use design-time analyzed results. However, on-the-fly processing often leads to low quality mappings, and design-time analysis becomes computationally costly for large-size problems and require huge storage for large number of applications. In this paper, we present a novel run-time mapping approach, where identification of an efficient mapping for a use-case is done by the online execution trace analysis of the active applications. The trace analysis facilitates for fast identification of the mapping while optimizing for the system resource usage and throughput of the active applications, leading to reduced energy consumption as well. By rapidly identifying the efficient mapping at run-time, the proposed approach overcomes the mappings' exploration time bottleneck for large-size problems and their storage overhead problem when compared to the traditional approaches. Our experiments show that on average the exploration time to identify the mapping is reduced 14× when compared to state-of-the-art approaches and storage overhead is reduced by 92%. Additionally, energy and resource savings are achieved along with identification of high quality mapping.**

***Index Terms*—Embedded Systems, Multiprocessor-Systems-on-Chip (MPSoCs), run-time mapping, design space exploration, throughput constraint**

I. Introduction

APPLICATION mapping on Multiprocessor-Systems-on-Chip (MPSoCs) has been identified as one of the most important problems in embedded systems design [1]–[3]. Under dynamic workloads, run-time mapping is required in order to optimize for performance and energy consumption towards fulfilling the end user demands. The run-time mapping can be accomplished either with or without previously analyzed results. For mapping without previously analyzed results, i.e., on-the-fly processing, a large body of research exists [4]–[7]. Efficient heuristics have been used to assign new arriving tasks on the system resources. These heuristics may not guarantee schedulability and a high quality mapping due to limited processing power at run-time.

Manuscript received August 01, 2014; revised December 20, 2014. This work is supported in part by the Tier 2 Singapore Ministry of Education Academic Research grant number R-263-000-B33-112 and German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing [SFB/TR 89]; http://invasic.de

A. K. Singh is with the Department of Computer Science, University of York, York YO10 5GH, UK (email: amit.singh@york.ac.uk)

A. Kumar is with the Department of Electrical and Computer Engineering, National University of Singapore, Singapore 117583 (e-mail: akash@nus.edu.sg).

M. Shaque and J. Henkel are with the Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe 76131, Germany (e-mail: muhammad.shafique@kit.edu; henkel@kit.edu).

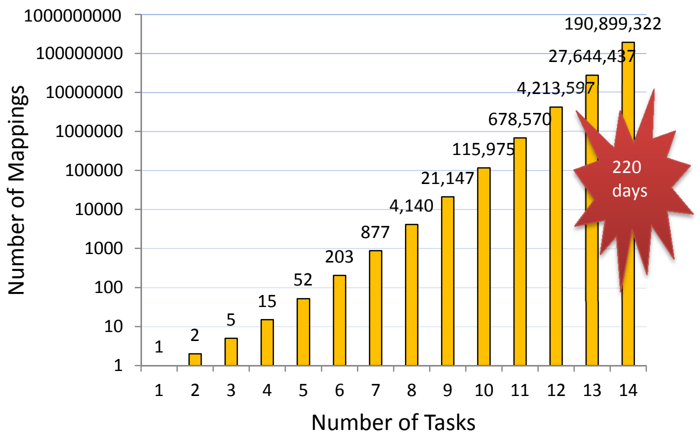


Fig. 1: Number of possible mappings with varying number of tasks.

To overcome the run-time processing bottleneck, recently, the focus has moved to shift the compute intensive analysis to design-time [8]–[13]. The analyzed results are used at run-time to facilitate efficient run-time mapping. Such mapping approaches also facilitate for a light-weight run-time platform manager, which is required in modern embedded systems (e.g., smart phones and tablets). During the analysis, Design Space Exploration (DSE) is performed to generate multiple mappings for an application or a use-case (i.e., a set of simultaneously active applications) [14]. These mappings are then used to handle dynamism in resource availability and throughput requirement at run-time. The literature has advanced for the DSE [15]–[19], but DSE process still imposes high computational cost. The simulation time to evaluate the design points (mappings) forms the real bottleneck in the DSE. Additionally, computational complexity of DSE brings major concern for the cases when the application/platform complexity (size) is high and the number of use-cases is huge due to large number of applications. Such cases require evaluation of a huge number of mappings.

An example: Fig. 1 shows the number of all the unique possible mappings as the application complexity increases in terms of number of tasks. The number of mappings follows

Bell Number [20]. For n = 14, a total of 190,899,322 mappings are possible (formula provided in later section), which will take approximately *220 days for evaluation* if we assume just 100 milliseconds (ms) to evaluate (simulate) one mapping. Therefore, computational complexity of DSE needs to be reduced in order to have an acceptable exploration time.

To accelerate the DSE, state-of-the-art efforts employ either *heuristics to prune the design space* [8]–[12] or *analytical estimations along with simulations to perform fast and accurate evaluations* [18], [21]–[24]. The design space is pruned by performing optimization for one or several metrics, e.g., performance [12], power [8], and jointly performance and power [9]–[11]. The approaches employing design space pruning have several drawbacks such as chances to miss the evaluation of efficient mappings. Furthermore, they use pure simulative evaluations and thus exhibit long exploration time that may not be acceptable. To reduce the exploration time, the DSE approaches in [18], [21]–[24] employ estimations along with simulations. They use analytical estimation models to rapidly identify the points of interest in the design space and then use simulative evaluations on the identified points for accurate evaluations.

It should be noted that most of the DSE strategies perform exploration for individual applications one after another (e.g., [12], [13], [15]–[17], [19]). Therefore, we might not get efficient mappings for all the active applications at run-time by mapping them one by one. This happens because the availability of system resources for different applications varies over time. *Therefore, jointly optimizing for all the active applications at the same time may lead to better results.*

There has been some focus on multiple applications DSE, where multiple application mapping scenarios are explored at design-time in order to handle dynamism in number of active applications (use-case or scenario) at run-time [25]–[28]. For n applications, there are 2^n possible use-cases (scenarios). This indicates that scenario based DSE is not scalable as the number of scenarios increases exponentially with the number of applications. Additionally, storage overhead to store mappings for various use-cases becomes huge. *Therefore, there is a need to devise a strategy for rapid identification of the efficient (maximum throughput) mapping at run-time for different use-cases in order to overcome the issues such as joint optimization for active applications, exploration time and storage overhead.*

Our Novel Contributions and Concept Overview: In order to address the above-discussed challenges, we propose:

1) A run-time trace analysis strategy to rapidly identify the efficient mapping for supporting a use-case. The strategy analyses the execution traces of applications within the use-case and performs mapping identification while maximizing for the throughput and resource usage.
2) To enable run-time trace analysis, a design-time strategy to extract and store the execution traces of individual applications that are used by the run-time platform manager during various use-case executions.
3) A trace storage overhead reduction strategy to reduce the memory requirement.

For each application, the design-time strategy stores execution traces for 1 task to 1 core mapping such that each core contains exactly 1 task. This kind of mapping enables to exploit maximum task level parallelism present in the application. The run-time identified mapping excels in terms of several performance metrics such as throughput, energy, resource usage, etc. The analysis strategy looks at the idle time of a core and sees if any other executions can be placed on the same core with minimal throughput degradation of the use-case applications towards satisfying the throughput constraints. Our approach performs fast, accurate and high quality identification at run-time. *To the best of our knowledge, this is the first work that addresses run-time trace-based analysis for rapid identification of the efficient mapping for a use-case.*

Paper Organization: The remainder of this paper is organized as follows. Section II introduces some related work. Section III and IV present system model and a motivational example to perform trace-based DSE, respectively. The proposed novel run-time mapping approach is described in Section V. Section VI describes our experiments and comparison to the state-of-the-art. Finally, Section VII concludes the paper.

II. Related Work

The mapping process can be accomplished either at *design-time* or *run-time*. Design-time mapping techniques (e.g., [29]–[33]) are suitable for static workload scenarios (e.g., a predefined set of applications to be mapped on a static platform) and thus cannot handle dynamism in applications incurred at run-time. Examples of dynamism could be supporting different use-cases at different moments of time. The need of run-time mapping techniques is witnessed to handle such dynamic workloads.

The run-time mapping techniques face the challenge to efficiently map tasks of required applications on the platform resources while keeping accurate knowledge of resource occupancy. The reported literature on run-time mapping has broadly spanned in two directions: *1)* on-the-fly mapping, and *2)* mapping using design-time DSE results.

The *on-the-fly mapping techniques* perform all the required computations as and when an application or a set of applications (referred to as use-case) need to be supported into the system. There are numerous efforts on on-the-fly mapping [4]–[7], [34]–[36]. These efforts use different kinds of heuristics to assign tasks of incoming applications on the system resources while putting the best efforts to optimize for one or several performance metrics such as overall execution time and energy consumption. Since the tasks are assigned (mapped) to resources by taking quick on-line decisions due to limited on-line processing capability, these techniques might lead to low quality mapping.

The *mapping techniques using design-time DSE results* achieve a better quality of mapping than the on-the-fly techniques as better mapping decisions taken at design-time are used [8]–[13]. During the design-time DSE, several mapping options are explored and those providing high quality for different kinds of run-time scenarios are stored to be used at run-time. The run-time process just selects the design-time computed mapping and configures the platform based

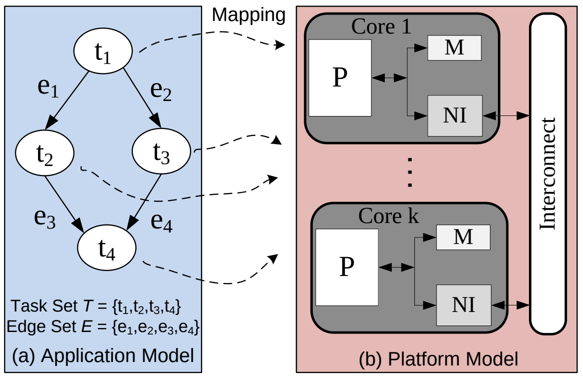


Fig. 2: Example application model, platform model, and mapping of application tasks on platform resources.

on the corresponding tasks to resources allocation. Although these approaches provide better quality of mapping, they incur high storage overhead requirement for large-size applications/platforms in order to keep the design-time explored mappings to be used at run-time.

