Process-Variation Aware Mapping of Real-Time Streaming Applications to MPSoCs for Improved Yield

Davit Mirzoyan[1], Benny Akesson[2], Kees Goossens[2]
[1] Delft University of Technology, [2] Eindhoven University of Technology
D.Mirzoyan@tudelft.nl, K.B.Akesson@tue.nl, K.G.W.Goossens@tue.nl

***Abstract*—As technology scales, the impact of process variation on the maximum supported frequency (FMAX) of individual cores in a MPSoC becomes more pronounced. Task allocation without variation-aware performance analysis can result in a significant loss in *yield*, defined as the number of manufactured chips satisfying the *application timing requirement*. We propose variation-aware task allocation for real-time streaming applications modeled as task graphs. Our solutions are primarily based on the throughput requirement, which is the most important timing requirement in many real-time streaming applications. The three main contributions of this paper are: 1) Using data flow graphs that are well-suited for modeling and analysis of real-time streaming applications, we explicitly model task execution both in terms of clock cycles (which is independent of variation) and seconds (which does depend on the variation of the resource), which we connect by an explicit binding. 2) We present two approaches for optimizing the yield. The approaches give different results at different costs. 3) We present exhaustive and heuristic algorithms that implement the optimization approaches. Our variation-aware mapping algorithms are tested on models of real applications, and are compared to the mapping methods that are unaware of hardware variation. Our results demonstrate yield improvements of up to 50% with an average of 31%, showing the effectiveness of our approaches.**

***Index Terms*—Process variation, Multiprocessor System-on-Chip, Synchronous Data Flow Graphs, Task Allocation**

I. Introduction

Aggressive technology scaling has enabled the integration of multiple processors and hardware accelerators on a single silicon chip die, known as a Multiprocessor System-on-Chip (MPSoC). The use of such systems is increasingly popular, as they have high computational power and low power consumption, which are the main requirements for many embedded systems. Scaling the minimum feature sizes in deep-sub-micron technologies, however, has also brought variations in key transistor parameters, such as channel length and device and interconnect width. This phenomenon, known as process variation [1], [2], significantly impacts the maximum supported frequency of individual cores in a MPSoC [3], [4]. It is shown in [5] that the variation in the longest path delay (the inverse of FMAX) of a Very Long Instruction Word (VLIW) processor, manufactured at 32nm technology, is up to 40%. Moreover, the impact of within-die variation on the system parameters is increasing as the technology scales, making it a limiting factor in efficient MPSoC design [3].

Binding of application tasks to the resources in a MPSoC without considering the impact of process variation can greatly compromise the performance and lead to a significant *yield* loss. In this work, the yield metric is not the hardware manufacturing yield, which determines the number of chips that meet the predefined frequency requirements. In our view, yield is defined at the application level, and shows the number of manufactured chips that satisfy the application timing requirement. Existing solutions, which propose variation-aware yield-driven task allocation and scheduling [6]–[9], use acyclic task graphs for application modeling and are based on latency requirements. Acyclic task graphs are not able to capture the cyclic data dependencies and the streaming behavior (i.e. iterative and overlapping execution) of real-time streaming applications [10]. In this work, we allow arbitrary task graphs that may include cyclic data dependencies. Our solutions are primarily based on the throughput requirement, which is the most important timing requirements in many real-time streaming applications, but latency requirements can also be addressed [11].

The three main contributions of this work are: 1) We base our solutions on Synchronous Data Flow Graphs (SDFG), which are well-suited for modeling and analysis of real-time streaming applications, and have multiple efficient techniques for throughput computation [10]. The novelty of our SDFG formulation lies in the *explicit modeling* of software execution in terms of clock cycles (which is independent of the variation in the hardware resource), and in terms of seconds (which does depend on the variation in the resource), which are linked by an explicit binding. 2) We present two approaches for optimizing the yield, *single-binding* and *multiple-bindings*. With the single-binding optimization approach, the objective is to find a binding at design time that maximizes the yield of all the manufactured chips, which have different variation in the resources. With multiple-bindings optimization approach, a set of bindings are found and stored at design time, and based on the variation in each manufactured chip, the right binding that satisfies the application timing requirement is selected at the run-time configuration stage. 3) We present exhaustive and heuristic algorithms that implement the optimization approaches. The exhaustive algorithms provide optimum results and can be applied to problems of small to medium size. The heuristic algorithms provide results close to optimum and are scalable to problems of large size. Our variation-aware mapping algorithms are compared to the mapping methods that are unaware of the variation in the hardware resources. Our results report yield improvements of up to 50% with an average of 31%, showing the effectiveness of our methods.

The rest of the paper is organized as follows: Section II presents the related work in the field. Section III introduces formal models of a hardware platform, an application SDFG and a binding. In Section IV, we present the single-binding and multiple-bindings approaches for optimizing the yield of real-time streaming applications. Section V illustrates the variation-aware exhaustive and heuristic algorithms that implement the optimization approaches. Section VI experimentally evaluates our methods and Section VII concludes the paper.

II. Related work

Several techniques have been proposed to minimize process variation at the circuit and microarchitectural levels [1], [12], but they are unable to hide the variation at the system level. There has been extensive research in the area of task allocation and scheduling for MPSoC [6]–[10], [13]–[15]. The researchers in [10], [14], [15] proposed methods to map throughput-constrained applications modeled as SDFGs to the resources in a MPSoC. None of them, however, considers the impact of process variation. With variation-unaware mapping approaches, the impact of process variation cannot be reflected by having different resources with different frequencies as the availability of a resource with a specific frequency is a matter of probability.

