Automated Bottleneck-Driven Design-Space Exploration of Media Processing Systems*

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, Netherlands
[2]Embedded Systems Institute, Eindhoven, Netherlands
Email: {y.yang, m.c.w.geilen, a.a.basten, s.stuijk, h.corporaal}@tue.nl

***Abstract*—Media processing systems often have limited resources and strict performance requirements. An implementation must meet those design constraints while minimizing resource usage and energy consumption. Design-space exploration techniques help system designers to pinpoint bottlenecks in a system for a given configuration. The trade-offs between performance and resources in the design space can guide designers to tailor and tune the system. Many applications in those systems are computationally intensive and can be modeled by a synchronous dataflow graph. We present a bottleneck-analysis-driven technique to explore the design space of those systems automatically and incrementally. The feasibility and efficiency of the technique is demonstrated with experiments on a set of realistic application models ranging from multimedia to digital printing.**

***Index Terms*—Synchronous dataflow, Design-space exploration, Bottleneck identification**

I. Introduction

Nowadays, system designers are challenged by the ever-increasing complexity of systems and the pressure for a shorter time-to-market. Designers are struggling to find a balanced design point that meets both resource and performance requirements. Design-Space Exploration (DSE) aims to provide designers with a profile of the system that shows all the trade-offs in the design space. Those trade-offs can help designers to dimension the system and to make important decisions on the system, e.g., the number of processors, the size of the memory, the bandwidth of the bus, and so on. However, the design spaces of those systems are normally very large and cannot be explored exhaustively. Bottleneck-driven DSE tries to explore the design space of the system efficiently by identifying resource bottlenecks and by exploring in specific directions, increasing the amount of bottleneck resources. Compared to exploration techniques that treat all resource dimensions equally, bottleneck-driven DSE can reduce the number of candidate design points to be explored considerably.

Although the identification of bottlenecks depends on the problem, the design flow of many bottleneck-driven DSEs can be seen as a specialization of the well known Y-chart method [1], [2]. Given a system design problem, design alternatives of the application and the architecture are represented via a set of parameters, and the metrics of interest are defined. Next, metrics are evaluated and the bottleneck parameters are identified. With this information, the system parameters are tuned. By iterating these steps, the design space can be explored.

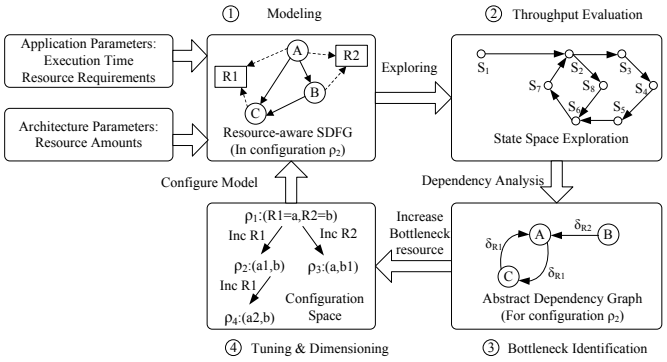


Fig. 1. Bottleneck-driven DSE for Resource-aware SDFG

Media processing systems, from mobile phones to digital printers, perform computations usually in a streaming way. Applications are frequently modeled by Synchronous Dataflow Graphs (SDFGs, [3]), and the systems normally have highly constrained resources (memory footprint, bandwidth, etc.). Once the mappings of the applications onto the components of the architecture template has been decided, it remains to dimension the architecture. In this paper, we develop a bottleneck-driven DSE flow for platform dimensioning of those systems that are modeled by an extended SDFG model: resource-aware SDFG [7]. Through the bottleneck-driven DSE, designers can obtain a design-space profile for the system and choose among the trade-offs in the profile.

Fig. 1 shows the overview of the flow. Given an application with a number of tasks with known execution times and resource requirements for a given platform with a number of resources, we use a resource-aware SDFG to model the system for a given mapping. For this paper, we choose throughput as the metric of interest. As the throughput cannot be computed directly from formulas, we execute the model and evaluate the throughput from the state space of the model. Through the analysis of the state space, we construct the abstract dependency graph, that captures the dependency of task executions on the availability of resources, and identify the resource bottlenecks in the system. We then increase the amount of one or more of the bottleneck resources and iterate the above steps. For example, for configuration ρ_2 in Fig. 1, we deduce that progress of A and C cyclically depends on availability of resource R1. Given that R1 thus is a potential bottleneck, we increase the amount of R1 to generate a new configuration ρ_4 which potentially has a better performance.