Further, the design-time DSE approaches are computationally costly and cannot finish the evaluation within a limited time for large-size problems. The main reason being the simulation time to evaluate each explored candidate mapping. Design space pruning (e.g., in [8]–[12]) and analytical estimations along with simulations (e.g., in [18], [21]–[24]) are employed to accelerate the DSE process. The design space pruning and analytical estimations may result in inefficient mappings due to discarding evaluation of efficient mappings and inaccurate estimations, respectively. Our previous studies have shown that design-time DSE strategies employing pruning provide mapping solutions that are non-optimal by 10%, i.e. 10% deviation from the optimal solution [12].

Despite taking several measures to accelerate the DSE process, the exploration time by existing design-time DSE strategies still remains quite high to be applied to identify the high quality mapping at run-time. Further, the DSE approaches require large storage space to store several potential mappings. In contrast to existing approaches, our approach performs fast and accurate identification of an efficient mapping at run-time in order to support a use-case, and overcomes the exploration time and storage space problem of design-time approaches. Our run-time approach analyses execution traces of the active applications within the use-case to identify the efficient mapping. The execution traces of all the applications that might need to be supported into the system in various combinations are extracted and stored at design-time in order to use them at run-time. Additionally, our approach leads to reduced energy consumption and better resource usage (supporting results in Section VI).

III. System Model and Problem Formulation

A. Application and MPSoC Platform Model

An **application** is a directed graph $AG = (T, E)$, where T is the set of tasks of the application and E is the set of directed edges representing dependencies amongst the tasks. Fig. 2(a) shows an example application graph model. Each task $t \in T$ has attributes *1)* execution time (ExecTime) and *2)* memory requirement, when mapped on a core. The ExecTime for each task is considered as its worst-case execution-time (WCET) and remains fixed. Each edge $e \in E$ represents data that is communicated between the dependent tasks.

An **MPSoC platform** is a directed graph $PG = (C, V)$, where C is the set of cores and V represents the connections amongst the cores. Fig. 2(b) shows an example platform graph model. Each core $c \in C$ consists of a processor (P), a local memory (M) and a network interface (NI). The communication amongst the cores is established by connecting them to a mesh-based interconnection network through the respective network interface. Each connection $v \in V$ connects two cores and contributes to the network. Dedicated connections are used to facilitate communications and thus latency (communication time) between cores remains constant. Examples of such network include circuit-switched networks (e.g., AEthereal [37]) that provide guaranteed throughput, implying constant latency.

B. Total Energy Consumption Model

The total energy (E_{total}) is computed as the sum of dynamic and static energy as follows.

$$E_{total} = E_{dynamic} + E_{static} \tag{1}$$

where $E_{dynamic}$ is computed as the sum of communication (E_{comm}) and computation (E_{comp}) energy, which are required to transfer and process the data, respectively [12].

The E_{comm} depends upon *1)* data volume, *2)* energy required to transfer one bit of data and *3)* distance between the communicating points (cores). Energy required to transfer one bit between core c_i and c_j is computed as follows.

$$E_{bit}(i,j) = E_{bit}^{link} \times (hops(i,j) - 1) + E_{bit}^{router} \times hops(i,j) \tag{2}$$

where $hops(i,j)$ are the number of routers between core c_i and c_j, and E_{bit}^{link} and E_{bit}^{router} are the energy consumed in link and router, respectively. The E_{comm} is estimated by summing over all communicating task pairs (edges).

$$E_{comm} = \sum_{\forall comm-cores} data(i,j) \times E_{bit}(i,j) \tag{3}$$

where $data(i,j)$ is the transferred data volume between communicating cores c_i and c_j. The computation energy to process all the tasks is estimated as follows.

$$E_{comp} = \sum_{\forall t \in T} t_{ExecTime} \times P_{dynamic} \tag{4}$$

where $P_{dynamic}$ is dynamic power consumption of a core.

The E_{static} for each core is computed as the product of overall execution time of the use-case (or application) and static power consumption of the core. For p used cores, total static energy is computed as $p \times E_{static}$, and unused cores are considered as power gated so that they don't contribute to the overall energy consumption.

C. Mapping Problem

Mapping of an application AG on MPSoC platform PG is represented as follows.

$$M_k : AG \xrightarrow[n]{map} PG \tag{5}$$

which assigns n application tasks to k used platform cores. Fig. 2 indicates mapping of an application with 4 tasks on platform cores, where t_1 and t_4 are mapped on core 1 and core k, respectively, and remaining tasks (t_2 and t_3) are mapped on some other platform cores. The application can use a maximum of 4 cores by allocating 4 tasks on 4 different cores. Each used core c_i gets associated with a set S_i, which contains the tasks mapped on c_i. A use-case mapping can be achieved in the similar way by considering the tasks of all the active applications.

The total number of mappings to assign n tasks on k cores subject to at least one task on each core follows *Stirling numbers of the second kind* that provides the number of ways to place n labeled balls (tasks) into k unlabeled boxes (cores) [38], and is computed as follows.

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}({}^kC_2)j^n \tag{6}$$

${}^kC_2$ is *k-choose-2*, and $S(n,n)$ = $S(n,1)$ = 1.

For n tasks, the total number of mappings using 1 core to n cores follows *Bell number* and can be computed as follows.

$$B_n = \sum_{k=0}^{n} S(n,k) \tag{7}$$

For n=10, B_n = 115,975 mappings. The number of mappings increases with number of tasks n and thus the overall evaluation time. Even employing design space pruning cannot overcome the evaluation time bottleneck as evaluation of each mapping takes several milliseconds [39]. Further, design space pruning lead to several other deficiencies as mentioned earlier.

At run-time, the mapping problem in this paper targets following constraints and objectives.
Constraints: number of available cores.
Objectives: minimize exploration time; maximize throughput and resource usage (leading to reduced energy consumption).

IV. Motivation for Trace-based Analysis

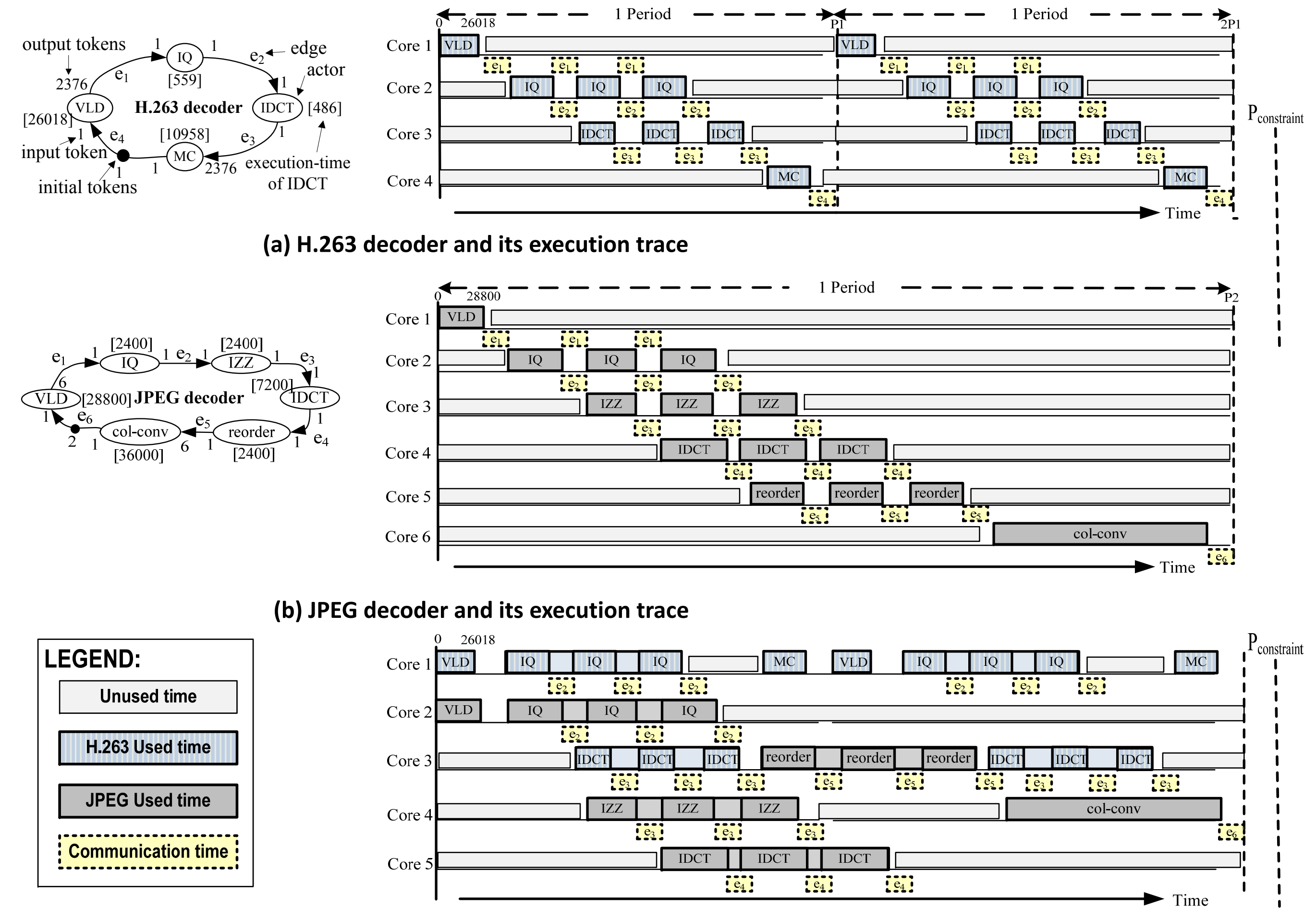
This section discusses the importance of trace-based analysis towards performing rapid DSE. Let us consider a simple example of a use-case containing two multimedia applications: H.263 and JPEG decoder, which needs to be supported into the system at run-time.

