Exploring Trade-offs between Performance and Resource Requirements for Synchronous Dataflow Graphs*

Yang Yang[1], Marc Geilen[1], Twan Basten[1,2], Sander Stuijk[1], Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology
[2]Embedded Systems Institute
{y.yang,m.c.w.geilen,a.a.basten,s.stuijk,h.corporaal}@tue.nl

***Abstract*—Synchronous dataflow graphs (SDFGs) are widely used to model streaming applications such as signal processing and multimedia applications. These are often implemented on resource-constrained embedded platforms ranging from PDAs and cell phones to automobile equipment and printing systems. Trade-off analysis between resource usage and performance is critical in the life cycle of those products, from tailoring platforms to target applications at design time to resource management at runtime. We present a trade-off analysis method for SDFGs based on model-checking techniques and leveraging knowledge from the dataflow domain. We develop results to prune the state space of an SDFG for multi-objective model checking without loosing optimality. To achieve scalability to large state spaces, we combine these pruning techniques with pragmatic heuristics. We evaluate our techniques with two sets of experiments. One set shows we can now do throughput-storage trade-off analysis for shared memory architectures, showing reductions in memory usage of 10-50% compared to existing distributed memory based analysis. A second set of experiments shows how our techniques support design-space exploration for the digital datapath of a professional printer system. Analysis times range from less than a second to at most several minutes.**

I. Introduction

Synchronous Dataflow Graphs (SDFGs, [18]) have been widely used to model and analyze streaming applications on embedded systems. These systems frequently have highly constrained resources (memory, bandwidth etc). Embedded applications cannot afford the resource requirements of their desktop counterparts. Embedded system designers need to keep the performance of these systems as high as possible while keeping the resource usage of these systems as low as possible. As these objectives compete with each other, there may be multiple Pareto-optimal points in the resource-performance metric space. So it is important to provide a method which can do trade-off analysis and help designers to tailor platforms for targeted applications and to choose runtime resource management policies. Our paper tackles this problem for the streaming application domain and provides trade-offs by analyzing SDFGs with model-checking techniques.

A simple example SDFG is depicted in Fig. 1 (taken from [26]). The nodes are called actors which represent the computations that are performed. The computation of an actor is atomic. Its name and *execution time* are denoted in the corresponding node of the graph. Actors transfer information to each other on FIFO *channels* via data items, called *tokens* (visualized as black dots). An essential property of synchronous dataflow graphs is that every time an actor *starts a firing* (starts execution), it consumes the same amount of tokens from its input ports, and that every time an actor *ends a firing* (ends execution), it produces the same amount of tokens from its output ports. These amounts are called the *rates* of the ports. Self-edges are used to limit the auto-currency of actors, i.e., the maximal number of simultaneous firings of the same actor. In the example, the auto-concurrency of actors is limited to one, via self-edges with one token.

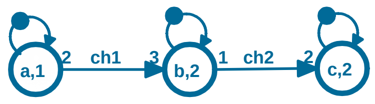


Fig. 1. Example SDFG

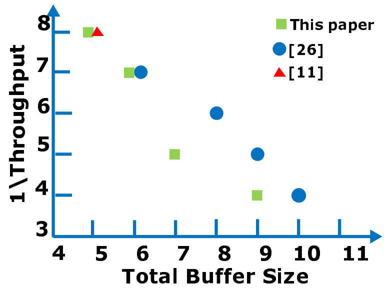


Fig. 2. Comparison among Methods

SDFG behavior is strongly influenced by applied scheduling policies and by resource constraints. They can be seen as putting constraints on the execution of the SDFG, and impact its performance. This is illustrated in Fig. 2. It shows the analysis results of the example SDFG with techniques taken from [11], [26]. [11] (triangles) explores *arbitrary* schedules and minimizes buffer size. However, it optimizes only one objective: memory. [26] (circles) analyzes the trade-offs between throughput and buffer size, but only exploring *self-timed* schedules with a *distributed buffer* resource model. Fig. 2 also shows the trade-offs obtained with our method (squares). We explore a larger part of the design space by allowing more freedom in scheduling and sharing of resources (assuming that sharing is possible in the platform) and we can therefore achieve better results.

*This work has been carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

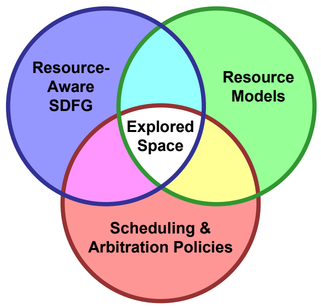


Fig. 3. Venn Diagram of Three Design Domains

The Venn diagram in Fig. 3 shows the concepts motivated by the above example. Each circle represents one design aspect which limits the valid schedules. The resource-aware SDFG domain represents the requirement that the behavior conforms to the functional specification of the graph in terms of dependencies and resource usage. The resource model domain represents constraints imposed by the (limited) available resources on the platform. The scheduling and arbitration policies model represents scheduling and arbitration strategies employed in the application and the platform. The sets are sets of executions of the resource-aware SDFG. The three circles define separate constraints on the set of executions. The constrained execution of the SDFG consists of executions in the intersection only. By carefully selecting the range in each domain to be explored, we can limit the design space to a tractable size, i.e., the intersection in Fig. 3.

In this paper, we use depth first search (DFS) of the non-deterministic state-space to analyze the trade-offs of applications which are modeled by resource-aware SDFGs, in a similar way to explicit-state model-checking techniques. Based on the concept illustrated in Fig. 3, resource information and scheduling rules are provided to limit the state space to be explored. By providing more flexibility in the resource models and scheduling policy design domains, we achieve better results compared to existing methods, even if the state space can only be explored partially. Heuristics and search constraints are provided to help users to accelerate the exploration of the state space. The resulting tool is the most flexible and widely applicable analysis tool for SDFGs available to date.

The remaining parts of the paper are structured as follows. Section 2 discusses related work. Section 3 introduces the SDFG model and its extension which takes resources and scheduling policies into consideration and develops the required theoretical results. Section 4 discusses the exploration techniques developed for this extended model. Implementation details and experimental evaluation can be found in Sections 5 and 6. Section 7 concludes.

II. Related Work

There are many research papers on finding an optimized SDFG schedule subject to one or more criteria [4], [5], [20], [21], [29], [16], [11], [22], [14]. [4] proposes Single Appearance Schedules (SAS), which are specific to single processor platforms and aim to minimize code size. [5] minimizes buffer size for SAS without buffer sharing. [20], [21], [16] allow sharing memory between channels to reduce the total memory usage. However, SAS are not necessarily optimal when other objectives than code size are to be optimized. For multi-processor platforms, where the schedule length does not necessarily lead to extra code size, non-SAS schedules can be better than SAS. [29] relaxes the single appearance constraint on schedules to further reduce the buffer size. [22] targets the minimization of context-switch cost. [14] minimizes total buffer size in throughput-optimal schedules. [30] extends the SDFG model to a variable-rate dataflow (VRDF) model to analyze buffer sizing for data-dependent inter-task communications. In [11], an exact method for exploring arbitrary schedules and generating minimum memory requirements for an SDFG is given which is based on model-checking [7] techniques. Our work is also based on model-checking techniques but differs from all the mentioned work, because it performs multi-objective trade-off analysis.