Wang *et al.* [6] introduced a new design metric called *performance yield*, defined as the probability of the assigned schedule meeting the predefined performance constraint. Assuming that the execution times of tasks follow a Gaussian distribution, they proposed a variation-aware scheduling algorithm that allocates and schedules tasks that have latency requirements in an acyclic task graph to MPSoC, such that the performance yield is maximized. Resource sharing in task allocation and scheduling under process variation has been studied by Chon and Kim [7]. They proposed an effective statistical static timing analysis technique, which schedules and binds tasks in an acyclic task graph to the resources in an MPSoC in the presence of resource sharing, such that the performance yield is maximized. Singhal and Bozorgzadeh [8] introduced the problem of stochastically optimal task allocation, which is to minimize the overall execution time of tasks in sequence and parallel under process variation. Huang and Xu [9] took into account the spatial correlation characteristics of systematic within-die variation and presented a scheduling algorithm that schedules tasks with latency constraints in an acyclic task graph, such that the performance yield is maximized. With their solution, a set of schedules is synthesized off-line and based on the variation for each chip, a run-time scheduler selects the right one, such that the latency constraint is satisfied whenever possible.

All the solutions in the above work that account for process variation use acyclic task graphs for application modeling and are based on latency requirements. Acyclic task graphs are not able to capture the iterative and overlapping execution of real-time streaming applications, which are primarily constrained by throughput requirements. We allow arbitrary task graphs that may include cyclic data dependencies. Our solutions are primarily based on throughput requirements but latency requirements can also be addressed [11]. We use SDF graphs that are well suited for modeling and analysis.

III. Formal Models

This section formally defines a hardware multiprocessor platform as a set of resources. We introduce a set of operating points (maximum supported frequencies) for each resource to reflect the impact of process variation. We define an SDFG model of an application, named an *unbound graph*, where the actors (tasks of an application) are characterized by execution times in clock cycles. The unbound graph is unaware of the binding of application actors to the hardware resources, and is hence decoupled from hardware variation. Later, we introduce an explicit binding of actors to the resources in a platform and define an SDFG model of an application, named a *bound graph*, where application actors are characterized by execution times in seconds. The bound graph is no longer decoupled from hardware variation, and it enables us to analyze the impact of variation on the application performance for different actor to resource bindings.

The presented techniques are general and apply to any system that implements the models in this section. Examples of such systems are CoMPSoC [16] and CA-MPSoC [17]

A. Model of a Hardware Platform

We refer to a hardware multiprocessor platform as a set of resources connected with each other by a hypothetical interconnection network. We denote the set of resources as R. Each resource is a generic processing element, such as a processor, DSP or a hardware accelerator. To model process variation in each of the resources, we characterize each resource by a *nominal operating point* (nominal maximum supported frequency) and a set of *possible operating points* (possible maximum supported frequencies). The nominal operating point (Definition 1) is the target speed specification of the resource, and what the manufacturing aims for, but due to process variation, the resource can have any of the multiple operating points (Definition 2). We assume a multiprocessor platform, where each resource is in a separate frequency domain and can be operated at any of its possible operating points. This assumption holds for Globally Asynchronous and Locally Synchronous (GALS) embedded designs.

Definition 1: (Nominal operating point of a resource) The function $ON : R \to \mathbb{R}^+$ returns the nominal operating point of a resource $r \in R$.

Definition 2: (Possible operating points of a resource) The function $OP : R \to \mathcal{P}(\mathbb{R}^+) \setminus \emptyset$ returns a non-empty set of all possible operating points of a resource $r \in R$.

The occurrence probability of each operating point of a resource is given by Definition 3. Note that the sum of the probabilities of all the operating points of any resource is 1.

Definition 3: (Probability of an operating point of a resource) The function $P : R \times \Omega \to \mathbb{R}^+$ returns the occurrence probability of an operating point $op \in OP(r)$ of a resource $r \in R$, where Ω is the set of all operating points

$$\Omega = \bigcup_{r \in R} OP(r) \tag{1}$$

Given that each resource is characterized by a set of possible operating points, there are multiple combinations of operating points for the overall number of resources. We refer to an instance of operating points for a sequence of the overall number of resources as a *system operating point*. The set of all possible system operating points is obtained by the Cartesian product of the individual sets of operating points of resources (Definition 4), and is an N-dimensional vector for N resources.

The probability that a set of resources has a certain system operating point is given by Definition 5.

Definition 4: (System operating points) The set OS of all possible system operating points for a set R of resources is defined as

$$\mathit{OS} = \prod_{r \in R} \mathit{OP}(r) \tag{2}$$

Definition 5: (Probability of a system operating point) The function $\mathit{SP} : \mathit{OS} \rightarrow \mathbb{R}^+$ gives the occurrence probability of a system operating point $os \in \mathit{OS}$ and is defined as

$$\forall os \in \mathit{OS}.\ \mathit{SP}(os) = \prod_{\substack{r \in R \\ op \in os}} P(r, op) \tag{3}$$

To illustrate the presented concepts, consider a platform comprising two resources. Each resource r is given by a nominal operating point $\mathit{ON}(r)$ (in cycles/second), a set of two possible operating points $\mathit{OP}(r)$, and the occurrence probabilities $P(r, op)$ of each operating point (Table I). The system operating points and the occurrence probability of each system operating point are obtained by Definitions 4 and 5, respectively, and are given in Table II.