*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.

We use configuration ρ_4 to configure the resource-aware SDFG and repeat the above steps automatically until the design space has been explored.

The paper is structured as follows. Sec. 2 introduces SDFGs and their resource-aware extension (steps 1, 2 in Fig. 1). Sec. 3 discusses bottleneck identification (step 3), while Sec. 4 presents the exploration of the resource configuration space (step 4). An experimental evaluation is given in Sec. 5. Sec. 6 discusses related work and Sec. 7 concludes.

II. Resource-aware SDFGs

The top part of Fig. 2 shows a simple example SDFG. The circular nodes are called *actors* and represent computations. Actor computation is atomic and its *execution time* is constant. Actor names and execution times are denoted inside the nodes. Actors transfer information to each other on FIFO *channels* (solid directed edges) via data items called *tokens* (black dots). An essential property of SDFGs is that every time an actor *fires* (executes), it consumes the same amount of tokens from its input ports and produces the same amount of tokens onto its output ports. These amounts are called the *rates*, and are attached to the ports in the figure.

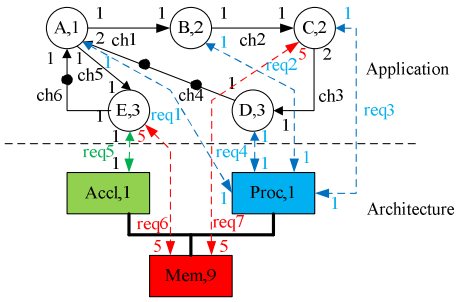


Fig. 2. Example Resource-aware SDFG

We assume a set $Ports$ of ports, and with each port $p \in Ports$ we associate a rate $Rate(p) \in \mathbb{N}\backslash\{0\}$. An SDFG is a tuple (A, C, τ) with a finite set A of actors, a finite set $C \subseteq Ports^2$ of channels and a mapping $\tau : A \mapsto \mathbb{N}$ that assigns to each actor $a \in A$ the time it takes to execute once, i.e., its execution time. 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. When an actor a starts its firing, it consumes $Rate(q)$ tokens from all input ports q. After time has progressed by $\tau(a)$, the actor finishes its firing and produces $Rate(p)$ tokens on every output port p.

Efficient algorithms to compute the throughput [4] and the required channel capacities of an SDFG [5], [6] exist. Channel capacities can be modeled in the SDFG [5], but it is hard to directly model other resources and handle situations such as resources shared by more than two actors. [7] extends SDFGs to resource-aware SDFGs to explore the trade-offs between multiple resources and throughput. We develop a bottleneck-driven analysis technique for resource-aware SDFGs.

For example, the SDFG in Fig. 2 is extended to a resource-aware SDFG by taking the architecture and the resource requirements into account. Rectangular blocks are added to model the resources: one processor $Proc$, one accelerator $Accl$ and one shared memory Mem with capacity 9. The dashed edges between the actors and the resources capture the resource requirements of actors. Actors A, B, C, D are mapped to $Proc$ and actor E is mapped to $Accl$. When C and E start firing, they claim 5 units of memory; when they end firing, they release 5 units of memory. For each actor, we annotate the amount of claimed and released resources at the actor resp. resource side of the corresponding requirement edge. We conservatively assume that resources are claimed and released at firing start and end, respectively.

Definition 1: (Resource-Aware SDFG) A *resource-aware SDFG* is a tuple $(A, C, \tau, R, Req, \rho)$ consisting of SDFG (A, C, τ), finite set R of resources, finite set $Req \subseteq A \times R$ of requirement edges, and resource configuration $\rho : R \mapsto \mathbb{N}$, denoting the available amount of resources. For each resource requirement edge $(a, r) \in Req$, $Clm(a, r)$ gives the amount of r claimed at the firing start of a and $Rel(a, r)$ gives the amount of r released at the firing end of a.

A *state* of a resource-aware SDFG $(A, C, \tau, R, Req, \rho)$ is a triple (δ, η, υ). Mapping δ associates with each channel the amount of tokens present in that channel in that state. Mapping η, called a *resource quantity*, 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). For example, in Fig. 2, the initial state $s_0 = (\langle 0, 0, 0, 2, 0, 1\rangle, \langle 0, 0, 0\rangle, \{(A, \{\}), (B, \{\}), (C, \{\}), (D, \{\}), (E, \{\})\}$.