The applications can be modeled using directed graphs (DGs) as described in Section III-A. However, we choose a special class (or subset) of DG called Synchronous Data Flow Graphs (SDFGs) (details in [40], [41]) that are often used to model multimedia applications with timing constraints. The reason behind choosing the SDF is that it provides easier modeling of multimedia applications and techniques to compute various SDF parameters such as throughput and storage requirements already exist. Without the loss of generality, other application models falling into DG category, e.g. CSDF, can also be considered. The left hand sides of Fig. 3(a) and (b) show SDFG models of H.263 and JPEG decoder, respectively. The nodes of an SDFG are referred to as *actors*, which implement functions to be executed by reading *tokens* (data) from the input edges and write the execution outcomes as tokens on the output channels. An actor consumes a fixed amount of tokens from the input edges and produces a fixed amount of tokens on the output edges upon firing (execution). An actor *fires* (executes) when there are sufficient input tokens on all of its input edges and sufficient buffer space on all of its output channels. These token amounts are also referred to as *rates*. The edges may contain *initial tokens*, as indicated by bullet points in Fig. 3(a) and (b). In Fig. 3, H.263 decoder contains 4 actors connected by edges and there are 6 actors in the JPEG decoder.

The multimedia applications are characterized by throughput constraints [42]. Throughput is determined as the inverse of the long term period (i.e. the average time needed for one iteration of the application). For the example H.263 decoder, period is equal to the summation of ExecTime(*VLD*), 2376×ExecTime(*IQ*), 2376×ExecTime(*IDCT*) and ExecTime(*MC*), where ExecTime is the WCET of respective actors. This period does not include network and memory access delays. It should be noted that actors *IQ* and *IDCT* have to execute 2376 times in one iteration (periodic execution) and the number of executions for each actor is referred to as *repetition vector* of the actor. The rate 2376 defines a resolution of 348 by 288 pixels for the used video frames. An SDFG with a throughput of 100 Hz has a period of 10 milliseconds (ms), i.e. takes 10 ms complete one iteration.

State-of-the art methodologies (like [8]–[12]) select the best mapping for all the actors (4 actors of H.263 & 6 actors of JPEG) from the design-time DSE mappings. The exhaustive DSE explores 115,975 mappings for a total of 10 actors, leading to unacceptable exploration time. Even employing heuristics based explorations to prune the design space cannot overcome the exploration (evaluation) time bottleneck as the design space still remains quite huge. Further, such explorations need large storage to store several potential mappings.

It may be beneficial to analyze the execution traces of active applications (e.g., H.263 & JPEG decoder) to identify the efficient mapping rapidly. The execution traces of the actors are captured as the start and end time of their active executions (firings) and are shown on the right hand side of Fig. 3(a) and (b) for H.263 and JPEG decoder, respectively. In the shown executions, each actor and edge is mapped on a different core and connection between cores, respectively. First, actor *VLD* executes for both the applications as there are sufficient input (initial) tokens on the incoming edges e_4 and e_6 for H.263 and JPEG, respectively. Thereafter, for H.263, it generates 2376 tokens to be transferred through e_1 to process them one by one by *IQ*, whereas 6 tokens are generated for JPEG. The transfer of tokens through edges and their processing by different actors follows the traces shown in Fig. 3(a) and (b). For easier understanding and space limitations, the shown trace considers rates as 3 in places of 2376 and 6. Thus, actors fire maximum 3 times. It should also be noted that applications



(a) H.263 decoder and its execution trace

(b) JPEG decoder and its execution trace

(c) Packed used and unused time intervals

Fig. 3: Execution trace analysis of applications' traces to identify an efficient mapping.

exhibit periodic executions, i.e. actor *VLD* for H.263 and JPEG fires again after finishing the execution of actors *MC* and *col-conv*, respectively, and similar execution patterns are followed in the upcoming periods.

Observation: An analysis can be performed to utilize the used and unused time intervals information represented with different shades in Fig. 3(a) and (b) *towards merging unused time (idle time) on one core with used times (busy times) on other core if the executions (used intervals) are not in parallel.* Thus, the core utilization will get increased by imposing more executions (used time intervals) on it and almost similar execution can be performed with less number of cores for both the applications.

A Potential Solution: Fig. 3(c) shows one potential solution to pack (merge) used and unused time intervals towards identifying an efficient actors to cores mapping, where only 5 cores are used for the 10 actors and similar execution patterns as that of Fig. 3(a) and (b) are achieved for both the applications. Thus, the efficient mapping can be rapidly identified by trace analysis and merging. However, *high quality and accurate identification is challenging.* In the following, we propose an approach that overcomes the aforementioned challenges.

V. Proposed Run-time Mapping Approach

An overview of our proposed run-time mapping approach is presented in Fig. 4, which operates in three main steps (explained in subsequent sections).

- Execution trace extraction of individual applications.
- Trace storage optimization to reduce storage overhead.
- Run-time trace analysis to identify the efficient mapping in terms of resource usage, throughput and energy consumption in order to support a use-case.

A. Execution Trace Extraction

The execution trace extraction is performed during the course of throughput computation for a given actors-to-cores mapping. For throughput computation (not a contribution of this paper), we deploy the technique of [39]. For each core, the technique first constructs static-order schedule that orders the execution of bound actors. Then, all the binding and scheduling decisions are modeled in a graph called binding-aware SDFG. Finally, self-timed state-space exploration of the binding-aware SDFG is performed and throughput is computed from the periodic part of the state-space.

Trace Capturing: The execution traces of actors of an application are captured as the start time (*ST*) and end time (*ET*) of their active executions (firings) during one periodic

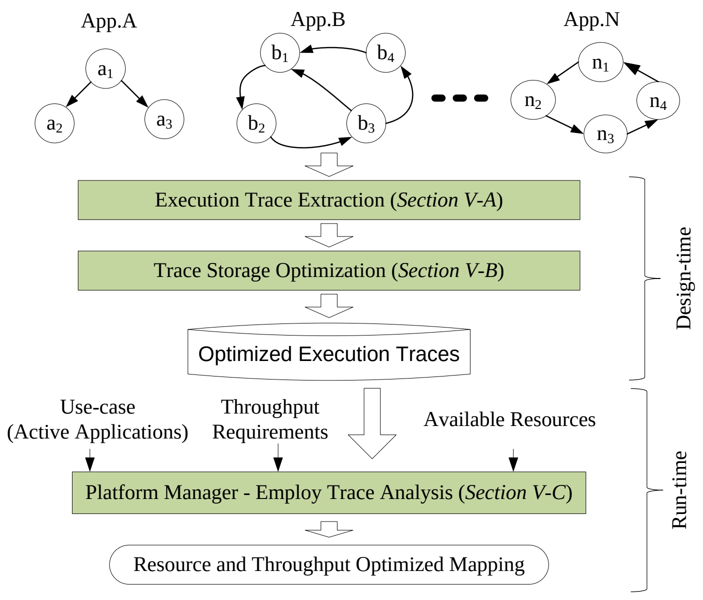

App.A
App.B
App.N
Execution Trace Extraction (Section V-A)
Trace Storage Optimization (Section V-B)
Optimized Execution Traces
Design-time
Use-case (Active Applications)
Throughput Requirements
Available Resources
Platform Manager - Employ Trace Analysis (Section V-C)
Resource and Throughput Optimized Mapping
Run-time

Fig. 4: Overall Flow of the proposed mapping approach showing the design-time and run-time steps.

execution. The traces are extracted for 1 actor to 1 core mapping (as shown in Fig. 3(a) and (b)) in order to exploit maximum task level parallelism. In case there is overlapping executions amongst multiple iterations, in steady state, there will be a periodic execution of the application containing some overlaps. This is due to the fact that in steady state self-timed execution the application executes periodically and one period may contain multiple iterations of an SDF graph application. We capture all the executions during one periodic execution and there might be overlapping executions as well. This indicates that our approach is applicable to different kinds of SDF graphs as they exhibit periodic execution behavior.

During trace capturing, we consider worst-case latency, i.e. the maximum possible hop in the architecture and thus results are valid for runtime mapping resulting in different hops. It should be noted that the communication latency for different hops in our considered architecture that uses dedicated end-to-end connections to model the mesh does not vary a lot, e.g. 10 cycles for one hop and 16 cycles for 6 hops. Further, the actors execution times are in the order of thousands of cycles, whereas edges execution times are in the order of dozens of cycles, and most of the time edges execute in parallel with actors. Therefore, the edges execution has only a slight impact on the overall execution of the application. This indicates that the worst-case latency can be chosen as a safe bet and also helps to cope with the worst-case behavior at runtime.

B. Trace Storage Optimization

We store the traces obtained in the previous step in an optimized way so that the storage overhead is reduced. For multiple executions of an actor a on a core t within a periodic execution of the application, the start and end time of the first execution ($ST_{1^{st}Fire}$ and $ET_{1^{st}Fire}$) and an addition factor (AF) is stored to get the start and end time of further executions within the first periodic execution. The total number of executions ($nrExecs$) of a on t within a period is also stored to keep a track of the overall executions on t. The addition factor (AF) is computed as follows:

$$AF = ExecTime(a) + Delay(e_{outgoing-from-a}) \quad (8)$$

where *ExecTime(a)* is obtained as ($ET_{1^{st}Fire} - ST_{1^{st}Fire}$) and delay of outgoing edge of a ($Delay(e_{outgoing-from-a})$) is obtained from the execution traces. In case of multiple outgoing edges, the edge having maximum execution time is considered. For all the executions of an edge, the latency remains constant as dedicated connections are used.