TABLE I
A PLATFORM COMPRISING TWO RESOURCES

Resource	$\boldsymbol{ON}(r)$	$\boldsymbol{OP}(r)$	$\boldsymbol{P}(r, op)$
r_1	10	8	0.3
		10	0.7
r_2	10	9	0.3
		10	0.7

TABLE II
SYSTEM OPERATING POINTS AND THEIR OCCURRENCE PROBABILITIES

$\boldsymbol{OS}$	(8 9)	(8 10)	(10 9)	(10 10)
$\boldsymbol{SP}(os)$	0.09	0.21	0.21	0.49

B. Model of an Unbound Graph

We model real-time streaming applications by means of Synchronous Data Flow Graphs (SDFG). The motivation behind this choice is that an SDFG model provides a good compromise between expressiveness, modeling ease, analysis potential and implementation efficiency. With an SDFG model, an application is captured by a directed graph, where the nodes (called actors) represent computations (tasks) that communicate with each other by sending streams of data-elements over their edges. We denote the set of all actors as A, where each actor requires a number of clock cycles to finish its execution (Definition 6). Note that the number of clock cycles required for an actor's execution can be different for each resource.

Definition 6: (Execution time of an actor in cycles) The function $\mathit{EC} : A \times R \rightarrow \mathbb{N}$ returns the number of cycles required to execute an actor $a \in A$ on a resource $r \in R$.

Definition 7 defines a model of a SDFG that is unaware of the binding of actors to resources. Each actor in the graph is characterized by a number of execution times in clock cycles of the resource for the resources to which it can be bound.

Definition 7: (Unbound graph) An unbound graph ug is a 4-tuple $\langle A, D, \mathit{Init}, \mathit{EC} \rangle$ with a set A of actors, a set $D = A \times A$ of dependency edges, a function $\mathit{Init} : D \rightarrow \mathbb{N}$ that gives the number of initial tokens for an edge $d \in D$, and the function $\mathit{EC} : A \times R$ that gives the execution times in clock cycles of actors A on a number of resources in the set R.

Figure 1 illustrates an example SDFG model of an H.263 Encoder application. It consists of five actors, which are connected to each other by means of seven dependency edges. Dependency edges d_3, d_6 and d_7 contain initial tokens, illustrated by black dots in the figure. The execution of an actor is called a firing. When an actor fires it removes a number of tokens from all its input ports and at the end of the firing (after its execution), it produces a number of tokens on each output port. The set of actor firings that restores the initial configuration of the graph is termed an iteration. During a single iteration of the graph, each actor can fire a number of times. This is given by the repetition vector of the graph (Definition 8). The repetition vector of the SDFG shown in Figure 1 is equal to (1, 99, 1, 99, 1) for actors (a_1, a_2, a_3, a_4, a_5), respectively.

Definition 8: (Repetition vector) The function $\gamma : A \rightarrow \mathbb{N}$ returns the number of times each actor $a \in A$ fires during a single iteration.

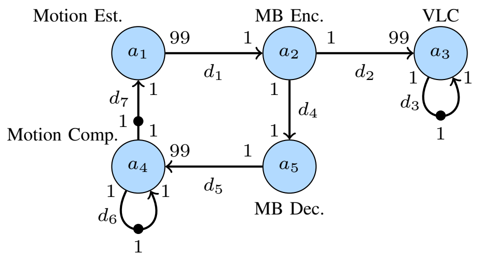

Fig. 1. Example SDFG model of H.263 Encoder.

C. Model of a Bound Graph

Each actor can be bound to a number of resources from the set R. The set of resources an actor can be bound to is given by Definition 9.

Definition 9: (Possible bindings of an actor) The function $\mathit{BP} : A \rightarrow \mathcal{P}(R) \setminus \emptyset$ returns the set of resources to which an actor $a \in A$ can be bound.

For a set A of actors and a set R of resources, there can be multiple bindings of actors to resources. The set of all possible actor to resource bindings can be obtained by the Cartesian product of the individual sets of possible bindings of actors (Definition 10).

Definition 10: (Binding) The set B of all possible actor to resource bindings is defined as

$$B = \prod_{a \in A} \mathit{BP}(a) \tag{4}$$

For each binding, the execution time of an actor in clock cycles is known. The execution time of an actor in seconds on a resource for a specific operating point is given by Definition 11.

Definition 11: (Execution time of an actor in seconds) The function $ET : A \times R \rightarrow \mathbb{Q}$ returns the execution time in seconds of an actor $a \in A$ on a resource $r \in R$ that has an operating point $op \in OP(r)$, and is defined as

$$ET(a,r) = \frac{EC(a,r)}{op} \tag{5}$$

For a specific system operating point and a specific binding of actors to resources, a model of a bound SDFG can be generated (Definition 12).

Definition 12: (Bound graph) A bound graph bg is a 3-tuple $\langle ug, b, os\rangle$ with an unbound graph ug, a binding $b \in B$ of actors A to resources R and a system operating point $os \in OS$.

The throughput of a SDFG is traditionally computed by means of Maximum Cycle Mean analysis (MCM) on the equivalent Homogeneous SDFG (HSDFG) (Definition 13). This implies that a conversion from SDFG to HSDFG is required [18].