The performance analysis work on SDFGs mainly focuses on throughput and latency. Throughput has been studied extensively in [9], [8], [12]. [9], [8] use Maximum Cycle Mean (MCM) analysis to compute throughput. This can only be used for Homogeneous SDFGs (HSDFGs). Conversion from an SDFG to an HSDFG is possible, but frequently leads to a sharp increase of the graph size making algorithms of [9], [8] fail. [12] avoids the costly conversion by analyzing the state space of SDFGs. It works well in practice for many graphs. Latency has only been studied recently [25], [13], [19] for SDFGs. [25] gives a heuristic that solves the latency-constrained resynchronization problem of an SDFG on multi-processors. [13] provides a heuristic to optimize latency under a throughput constraint. [19] provides bounds on maximum latency for jobs with different types of inputs. The techniques presented in the current paper allow us to investigate trade-offs between performance metrics and resource usage in general. We focus on throughput and extend [12] by relaxing the self-time scheduling constraint and by allowing buffer sharing.

Previous work on trade-off analysis of SDFGs is mostly limited to single processor platforms [6], [31]. [6] explores the trade-off between code size and data memory. [31] gives a CD2DAT example to show the trade-off between code, data memory and execution time for SAS, based on an evolutionary algorithm. Only recently, trade-offs for SDFGs on multiprocessor platforms are investigated [26]. [26] gives an exact method to explore the trade-off between total buffer size and throughput for multiprocessor platforms based on techniques taken from [11], [12]. [28] extends it to include cyclo-static dataflow graphs and provides a fast approximation algorithm to tackle graphs with many similar Pareto points. Our work generalizes [26] with respect to SDFG analysis by extending to multiple objectives and by relaxing assumptions on scheduling and resource models.

[17] provides a design-space exploration (DSE) framework for multiprocessor systems-on-chip based on SDFG specifications. The framework focusses on a single objective, the makespan of an SDFG, and the SDFGs are limited to HSDFGs without cyclic dependencies.

Model checking [3], [7] is widely used in system verification such as hardware verification and protocol verification. Recently it is also used for scheduling and scheduling related problems [2], [1], [11], [15]. However, multi-objective model checking is only studied recently and is limited to qualitative property verification [10] for stochastic models. Those techniques cannot be applied to trade-off analysis between resources and performance for SDFGs. [24] incorporates a SAT solver, a model checking technique, with an evolutionary algorithm for DSE of a task-graph model and uses list scheduling to find a feasible schedule. Our paper generalizes the trade-offs analysis of SDFGs as a multi-objective model-checking problem and tries to prune the state space by leveraging knowledge from both dataflow models and multi-objective optimization.

III. Resource-aware SDF Model

Formally, an SDFG is defined as follows. We assume a set Ports of ports, and with each port $p \in \mathit{Ports}$ we associate a finite rate $\mathit{Rate}(p) \in \mathbb{N}\backslash\{0\}$ (where we assume that $0 \in \mathbb{N}$). An actor a is a tuple $(\mathit{In}, \mathit{Out})$ consisting of a set $\mathit{In} \subseteq \mathit{Ports}$ of input ports (denoted by $\mathit{In}(a)$), a set $\mathit{Out} \subseteq \mathit{Ports}$ ($\mathit{Out}(a)$) with $\mathit{In} \cap \mathit{Out} = \varnothing$

An SDFG is a tuple (A, C, τ) with a finite set A of actors, a finite set $C \subseteq \mathit{Ports}^2$ of channels and a mapping $\tau : A \mapsto \mathbb{N}$. The source of every channel is an output port of some actor; the destination is an input port of some actor. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actor. For every actor $a = (I, O) \in A$, we denote the set of all channels that are connected to ports in I (O) by $\mathit{InC}(a)$ ($\mathit{OutC}(a)$). The mapping $\tau : A \mapsto \mathbb{N}$ assigns to each actor $a \in A$ the time it takes to execute the actor once, i.e., its execution time.

When an actor a starts its firing, it consumes $\mathit{Rate}(q)$ tokens from all $(p, q) \in \mathit{InC}(a)$. After time has progressed by $\tau(a)$, the actor finishes its firing and produces $\mathit{Rate}(p)$ tokens on every $(p, q) \in \mathit{OutC}(a)$. For distribution of tokens on channels, we define the following concept.

A channel quantity on the set C of channels (representing for instance the number of tokens) is a mapping $\delta : C \mapsto \mathbb{N}$. If δ_1 is a channel quantity on C_1 and δ_2 is a channel quantity on C_2 with $C_1 \subseteq C_2$, we write $\delta_1 \preceq \delta_2$ if and only if for every $c \in C_1$, $\delta_1(c) \leq \delta_2(c)$. $\delta_1 + \delta_2$ and $\delta_1 - \delta_2$ are defined by pointwise addition of δ_1 and δ_2 and substraction of δ_2 from δ_1; $\delta_1 - \delta_2$ is only defined if $\delta_2 \preceq \delta_1$.

The amount of tokens read at the start of a firing of some actor a now can be described by a channel quantity $\mathit{Rd}(a) = \{((q,p), \mathit{Rate}(p)) \mid (q,p) \in \mathit{InC}(a)\}$, produced tokens by channel quantity $\mathit{Wr}(a) = \{((p,q), \mathit{Rate}(p)) \mid (p,q) \in \mathit{OutC}(a)\}$.

SDFGs with rates which lead to deadlocks or an unbounded amount of tokens on some of its channels are called *inconsistent*. Consistency [18] is known as a necessary condition to allow a deadlock-free execution of SDFG within a bounded channel quantity on all channels.

Definition 1: (Repetition Vector, Consistency) A repetition vector γ of an SDFG (A, C, τ) is a function $\gamma : A \mapsto \mathbb{N}$ such that for each channel $(p, q) \in C$ from actor $a \in A$ to $b \in A$, $\mathit{Rate}(p) \cdot \gamma(a) = \mathit{Rate}(q) \cdot \gamma(b)$ (called a *balance equation*). A repetition vector is called non-trivial if and only if $\gamma(a) > 0$ for all $a \in A$. An SDFG is called *consistent* if and only if it has a non-trivial repetition vector. A consistent, connected SDFG has a unique smallest non-trivial repetition vector, which is designated as *the repetition vector* of the SDFG.

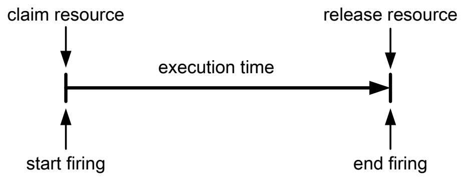


Fig. 4. Firing of an Actor

Since consistency is easy to check, we only consider consistent SDFGs. Further, we assume connectedness of SDFGs.

A resource-aware SDFG extends an SDFG by annotating actors with resource requirements. In order to describe the amount of resources, we define a concept similar to the channel quantity.

Definition 2: (Resource Quantity) A *resource quantity* on a set R of resources is a mapping $\eta : R \mapsto \mathbb{N}$. If η_1 and η_2 are resource quantities, the relation $\eta_1 \preceq \eta_2$ and operators $\eta_1 + \eta_2$ and $\eta_1 - \eta_2$ are defined similar to channel quantities.