An execution $\sigma = s_0 s_1 s_2 \cdots$ of a system is captured by a sequence of state transitions of the resource-aware SDFG model. For example, when actor A in Fig. 2 starts firing, it consumes the initial tokens on $ch4$ and $ch6$, claims processor $Proc$ and its state goes from the initial state s_0 to state $s_1 = (\langle 0, 0, 0, 0, 0, 0\rangle, \langle 1, 0, 0\rangle, \{(A, \{1\}), (B, \{\}), (C, \{\}), (D, \{\}), (E, \{\})\}$. Then the time advances for the minimal remaining execution time of any of the active actors, 1 time unit in this example. Actor A then ends firing, produces 1 token on both $ch1$ and $ch5$ and releases $Proc$, etc. Resource conflicts (e.g. C and E compete for Mem) and scheduling freedom (e.g. the firing order of B and E) for a system allow multiple possible transitions and lead to different executions of the system. Fig. 3 shows the state space of the example under resource configuration $\rho = \langle 1, 1, 9\rangle$.

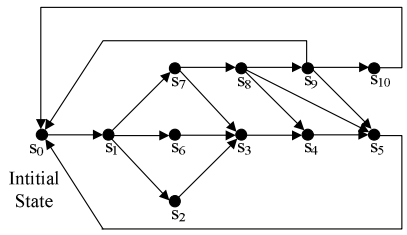


Fig. 3. State space of the example of Fig. 2

In general, during a state transition from one state to another state, some actors end firing, produce tokens on channels and

release occupied resources while some other actors start firing, consume tokens and claim required resources.

As the amount of resources ρ of a resource-aware SDFG is finite, the state space of a resource-aware SDFG is also finite. After a finite number of transitions, an infinite execution will revisit some states infinitely often.

Definition 2: (SIMPLE EXECUTION) An infinite execution σ is a *simple execution* if and only if it is of the form $\sigma = \sigma_{pre} \cdot \sigma_{per}^{\omega}$, with a finite prefix execution σ_{pre} followed by a repetition of finite execution σ_{per}.

Fig. 3 shows multiple simple executions, e.g. $\sigma_1 = (s_0 s_1 s_2 s_3 s_4 s_5)^{\omega}$ and $\sigma_2 = (s_0 s_1 s_7 s_8 s_9 s_{10})^{\omega}$ both with empty prefix.

A simple execution generates a finite schedule consisting of a prefix schedule and a periodic schedule. Moreover, the performance of the execution, i.e. the throughput, depends on σ_{per} [4], and can be computed efficiently (step 2 in Fig. 1). Given an execution σ, the resource usage is the least upper bound of the resource quantities along the execution. If σ is simple, $\sigma = \sigma_{pre} \cdot \sigma_{per}^{\omega}$, the resource usage is the maximum of the resource quantities between σ_{pre} and σ_{per}. Given the nice properties of simple executions, for efficiency reasons, we limit our attention to simple executions. Different simple executions for a given resource configuration ρ of a resource-aware SDFG have different throughputs and resource usages.

After throughput evaluation of the resource-aware SDFG with configuration ρ, we use dependency analysis to identify the bottlenecks of the resource-aware SDFG with the given ρ.

III. BOTTLENECK IDENTIFICATION

The maximal throughput of a system may be limited by the amount of available resources (e.g. the number or speed of processors, the size of memory, the bandwidth of a bus). In [5], [6], a dependency graph and its abstract dependency graph are introduced to capture the dependencies on channel capacities between actor firings of an SDFG. The dependencies are used to analyze the bottlenecks in channel capacities. In this paper, we adapt those concepts to analyze resource-aware SDFGs. In an execution, for example, one or more actors may not start firing while waiting for the other running actors to end firing, so that they can claim the released resources. Increasing the amount of resources may enable the waiting actors to start firing earlier and possibly increase the throughput. We would like to detect such situations as indications of a potential bottleneck. The dependency of the start of firing of an actor on a resource released or tokens produced by the end of firing of another actor is called a *causal dependency*.

Definition 3: (CAUSAL DEPENDENCY) A firing of actor a causally depends on the firing of actor b if and only if the firing of a claims resources or consumes tokens that are released or produced by the firing end of b without any time progress between the firing start of a and the firing end of b.