For k^{th} firing of the actor a, the start time $ST_{k^{th}Fire}$ and end time $ET_{k^{th}Fire}$ are calculated as follows:

$$\begin{aligned} ST_{k^{th}Fire} &= ST_{1^{st}Fire} + (k-1) \times AF \\ ET_{k^{th}Fire} &= ST_{k^{th}Fire} + ExecTime(a) \end{aligned} \quad (9)$$

It is evident that $ST_{1^{st}Fire}$, $ET_{1^{st}Fire}$, AF and $nrExecs$ of an actor a are sufficient to construct the remaining executions of a within the first periodic execution of the application, e.g. remaining executions of actor IQ of H.263 decoder shown in Fig. 3(a) can be constructed for the first period (from 0 to P1). In order to construct the execution traces of an actor over different periodic executions of the application, period P of the application also needs to be stored. For example, for second periodic execution (from P1 to 2P1) of H.263 decoder in Fig. 3(a), $ST_{1^{st}Fire}$ and $ET_{1^{st}Fire}$ of actor IQ are constructed by adding period $P1$ to the respective stored values of $ST_{1^{st}Fire}$ and $ET_{1^{st}Fire}$ for first periodic execution (from 0 to P1), and the start time and end times for the remaining executions (nrExecs-1) of IQ are achieved by employing Equation 9. Therefore, $ST_{1^{st}Fire}$, $ET_{1^{st}Fire}$, AF and $nrExecs$ of each actor of the application needs to be stored along with the application period P to construct the whole execution trace.

Such storing shows significant reduction in the storage overhead. We demonstrate trace storage optimization for the multiple executions of actor IQ on core 2 for the H.263 decoder as shown in Fig. 3(a). Actor IQ executes 2376 times on core 2 in practice. For 2376 executions on core 2, we need to store only $ST_{1^{st}Fire}$ and $ET_{1^{st}Fire}$ from traces, computed AF and $nrExecs$ as 2376. These information are sufficient to calculate ST and ET of any firing, and overall used time interval by IQ on core 2.

It should be noted that for the cases when execution pattern on a core is not consistent (uniform), i.e. executions are not always separated by a fixed time interval, the start time and end time of all the executions are stored for one periodic execution. The period of the application is also stored to construct the execution traces in different periodic executions as described earlier. Therefore, in such cases, trace storage optimization cannot be performed for the executions within a period.

C. Run-time Trace Analysis to Identify Mapping

In order to map applications at run-time, the platform manager is invoked to find a mapping. The manager takes the execution traces of the applications to be supported on the platform and identifies the resource and throughput optimized mapping based on the number of available cores in the platform in order to satisfy the throughput requirements. These active applications form a use-case. The platform manager is

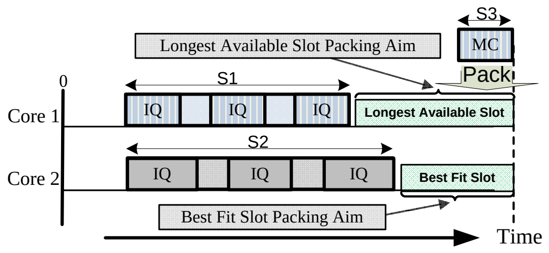


Fig. 5: Slots formation and aims of proposed approaches to pack the next slot.

invoked to find a mapping whenever the use-case, i.e. the set of active applications to be supported at run-time changes. The manager adopts a trace analysis strategy to rapidly identify the mapping. Our strategy tries to maximize throughput by allowing minimum throughput degradation while packing the execution traces of applications on a given number of available cores. The executions are tried to be packed as soon as possible so that there is minimum stretching in the overall execution trace, resulting in minimum throughput degradation.

The proposed trace analysis strategies utilize used (filled) and unused (empty) slots of cores, where *used slots* are formed by combining multiple adjacent executions (firings) of the same actor if the gap between adjacent executions is less than smallest execution time amongst all the actors. The start and end time of a slot Sn (ST_{Sn} and ET_{Sn}) are determined as follows:

$$\begin{aligned} ST_{Sn} &= ST_{FirstFire} \\ ET_{Sn} &= ET_{LastFire} \end{aligned} \tag{10}$$

provided the smallest execution cannot fit in between any of the adjacent executions between the first and last firing. The idle periods define unused slots. The communication overhead between actors is considered by assigning start time of each slot after the dependent actor/edge executions. Fig. 5 shows the formed used slots of actors IQ of H.263 and JPEG decoder when mapped on two separate cores *core 1* and *core 2* (executions taken from Fig. 3 (a) and (b)). The formed slots are $S1$ and $S2$, which contain 3 adjacent executions. An actor may fire a large number of times, e.g., IQ of H.263 decoder fires 2376 times in practice, therefore, forming a slot of 2376 firings will facilitate for coarse-grained (block level) processing, resulting in faster computations when compared to the firing level (fine-grained) processing.

In order to pack a used slot (e.g., $S3$) of a core on an unused slot from various unused slots of different cores, we proposed two strategies. The first strategy aims to choose *longest available (unused) slot* as shown in Fig. 5. For a used slot to be packed (S_{pack}), the available capacity AC (in cycles) of each core is determined as follows:

$$AC_{core} \;=\; ET_{S_{pack}} - ET_{S_{LastPackedOn\_core}} \tag{11}$$

where ET represents the end time. For example, in Fig. 5, available capacity of *core 1* to pack $S3$ is the difference of ET_{S3} and ET_{S1}. If there is no earlier packed slot on the core, the value of $ET_{S_{LastPackedOn\_core}}$ is zero. The second strategy aims to choose the *smallest possible slot, i.e. the best fit slot*.

Algorithm 1: Longest Available Slot Packing

```
Input: Apps, thrConstraints, ExecTraces of Apps using $k$ cores,
       available cores $p$.
Output: efficient mapping using a maximum of $p$ cores.
for each application app do
    $Papp_{constraint} = \frac{1}{app_{thrConstraint}}$;
end
Find Least Common Multiple $LCM$ of periodic constraints
($Papp_{constraint}$) of all Apps;
for each application app do
    $nrOfExecution[app] = \frac{LCM}{Papp_{constraint}}$;
end
Construct execution traces of each application based on
$nrOfExecution$;
Construct Used Slots of $k$ cores;
Sort all slots in ascending order based on their ET;
//pack slots of $k$ cores to $p$ cores
nrEmptyCore = p;
for each slot $s$ do
    if packing of actor in $s$ is not fixed then
        Calculate available capacities ACs of all the cores ($\in$
        p) by Equation 11;
        $[UsedCores, AC_{UsedCores}] = Filter(p, ACs)$;
        $HAC_{UsedCore} =$
        FindMax($UsedCores, AC_{UsedCores}$);
        if ((s==0 || (($D_s$>$HAC_{UsedCore}$ || $ET_s$ > ST of an
        unpacked slot containing a packed actor) &&
        (nrEmptyCore>0)) then
            Pack $s$ to an available core $avail_{core} \in$ p;
            Fix packing of actor in $s$ to $avail_{core}$;
            $nrEmptyCore - -$;
        else
            Pack $s$ to $UsedCore$;
            Fix packing of actor in $s$ to $UsedCore$;
        end
    else
        Pack $s$ to earlier fixed core (for the actor in $s$);
    end
end
```

*1) **Longest Available Slot Packing (LASP)**:* The LASP trace analysis strategy utilizes **longest available slot** on a core to pack a new slot (Fig. 5). The LASP strategy is presented in Algorithm 1. The algorithm takes run-time active applications (Apps), their throughput constraints and execution traces, and number of available cores (p) as input, and provides the efficient mapping using a maximum of p cores. For each active application, first, the periodic constraint ($Papp_{constraint}$) is found, which is the inverse of application throughput constraint. Then, the least common multiple (LCM) of periodic constraints of all the active applications is identified. This helps to identify exact number of periodic executions of each active application ($nrOfExecution[app]$) within LCM. For example, in Fig. 3, the number of periodic executions of H.263 & JPEG decoder within LCM is 2 & 1 respectively. Thereafter, for the found $nrOfExecution$, execution traces of each application are constructed. In order to perform block level processing, the execution traces are used to construct used slots as described earlier. The slots are then sorted in ascending order based on their end time (ET) in order to facilitate earlier packing of early finished slots so that the executions are minimally delayed.

The algorithm packs all the sorted slots of k cores to p available cores. The packing of slots follows the steps described in Algorithm 1. The first slot is packed on a completely free core ($avail_{core}$) and the packing of actor in the slot is fixed to the same core. Similar fixing is applied to actors in other slots as well. Such fixing ensures the packing of all the executions of an actor to the same core and thus avoids overhead of actor migration to a different core during various executions of the same actor. The remaining slots are packed on a used or completely free core depending upon the available capacities of the used cores ($AC_{UsedCores}$), already packed actor, and number of completely free cores ($nrEmptyCore$). If no completely free core is left, the slot is packed on a used core ($UsedCore$) having highest available capacity ($HAC_{UsedCore}$), which is selected amongst the used cores ($UsedCores$) by using function **FindMax()**. While packing a slot, its communication dependencies are taken into account by assigning the start time of the slot only after the execution of the longest time consuming edge connecting with an actor contained in the already packed slots. In case a completely free core is available and the slot interval (Ds) is greater than $HAC_{UsedCore}$, the slot is packed to the free core ($avail_{core}$). The value of Ds is calculated as follows:

$$D_s = ET_s - ST_s \tag{12}$$

where ET_s and ST_s are the end and start time of the slot.

LASP Demonstration: The LASP strategy has been demonstrated to rapidly identify the mapping for actors of H.263 decoder and JPEG decoder on 4 cores. The demonstration considers execution traces of H.263 decoder and JPEG decoder as shown in Fig. 6(a) and (b), respectively. The resulting mapping after applying LASP is presented in Fig. 6(c). The intra-core and some other communications are not shown to have a better representation of the main message. In order to identify the mapping, first, slots (S1 to S14) are formed as shown in Fig. 6(a) and (b), and then sorted based on their end time in the ascending order. The slot numbering S1, ..., S14 is in the required sorted fashion. The packing of slots by LASP considers Highest Available Capacity (HAC) core, i.e. Longest Available Slot (LAS) to pack most of the slots. For example, LAS for packing slots S3 and S7 is on Core 1, and on Core 4 for S14 as shown in Fig. 6(c). The mapping of actors on cores is computed by extracting all the unique actors in the packed slots on the cores.