The amount of resources claimed by some actor a can now be described by a resource quantity $\mathit{Clm}(a)$; released resources by resource quantity $\mathit{Rel}(a)$. We conservatively assume that resources are claimed and released at firing start and end, respectively (see Fig. 4).

Definition 3: (Resource-Aware SDFG) A *resource-aware SDFG* is a tuple $(A, C, \tau, R, R_C, \mathit{Clm}, \mathit{Rel})$ consisting of an SDFG (A, C, τ), a finite set R of resources, a resource quantity R_C denoting resource limitations, a mapping $\mathit{Clm} : A \mapsto (R \mapsto \mathbb{N})$ and a mapping $\mathit{Rel} : A \mapsto (R \mapsto \mathbb{N})$. The mappings Clm and Rel associate a resource quantity to each $a \in A$, which denotes the resources it claims and releases at the start and end of its firing, respectively.

As with inconsistent rates for tokens on channels, it is possible that inappropriate resource claims and releases of a resource-aware SDFG lead to deadlock or unbounded resource accumulation. Therefore, resource consistency is a necessary condition for a meaningful analysis.

Definition 4: (Resource Consistency) A resource-aware SDFG is *resource consistent* if and only if it is consistent and its repetition vector γ satisfies the following resource balance equation: $\sum_{a \in A} \mathit{Clm}(a) \cdot \gamma(a) = \sum_{a \in A} \mathit{Rel}(a) \cdot \gamma(a)$.

Resource consistency is also straightforward to check. In the remainder of this paper, we therefore only consider resource-consistent SDFGs.

A *state* of a resource-aware SDFG (A, C, τ, R, R_C, Clm, Rel) is a triple (δ, η, υ). Channel quantity δ associates with each channel the amount of tokens present in that channel in that state. Resource quantity η associates with each resource $r \in R$ the amount *used* of that resource in that state. To keep track of time progress, actor status $\upsilon : A \mapsto \mathbb{N}^{\mathbb{N}}$ associates with each actor $a \in A$ a multiset of numbers representing the remaining times of different active firings of a. We assume that the initial state of a resource-aware SDFG is given by some initial token distribution δ_0, initial resource usage η_0 (not necessarily zero) and no actor firing, which means the initial state equals $(\delta_0, \eta_0, \{(a, \{\}) \mid a \in A\})$ (with $\{\}$ the empty multiset).

The use of a multiset of numbers to keep track of actor progress allows multiple simultaneous firings of the same actor (auto-concurrency). By adding self-loops to actors with a number of initial tokens equivalent to the desired maximal auto-concurrency degree, the auto-concurrency can be limited.

The dynamic behavior of a resource-aware SDFG is described by *transitions*. There are three types of different transitions: start of actor firings, end of actor firings, and time progress through clock ticks.

Definition 5: (TRANSITION) A *transition* of a resource-aware SDFG $(A, C, \tau, R, R_C, Clm, Rel)$ from state $(\delta_1, \eta_1, \upsilon_1)$ to state $(\delta_2, \eta_2, \upsilon_2)$ is denoted by $(\delta_1, \eta_1, \upsilon_1) \xrightarrow{\beta} (\delta_2, \eta_2, \upsilon_2)$ where label $\beta \in \{A \times \{start, end\}\} \cup \{clk\}$ denotes the type of transition.

- Label $\beta = (a, start)$ corresponds to the firing start of actor $a \in A$. This transition results in $\delta_2 = \delta_1 - Rd(a)$, $\eta_2 = \eta_1 + Clm(a)$ and $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \uplus \{\tau(a)\}]$ (where $\uplus$ denotes multiset union). It may occur if $Rd(a) \preceq \delta_1$ and $Clm(a)+\eta_1 \preceq R_C$ and no end transition is enabled.
- Label $\beta = (a, end)$ corresponds to the firing end of actor $a \in A$. This transition results in $\delta_2 = \delta_1 + Wr(a)$, $\eta_2 = \eta_1 - Rel(a)$ and $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \setminus \{0\}]$ (where $\setminus$ denotes multiset difference). It is enabled if $0 \in \upsilon_1(a)$.
- Label $\beta = clk$ denotes a clock transition, which is enabled if no end transition is enabled. This transition results in $\delta_2 = \delta_1$, $\eta_1 = \eta_2$ and $\upsilon_2 = \{(a, \upsilon_1(a) \ominus 1) \mid a \in A\}$ (where $\upsilon_1(a) \ominus 1$ denotes a multiset of natural numbers containing the elements of $\upsilon_1(a)$, which are all positive, reduced by one).

In contrast with traditional SDFGs, due to resource constraints, not all start transitions with sufficient input tokens may actually be able to start simultaneously. There may exist multiple combinations of start transitions of actors with sufficient input tokens that can start at the same time and keep resource usage within resource constraints for the resulting states. Note that *end* transitions are not constrained by resources and are always executed eagerly.

An *execution* of a resource-aware SDFG is a finite or infinite alternating sequence of states and transitions: $\sigma = s_0 \xrightarrow{\beta 0} s_1 \xrightarrow{\beta_1} \cdots$ (not necessarily starting with the initial state of the SDFG). When the labels are not relevant, we also write $\sigma = s_0 s_1 s_2 \ldots$. We use $|\sigma|$ to denote the length of execution σ (the number of transitions); $|\sigma| = \infty$ if σ is infinite. We use σ^n to denote the execution up to and including state s_n (when $|\sigma| \geq n$), $t(\sigma)$ to denote the number of *clk* transitions in σ and $\sigma(i)$ to denote state s_i.

We make the following observations. *end* transitions have priority over other transitions. If a number of subsequent start transitions is taken, then the order in which they are taken has no impact on the resource usage or resulting state. When no more *start* transitions are selected, a *clk* transition occurs, possibly leading to new *end* transitions and so on. This means we can view an execution as a repetition of the following phases; (i) execute all enabled *end* transitions, (ii) execute some set of *start* transitions in arbitrary order, (iii) execute a single *clk* transition. Now it is easy to see that an execution can be fully characterized by a sequence of (possibly empty) multisets of actors that execute a *start* transition at all time instants. We call such a multiset $d_k \in \mathbb{N}^A$ of starting actor firings at time instant k a *decision*. For notational convenience, we call the set $\mathbb{N}^A$ of all multisets of actors, the set D of decisions. An execution is equivalently characterized by the sequence $d_k \in D$, $k \geq 0$.

Obviously, there are multiple, different executions, with different decisions. We can define rules to guide the execution to make decisions at those specific states and we call those rules a *scheduling policy*, defined as follows.

A *scheduling policy* is a function $\pi : D^* \mapsto 2^D$, which decides which of the enabled decisions are allowed to be selected. A policy is called deterministic if $\pi(\sigma)$ contains a unique decision for every finite execution σ.

Streaming applications are expected to continue executing indefinitely. Therefore, for our analyses, we are interested in infinite executions and their properties.

IV. EXPLORATION

In traditional state-space analysis of SDFGs, only throughput is considered as a performance metric. In this work, we want to consider multiple objectives of different types, such as peak resource usage for each of the resources of the graph, together with throughput of the graph. We limit ourselves to resource-aware SDFGs with a *finite* state space. This is typically the case, for example when the graph is strongly connected, or when every actor actually uses some resources, or when every channel in the graph has a finite buffer capacity. An important consequence is that every infinite execution necessarily revisits at least one state infinitely often.