The *causal dependencies* can be classified into two types, *channel dependencies* and *resource dependencies*.

Definition 4: (CHANNEL DEPENDENCY) A causal dependency caused by producing and consuming tokens on channel $c \in C$ is a *channel dependency*, denoted by δ_c, or $\delta_c(x, y)$ to make the involved firings or actors x and y explicit.

Definition 5: (RESOURCE DEPENDENCY) A causal dependency caused by releasing and claiming resource $r \in R$ is a *resource dependency*, denoted by δ_r, or $\delta_r(x, y)$ to make the involved firings or actors x and y explicit.

A *channel dependency* only exists between two actors that connect to the same channel, and can thus be easily detected. The detection of a *resource dependency* depends on how the resource is shared. If it is only shared between two actors, it can also be detected easily. For a resource shared by more than two actors, the resource dependencies cannot be easily detected as we do not know exactly how released resources are distributed among the waiting actors. Given a resource $r \in R$ that is shared by more than two actors and a time instance t, the producer set A_p contains firings that end and produce the resource at t and the consumer set A_c contains firings that start and consume r at t. Without loosing any resource dependencies, we assume that the resource dependency exists between every firing in A_p and every firing in A_c if and only if the available amount of r, $R_{avail}(r)$, at the time t before release is less than the total amount of r that is claimed, i.e. $\delta_r(p, c)$, exists for every pair $(p, c) \in A_p \times A_c$ if and only if $R_{avail}(r) < \sum_{c \in A_c} Clm(c, r)$ (assuming the obvious interpretation of Clm for firings). This assumption simplifies the detection of resource dependencies, but it also introduces false dependencies, that are discussed later in detail.

Causal dependencies between actor firings can be captured as follows.

Definition 6: (CAUSAL DEPENDENCY GRAPH) Given a simple execution $\sigma = \sigma_{pre} \cdot \sigma_{per}^{\omega}$ of a resource-aware SDFG, the causal dependency graph (D, E) contains a node $a_k \in D$ for the k-th firing a_k of the actor $a \in A$ in σ_{per} and the set of dependency edges E contains an edge $\delta_c(a_k, b_l)$ or $\delta_r(a_k, b_l)$ if and only if there exists a causal dependency for channel c or resource r between firings a_k and b_l.

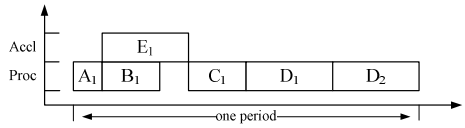
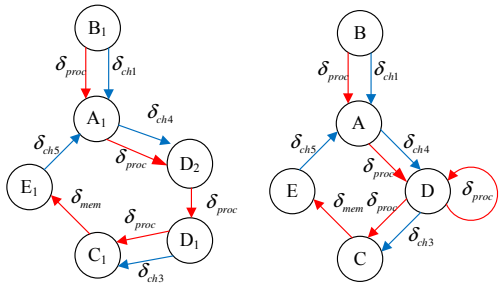


Fig. 4. An execution fragment of the example of Fig. 2



(a) Dependency Graph (b) Abstract Dep. Graph

Fig. 5. Dependency Graph and Abstract Dependency Graph

For the execution period of the example of Fig. 2 given in Fig. 4, the dependency graph is given in Fig. 5(a).

Dependencies between actor firings in the σ_{per} of a simple execution σ can form cyclic dependencies, called a *causal dependency cycle*. For example, in Fig. 5(a), dependency edges

$\delta_{proc}(A_1, D_2)$, $\delta_{proc}(D_2, D_1)$, $\delta_{proc}(D_1, C_1)$, $\delta_{mem}(C_1, E_1)$ and $\delta_{ch5}(E_1, A_1)$ form a causal dependency cycle.

Definition 7: (CAUSAL DEPENDENCY CYCLE) A *causal dependency cycle* is a simple cycle in the dependency graph.

The throughput of a simple execution is limited by some of those cycles in its dependency graph. For example, the throughput of the execution in Fig. 4 is limited by the dependency cycles that contain $\delta_{mem}(C_1, E_1)$. If a resource dependency δ_r appears in a causal dependency cycle, the throughput may increase if we can remove the dependency by increasing the amount of r, e.g. the increase of Mem allows E and C to fire at the same time. Resource dependencies not in cycles are not critical to the performance. For example, $\delta_{proc}(B_1, A_1)$ is not on a dependency cycle; it is not critical to the throughput of the execution in Fig. 4. If we delay B_1 one time unit, the throughput remains the same.