Definition 13: (Throughput of a bound graph) Throughput of a bound graph bg is defined as $T(ug,b,os) = 1/MCM(bg')$, where $MCM(bg')$ is the maximum cycle mean over all cycles in the equivalent HSDFG bg'. The cycle mean of each cycle c equals the sum of the execution times of actors in the cycle divided by the number of initial tokens on the cycle.

$$MCM(bg') = \max_{c \in C_{bg}} \sum_{a \in c} ET(a)/Init(c) \tag{6}$$

IV. Yield Optimization Problems

In this section, we define our optimization problem for real-time streaming applications. The optimization objective is to maximize the yield, which is the number of manufactured chips that satisfy the application minimum throughput requirement, denoted t_{req}. We present two approaches for optimizing the yield, *single-binding* and *multiple-bindings*. With the single-binding optimization approach, the objective is to find a single binding at design time, such that the yield is maximized. With this approach, all the manufactured chips, which have different system operating points (variation) in the resources, have an identical binding. The objective with the multiple-bindings optimization approach is to find and store a set of bindings at design time, and based on the system operating point (variation) of each manufactured chip, the right binding that satisfies the throughput requirement is selected at the run-time configuration stage. The run-time binding selection for each chip is done only once at the system initial configuration stage through the operating system, and is not detrimental to real-time deadlines.

A. Single Binding

With the single-binding optimization approach, the objective is to find a binding that maximizes the yield. For a given binding $b \in B$, the different chips that have different system operating points (variation) can have different throughputs. Figure 2 depicts the throughput $T(bg) = T(ug,b,os)$ of the bound graph bg for the different system operating points $os \in OS$ of the chips, given a fixed binding $b \in B$. In our modeling framework, the yield of a binding $b \in B$ over all the system operating points $os \in OS$, where each system operating point has a probability-weight $SP(os)$, is given in Definition 14.

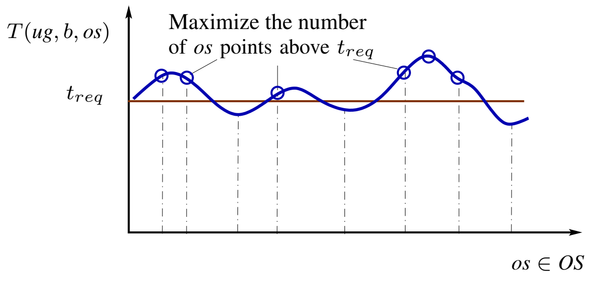

Fig. 2. Throughput against system operating point for a fixed binding. Yield is given by the number of os points (with associated probabilities) above t_{req}.

Definition 14: (Yield of a binding) Given an unbound graph ug and a binding $b \in B$, the function Y gives the number of chips satisfying the requirement t_{req} over all system operating points $os \in OS$

$$\forall b \in B.\ Y(b) = \sum_{os \in OS} \begin{cases} SP(os) & \text{if } T(ug,b,os) \geq t_{req} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

The objective of the single-binding optimization approach is formulated as: Given a set A of actors and a set R of resources, find a binding $b_{out} \in B$ of actors to resources, such that the yield $y_{max} = Y(b_{out})$ is maximized.

B. Multiple Bindings (Run-Time Configuration)

With the multiple-bindings optimization approach, the objective is to find a binding for each system operating point $os \in OS$ at design time, such that the throughput requirement of the bound graph at that system operating point is satisfied. If there is no binding that satisfies the requirement, then the chips with that particular system operating point cannot be used, reducing the yield. Therefore, a set of bindings are found and stored at design time. Based on the system operating point (variation) of each manufactured chip, the right binding that satisfies the throughput requirement is then selected at the run-time configuration stage. Per chip binding selection always results in higher or equal yield as compared to the case where a single binding is selected for all the chips. The downside of the approach is that multiple bindings are stored for the configuration stage and diverse application instances are present for the same product, which can complicate the processes of software maintenance and upgrading.

The objective of the multiple-bindings optimization approach is formulated as: Given a set A of actors and a set R of resources, find a set $B_{out} \subseteq B$ of bindings that includes a binding for each $os \in OS$, such that the yield y_{max} is maximized.

V. Implementation Algorithms

As shown in Section IV, to maximize the yield by either of the presented optimization approaches, various bindings of application actors to resources have to be explored. In this section we present two algorithms for the evaluation of bindings, an *exhaustive* approach and a *heuristic* algorithm. With the exhaustive approach, we evaluate all binding possibilities (as given by Definition 10) to find a binding for all chips (single-binding optimization) or a binding per chip (multiple-bindings optimization). This approach enables us to find the maximum improvement in yield (optimum solution), but is computationally too expensive for problems of large size (i.e. large number of actors and resources). To overcome this limitation, we also implemented a heuristic algorithm that prunes the search space and obtains results close to the optimum.

A. Exhaustive Algorithm

The exhaustive algorithm for the single-binding optimization approach is shown in Algorithm 1. As input, the algorithm requires an application graph *ug* with an associated throughput requirement t_{req} and a set R of resources, where each resource is given by a set $OP(r)$ of possible operating points and associated occurrence probabilities $P(r, op)$. As can be seen, the algorithm exhaustively evaluates the yield (Definition 14) of all possible bindings, and returns the binding b_{out} that results in the highest yield $y_{max} = Y(b_{out})$.