While exploring the state space of a resource-aware SDFG for the trade-offs between metrics of interest, we want to prune suboptimal executions as much as possible. We consider two important classes of metrics. First, MAX quantities record the maximum values attained in any of the states along an execution, for instance peak memory usage.

Definition 6: (MAX QUANTITY) Given an execution σ of a resource-aware SDFG, $\sigma = s_0 s_1 s_2 \ldots$. Each state s_i is assumed to have a corresponding quantity of interest $q(s_i)$. We define $q(\sigma) = \max\{q(s_i) \mid 0 \leq i < |\sigma|\}$ to denote the quantity q for execution σ. An important property of this type of quantity is *monotonicity*, i.e., when $m \leq n$, then $q(\sigma^m) \leq q(\sigma^n)$. Without loss of generality, we assume that we are interested in executions with *minimal* values for MAX quantities.

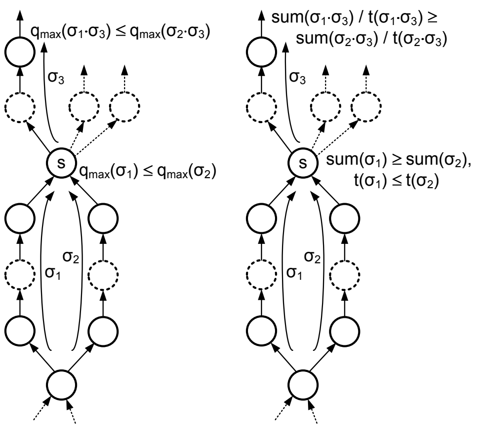
Note that, because the state space is finite, the maximum over an infinite execution is well-defined.

Peak resource usage can be defined as a set of MAX quantities. Given an execution of a resource-aware SDFG $\sigma = s_1 s_2 s_3 \ldots$, with $s_i = (\delta_i, \eta_i, \upsilon_i)$, the *resource usage* $Ru(\sigma) = \sup\{\eta_0, \eta_1, \eta_2, \cdots\}$ is the least upper bound of the resource quantities in all states in the execution. Each of the individual resources in $Ru(\sigma)$ corresponds to a MAX quantity. Our theoretical development in this section does not make any assumptions about the specific MAX quantities considered, but the experimental evaluation of Section VI considers resource usage.

Due to the monotonicity of MAX quantities, it is easy to prove the following proposition.

Proposition 1: Given two finite (partial) executions σ_1 and σ_2 that start from the same state and end in the same state. If $q(\sigma_1) \leq q(\sigma_2)$ for a MAX quantity q, then for any execution $\sigma_b = \sigma_2 \cdot \sigma_3$ (where $\cdot$ denotes concatenation), execution $\sigma_a = \sigma_1 \cdot \sigma_3$ has $q(\sigma_a) \leq q(\sigma_b)$.

Proof: Since $q(\sigma_1 \cdot \sigma_3) = \max(q(\sigma_1), q(\sigma_3)) \leq \max(q(\sigma_2), q(\sigma_3)) = q(\sigma_2 \cdot \sigma_3)$, it follows that $q(\sigma_a) \leq q(\sigma_b)$. ■



(a) MAX Quantity (b) AVG Quantity

Fig. 5. Pruning the execution space.

Fig. 5(a) illustrates Prop. 1. The proposition can be used to discontinue the exploration of (partial) execution σ_2 when arriving in state s.

Another important class of metrics are long-run time-averages, such as average resource usage or average throughput. We call them AVG quantities.

Definition 7: (AVG QUANTITY) Given an execution σ of a resource-aware SDFG, $\sigma = s_0 s_1 s_2 \ldots$. If q is an AVG quantity, then $q(\sigma) = \lim_{N \to |\sigma|} \frac{\sum_{i=0}^{N} q(s_i)}{t(\sigma^N)}$ if this limit exists and it is undefined otherwise. We assume we are interested in executions with *maximal* values for AVG quantities.

Note that, in general, for certain very irregular executions, the limit may not exist. It is easy to show however that for optimal schedules (with maximum AVG) the limit does exist.

For simplicity, we have defined MAX and AVG quantities as functions of the sequence of states only, but quantities can also be based on information in the transitions. However, formalizing this would only complicate notation.

The throughput $Th_a(\sigma)$ of an arbitrary actor a of an SDFG in an execution σ can now be defined as an AVG property, namely the long-run average number of *start* transitions of a in σ. Like in traditional SDFG throughput analysis, the throughputs of all actors are related through their firing ratios expressed by the repetition vector. As is common, we define a normalized notion of throughput for the graph as a whole. The throughput of an execution σ of a resource-aware SDFG with repetition vector γ is defined as $Th(\sigma) = \frac{Th_a(\sigma)}{\gamma(a)}$. As for MAX quantities, our analysis is not limited to throughput, but the experimental evaluation considers throughput.

AVG quantities do not have the monotonicity property of MAX properties. Which one of the various executions ending in the same state is better depends on the future execution sequence. In order to decide locally at any given state whether an execution is guaranteed to be better, we have to use more strict conditions.

Proposition 2: Given two finite (partial) executions σ_1 and σ_2 that start in the same state and end in the same state. If $\sum_{i=0}^{|\sigma_1|} q(\sigma_1(i)) \geq \sum_{i=0}^{|\sigma_2|} q(\sigma_2(i))$ and $t(\sigma_1) \leq t(\sigma_2)$ for an AVG quantity q, then for any execution $\sigma_b = \sigma_2 \cdot \sigma_3$, execution $\sigma_a = \sigma_1 \cdot \sigma_3$ has $q(\sigma_a) \geq q(\sigma_b)$.

Proof: Let $S_1 = \sum_{i=0}^{|\sigma_1|} q(\sigma_1(i))$ and $S_2 = \sum_{i=0}^{|\sigma_2|} q(\sigma_2(i))$. Then, $q(\sigma_a) = \lim_{N \to |\sigma_3|} \frac{S_1 + \sum_{i=0}^{N} q(\sigma_3(i))}{t(\sigma_1) + t(\sigma_3^N)} \geq \lim_{N \to |\sigma_3|} \frac{S_2 + \sum_{i=0}^{N} q(\sigma_3(i))}{t(\sigma_2) + t(\sigma_3^N)} = q(\sigma_b)$. ■

Prop. 2 is illustrated in Fig. 5(b). Also in this example, the exploration of execution fragment σ_2 can be terminated when arriving in state s. When considering MAX and AVG quantities simultaneously, the conditions in Prop. 1 and 2 need to be satisfied for all the respective quantities in order to discontinue the exploration of σ_2.

The ultimate goal of the exploration we are developing is to find Pareto-optimal executions, that capture the optimal trade-offs between the metrics of interest. In the end, we are particularly interested in *infinite* executions starting from the *initial state* of the SDFG. It is convenient however to define Pareto optimality for arbitrary executions (and for arbitrary metric spaces).

Definition 8: (PARETO-OPTIMAL EXECUTION) An execution σ_1 dominates another execution σ_2 if and only if σ_1 is not worse than σ_2 in any of the metrics of interest. An execution σ is Pareto optimal if and only if it is not dominated by any other execution. A Pareto-optimal execution with its metric values is called a *Pareto point*.

The following result follows immediately.