Definition 8: (BOTTLENECK) A resource $r \in R$ in a resource-aware SDFG is a bottleneck of execution σ under configuration ρ if and only if increasing r in configuration ρ is needed for an increase of the throughput of σ.

Unfortunately, the size of the dependency graph can be very large. For example, the minimal number of actor firings in one period of the sample rate graph of [8] is 612. Therefore, we use an abstract representation of the dependency graph to capture all dependencies between the firings in σ_{per}.

Definition 9: (ABSTRACT DEPENDENCY GRAPH) Given a causal dependency graph (D, E), the abstract causal dependency graph (D_a, E_a) contains a node $d_a \in D_a$ for each actor $a \in A$ and a dependency edge $\delta(d_a, d_b) \in E_a$ for each dependency edge $\delta(a_k, b_l) \in E$.

Fig. 5(b) shows the abstract dependency graph of Fig. 5(a).

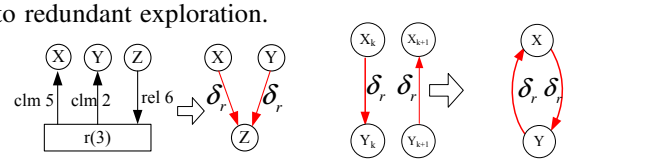
By construction, any dependency cycle in the dependency graph gives a dependency cycle in the abstract graph. Dependency edges related to bottleneck resources thus appear at least once in a dependency cycle of the abstract dependency graph. We can therefore detect resource dependencies in the abstract graph to identify potential bottlenecks (step 3 in Fig. 1).

Proposition 1: If a resource r is a bottleneck, then the abstract dependency graph has a dependency cycle containing a resource dependency for r.

Three items remain to be discussed. First, causal dependencies are defined based on firings. However, an execution may enter into a deadlock state, where no actor is able to fire. We need to redefine the causal dependency concept in this case. With this adapted definition, an abstract dependency graph can be derived as before.

Definition 10: (CAUSAL DEPENDENCY IN DEADLOCK) In a deadlock state, a firing of actor a causally depends on a firing of actor b if and only if the firing of a needs tokens that may be produced by or resources that may be released by a firing of actor b.

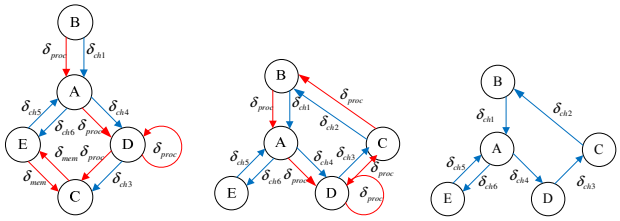
Second, the dependency assumption in the shared resource case and the use of the abstract dependency graph instead of the dependency graph can lead to false dependencies. Fig. 6 shows two examples. In Fig. 6(a), firing start X claims 5 units of resource r and firing start Y claims 2 units; the available amount is 3 units before firing end Z releases 6 units. If Y uses the available amount of r and X uses the amount of r released by Z, then only X depends on Z and the dependency between Y and Z is false. However, in our definition, X and Y both depend on Z. In Fig. 6(b), there is no cycle in the dependency graph, but a false dependency cycle exists in the abstract dependency graph. The false dependencies in the abstract dependency graph can cause some non-bottleneck resources to be detected as bottleneck resources and may lead to redundant exploration.



(a) False Resource Dependency (b) False Dependency Cycle

Fig. 6. False Dependencies

Third, multiple simple executions can exist in the state space of a resource-aware SDFG, each with an abstract dependency graph. Often, bottleneck resources are the same. For efficiency, we build a dependency graph that merges all dependencies into one graph. This may again lead to false dependency cycles. Again, real bottleneck resource dependencies always exist in the merged graph and the real bottlenecks are always detected. We gain efficiency by tolerating those false detections while keeping the exploration safe.