Constraints checking and stalls installation: The throughput of active applications are estimated after all the slots are packed on p available cores (e.g., on 4 cores from 10 cores as shown in Fig. 6). For each application App, throughput is estimated as follows:

$$thr_{App} = \frac{nrOfExecution[app]}{ET_{AppLastSlot}} \tag{13}$$

The provided packing (mapping) satisfies throughput constraint for all the applications if the estimated throughput values are greater than or equal to the respective throughput constraints. The executions of applications are repeated in further periodic executions, where the overall period is considered as the longest obtained period (i.e., smallest estimated throughput *thrSmallest*) amongst all the applications. For applications finishing their executions earlier than the longest obtained period, the executions are started by introducing appropriate delays (stalls) so that periodicity of executions are maintained. For each application, the delays are calculated as follows:

$$Delay_{App} = \frac{1}{thr_{App} - thrSmallest} \tag{14}$$

These delays (stalls) might affect the overall application throughput. However, they are required to ensure the periodicity of the applications. It has been observed that the overall application throughput with and without stalls differs by a minimal amount, which has been evaluated in the next section. Thus, stalls can be introduced to ensure the periodicity.

It should be noted that we have considered worst-case execution times (WCETs) of tasks and they remain fixed. This also ensures to model the worst-case run-time behavior of the application. During run-time execution, we enforce start times of tasks based on the respective WCETs and the intended delays to start the execution. This provides us an easy way to ensure the periodicity of the executions. However, in case a task's execution time varies at run-time and is less than the WCET, the created slacks can be utilized by employing the dynamic voltage and frequency scaling (DVFS) techniques to achieve energy savings [43], [44], but our focus is not to exploit DVFS potentials and our approach is orthogonal to DVFS.

Advantages: The LASP strategy maps executions to p available cores by giving more priority to already used cores ($\in p$). Therefore,*resource utilization of the used cores increases.* For example, Fig. 6(c) shows that the utilization of cores is increased when compared to the utilization with original mappings (Fig. 6(a) and (b)). Additionally, the strategy packs the slots as soon as possible (ASAP) in order to facilitate earlier packing of early finished slots, resulting in minimal delays in the overall execution. Further, the strategy may pack slots (executions) on lower number of cores than the available ones while achieving the same throughput, e.g., executions of H.263 and JPEG decoder shown in Fig. 6(a) and (b)) can get packed on 6 cores even if 8 cores are available. Therefore, savings in energy consumption as well can be achieved by applying power gating to the unused cores.

The **complexity** of LASP depends upon the number of formed slots n_s that need to be packed on p available cores. For a given value of n_s and p, the worst-case complexity (C) is calculated as the maximum number of operations that need to performed for all the slots. A maximum of $n_s \times p$ operations are possible as p cores need to be evaluated for each slot. Therefore, the complexity of the trace analysis is O($n_s p$). The timing overheads for various application scenarios are provided in next Section (Section VI-E.)

*2) **Best Fit Slot Packing (BFSP)**:* The BFSP strategy follows similar steps as that of LASP, but uses the best fit slot instead of the slot having highest available capacity. Therefore, instead of calling function **FindMax()** within Algorithm 1, it calls function **FindBestFit()**, which is described in Algorithm 2. The function scans from minimum to maximum available

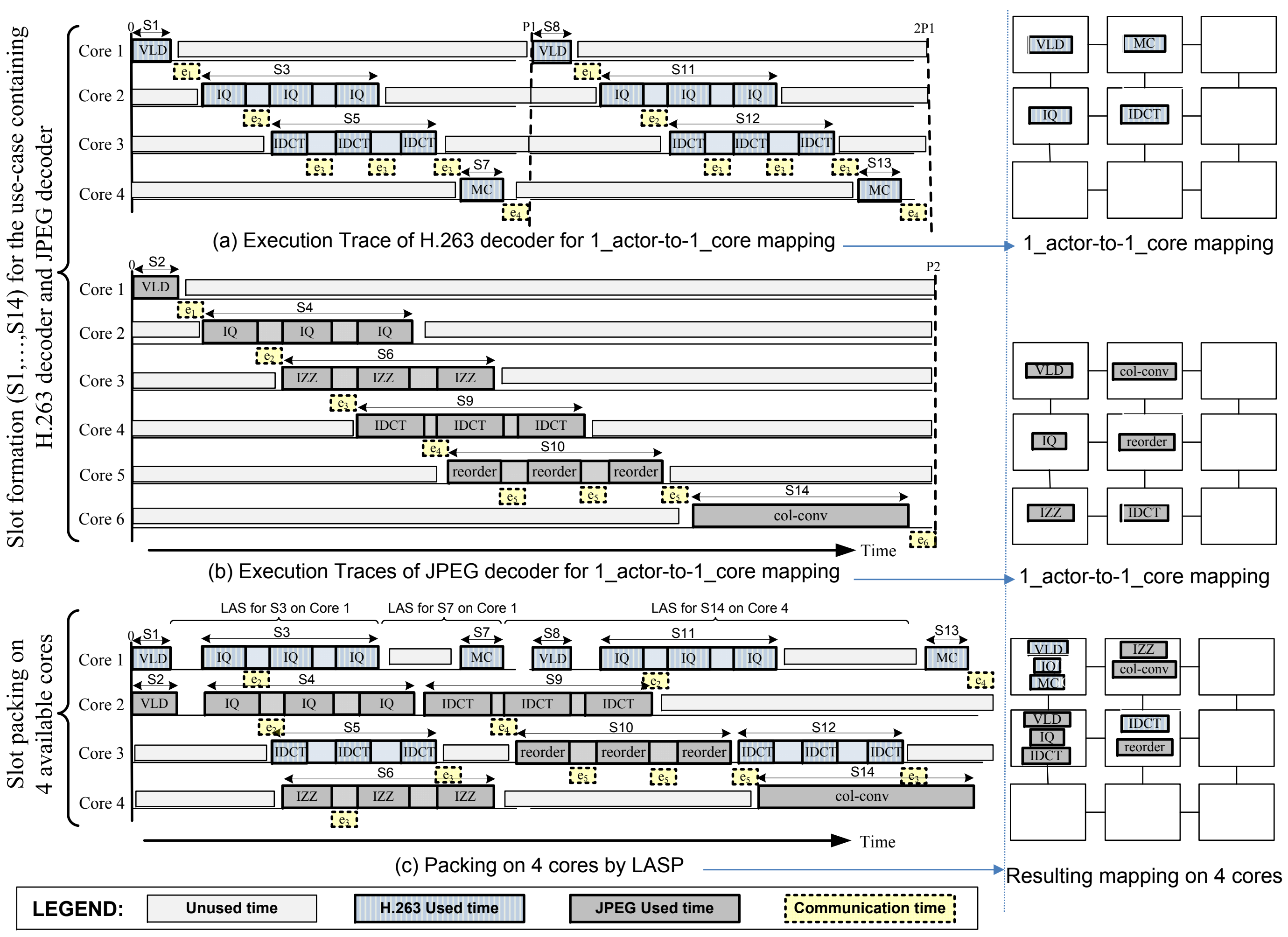

Slot formation (S1,…,S14) for the use-case containing H.263 decoder and JPEG decoder
(a) Execution Trace of H.263 decoder for 1_actor-to-1_core mapping
1_actor-to-1_core mapping
(b) Execution Traces of JPEG decoder for 1_actor-to-1_core mapping
1_actor-to-1_core mapping
Slot packing on 4 available cores
LAS for S3 on Core 1
LAS for S7 on Core 1
LAS for S14 on Core 4
(c) Packing on 4 cores by LASP
Resulting mapping on 4 cores
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
VLD
IQ
IDCT
MC
IZZ
reorder
col-conv
Time
LEGEND:
Unused time
H.263 Used time
JPEG Used time
Communication time

Fig. 6: Slot formation for the execution traces of H.263 and JPEG decoder, and resulting mapping (packing) after applying LASP.

Algorithm 2: FindBestFit()

```
Input: $UsedCores$, $AC_{UsedCores}$.
Output: Best fit core $BF_{core}$.
Sort $UsedCores$ based on $AC_{UsedCores}$ in ascending order;
for each core c (∈ $UsedCores$) do
    if $D_s \leq AC_c$ then
        $BF_{core} = c$;
        break;
    end
end
```

capacity cores and terminates as soon as a core that can provide a fit is found.

BFSP Demonstration: The BFSP demonstration considers the same execution traces (H.263 decoder and JPEG decoder) as shown in Fig. 6(a) and (b), and tries to identify a mapping for the same number of available cores (4 cores) as that of LASP demonstration. The resulting mapping after applying BFSP is presented in Fig. 7. Similar to LASP, the intra-core and some other communications are not shown to have a better representation of the main message. In the resulting packing, the Best Fit Slot (BFS) to pack slots S3, S7 and S14 are available on core 2, core 3 and core 1, respectively, as shown in Fig. 7. The best fit slots are the ones that can accommodate a slot by leaving minimum possible unused time on a core. The mapping of actors on cores is computed in the similar manner as in LASP.

The BFSP also provides similar advantages as that of LASP. However, as it chooses the best fit slot, the higher available slots are left that might get used by other unpacked slots. Therefore, in some suitable cases, it might provide better packing than LASP and the same has been demonstrated in the next section. However, if the longest available slots are left unused, a lower load balancing will be achieved. The computational complexity of BFSP is similar to LASP.