Corollary 1: Given a resource-aware SDFG with a metric space of arbitrarily many MAX and AVG quantities and two finite (partial) executions σ_1 and σ_2 that satisfy the conditions of Prop. 1 and 2 for all these quantities. Then σ_1 dominates σ_2.

Corollary 1 can be used to prune the search space when searching for Pareto points. However, the number of possibly Pareto-optimal executions is still extremely large. When the metric space has at most one AVG quantity, besides arbitrarily many MAX quantities, we can further prune the search space. It turns out that we can limit the search to so-called *simple executions.*

Definition 9: (SIMPLE EXECUTION) A simple execution is an infinite execution starting from the initial state of the SDFG that is composed of two parts: a finite length prefix execution σ_{pre}, not containing any duplicate states, and an infinite periodic repetition of execution σ_c that is a cycle that starts and ends in the final state of σ_{pre} and has no duplicate states either.

An important property of a simple execution $\sigma = \sigma_{pre} \cdot \sigma_c^\omega$ (with σ_c^ω denoting an infinite repetition of σ_c) is that the value of AVG quantities such as throughput is fully determined by the periodic part σ_c. For example, $Thr(\sigma) = Thr(\sigma_c)$. The values for MAX quantities are determined by the finite execution $\sigma_{pre} \cdot \sigma_c$; for example, $Ru(\sigma) = Ru(\sigma_{pre} \cdot \sigma_c)$. These observations are used in the proof of the following crucial proposition.

Proposition 3: Given an arbitrary infinite execution σ of a resource-aware SDFG, $\sigma = s_0 s_1 s_2 \ldots$, and a metric space consisting of one AVG quantity q and an arbitrary number of MAX quantities $p_1, p_2, \cdots, p_m$. Then, there exists a simple execution σ_s that dominates σ in this metric space.

Proof: (Sketch.) The states in σ can be divided into two sets: S_T and S_R, such that states in S_T are only visited finitely often while the states in S_R are visited infinitely often. As the number of states in S_T is finite, after a finite length of execution σ_{pre}, new state transitions only happen in S_R. The infinite path through states of S_R essentially consists of (possibly nested) repeated visits of simple cycles in the state space. As the number of the states in S_R is finite, the number of different simple cycles is also finite. Assume we find M_N simple cycles for $\sigma_N = s_0 s_1 s_2 \dots s_N$ after N transitions. From the property of AVG, we have $q(\sigma) = \lim_{N\to\infty} \frac{M_{N,1}\cdot q(\sigma_{c_1}) + M_{N,2}\cdot q(\sigma_{c_2}) + \cdots + M_{N,k}\cdot q(\sigma_{c_k})}{M_{N,1}+M_{N,2}+\cdots+M_{N,k}}$ where $\sum_{i=1}^{k} M_{N,i} = M_N$ and $M_{N,i}$ the number of complete visits of simple cycle σ_{c_i} after N states. So $q(\sigma) \leq q_{max}$, where $q_{max} = max(q(\sigma_{pre}), q(\sigma_{c_1}), \cdots, q(\sigma_{c_k}))$. Let $\sigma_s = \sigma_{pre1} \cdot \sigma_{c_{max}}^{\omega}$ be an execution such that σ_{pre1} is a prefix of σ, which eventually visits simple cycle c_{max} infinitely often, where c_{max} is the cycle with maximum property value q_{max}. Then $q(\sigma_{c_{max}}) = q_{max}$ and $q(\sigma_s) = q_{max} \geq q(\sigma)$. Furthermore, $p_i(\sigma_s) = max(p_i(\sigma_{pre1}),\, p_i(\sigma_{c_{max}})) \leq p_i(\sigma)$, where $p_i(\sigma) = max(p_i(\sigma_{pre}), p_i(\sigma_{c_1}), \cdots, p_i(\sigma_{c_n}))$. So, σ_s dominates σ. ■

Note that with multiple AVG quantities, trade-offs between these quantities can be achieved if two different schedules with different values for AVG quantities can be alternatively applied in arbitrary ratios. This is why the proof of Prop. 3 does not hold for this situation, and what complicates the analysis if multiple AVG quantities have to be considered.

From the above proposition, we know that we only have to consider simple executions, because for arbitrary executions, we can always find a simple execution that dominates it. So we can use a DFS based algorithm to find all simple cycles and use conditions from Prop. 1 and 2 to prune the search space during exploration. These observations form the basis of Algorithm 1.

Algorithm 1 DFS algorithm

```
Input: A Resource-aware SDFG $G$ with initial state $s_0$
Input: Resource constraints $R_C$ and Scheduling policies $\Pi$
Output: A set $P$ of Pareto points
procedure EXPLORE($G$, $s_0$)
    StateStack $Q = < s_0 >$
    StateSpace $S = \{s_0\}$
    while LENGTH($Q$)> 0 do
        $s_{curr} = Q.top()$
        $s_{next}$ =NEXTSTATE($s_{curr}$);
        if $s_{next} \notin Q$ then
            if $s_{next} \notin S$ then                    ▷ new state
                $Q.push(s_{next})$
                $S.insert(s_{next})$
            else                                     ▷ revisited state
                if MAYBEPARETOOPT($Q, s_{next}$) then
                    $Q.push(s_{next})$
                end if
            end if
        else                                            ▷ cycle found
            $p$ =COMPUTEQUALITYMETRICS($Q$)
            merge $p$ into Pareto set $P$
        end if
    end while
    return $P$
end procedure

procedure NEXTSTATE($s$)
    $F$=TOKENSENABLEDFIRINGS($s$)
    $R$=RESOURCECONSTRAINT($s$,$R_C$) ▷ avail. resources
    $D$=UNEXPLOREDDECISIONBRANCHES($s$,$F$, $R$, $\Pi$)
    if $D \neq \emptyset$ then
        $d$=SELECTONEDECISION($D$)
        $s_{next}$=TAKEDECISION($s$,$d$)
    else
        $Q.pop()$
        if $Q = \emptyset$ then
            finish exploration
        else
            $s = Q.top()$                             ▷ back-track
            $s_{next}$ =NEXTSTATE($s$)
        end if
    end if
    return $s_{next}$
end procedure
```

If we encounter a state which is already on the DFS stack, we have closed a simple cycle and we can analyse the cycle for its AVG quantity, store the result (if not dominated), and back-track. Moreover, if we encounter a state which is not on the DFS stack, but which we have visited before, then we check Pareto dominance of any of the previous visits of the state over the current visit (via Cor. 1). If it is dominated, we back-track; otherwise we have to revisit the state. It is easy to see that the approach terminates because there is only a finite number of states and states on the DFS stack cannot be revisited.

Theorem 1: Algorithm 1 finds all Pareto points given a metric space consisting of some number of MAX quantities and at most one AVG quantity.

Proof: (sketch) From Prop. 3, it follows that it is sufficient to explore only simple executions. This supports that the algorithm back-tracks as soon as a state is found that is already on the DFS stack. The second condition that the algorithm uses to back-track is when a state is found that has been explored before with properties that dominate the ones of the current path (based on Cor. 1). We show that this is sound.