Algorithm 1 Exhaustive algorithm: Single-binding.

```
Require: $ug$, $t_{req}$, $R$, $OP(r)$, $P(r, op)$
1: $y_{max} \leftarrow 0$
2: for all $b \in B$ do
3:     if $Y(b) > y_{max}$ then
4:         $b_{out} \leftarrow b$
5:         $y_{max} \leftarrow Y(b)$
6:     end if
7: end for
8: return $b_{out}$, $y_{max}$
```

Algorithm 2 illustrates the exhaustive algorithm for the multiple-bindings optimization approach. As shown, for each system operating point $os \in OS$, the algorithm exhaustively evaluates all possible bindings to find a binding that satisfies the requirement t_{req}. The first binding that satisfies the requirement t_{req} is stored for each system operating point. It is possible that no binding can satisfy t_{req} for a particular $os \in OS$, resulting in reduced yield. The algorithm returns a set B_{out} of bindings that includes a binding for individual $os \in OS$ and an estimated yield y_{max} for all the chips.

The exhaustive approach enables us to find the optimum solution, which provides the maximum improvement in yield. The limitation of the approach is that it is computationally too expensive for problems of large size (i.e. large number of actors and resources). The total number of bindings to evaluate is $|R|^{|A|}$, where $|R|$ is the number of resources and $|A|$ is the number of actors in an application. We, hence, proceed by presenting a heuristic algorithm that prunes the search space and obtains results close to the optimum.

Algorithm 2 Exhaustive algorithm: Multiple-bindings.

```
Require: $ug$, $t_{req}$, $R$, $OP(r)$, $P(r, op)$
1: $y_{max} \leftarrow 0$, $B_{out} \leftarrow \emptyset$
2: for all $os \in OS$ do
3:     for all $b \in B$ do
4:         if $T(ug, b, os) > t_{req}$ then
5:             $B_{out} \leftarrow B_{out} \cup \{b\}$ //save $b$ for current $os$
6:             $y_{max} \leftarrow y_{max} + SP(os)$
7:             BREAK
8:         end if
9:     end for
10: end for
11: return $B_{out}$, $y_{max}$
```

B. Heuristic Algorithm

With the heuristic algorithm, only a small number of bindings from the total number of possibilities are explored. The bindings that are evaluated by the heuristic algorithm are generated by a two-phase procedure, *initial resource allocation* and *allocation optimization*. In the initial resource allocation, an initial binding of application actors to resources is derived. This initial binding later undergoes an optimization stage where the actors are moved from one resource to another to either improve the yield (single-binding optimization) or the throughput for each chip (multiple-bindings optimization).

In the initial resource allocation, the actors whose execution time have a large impact on the throughput of an application, referred to as *critical actors*, are considered first. The criticality of an actor $a \in A$ is estimated by the product of its repetition vector $\gamma(a)$ (Definition 8) and average execution time (Definition 6) in a number of cycles over all the resources (Definition 15). This is an approximate way of determining the criticality, as it intuitively estimates the average computational demand of an actor.

Definition 15: (Actor criticality) The function $C : A \to \mathbb{Q}$ returns the criticality of an actor $a \in A$, and is defined as

$$\forall a \in A.\ C(a) = \gamma(a) \cdot \frac{1}{|R|} \sum_{r \in R} EC(a, r) \tag{8}$$

When allocating the actors to the resources, the initial resource allocation tries to balance the load (in terms of execution time in seconds during an iteration) on the resources. The load of a resource is computed by the sum of products of the repetition vectors and the execution times in seconds of the actors bound to the resource (Definition 16).

Definition 16: (Resource load) The function $L : R \to \mathbb{Q}$ returns the load of a resource $r \in R$, and is defined as

$$\forall r \in R.\ L(r) = \sum_{\substack{a \in A \\ a \text{ bound to } r}} \gamma(a) \cdot ET(a, r) \tag{9}$$

Algorithm 3 shows the heuristic algorithm for the single-binding optimization approach. In the first part of the algorithm (lines 2–9), initial resource allocation is performed. The actors, sorted in decreasing order of criticality, are allocated to the resources, such that the load on the resources is balanced. Each time an actor is to be bound to a resource, the resource with the lowest load is selected. If the resources are not allocated yet, an actor is bound to the resource with the highest nominal operating point (Definition 1). This is done to ensure that the actors with higher criticality are allocated to resources with higher computational power (resources with higher nominal operating points are on average faster). In the second part of the algorithm (lines 11–19), allocation optimization is performed. The allocation of each actor in increasing order of criticality is reconsidered. Each time an actor is moved from one resource to another, the new binding is evaluated for yield. The algorithm returns a binding b_{out} that has the highest yield $y_{max} = Y(b_{out})$ among the bindings that have been explored.

Algorithm 3 Heuristic algorithm: Single-binding.

```
Require: $ug$, $t_{req}$, $R$, $OP(r)$, $P(r, op)$
1: $y_{max} \leftarrow 0$
2: $\forall r \in R.\ L(r) \leftarrow 0$
3: Sort $A$ in decreasing $C(a)$
4: Sort $R$ in decreasing $ON(r)$
5: for all $a \in A$ do
6:     Sort $R$ in increasing $L(r)$
7:     Bind first $a$ to first $r$
8:     Update $L(r)$
9: end for //initial binding $b$ retrieved
10:
11: for all $a \in A$ do
12:     for all $r \in R$ do
13:         if $Y(b) > y_{max}$ then
14:             $b_{out} \leftarrow b$
15:             $y_{max} \leftarrow Y(b)$
16:         end if
17:         Bind $a$ to $r$ //new binding $b$ retrieved
18:     end for
19: end for
20: return $b_{out}$, $y_{max}$
```