(a) $\rho = \langle 1, 1, 9 \rangle$ (b) $\rho = \langle 1, 1, 10 \rangle$ (c) $\rho = \langle 3, 1, 10 \rangle$

Fig. 7. Bottleneck Identification

IV. DESIGN SPACE EXPLORATION FLOW

From Sec. III, we know that bottleneck information of a resource-aware SDFG is embedded in its abstract dependency graph. We can compute the SDFG's maximal achievable throughput without resource limitations efficiently from its strongly connected components through state-space analysis [4]. In practice, we always perform a DSE within some resource ranges, for instance, given by cost. In [5], [6], channel-buffer bottlenecks of an SDFG are detected through an abstract dependency graph and used to guide the exploration of buffer configurations. In this paper, we also infer the potential resource bottlenecks and increase the amount of relevant ones incrementally until the graph reaches the maximal throughput or the maximal resource configuration that we want to explore. For example, in Fig. 7(a), Mem and $Proc$ are potential bottlenecks. We increase Mem from 9 to 10, and the Mem dependency disappears in Fig. 7(b). When increasing $Proc$ from 1 to 3, all resource dependencies disappear (Fig. 7(c)) and maximal throughput is reached.

Fig. 8 gives the DSE algorithm. It uses an initial resource configuration to configure a resource-aware SDFG and explores the state space of the configured graph. Upon detecting a cycle in the state space, throughput and resource usage

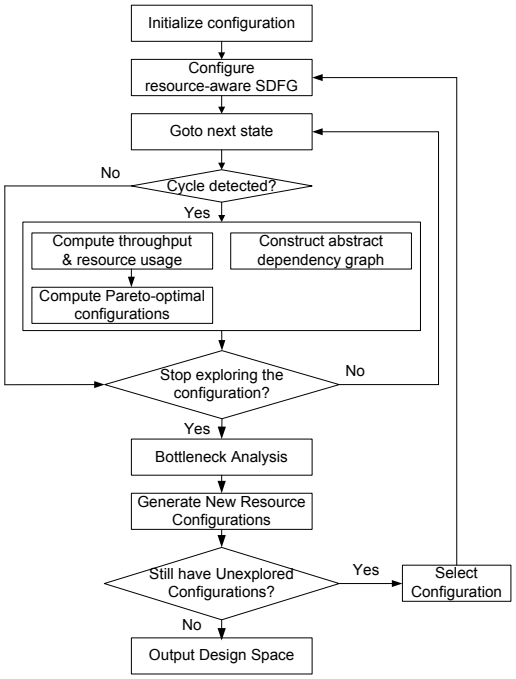


Fig. 8. Design-Space Exploration Flow

of the execution are computed. Pareto-optimal design points among all explorations are kept as the output of the DSE. When detecting a cycle, causal dependencies are added to the abstract dependency graph. As multiple cycles can exist in the state space, the abstract dependency graph is complete only after the state-space exploration stops. Bottleneck analysis is then performed by identifying resource dependencies in strongly connected components of the dependency graph. Identified bottleneck resources are each increased a minimal step, specified by the user. Thus, a new set of unexplored configurations is generated and pushed into the configuration queue. Breadth-first search is used to search the configuration space. Dynamic programming is used to avoid redundant explorations of configurations that have already been explored. The algorithm terminates if the configuration queue is empty.

V. Experimental Evaluation

To evaluate the DSE flow, we experiment with a set of DSP, multimedia and printer datapath models on an Intel 2.2 GHz Core^{TM}2 with 4GB RAM. Since exhaustive exploration of the configuration space is infeasible, we compare our algorithm with the approach in [7] that does not use bottleneck information but that samples the resource configuration space according to a grid and explores each grid point (configuration) separately. Through bottleneck analysis, we expect that many grid points do not need to searched by the bottleneck-driven DSE, reducing the exploration time. Our method has been implemented in the freely available SDF3 [9] tool.

In the first experiment, we compare the execution times of the approaches on six resource-aware SDFG models (adapted from six SDFGs by adding resource requirements). The set contains a modem [8], a satellite receiver [10] and a sample-rate converter [8] from the DSP domain and an MP3 decoder [5] and an H.263 decoder [5] from the multimedia domain. We also use the often used artificial bipartite graph from [8].