Let c be a simple cycle, part of a Pareto-optimal simple execution. Consider all optimal simple executions ending in the cycle c with an optimal prefix in the following sense: σ is an optimal prefix to the cycle c if it is a simple path to c and for every prefix σ' of σ, σ' is an optimal path to its final state. Consider the first such optimal execution explored by the algorithm. Because the prefix is optimal, no back-tracking occurs during the exploration until a state s of c is reached. Then the exploration of c is followed until it is complete (returning to s) or a state $s' \neq s$ on c is found that has already been visited with dominating properties. Suppose the latter is true. Any state visited but not on the stack has already been fully explored. Since it is dominating and also a state of c, it was explored after the first state s on c was found. Hence, there exists a path from s to s' which is better than the part of c explored. This would mean that the cycle c is not optimal, a

TABLE I
OPTIONS FOR EXPLORATION

Design Domain	Option
Scheduling	stack size
	iteration number
	branch selection rule
	branching width
	partial order among actors
	channel quantities bounds
Resources	resources bounds
Search Algorithm	back-track step
	cycle count
	time limit

contradiction. Thus such a case cannot occur, cycle c is fully explored, and the optimal simple execution is found. ■

V. IMPLEMENTATION

In this section, we discuss the decisions we made for the implementation of our method. It is based on Alg. 1, but it implements several features to facilitate the exploration of large state spaces. The price to be paid of using these features is (potential) loss of optimality, but the result is a widely applicable, versatile tool.

We implemented our method in the SDF3 toolset [27]. Like in many model-checking tools, we use a hash table as the data structure to store the visited states for quick checking. However, to allow exploration of large state spaces efficiently, we cannot only depend on a good data structure. We have to limit the size of the explored state space to keep our tool fast while ensuring that the exploration can find enough interesting design points.

Fig. 3 introduces the concept that the exploration of the state space of an SDFG can be controlled by adjusting the resource and scheduling design domains. By providing options to configure the two design domains, we can guide the exploration to search different parts of the state space and try to find the design points with different quality metrics. The exploration options we implemented as configurable options in our heuristic search are listed in Table I. These options are divided into 3 groups: Scheduling options, Resource options, and Search-algorithm options.

In the scheduling domain, the length of schedules is often an important design constraint for embedded systems. Stack size limits the depth of the DFS algorithm and indirectly puts constraints on the length of schedules. For SDFG $G = (A, C, \tau)$ with repetition vector γ, an iteration is a set of actor firings such that for each actor $a \in A$, the set contains $\gamma(a)$ firings of a. By limiting the number of iterations to n, the schedule length is not longer than $n \sum_{a \in A} \gamma(a)$. The branch selection rule and branching width options are used together to limit the number of schedules explored. The branch selection rule attempts to choose the most interesting schedules and the branching width limits the number of branches explored. For example, we implemented a fairness rule to guide the exploration to ensure fairness in the firings of actors. The rule guides the algorithm to select branches based on a cost which is computed from the repetition vector γ of the SDFG. The cost of a decision d is $C(d) = \sum_{a \in A} d_a c_a$, with d_a the number of firing starts of actor a in decision d and $c_a = f_a/\gamma(a)$ the cost of one firing of actor a, where f_a is the

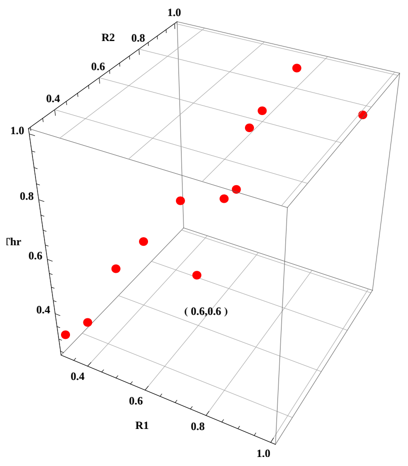


Fig. 6. Grid Exploration

accumulated firing count of actor a since the start of execution. The larger the cost of one actor, the more firings of one actor have been executed and the smaller the chance that the actor is selected for the next firing start. In this way, we ensure fairness in actor firings. The fairness rule can avoid that the algorithm selects actors greedily based on the order in the data structure when exploring the state space partially. Experiments show that the results with the fairness rule turned on are better than the results without it. Another scheduling constraint that can be defined is a (possibly partial) priority ordering among actors. When actors compete for the same resource, they will be assigned according to their priority. The search algorithm will only explore options that satisfy this priority order. Bounds on channel quantities can provide another constraint on the explored executions. The bounds limit the maximum number of tokens that can be stored in each channel. As the token distribution over channels can be viewed as the memory of the past execution, the bounds limit the depth of the memory and thus influence the future execution.

In the resources design domain, we use resource bounds to constrain the exploration. As said before, the resources of embedded system are typically highly constrained. The limited resources will influence the scheduling of an SDFG and result in different performance numbers. By setting different resource constraints, we can explore the state space only with interesting resource constraints.

The search algorithm itself can also be configured. To ensure that the search algorithm covers a large state space, we use the back-track step to avoid that the search becomes trapped into some local region. For example, if the back-track step is set to 5, the search algorithm back-tracks at least 5 states before it starts searching again. As the number of simple cycles can be very large, we also use a maximum cycle count as a termination condition to stop the search algorithm if the number of cycles found reaches the limit. A time limit for one exploration is another natural termination condition for designers to control the time budget of the exploration. We can explore the design space very quickly by combining the various options. For this purpose, we developed a grid search strategy, where the resource space of an SDFG is divided into a grid. Each grid point is explored with a fixed time budget, using the fairness rule to guide the exploration. Fig. 6 shows an example for an SDFG with two types of resources $R1$ and $R2$. And $(0.6, 0.6)$ is one of the grid points in the normalized resource space. This strategy has two advantages. First, it is scalable, as the exploration of the whole design space can be distributed to a multi-core system or PC clusters. Second, the total exploration time $T_t = \frac{(T_p+T_{ov})N_p}{N_{proc}}$ is controllable by the designer, where T_p is the time budget for one grid point and T_{ov} is the overhead for setup and cleanup of exploration (such as memory allocation and deallocation), N_p is the number of grid points and N_{proc} is the number of processors. For large state spaces, the overhead can be omitted as T_p dominates.

VI. Experimental Evaluation

To show the use and versatility of our tool, we perform two different sets of experiments on an Intel Core$^{\text{TM}}$2 at 2.2 GHz with 4GB of RAM. In the first experiment, we explore the trade-off between the throughput and the required buffer space for the channels of an SDFG when assuming that memory can be shared among channels. This trade-off is important for the efficient implementation of multimedia and signal processing applications on embedded systems. Earlier work [26], [28] has only explored the trade-offs when assuming that memory cannot be shared among channels. As there are no comparable tools for DSE of SDFGs, we choose to compare our results with [26]. Our experiment shows that we can efficiently explore this trade-off space and reduce memory use when compared to using a distributed memory model. In the second experiment, we apply our tool to a real-life industrial case study where the design space of the digital datapath of a professional printer is explored. We show that our tool allows to efficiently explore the design space of such a datapath.

A. *Throughput vs Memory Trade-offs*

The benchmark set for this experiment contains a modem [6], a satellite receiver [23] and a sample-rate converter [6] from the DSP domain and an MP3 decoder [26] and an H.263 decoder [26] both from the multimedia domain. We also use an example from [26] and the frequently used bipartite SDFG from [6]. For each of the SDFGs, we explore the trade-off between throughput and buffer memory requirement. The search parameters of our algorithm are set as follows. The range of the iteration number is from 1 to 3. The range of the branching width is from 2 to 3 and the fairness rule is used. The backtrack step range is from 1 to 2. The memory scan range is from the lower bound of [11] to the upper bound of [26] and is uniformly divided into 10 steps for our grid search. The time budget for each exploration is 1 second for the first part of the experiment and 60 seconds for the second part.