VI. Experimental Results

The proposed approach has been implemented as an extension of the publicly available SDF^3 tool set [41]. Experiments are performed on a Quad Core processor at 2.4 GHz. As a benchmark to evaluate the quality of the approach, models of real-life streaming multimedia applications H.264 decoder (3 actors), H.264 encoder (9 actors), H.263 decoder (4 actors), H.263 encoder (5 actors), MPEG-4 decoder (5 actors), JPEG decoder (6 actors), and MP3 decoder (14 actors) have been considered to form various use-cases. For each application,

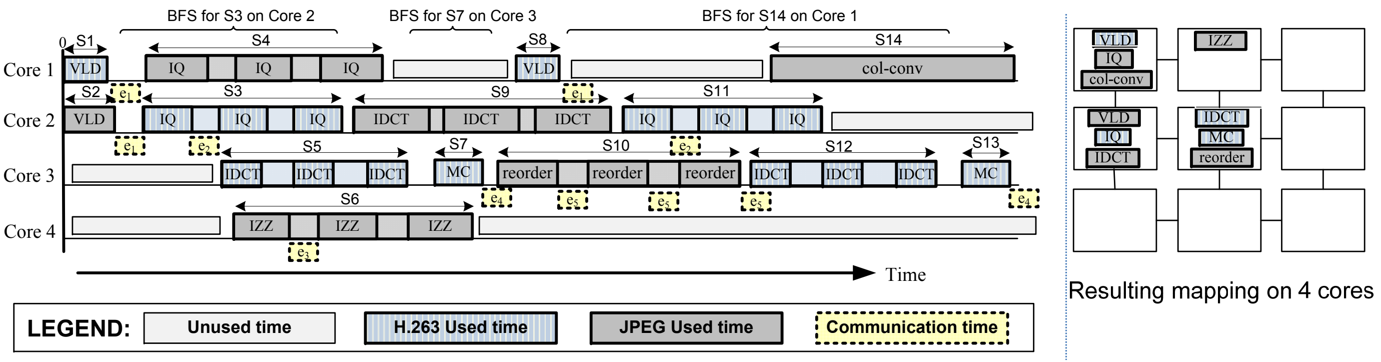


Fig. 7: Resulting mapping (packing) after applying BFSP.

TABLE I: Approaches considered for comparison

Approaches	Abbreviation	References
Simulation-based Exhaustive DSE	EDSE	[10]
Simulation-based Pruned DSE	PDSE	[12]
Simulation and Analysis based DSE	SADSE	[23], [24]
Load Balancing	LB	[26]
On-the-fly Nearest Neighbor	NN	[4]
On-the-fly Communication-aware Nearest Neighbor	CNN	[5]
Run-time Mapping utilizing DSE results	HybridMap	[8]–[12], [28]
Longest Available Slot Packing	LASP	Proposed
Best Fit Slot Packing	BFSP	Proposed

its throughput requirement and other details such as execution times of actors and input/output tokens on edges are specified in the application model. To support the mapping of a use-case by existing approaches, its application models are merged into one application graph and their execution rates are specified. The target platform contains homogeneous ARM7TDMI cores.

We present results obtained from our approach to find the efficient mapping solution and compare them with various existing approaches reported in the literature, which are abbreviated in Table I. The EDSE flow evaluates all possible mappings for a given number of available cores, whereas PDSE prunes the mapping space. The EDSE and PDSE employ simulations to evaluate the mappings in order to compute their throughput. SADSE employs simulations and estimations to accelerate the evaluation process. The estimations are performed to identify the efficient mapping in terms of throughput and then simulation is employed on the mapping to achieve accurate results. The LB identifies a mapping such that loads on available cores are balanced and simulation is employed to evaluate the mapping. The on-the-fly heuristics find a mapping at run-time and then throughput of the mapping is computed. The NN strategy tries to map the communicating actors on the neighboring cores, whereas CNN strategy tries to map the maximum communicating pairs of actors on the same core. In HybridMap, the DSE is performed in advance and the DSE results are stored to be used at run-time towards facilitating efficient mapping. The LASP and BFSP identify the efficient mapping by employing proposed estimation methods.

A. Speed-up and Throughput

Fig. 8 shows speed-up of various exploration approaches when normalized with respect to (w.r.t.) EDSE at different

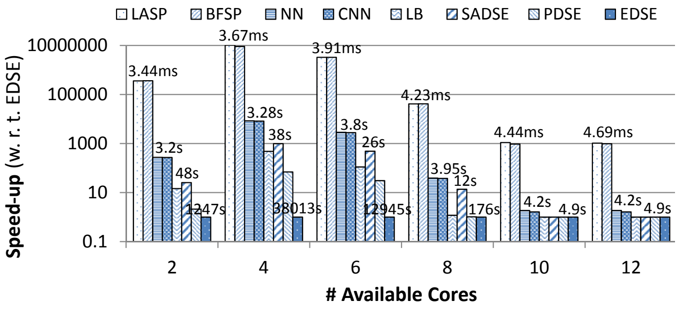


Fig. 8: Speed-up and exploration time of various approaches w.r.t. EDSE.

number of available cores for the use-case containing H.263 decoder and encoder. The figure also indicates the exploration time for EDSE, SADSE, CNN and LASP. The exploration time of the approaches highly depends upon the number of mappings to be simulated. For each mapping, evaluation time (contributing to the overall exploration time) includes time to find the mapping and its throughput computation. The EDSE cannot finish exploration within a limited time in some cases as a huge number of mappings need to be evaluated, and exploration time for such cases is calculated based on the evaluation time for one mapping and the total number of mappings. The number of evaluated mappings by PDSE for a use-case containing n actors is computed as follows.

$$\begin{aligned} PDSE(n,p) &= 1 +^{n} C_2 +^{(n-1)} C_2 +^{(n-2)} C_2 + \\ &\quad ^{(n-3)}C_2 + ... +^{(p+1)} C_2 \\ &= 1 + \sum_{k=1}^{n-1} (^{(k+1)}C_2) - \sum_{k=1}^{p-1} (^{(k+1)}C_2) \end{aligned} \tag{15}$$

where first a mapping using n cores is evaluated (1 mapping), then the mappings using $(n-1)$ cores ($^{n}C_2$ mappings) and so on until the mappings using p cores ($^{(p+1)}C_2$) are evaluated. The approach selects the best mapping (in terms of throughput) at m cores to evaluate mappings using $(m-1)$ cores. Table II shows the number of mappings to be evaluated at varying number of available cores (2 to 12) by EDSE and PDSE, which are computed by Equation 6 and 15, respectively, for the H.263 decoder/encoder use-case (i.e., n = 9).

The number of evaluated mappings beyond n cores is one as the use-case will use a maximum of n cores. Therefore, for higher number of available cores, the exploration strategies

TABLE II: Number of Mappings for 9 actors

	Number of available Cores					
Approaches	2	4	6	8	10	12
EDSE	255	7770	2646	36	1	1
PDSE	120	111	86	36	1	1

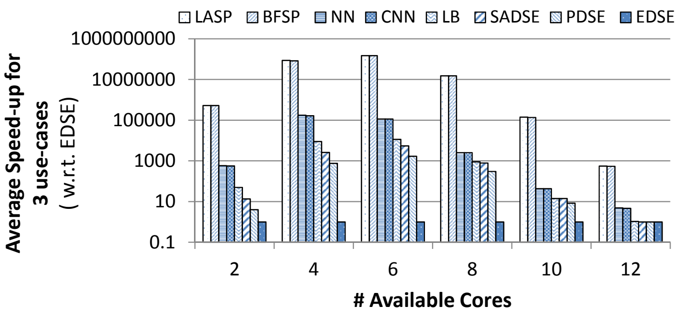


Fig. 9: Average speed-up over different use-cases w.r.t. EDSE.

take almost the same time as those in n cores, as shown in Fig. 8. It can also be observed that LASP and BFSP perform very fast exploration to identify a mapping along with its throughput estimation. Further, the speed-up of LASP is higher than that of BFSP because BFSP performs some additional computations to find the best slot. On an average, LASP achieves a speed-up of 4115x when compared to the SADSE, and 1021x compared to on-the-fly CNN approach. Fig. 9 shows average speed-up at different number of available cores over 3 use-cases: H.263 decoder/encoder, JPEG/MPEG decoder and H.264 decoder/encoder. It can be observed that our approach achieves significant speed-up when compared to existing approaches.

Fig. 10 shows time to find the best possible mapping (in milliseconds) by different on-the-fly processing approaches at varying number of available cores. In contrast to earlier timing results, the shown times for CNN and NN do not include throughput computation time for the found mapping. Differently from CNN and NN, our approaches HASP and BFSP identify the mapping along with it's throughput estimation. These approaches find just one mapping based on the available resources by taking quick online decisions. Therefore, the timing results will always be better than other approaches that find multiple mappings, e.g., EDSE, PDSE and SADSE. Further, these approaches take less time than LB as LB tries for several allocation options to find a load balanced mapping. On an average, LASP provides a speed-up of 14× and 16× when compared to NN and CNN, respectively.

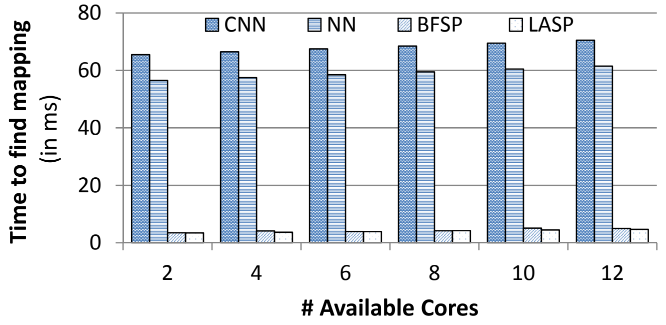


Fig. 10: Time to find mapping.

TABLE III: Best mapping throughput ($\times$ 10^{-10}/time-units)

	Number of available Cores						
Approaches	1	2	3	4	5	6	7
PDSE/SADSE	1916	2643	4636	4649	4692	7901	7901
LB	1916	1848	1854	1868	1868	1868	1868
NN	1916	2549	3212	4319	4456	6308	6656
CNN	1916	2595	3723	4339	4540	6372	6940
LASP	1916	2643	4626	4649	4672	7783	7783
BFSP	1916	2643	4629	4649	4692	7783	7783

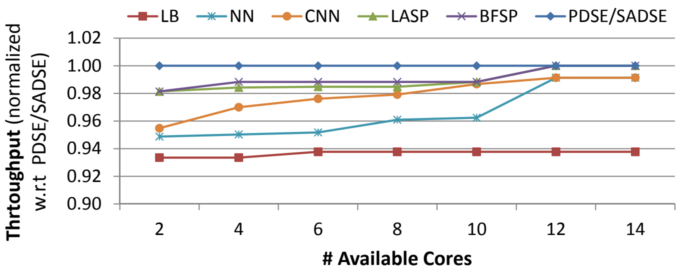


Fig. 11: Throughput comparison at different number of available cores for MP3 decoder.