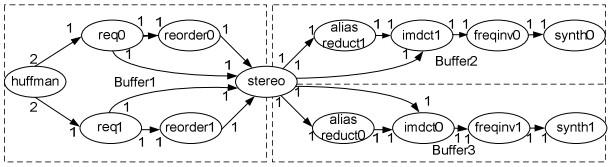


Fig. 9. MP3 Buffer Configuration

For each model, we explore the trade-offs between throughput and the size of one shared memory. As the state space for one specific SDFG is very large, we heuristically limit the state-space search. The memory (resource configuration) range is from the lower bound of [11] to the upper bound of [5] and it is uniformly divided into 10 steps. The left column for each model in Table I gives execution times and the numbers of explored configurations for both the grid (Non-BD) and bottleneck-driven search (BD).

The results show that bottleneck-driven DSE has two effects on the execution time. On the one hand, it avoids the exploration of some unnecessary resource configurations. On the other hand, the bottleneck analysis brings some overhead. For Bipartite, the two approaches explore the same configurations and the number of simple executions for each configuration is very large, so the bottleneck-driven approach is worse. For Modem, many unnecessary configurations can be avoided by bottleneck-driven DSE and the analysis overhead is more than compensated by the reduction in configurations. The other models give results in between these two extremes.

To test bottleneck analysis for multiple resources and large configuration spaces, we did experiments with distributed memory. For MP3, for example, Fig. 9 shows the buffer sharing among actors for three different buffers. The right column for each model in Table I gives the results. Substantial reductions are obtained in all cases. As expected, the performance of bottleneck-driven DSE improves with increasing numbers of resources.

The second experiment is a printer case study provided by Océ (www.oce.com). We aim to dimension the memory and bus usage of three different printer architectures, one reference architecture, one with faster processing units, and one with additional processing units. Table II shows that bottleneck analysis reduces the number of explored configurations, and even if the overhead for bottleneck analysis is substantial (Arch 3), the overall execution time reduction is still good.

The configuration space of grid search with and without bottleneck analysis for the first printer architecture is shown in Fig. 10. The (blue) squares are configurations explored without bottleneck analysis. The (green) triangles are configurations explored with bottleneck analysis. Thanks to the bottleneck identification, exploration stops increasing specific resources if they are no longer a potential bottleneck of the system. The (red) circles show the resource usage of the found Pareto points. They do not coincide with grid points because the actual resource usage may be less than the configured amounts.

VI. Related Work

The Y-chart method [1], [2] is widely used to analyze embedded systems and as the basis for DSE [12], [13]. However, [12], [13] formulate the DSE problem as an integer programming problem and solve it with evolutionary

TABLE I
EXECUTION TIME COMPARISON

	Bipartite		Sample Rate		Modem		Satellite		MP3		H.263(QCIF)	
No. of Mem buffers	1	2	1	3	1	2	1	2	1	3	1	2
Conf No. Non-BD	168	286	288	1176	336	125	264	245	408	360	264	45
Exec Time Non-BD (s)	134.3	92.4	523.7	404.831	359.2	83.5	562.2	585.337	577.9	237.8	252.5	85.7
Conf No. BD	168	168	216	181	89	19	90	27	118	89	264	14
Exec Time BD (s)	181.9	86.1	496.5	89.3	116.8	16.0	207.9	62.92	235.3	145.6	259.3	19.5
Conf Reduction	0%	41%	25%	85%	73%	85%	66%	89%	71%	75%	0%	69%
Exec Time Reduction	-35%	6%	5%	78%	67%	80%	63%	89%	59%	39%	-3%	77%

TABLE II
PRINTER ARCHITECTURE COMPARISON

	Arch 1	Arch 2	Arch 3
Conf No. Non-BD	110	110	110
Exec Time Non-BD (s)	71.9.6	128.2	120.0
Conf No. BD	37	44	58
Exec Time BD (s)	40.2	78.3	100.0
Conf Reduction	66%	60%	47%
Exec Time Reduction	44%	39%	20%