To the best of our knowledge, our tool is the first that allows to analyze the throughput-memory trade-off when memory can be shared among channels. The Pareto points found by our tool are more resource efficient compared to the Pareto points found by [26] which does not allow sharing memory among channels. To investigate the impact of sharing resources, we compare our results to the Pareto points found by [26]. The results of [26] are known to be optimal when memory cannot be shared. Though we cannot compare the results with the optimal results with shared memory, as they are not known, we compare our results with the experimental results when

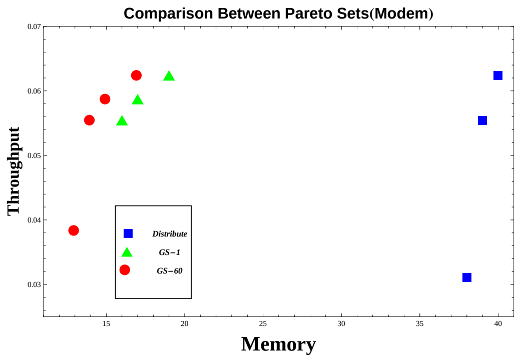


Fig. 7. Modem Pareto Points

exploring longer (60s) for each configuration. The comparison of the results shows that the results can be improved for some graphs by using a longer exploration time. However, the total time spent on the exploration is also increasing very quickly. In order to quantify the difference between various results, we define the average memory reduction MR_{avg} as a metric to compare a Pareto set S_{new} (our result) with a reference Pareto set S_{ref} (the result found by the algorithm in [26]). The memory reduction for each reference point $r \in S_{ref}$ is the maximal memory reduction of its counterpart $a \in S_{new}$ which has throughput $Th(a)$ not less than throughput $Th(r)$.

$$MR_{avg} = \frac{1}{|S_{ref}|} \sum_{r \in S_{ref}} \max_{a \in S_{new}} d(r, a) \tag{1}$$

where

$$d(r, a) = \begin{cases} mem(r) - mem(a) & Th(r) \leq Th(a) \\ 0 & Th(r) > Th(a) \end{cases} \tag{2}$$

The results of the experiments are summarized in Table II. It shows the number of actors and channels in each graph and the minimal buffer space for the smallest positive throughput, the maximal throughput that can be achieved. It also shows the number of Pareto points, the execution time of the tool and the statistical information about memory reductions achieved by sharing of memory. The 60 seconds results are shown in brackets if they are different from the 1 second results. Fig. 7 shows the Pareto points of the Modem model found by the two algorithms. Sharing memory reduces memory by more than 50% for this particular case. The results of Table II show that by sharing memory among actors, the required memory can be reduced from 10% to 50% in most cases. The fact that the minimally obtained resource reduction is positive in all cases shows that we can always achieve the same throughput as the throughput found by [26]. Although 60 seconds results are sometimes better than 1 second results, a 60 second budget results in much longer overall analysis times. The substantial memory reductions and obtained throughput results together with the analysis efficiency indicate that our techniques perform well. Fig. 8 shows the Pareto points of the H.263 decoder (QCIF frame size). The reason for the low average reduction in memory per Pareto point, is the large number of Pareto points found by the algorithm of [26] in the upper right corner of the graph, which are dominated by the one nearby Pareto point found by our tool by a small margin,

TABLE II
EXPERIMENTAL RESULTS OF BENCHMARKS

	Example[25]	Bipartite	Sample Rate	Modem	Satellite	MP3	H.263(QCIF)
actors/channels	3/2	4/4	6/5	16/19	22/26	13/12	4/3
Min.Throughput	1.25×10^{-1}	3.09×10^{-3}	$1.00(1.02)\times10^{-3}$	5.56×10^{-2}	7.60×10^{-4}	1.90×10^{-7}	1.52×10^{-6}
Min.BufferSize	4	26	23	16(13)	962	11	595
Max.Throughput	2.50×10^{-1}	3.97×10^{-3}	1.04×10^{-3}	6.25×10^{-2}	9.47×10^{-4}	2.68×10^{-7}	3.01×10^{-6}
Min.BufferSize	7	32	31	19(17)	1220	14	1190
Pareto points	4	7	7(5)	3(4)	3	3	3
Exec. time(min)	0.45(0.45)	1.17(2.55)	5.52(147)	2.34(24.7)	5.56(147)	5.18(95.5)	2.22(18.6)
Max. Memory Reduction	22.2%	13.3%	30.3%	57.9%(65.8%)	37.7%	50.0%	50.1%
Min. Memory Reduction	10.0%	7.1%	8.8%(28.2%)	52.5%(57.5%)	21.0%	46.2%	0.5%
Avg. Memory Reduction	14.9%	10.7%	22.4%(29.2%)	55.6%(61.6%)	29.4%	48.1%	3.2%
Std. Deviation	0.05	0.02	0.10(0.01)	0.02(0.03)	0.08	0.02	0.08

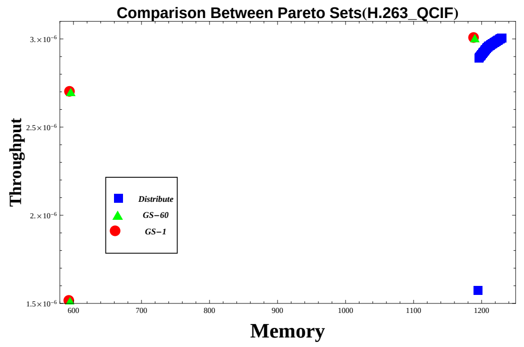


Fig. 8. H.263(QCIF) Pareto Points

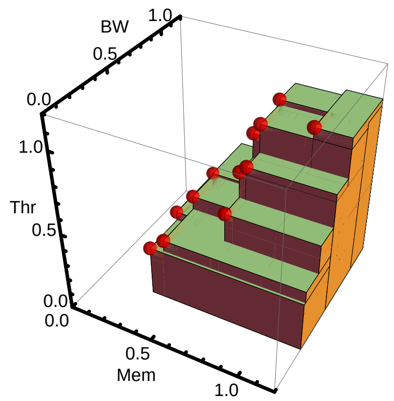


Fig. 9. Normalized Design Space of A Printer Use Case

while the single, lower throughput point is improved quite a lot. The average memory-reduction metric defined above does not capture this situation very well. The execution time of our method is reasonable, though it is longer than the execution time of the reference algorithm.

B. Printer Case Study

In this case study, we analyze the datapath of a professional printer, provided by Océ (www.oce.com). The processing units of the datapath share memory and the memory bus. Twelve use cases which are frequently seen in the daily use of a printer are investigated. We model these use cases as resource-aware dataflow graphs and analyze them with our tool.