The heuristic algorithm for the multiple-bindings optimization approach is illustrated in Algorithm 4. For each system operating point $os \in OS$, the algorithm performs initial resource allocation and allocation optimization, such that a binding is found that satisfies the requirement t_{req}. In initial resource allocation (for each $os \in OS$), when the resources are not allocated yet, an actor is bound to the resource with the highest operating point based on the current os (line 5 in Algorithm 4). This ensures that the actors with higher criticality are allocated to faster resources for each system operating point. After initial resource allocation, allocation optimization is performed (lines 12–25). The allocation of each actor in increasing order of criticality is reconsidered. Each time an actor is moved from one resource to another, the new binding is evaluated for throughput. The optimization for each $os \in OS$ stops when a binding is found that satisfies the requirement t_{req}. The algorithm returns a set B_{out} of bindings that includes a binding for individual $os \in OS$ and an estimated yield y_{max} for all the chips.

The number of bindings to explore with the heuristic algorithm for yield estimation for all chips (single-binding approach) or for throughput estimation per chip (multiple-bindings approach) is $|A| \cdot (|R| - 1)$. Given a large problem, $|A| \cdot (|R| - 1)$ is considerably lower than the total number $|R|^{|A|}$ of bindings evaluated by the exhaustive approach, i.e. $|A| \cdot (|R| - 1) << |R|^{|A|}$.

Algorithm 4 Heuristic algorithm: Multiple-bindings.

```
Require: $ug$, $t_{req}$, $R$, $OP(r)$, $P(r, op)$
1: $y_{max} \leftarrow 0$, $B_{out} \leftarrow \emptyset$
2: Sort $A$ in decreasing $C(a)$
3: for all $os \in OS$ do
4:     $\forall r \in R.\ L(r) \leftarrow 0$
5:     Sort $R$ in decreasing speed based on $os$
6:     for all $a \in A$ do
7:         Sort $R$ in increasing $L(r)$
8:         Bind first $a$ to first $r$
9:         Update $L(r)$
10:    end for //initial binding $b$ retrieved
11:
12:    for all $a \in A$ do
13:        if $T(ug, b, os) > t_{req}$ then
14:            $B_{out} \leftarrow B_{out} \cup \{b\}$ //save $b$ for current $os$
15:            $y_{max} \leftarrow y_{max} + SP(os)$
16:            BREAK
17:        end if
18:        for all $r \in R$ do
19:            Bind $a$ to $r$ //new binding $b$ retrieved
20:            if $T(ug, b, os) > t_{req}$ then
21:                BREAK
22:            end if
23:        end for
24:    end for
25: end for
26: return $B_{out}$, $y_{max}$
```

VI. Experimental Results

In this section, we describe our experimental setup and present the results of our experiments. We illustrate the improvements in yield that are achieved by our variation-aware mapping algorithms over the mapping approaches that are unaware of hardware variation. Additionally, we analyze how well the heuristic algorithms perform as compared to the optimum results.

A. Setup

Our variation-aware mapping algorithms for real-time streaming applications are evaluated on a number of real DSP and multimedia applications modeled as SDFGs. From the DSP domain, the set contains a Sample-Rate Converter and Modem, and from the multimedia domain an H.263 Encoder (Figure 1), H.263 Decoder, MP3 Playback, MP3 Decoder and Satellite Receiver. These application SDFGs are the unbound graphs in our formal framework. The SDFGs of the applications, including their execution times in clock cycles, can be found in [19], and are not presented in this paper because of limited space.

These applications are allocated to a hypothetical MPSoC with 2-5 heterogeneous resources with nominal operating points (Definition 1) of 380, 380, 380, 440 and 500 MHz, respectively. The set of possible operating points of each resource is obtained by making assumptions on the impact of process variation on the nominal operating point (FMAX) of the resources. To reflect the impact of within-die variation on the nominal operating point, we assumed mean degradations of 3%, 6% and 15% for the resources with 380, 440 and 500 MHz nominal operating points, respectively. As shown in [4], within-die variations result in uncorrelated delay variations in various devices in a chip, and are higher for devices with lower logic depth (faster resources). That is why we assumed different mean degradations for the resources. Die-to-die variations, on the other hand, result in correlated delay variations in the devices, and are not dependent on the logic depth. For this variation, we assumed a standard deviation of 3.3% ($3\sigma = 10\%$) for all three resources. These numbers are selected as they agree with the data on process variation in current technology nodes [5]. To obtain the set of possible operating points, we discretized the FMAX probability distribution function (PDF) of each resource into 5 discrete points. The discretized model of FMAX PDF for the resource with 380 MHz nominal operating point is shown in Figure 3. The more points from the distribution we select, the higher accuracy we achieve. By choosing 5 points, we sacrifice accuracy in the interest of reducing the run-times of the algorithms.