Table III shows the throughput values of the efficient (highest throughput) mapping for different number of available cores for H.263 decoder/encoder use-case when various approaches are employed. Similar results are obtained for other use-cases. A couple of observations can be made from Table III. *1)* Throughput obtained by approaches employing simulations (e.g., PDSE and SADSE) is the same. *2)* For few cases, BFSP achieves better throughput than LASP due to better slot packing. *3)* The throughput obtained by proposed estimation approaches LASP and BFSP are close to that of the approaches employing simulations. On an average, the throughput values deviate by 0.77% for LASP and 0.63% for BFSP when compared to simulation based approaches. This deviation is obtained due to slight error in the estimations. Therefore, estimated throughput values are close to the accurate values.

Fig. 11 shows the throughput values of the efficient mapping at different number of available cores when various approaches are employed for the use-case containing MP3 decoder. The throughput values are normalized w.r.t the approaches PDSE/SADSE. It can be observed that at different number of available cores, our fast approaches (LASP and BFSP) achieve almost the same quality of mapping as that of existing approaches that employ simulations, whereas other approaches lead to inferior results due to inefficient actors to cores allocation decisions. The slight deviation in the quality (throughput) by our approaches w.r.t. PDSE/SADSE is obtained mainly due to negligible error in the estimations. Therefore, estimated throughput values are close to the accurate values.

B. Storage Overhead

Table IV shows storage requirement by different applications when various approaches are employed. The approaches EDSE, PDSE, SADSE and HybridMap store the same number of mappings to be used at run-time. In contrast, our approaches LASP and BFSP store execution traces of the applications. For our approach, the table shows storage overhead without (No-optimiz.) and with (Optimiz.) trace storage optimization. On an average, our approaches reduce the storage requirement

TABLE IV: Storage overhead (kB) of different approaches

# Apps	# use-cases	# mappings	Storage Requirement: EDSE/ PDSE/ SADSE/ HybridMap	LASP/BFSP No-optimiz.	LASP/BFSP Optimiz.
3	7	72	3.4	5.8	3.2
4	15	184	10.5	9.5	5.7
5	31	432	27.8	12.4	6.2
6	63	1024	75.7	15.9	9.5
7	127	2944	313.8	16.4	11.5

by 92% when compared to existing approaches. The number of mappings is computed as the summation of number of required mappings for each use-case that can be formed by the applications. For a use-case containing n actors, a total of n mappings using 1 core to n cores are required in order to handle the dynamism in resource availability. For m applications, a total of $(2^m - 1)$ use-cases are possible. For 3 applications, JPEG, H.264 decoder and H.264 encoder are considered. For 4^{th}, 5^{th}, 6^{th} and 7^{th} applications, additional applications MPEG, H.263 decoder, H.263 encoder and MP3 are considered. A couple of observations can be made from Table IV. *1)* LASP and BFSP require less storage space when compared to existing approaches as LASP/BFSP store execution traces of only the active applications, whereas other approaches store all the mappings for different possible use-cases that are possible by varying combination of the active applications. *2)* For lower number of applications, storage requirement of existing and proposed approaches are almost the same as the existing approaches need to store a small number of mappings, whereas the storage requirement differs significantly for large number of applications. *3)* Our trace storage optimization process shows further reduction in the storage requirement.

C. Energy and Resource Savings

The total energy consumption is computed as the sum of dynamic and static energy by employing the energy consumption model introduced in Section III-B. Fig. 12 shows energy consumption and number of used cores for different number of available cores when PDSE and LASP are employed. The results shown are for a use-case containing H.264 decoder and H.264 encoder. The approach PDSE provides better results than other approaches such as LB and NN, similar to the throughput results as shown in Table III and Fig. 11. Therefore, the energy and resource savings results obtained by other inferior approaches than that of PDSE are not reported and only PDSE has been considered to compare with our proposed approaches. On an average, LASP reduces energy consumption and resources used by 5% and 15% respectively when compared to PDSE. The energy consumption and resource usage (number of used cores) by PDSE and LASP are almost the same for lower number of available cores as both the approaches achieve similar mappings that use all the available cores. For higher number of available cores, LASP uses less number of cores than PDSE. The PDSE uses a maximum of 12 cores that is the same as the number of actors in the use-case, whereas LASP uses a maximum of 9 cores as it tries to use minimum number of cores while providing the same throughput. Moreover, LASP consumes lower energy than PDSE for higher number of available cores as LASP uses lesser number of cores than PDSE.

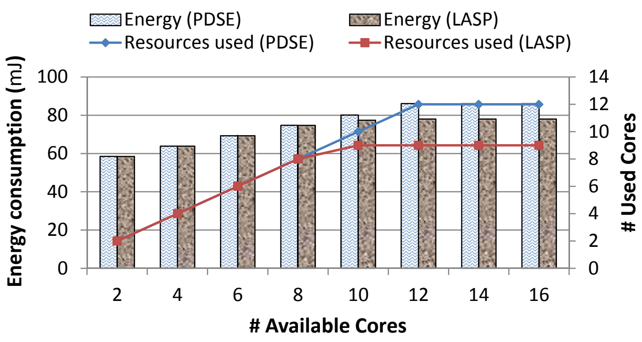


Fig. 12: Energy consumption and resource usage.

TABLE V: Stall installation penalty (%) in various use-cases containing different combinations of decoder (de.) and encoder (en.)

Use-case1: H.263 de.	Use-case1: H.263 en.	Use-case2: JPEG de.	Use-case2: MPEG de.	Use-case3: H.264 de.	Use-case3: H.264 en.
1.40%	0.00%	2.20%	0.00%	3.05%	0.00%

D. Stalls Installation Penalty

The stalls introduced to ensure the periodicity of the applications might degrade their overall throughput. In order to evaluate the throughput degradation, we have computed stalls installation penalty as the percentage deviation in the achieved throughput without and with stalls. Table V shows the stall installation penalty for different applications considered in various use-cases when LASP is employed. For each use-case, the penalty has been averaged over different number of used cores to support the use-case. It can be observed that there is no penalty for one of the applications in each use-case. The reason being that this application is minimum throughput (longest period) one and its periodicity is automatically ensured. The periodicity of the other application that finishes earlier than the longest period is maintained by introducing appropriate delays (stalls) computed by Equation 14. For such applications, the overall throughput with and without stalls differs by a very minimal amount (e.g., 1.40% for H.263 decoder), as shown in Table V. On an average, the throughput differs by only 2.21%. It can also be observed that through degradation by stalls installation is maximum for H.264 decoder as large delay needs to be introduced to ensure the periodicity with H.264 encoder. Since throughput degradation is quite small and our approach provides several other advantages (e.g., low exploration time, storage overhead and energy consumption), the small stalls installation penalty can be compensated.

E. Run-time Suitability

Table VI shows the time required (in milliseconds) to map various use-cases on 4 available cores when HybridMap and LASP are employed. For run-time suitability, the HybridMap approach seems to be the most suitable existing approach as it identifies a high quality mapping rapidly by utilizing the design-time DSE explored results. Further, the HybridMap approach provides better quality mapping and takes lesser time to find the mapping than other run-time mapping approaches such as NN and CNN that try to identify a mapping without any

TABLE VI: Time required (in ms) to map various use-cases

Use-case	HybridMap			LASP		
Applications	Select	Config.	Total	Explor.	Config.	Total
H.263 decoder H.263 encoder	0.15	8.00	8.15	3.67	8.00	11.67
JPEG decoder MPEG decoder	0.17	8.00	8.17	5.13	8.00	13.13
H.264 decoder H.264 encoder	0.18	8.01	8.19	5.36	8.01	13.37

prior analysis. Therefore, it has been considered to compare with our proposed run-time approach.

The total time to map a use-case consists of time to find the actors to cores allocation and then accordingly configure the platform. In HybridMap, the efficient mappings using different number of cores are computed at design-time and stored for using at run-time. Thus, mapping is not computed at run-time, but selected from the storage. Therefore, for HybridMap, the time required for selection and configuration contribute to the total time, whereas time required for exploration and configuration contribute to the total time of LASP. The HybridMap takes lesser time for mapping than that of LASP and thus can guarantee for schedulability (fast mapping). However, HybridMap requires large storage to keep the various mappings and large DSE time (shown earlier). Our approach overcomes above issues while taking similar orders of mapping time as that of HybridMap. Additionally, our approach provides high quality mapping and takes much less time than on-the-fly approaches ([4]–[7]), which perform all the computations (e.g., mapping and throughput computations) at run-time, as shown in Section VI-A. Therefore, the issues of HybridMap and on-the-fly approaches are overcome by our approach, making it very suitable for efficient run-time management.

VII. Conclusions

We present a novel run-time trace analysis strategy to rapidly identify the maximum throughput mapping to support a use-case while optimizing for throughput and resource usage. The strategy analyzes execution traces of use-case applications to facilitate rapid identification. Experiments show that storage overhead and energy consumption are also reduced in addition to the exploration time while providing near-optimal mapping solutions. In future, we plan to extend/explore the approach for heterogeneous MPSoCs in order to identify the premier mapping for a variety of available cores.

References

[1] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, "Mapping of applications to MPSoCs," in *Proceedings of IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2011, pp. 109–118.

[2] R. Marculescu, U. Ogras, L.-S. Peh, N. Jerger, and Y. Hoskote, "Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 1, pp. 3–21, 2009.

[3] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends," in *Proceedings of ACM Design Automation Conference (DAC)*, 2013, pp. 1:1–1:10.

[4] E. L. d. S. Carvalho, N. L. V. Calazans, and F. G. Moraes, "Dynamic task mapping for MPSoCs," *IEEE Des. Test of Comp.*, vol. 27, no. 5, pp. 26–35, 2010.

[5] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms," *Elsevier Journal of Systems Architecture (JSA)*, vol. 56, pp. 242–255, 2010.

[6] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 16, pp. 24–33, 2008.