The first set of experiments considers single-use-case analysis for one specific architecture configuration, so in this particular case, we are not looking for trade-offs, but to evaluate metrics in a particular design solution. The metrics we are interested in are the peak and average usage of resources in the datapath and the throughput of the datapath for those use cases. As the scheduling policy for these use cases is deterministic, there is only one simple execution for each use case. It is important to note that, because of this, we are not limited to only one AVG quantity, so we can consider both throughput and average resource usage. From Section 4, we know that the throughput and resource usage can be easily computed from the prefix and the periodic part of the execution. Table III shows the execution time of our algorithm and the number of states of the execution. For most of the use cases, the execution time of the algorithm is less than 1 second. The two exceptions are use cases with large but slightly different actor execution times that cause a periodic execution phase with a large number of states.

The second set of experiments concerns the design-space exploration of printer architectures. We study the trade-off among peak memory and bandwidth usage with performance (throughput), obtained by different schedules. By using our grid search method, we can get the profile of the design space of a specific architecture, which can help a system designer make decisions on questions like how much memory and how much bandwidth are needed for some specific performance requirement.

Fig. 9 shows the normalized 3-dimensional Pareto space in the design space of a particular use case for some platform configuration. Without giving the detailed analysis results, we illustrate how our method can be used for design space exploration. By considering options of adjusting the speed of processing units or adding additional processing unit instances, we explore three different platform configurations. From the analysis we see that in the first configuration, the system performance is limited by some image processing units. The first configuration is used as a reference. The second configuration increases speed of critical processing units by 30%. The third configuration adds additional image processing units to the platform. By increasing the speed of processing units in the second configuration, the maximal performance is also increased by 30%. However, the bottleneck does not change, and the platform cannot reach its maximal performance (that is determined by the scanner and printing components). In the third configuration, we add additional resource instances

TABLE III
ANALYSIS OF PRINTER USE CASES

UseCase No.	1	2	3	4	5	6	7	8	9	10	11	12
actors/channels	5/6	12/15	9/9	3/3	3/3	8/8	9/9	11/13	14/19	14/19	5/6	8/8
exec. time(s)	0.876	0.846	54.994	0.365	0.297	0.254	0.257	0.304	6.587	0.306	0.385	0.244
state count	2204	1888	12651	854	422	10	11	383	3470	346	946	10

to remove the system bottleneck identified by the previous two experiments and the maximal performance of the system is achieved. Finally, as different priorities of processing units might lead to different executions with different resource usage and performance, we investigate the sensitivity for different scheduling priorities in the system architecture. The exploration results show that some Pareto points in the design space can be achieved by giving the bottleneck units higher priorities. When resources are sufficient, priorities do not influence the performance.

The results of this case study show that our tool is sufficiently flexible to support design-space exploration. It allows to explore the trade-offs between several objectives, and to investigate scheduling policies for shared resources.

VII. CONCLUSIONS

In this paper, we consider the trade-off analysis problem for SDFGs as a multi-objective model-checking problem. Pareto dominance and SDFG-specific information are used to prune the search space. Some theoretical results are provided as foundation for the exploration of the state space. We implemented a highly scalable algorithm with many configuration options. Two case studies show that our tool can explore the design space very quickly while providing a good characterization of the available trade-offs. Future directions include theoretical studies to find more efficient ways to explore the state space, for example by utilizing the structure of SDFGs, and to prune the state space by model-checking techniques such as symmetry reduction and partial-order reduction.

REFERENCES

[1] Y. Abdeddaïm and O. Maler, "Job-shop scheduling using timed automata," in *Computer Aided Verification*, 2001, pp. 478–492.

[2] K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine, "A framework for scheduler synthesis," in *The Proc. of 20th IEEE Real-Time Systems Symposium.*, 1999, pp. 154–163.

[3] C. Baier and J. P. Katoen, *Principles of Model Checking*. The MIT Press, May 2008.

[4] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs," in *VLSI Signal Processing, VI*, 1993, pp. 188–196.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Dordrecht: Kluwer Academic Publishers, 1996.

[6] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal on VLSI Signal Process. Syst.*, vol. 21, no. 2, pp. 151–166, 1999.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[8] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 4, pp. 385–418, 2004.

[9] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system performance analysis," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 889–899, 1998.

[10] K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis, "Multi-objective model checking of Markov decision processes," in *Logical Methods in Computer Science*, vol. 4, no. 4, 2008, pp. 1–21.

[11] M. C. W. Geilen, T. Basten, and S. Stuijk, "Minimizing buffer requirements of synchronous dataflow graphs with model-checking," in *DAC'05 Proc*, 2005, pp. 819–824.

[12] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. D. Theelen, and M. R. Mousavi, "Throughput analysis of synchronous data flow graphs," in *ACSD'06 Proc*, IEEE 2006, pp. 25–34.

[13] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen, "Latency minimization for synchronous data flow graphs," in *DSD'07 Proc*, 2007, pp. 189–196.

[14] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing memory requirements in rate-optimal schedules," in *Proc. of Application Specific Array Processors*, 1994, pp. 75–86.

[15] Z. Gu, X. He, and M. Yuan, "Optimization of static task and bus access schedules for time-triggered distributed embedded systems with model-checking," in *DAC '07*. ACM, 2007, pp. 294–299.

[16] J. Hahn and P. H. Chou, "Buffer optimization and dispatching scheme for embedded systems with behavioral transparency," in *EMSOFT '07*. ACM, 2007, pp. 94–103.

[17] C. Lee, S. Kim, and S. Ha, "A systematic design space exploration of mpsoc based on synchronous data flow specification," *Journal of Signal Processing Systems*.

[18] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Comp.*, vol. 36, no. 1, pp. 24–35, 1987.

[19] O. Moreira and M. Bekooij, "Self-timed scheduling analysis for real-time applications," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, p. 14, 2007.

[20] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 177–198, 2001.

[21] P. K. Murthy and S. S. Bhattacharyya, *Memory Management for Synthesis of DSP Software*. Boca Raton, Florida: CRC Press, 2006.

[22] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proc. of Application-Specific Array Processors*, 1993, pp. 285–296.

[23] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *Int. Conf. on Acoustics, Speech, and Signal Processing, Proc*, 1995, pp. 2651–2654.

[24] T. Schlichter, M. Lukasiewycz, C. Haubelt, and J. Teich, "Improving system level design space exploration by incorporating sat-solvers into multi-objective evolutionary algorithms," in *Emerging VLSI Technologies and Architectures, 2006. IEEE*

[25] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York, NY, USA: Marcel Dekker, Inc., 2000.

[26] S. Stuijk, M. C. W. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC'06 Proc*, 2006, pp. 899–904.

[27] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free ," in *ACSD'06, Proc*. http://www.es.ele.tue.nl/sdf3, 2006, pp. 276–278.

[28] S. Stuijk, M. C. W. Geilen, and T. Basten, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. Comp.*, vol. 57, no. 10, pp. 1331–1345, 2008.

[29] W. Sung and S. Ha, "Memory efficient software synthesis with mixed coding style from dataflow graphs," *IEEE Trans. on VLSI Syst.*, vol. 8, no. 5, pp. 522–526, 2000.

[30] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication," *IEEE Real-Time and Embedded Technology and Applications Symposium*, vol. 0, pp. 183–194, 2008.

[31] E. Zitzler, J. Teich, and S. S. Bhattclcharyya, "Evolutionary algorithms for the synthesis of embedded software," *IEEE Trans. on VLSI Syst.*, vol. 8, no. 4, pp. 452–455, 2000.