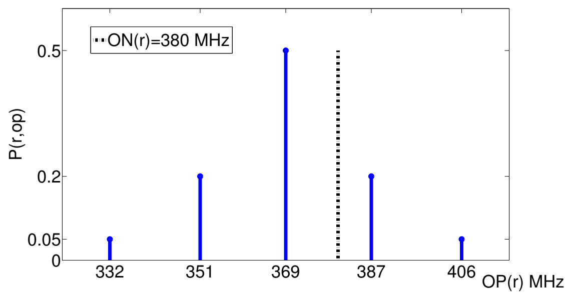

Fig. 3. Discretized FMAX PDF of a resource with 380 MHz nominal operating point, resulting in possible operating points and associated probabilities

As given in Definition 13, the throughput of a SDFG is traditionally computed by means of MCM analysis on the equivalent HSDFG. This requires a conversion from the SDFG to an equivalent HSDFG, which can be considerably larger in size (in terms of the number of actors) than the original SDFG, making the approach inefficient for SDFG throughput analysis. In our work, we use the SDF3 tool for throughput analysis [20]. To compute the throughput of a SDFG, SDF3 uses State Space Exploration, which works directly on a SDFG and gives results equivalent to MCM analysis [21]. Depending on what features are enabled in SDF3, the run-time of the tool can vary [22]. In our experiments, SDF3 is used for only throughput analysis, and no additional features are required. This results in run-times in the order of micro-seconds.

B. Evaluation Results

We compare the results of our optimization algorithms to those of variation-unaware nominal frequency-based mapping methods, where the binding of actors to resources is derived based on the nominal operating points of the resources. The purpose of the experiments is to show the importance of variation-aware mapping. We first show the maximum improvement in yield (optimum result) that is achieved by the variation-aware exhaustive mapping algorithms for the H.263 Decoder, H.263 Encoder, MP3 Playback and Sample Rate applications of medium size (6 actors the largest). Later, we evaluate the performance of the heuristic mapping algorithms as compared to the optimum results for the same applications of medium size, and we apply the heuristic algorithms to the Modem, MP3 Decoder and Satellite Receiver applications of large size (22 actors the largest).

Figure 4 illustrates the yield for the H.263 Decoder, H.263 Encoder, MP3 Playback and Sample Rate as a result of variation-aware exhaustive and nominal frequency-based mapping algorithms. These applications have a small to medium number of actors (6 actors the largest), enabling the use of the exhaustive mapping algorithms. The exhaustive algorithms for the single-binding and multiple-bindings optimization approaches are denoted as VA-SBE and VA-MBE, respectively. For nominal frequency-based mapping, there can be multiple bindings of actors to resources that satisfy the throughput requirement for the nominal operating points of the resources. A binding that just satisfies the requirement can potentially result in very low yield, as any negative variation in the nominal operating points of the resources can lead to a violation. For a fair comparison, for the nominal frequency-based mapping, we choose a binding that satisfies the throughput requirement and gives the highest throughput among all other bindings. The algorithm for the described nominal frequency-based mapping is denoted as NCE in Figure 4.

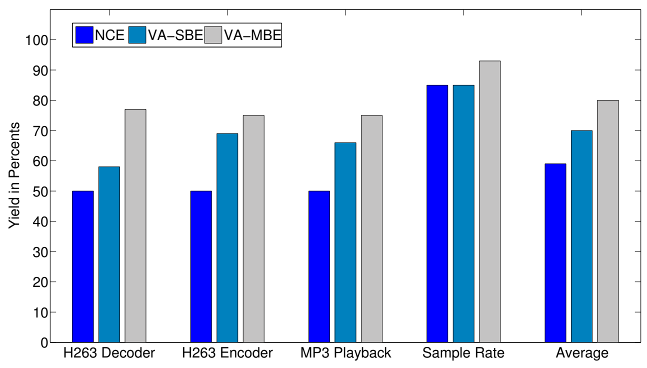

Fig. 4. Yield of applications as a result of NCE, VA-SBE and VA-MBE mapping algorithms.

Figure 4 shows yield improvements of up to $69\% - 50\% = 19\%$ (H.263 Encoder) achieved by VA-SBE over the variation-unaware NCE mapping. As expected, VA-MBE performs better than VA-SBE, resulting in yield improvements of up to 27% over NCE mapping. Figure 4 additionally illustrates the average yield for the set of applications. The variation-unaware NCE mapping results in 59% average yield, which is improved to 70% and 80% by VA-SBE and VA-MBE, respectively. These results show the importance of variation-awareness in the resource allocation process. The run-times of the exhaustive algorithms for the applications in Figure 4 are in the order of 1 hour on a dual core 2.8 GHz machine.

Figure 5 shows the yield achieved by the variation-aware heuristic and nominal frequency-based mapping approaches for the complete set of applications. The nominal frequency-based mapping, denoted as NCH, is also implemented by the same heuristic algorithm presented in Section V. With NCH mapping, application actors are initially allocated to the resources for the nominal operating points, followed by an allocation optimization, where the initial allocation is optimized for higher throughput. Figure 5 shows that for the H.263 Decoder, H.263 Encoder, MP3 Playback and Sample Rate, the heuristic NCH mapping gives the same results as the exhaustive NCE mapping illustrated in Figure 4.

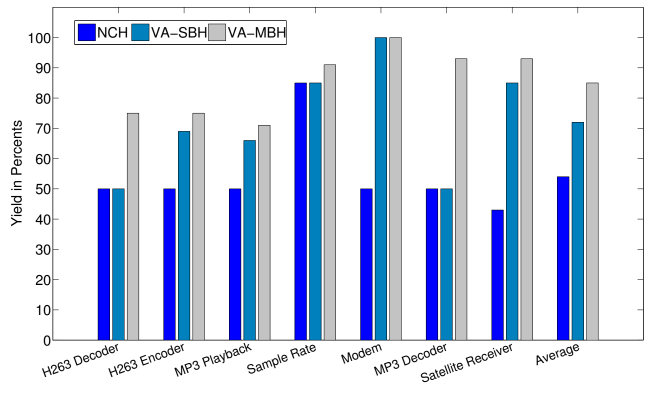

Fig. 5. Yield of applications as a result of NCH, VA-SBH and VA-MBH mapping algorithms.