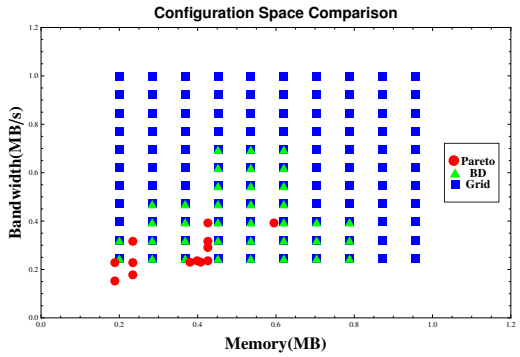


Fig. 10. Comparison between Configuration Spaces

algorithms. These approaches do not work for problems that cannot be formulated as an integer programming problem (e.g. throughput analysis for SDFGs). Bottleneck analysis is an important aspect of system performance analysis [14]. Work has been done on bottleneck analysis in areas such as system design [1], [2], hardware optimization [15], [16], network flow analysis [17], program optimization [18], and trade-off analysis [5], [6]. [18] analyzes program traces and constructs a dependency graph of program execution. By analyzing the dependencies in the critical path of the dependency graph, the bottleneck can be identified and performance improved. [5], [6] extract the dependency graph from the state space of an SDFG. By analyzing the critical cycles in the dependency graph, bottleneck buffers and trade-offs between buffer size and throughput are found. [5], [6] only allow distributed resources and only deterministic self-timed execution is possible. [7] propose a resource-aware SDFG model that allows generic resource analysis. Our work is inspired by [5], [6] and proposes a bottleneck-driven DSE technique for resource-aware SDFGs for automatic system dimensioning.

VII. CONCLUSIONS

We developed a bottleneck-driven DSE approach to explore the design space of a media processing system captured by a resource-aware SDFG. The approach guides the search by information collected during the evaluation of metrics of interest. Experimental results show that, for systems with multiple resources and large configuration spaces, the bottleneck-driven approach saves up to 89% in analysis time compared to a brute-force approach. Future directions include applying the bottleneck analysis to resource-aware SDFGs with specific scheduling strategies, and analyzing the sensitivity of bottleneck analysis to task execution time variation. The approach is not necessarily limited to resource-aware SDFGs and can potentially be applied to other models of computation. It can also be combined with mapping exploration techniques to explore an even larger part of the overall design space.

REFERENCES

[1] F. Balarin, *et al.*, *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer, 1997.

[2] B. Kienhuis, *et al.*, "An approach for quantitative analysis of application-specific dataflow architectures," in *Application-Specific Systems, Architectures and Processors 1997 Proc, IEEE*, 1997, pp. 338–349.

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

[4] A. Ghamarian, *et al.*, "Throughput analysis of synchronous data flow graphs," in *ACSD'06 Proc, IEEE*, 2006, pp. 25–34.

[5] S. Stuijk, *et al.*, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC'06 Proc, ACM*, 2006, pp. 899–904.

[6] ——, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. Comp.*, vol. 57, no. 10, pp. 1331–1345, 2008.

[7] Y. Yang, *et al.*, "Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs," in *Estimedia '09 Proc, IEEE*, 2009, pp. 96–105.

[8] S. S. Bhattacharyya, *et al.*, "Synthesis of embedded software from synchronous dataflow specifications," *Journal on VLSI Signal Process. Syst.*, vol. 21, no. 2, pp. 151–166, 1999.

[9] S. Stuijk, *et al.*, "SDF^3: SDF For Free," in *ACSD'06 Proc, IEEE*, 2006, pp. 276–278.

[10] S. Ritz, *et al.*, "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.

[11] M. C. W. Geilen, *et al.*, "Minimizing buffer requirements of synchronous dataflow graphs with model-checking," in *DAC'05 Proc, ACM*, 2005, pp. 819–824.

[12] M. Lukasiewycz, *et al.*, "Efficient symbolic multi-objective design space exploration," in *ASP-DAC '08 Proc*. IEEE, 2008, pp. 691–696.

[13] A. D. Pimentel, *et al.*, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. Comp.*, vol. 55, no. 2, pp. 99–112, 2006.

[14] R. K. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling*. John Wiley & Sons, April 1991.

[15] F. Gao and S. Sair, "Long-term performance bottleneck analysis and prediction," in *ICCD 2006*, 2006, pp. 3–9.

[16] J. Hu, *et al.*, "System-level buffer allocation for application-specific networks-on-chip router design," *IEEE Trans. CAD*, vol. 25, no. 12, pp. 2919–2933, 2006.

[17] H. Chen and A. Mandelbaum, "Discrete flow networks: Bottleneck analysis and fluid approximations," *Mathematics of Operations Research*, vol. 16, no. 2, pp. 408–446, 1991.

[18] M. Agarwal and M. I. Frank, "Spartan: A software tool for parallelization bottleneck analysis," in *IWMSE '09*, 2009, pp. 56–63.