[7] O. Moreira, J. J.-D. Mol, and M. Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, 2007, pp. 1557–1564.

[8] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 692 –707, 2010.

[9] C. Ykman-Couvreur, P. A. Hartmann, G. Palermo, F. Colas-Bigey, and L. San, "Run-time resource management based on design space exploration," in *Proceedings of IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2012, pp. 557–566.

[10] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems," in *Proceedings of IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2002, pp. 112–119.

[11] C.-L. Chou and R. Marculescu, "Designing heterogeneous embedded network-on-chip platforms with users in mind," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 29, no. 9, pp. 1301–1314, 2010.

[12] A. K. Singh, A. Kumar, and T. Srikanthan, "Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 1, pp. 9:1–9:29, 2013.

[13] C. Lee, S. Kim, and S. Ha, "Efficient Run-time Resource Management of a Manycore Accelerator for Stream-based Applications," in *Proceedings of IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2013, pp. 51–60.

[14] L. Benini, D. Bertozzi, and M. Milano, "Resource Management Policy Handling Multiple Use-Cases in MPSoC Platforms Using Constraint Programming," in *Proceedings of International Conference on Logic Programming (ICLP)*, 2008, pp. 470–484.

[15] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, "An industrial design space exploration framework for supporting run-time resource management on multi-core systems," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2010, pp. 196–201.

[16] B. Giovanni, L. Fossati, and D. Sciuto, "Decision-theoretic design space exploration of multiprocessor platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 29, pp. 1083–1095, 2010.

[17] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini, "An Integrated Open Framework for Heterogeneous MPSoC Design Space Exploration," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, vol. 1, 2006, pp. 1 –6.

[18] Z. J. Jia, A. Pimentel, M. Thompson, T. Bautista, and A. Nunez, "NASA: A generic infrastructure for system-level MP-SoC design space exploration," in *Proceedings of IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2010, pp. 41 –50.

[19] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich, "Efficient symbolic multi-objective design space exploration," in *Proceedings of IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2008, pp. 691–696.

[20] H. Becker and J. Riordan, "The Arithmetic of Bell and Stirling numbers," *American journal of Mathematics*, pp. 385–394, 1948.

[21] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," *SIGPLAN Not.*, vol. 37, no. 7, pp. 18–27, 2002.

[22] J. Kim and M. Orshansky, "Towards formal probabilistic power-performance design space exploration," in *Proceedings of ACM Great Lakes symposium on VLSI (GLSVLSI)*, 2006, pp. 229–234.

[23] R. Piscitelli and A. Pimentel, "Design space pruning through hybrid analysis in system-level design space exploration," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 781 –786.

[24] A. K. Singh, A. Das, and A. Kumar, "RAPIDITAS: RAPId Design-Space-Exploration Incorporating Trace-Based Analysis and Simulation," in *Proceedings of IEEE Euromicro Conference on Digital System Design (DSD)*, 2013, pp. 836–843.
[25] P. van Stralen and A. Pimentel, "Scenario-based design space exploration of MPSoCs," in *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 2010, pp. 305 –312.
[26] S. Stuijk, M. Geilen, and T. Basten, "A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour," in *Proceedings of IEEE Euromicro Conference on Digital System Design (DSD)*, 2010, pp. 548 –555.
[27] G. Palermo, C. Silvano, and V. Zaccaria, "Robust optimization of SoC architectures: A multi-scenario approach," in *Proceedings of IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2008, pp. 7 –12.
[28] W. Quan and A. D. Pimentel, "A Scenario-based Run-time Task Mapping Algorithm for MPSoCs," in *Proceedings of ACM Design Automation Conference (DAC)*, 2013, pp. 131:1–131:6.
[29] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano, "Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2006, pp. 3–8.
[30] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A methodology for mapping multiple use-cases onto networks on chips," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2006, pp. 118–123.
[31] G. Chen, F. Li, S. Son, and M. Kandemir, "Application mapping for chip multiprocessors," in *Proceedings of ACM Design Automation Conference (DAC)*, 2008, pp. 620–625.
[32] C. Marcon, E. Moreno, N. Calazans, and F. Moraes, "Comparison of network-on-chip mapping algorithms targeting low energy consumption," *IET Computers Digital Techniques*, pp. 471 –482, 2008.
[33] H. Javaid and S. Parameswaran, "A design flow for application specific heterogeneous pipelined multiprocessor systems," in *Proceedings of ACM Design Automation Conference (DAC)*, 2009, pp. 250–253.
[34] L. Chen, T. Marconi, and T. Mitra, "Online scheduling for multi-core shared reconfigurable fabric," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 582 –585.
[35] J. Huang, A. Raabe, C. Buckl, and A. Knoll, "A workflow for runtime adaptive task allocation on heterogeneous MPSoCs," in *Proceedings of IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2011, pp. 1 –6.
[36] F. Wang, Y. Chen, C. Nicopoulos, X. Wu, Y. Xie, and N. Vijaykrishnan, "Variation-aware task and communication mapping for mpsoc architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, no. 2, pp. 295 –307, 2011.
[37] K. Goossens, J. Dielissen, and A. Radulescu, "AEthereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Des. Test*, vol. 22, no. 5, pp. 414–421, 2005.
[38] C. Cooper and R. E. Kennedy, "Patterns, Automata, and Stirling Numbers of the Second Kind," *Math. Comput. Educ.*, vol. 26, no. 2, pp. 120–124, 1992.
[39] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," in *Proceedings of IEEE Conference on Application of Concurrency to System Design (ACSD)*, 2006, pp. 25–36.
[40] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, pp. 24–35, 1987.
[41] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Proceedings of IEEE Conference on Application of Concurrency to System Design (ACSD)*, 2006, pp. 276–278.
[42] M. A. Bamakhrama and T. Stefanov, "Managing latency in embedded streaming applications under hard-real-time scheduling," in *Proceedings of IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2012, pp. 83–92.
[43] A. K. Singh, A. Das, and A. Kumar, "Energy Optimization by Exploiting Execution Slacks in Streaming Applications on Multiprocessor Systems," in *Proceedings of ACM Design Automation Conference (DAC)*, 2013, pp. 115:1–115:7.
[44] P.-H. Tseng, P.-C. Hsiu, C.-C. Pan, and T.-W. Kuo, "User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices," in *Proceedings of ACM Design Automation Conference (DAC)*, 2014, pp. 85:1–85:6.

Amit Kumar Singh (M09) received the B.Tech. degree in Electronics Engineering from Indian School of Mines, Dhanbad, India, in 2006. Thereafter, he worked with HCL Technologies, India for year and half before starting his PhD at School of Computer Engineering, Nanyang Technological University (NTU), Singapore, in 2008. He completed his PhD in 2012. From February 2012 to August 2014, he worked with the Department of Electrical and Computer Engineering, National University of Singapore (NUS) as a post-doctoral researcher. Since September 2014, he is working with Department of Computer Science, University of York, UK. His research interests include system level design-time and run-time optimizations of 2D and 3D multi-core systems with focus on performance, energy, temperature, and reliability. He has published over 35 papers in these areas in leading international journals/conferences. Dr. Singh was the receipt of PDP 2015 Best Paper Award, HiPEAC Paper Award, and GLSVLSI 2014 Best Paper Candidate. He is TPC member of IEEE/ACM conferences like ISED and MES, and has served as session chair in conferences like APESER and DATE.

Muhammad Shafique (M11) received the Ph.D. degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in 2011. He is currently a Research Group Leader at the Chair for Embedded Systems, KIT. He has over ten years of research and development experience in power-/performance-efficient embedded systems in leading industrial and research organizations. He holds one U.S. patent. His current research interests include design and architectures for embedded systems with focus on low power, reliability, and adaptivity. Dr. Shafique was the recipient of 2015 ACM/SIGDA Outstanding New Faculty Award, six gold medals, the CODES+ISSS 2011 and 2014 Best Paper Awards, AHS 2011 Best Paper Award, DATE 2008 Best Paper Award, DAC 2014 Designer Track Poster Award, ICCAD 2010 Best Paper Nomination, several HiPEAC Paper Awards, and the Best Master Thesis Award. He is the TPC co-Chair of ESTIMedia 2015 and has served on the TPC of several IEEE/ACM conferences like ICCAD and DATE.

Akash Kumar (M05-SM13) received the B.Eng. degree in computer engineering from the National University of Singapore (NUS), Singapore, in 2002. He received the joint Master of Technological Design degree in embedded systems from NUS and the Eindhoven University of Technology (Tue), Eindhoven, The Netherlands, in 2004, and received the joint Ph.D. degree in electrical engineering in the area of embedded systems from TUe and NUS, in 2009. Since 2009, he has been with the Department of Electrical and Computer Engineering, NUS. Currently, he is an Assistant Professor in the department. His research interests include design, analysis and resource management of low-power and fault-tolerant embedded multiprocessor systems. He has published over 80 papers in leading international electronic design automation journals and conferences on these topics. He is also a member of technical program committees of major conferences in the design automation area like, DAC, DATE, ASPDAC, etc.

Jörg Henkel (M95-SM01-F15) is currently with the Karlsruhe Institute of Technology (KIT), Germany, where he is directing the Chair for Embedded Systems (CES). Dr. Henkel received the masters and the Ph.D. (Summa cum laude) degrees, both from the Technical University of Braunschweig, Germany. He then joined the NEC Laboratories, Princeton, NJ, USA. He holds ten U.S. patents. His current research interests include design and architectures for embedded systems with focus on low power and reliability. Prof. Henkel was the recipient of the 2008 DATE Best Paper Award, the 2009 IEEE/ACM William J. McCalla ICCAD Best Paper Award, the CODES+ISSS 2011 and 2014 Best Paper Awards. He was the Chairman of the IEEE Computer Society, Germany Section, and the Editor-in-Chief of the ACM Transactions on Embedded Computing Systems. He is also an Initiator and the Spokesperson of the national priority program called Dependable Embedded Systems of the German Science Foundation, and the General Chair of ICCAD 2013. He is a Fellow of the IEEE.