Figure 5 shows that for the H.263 Encoder, MP3 Playback and Sample Rate, the heuristic algorithm for the single-binding mapping approach, denoted as VA-SBH, gives the optimum results as found by the computationally expensive VA-SBE exhaustive algorithm shown in Figure 4. For the H.263 Decoder, VA-SBH results in 8% lower yield than the optimum result found by VA-SBE. The heuristic algorithm for the multiple-bindings mapping approach, denoted as VA-MBH, gives the optimum result for the H.263 Encoder. For the H.263 Decoder, MP3 Playback and Sample Rate, VA-MBH results in 2%, 4% and 2% lower yield, respectively. These results show that the heuristic algorithms provide results close to the optimum.

Figure 5 additionally illustrates the yield improvements achieved by the variation-aware heuristic mapping algorithms for the Modem, MP3 Decoder and Satellite Receiver applications of large size (22 actors the largest). VA-SBH and VA-MBH provide improvements in yield of up to 50%. The run-times of the heuristic algorithms for the applications of large size are in the order of 15 minutes on a dual core 2.8 GHz machine. The exhaustive algorithms for these applications are infeasible and cannot be applied. This shows how effectively the heuristic algorithms can be applied to problems of large size.

Our experiments showed that many of the bindings selected for an application by VA-MBH are identical, and that not more than 10 different bindings are selected for any of the applications. This observation shows the applicability of the multiple-bindings optimization approach as it imposes low storage requirements.

VII. Conclusions

This paper introduces two approaches, single-binding and multiple-bindings, for mapping real-time streaming applications to MPSoC for maximized yield under process variation. For application modeling we use Data Flow graphs, which can capture the iterative and overlapping execution of real-time streaming applications and have multiple efficient techniques for throughput computation. The novelty of our SDF formulation lies in the explicit modeling of software execution 1) in terms of clock cycles (which is independent of hardware variation), and 2) in terms of seconds (which does depend on the hardware variation), which are linked by an explicit binding. We present exhaustive and heuristic algorithms that implement the single-binding and multiple-bindings optimization approaches. Our results show that: 1) Variation-awareness is important in the resource allocation process, resulting in yield improvements of up to 50% with an average of 31% over the mapping methods that are unaware of hardware variation. 2) The heuristic mapping algorithm effectively reduces the exponential complexity of the exhaustive algorithm, while only giving slight reduction in yield (4% on average). 3) The run-time storage requirements for the multiple-bindings optimization approach are very low as only a few bindings are selected and stored for an application.

References

[1] O. Unsal *et al.*, "Impact of parameter variations on circuits and microarchitecture," *MICRO*, vol. 26, no. 6, 2006.

[2] *International Technology Roadmap for Semiconductors*, 2009 Edition.

[3] K. Bowman *et al.*, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *Solid-State Circuits*, vol. 37, no. 2, 2002.

[4] M. Eisele *et al.*, "The impact of intra-die device parameter variations on path delays and on the design for yield of low voltage digital circuits," *VLSI*, vol. 5, no. 4, 1997.

[5] M. Miranda *et al.*, "Variability aware modeling of SoCs: From device variations to manufactured system yield," in *Proc. ISQED*, 2009.

[6] F. Wang *et al.*, "Variation-aware task allocation and scheduling for MPSoC," in *Proc. ICCAD*, 2007.

[7] H. Chon and T. Kim, "Timing variation-aware task scheduling and binding for MPSoC," in *Proc. ASP-DAC*, 2009.

[8] L. Singhal and E. Bozorgzadeh, "Process variation aware system-level task allocation using stochastic ordering of delay distributions," in *Proc. ICCAD*, 2008.

[9] L. Huang and Q. Xu, "Performance yield-driven task allocation and scheduling for MPSoCs under process variation," in *Proc. DAC*, 2010.

[10] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. DAC*, 2007.

[11] O. Moreira and M. Bekooij, "Self-timed scheduling analysis for real-time applications." in *Proc. EURASIP*, 2007.

[12] J. Tschanz *et al.*, "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," in *Proc. CCSS*, 2002.

[13] T. D. Braun *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Parallel Distrib. Comput.*, vol. 61, no. 6, 2001.

[14] A. Bonfietti *et al.*, "Throughput constraint for synchronous data flow graphs," in *Proc. CPAIOR*, 2009.

[15] A. Bonfietti *et al.*, "An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms," in *Proc. DATE*, 2010.

[16] A. Hansson *et al.*, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, 2009.

[17] A. Shabbir *et al.*, "CA-MPSoC: An automated design flow for predictable multi-processor architectures for multiple applications," *J. Syst. Archit.*, vol. 56, no. 7, 2010.

[18] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Oct. 2000.

[19] "SDF example applications," www.es.ele.tue.nl/sdf3/download/examples.

[20] S. Stuijk *et al.*, "SDF3: SDF for free," in *Proc. ACSD*, 2006.

[21] A. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," in *Proc. ACSD*, 2006.

[22] S. Stuijk *et al.*, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proc. DAC*, 2006.