J. Parallel Distrib. Comput. 73 (2013) 1337–1350

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Maximum-throughput mapping of SDFGs on multi-core SoC platforms

Alessio Bonfietti [a,*], Michele Lombardi [a], Michela Milano [a], Luca Benini [b]

[a] DISI, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
[b] DEI, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy

HIGHLIGHTS

- We face Max-Throughput Mapping and Scheduling of streaming applications (SDF) on MPSoC platforms.
- We develop a Constraint-based solver relying on an incremental algorithm to narrow the search space.
- The method is complete, but we devise heuristics to quickly guide search to high quality solutions.
- We perform an extensive evaluation to assess the method effectiveness and scalability.
- Adding incrementality speeds-up tree-search pruning by orders of magnitude.

ARTICLE INFO

Article history:
Received 22 December 2011
Received in revised form
15 May 2013
Accepted 23 May 2013
Available online 11 June 2013

Keywords:
Scheduling
Constraint programming
Mapping
Multi-core platforms
Acceleration of parallel execution

ABSTRACT

Data-Flow models are attracting renewed attention because they lend themselves to efficient mapping on multi-core architectures. The key problem of finding a maximum-throughput allocation and scheduling of Synchronous Data-Flow graphs (SDFGs) onto a multi-core architecture is NP-hard and has been traditionally solved by means of heuristic (incomplete) algorithms with no guarantee of global optimality. In this paper we propose an exact (complete) algorithm for the computation of a maximum-throughput mapping of applications specified as SDFG onto multi-core architectures. This is, to the best of our knowledge, the first complete algorithm for generic SDF graphs, including those with loops and a finite iteration bound. Our approach is based on Constraint Programming, it guarantees optimality and can handle realistic instances in terms of size and complexity. Extensive experiments on a large number of SDFGs demonstrate that our approach is effective and robust.

1. Introduction

As the number of processors integrated on a single chip increases with the fast pace dictated by Moore's Law, multi-core systems-on-chip (MPSoCs) are becoming truly distributed systems at the micro-scale. A typical MPSoC [9,24] features a number of computing tiles connected through a network-on-chip (NoC). A tile hosts a processor and a local memory hierarchy, and communicates with other tiles using communication services provided by the NoC interface. Processors are often highly optimized for domain-specific computation, with specialized instruction sets and support for vectorial data-parallel execution. While intra-tile parallelism is typically expressed through language intrinsics or automatically discovered by compilers, inter-tile communication is relatively expensive in time and power and it should be made explicit by the programmer. Thus, data-flow (streaming) models, which express computations as collection of processes communicating through explicit channels with precisely defined production and consumption rules, match very well the nature of the underlying execution platforms [29].

From the application viewpoint, requirements for high performance and low power have increased at a breakneck speed in many embedded computing domains like wireless communication, imaging, audio and video processing and graphics, pushed by the demand for higher communication bandwidth, multimedia quality and realistic rendering. Applications in these areas are highly parallelizable and feature significant functional parallelism, which can effectively be expressed through a data-flow model of computation, where data is processed in (pipelined) sequences of computing stages with forks and loops to express alternatives and state.

As discussed above, technology and architectural evolution as well as application trends are motivating the use of data-flow programming in embedded computing. For this reason increased research effort is being focused on developing methods and tools for

* Corresponding author.
E-mail address: alessio.bonfietti@unibo.it (A. Bonfietti).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.05.004

efficiently mapping data-flow applications onto many-core MP-SoC platforms [21]. The theoretical foundations of the data-flow model of computation were studied in the seventies and eighties [30], with the definition of several flavors of graph notations to formally and precisely express various classes of data-flow computational models, spanning the expressiveness vs. analyzability trade-off curve [7]. Synchronous data-flow (SDF) is one of the most widely used models (for details see Section 2), as it is sufficiently semantically rich to express practical computations, while being still analyzable with reasonable efficiency [31]. As of today, several commercial and academic programming environments are available for SDF application specification, analysis and mapping [21,7].

As most of the data-flow applications are subject to real-time constraints, a key problem that must be addressed by SDF mapping tools is throughput-constrained mapping (and/or throughput maximization). An informal definition of SDF execution throughput (see Section 2 for a formal definition) is the number of executions of an SDF graph in a unit of time. Applications usually come with throughput constraints, such as decoded frames per second, or processed polygons per second, and the key objective of a mapping tool is to find an allocation and scheduling of SDF nodes on computing tiles so that application throughput constraints are met. This is a NP-hard problem, and it is usually solved by sequential decomposition and incomplete search [32,43]. Additionally, even though SDF execution ultimately becomes periodic, the execution sequence within one period and the aperiodic initial transient can be very long. This greatly complicates throughput computation during the search of mapping and scheduling alternatives even for SDF graphs with a low number of nodes. Hence, complete search approaches were believed to be computationally intractable even for the simplest SDF instances.

We propose an algorithmic framework for allocation and scheduling of synchronous-data flow applications on a target homogeneous multiprocessor platform; the approach is complete, namely if a throughput requirement is specified, a feasible solution is guaranteed to be found if it exists; in general, the solver always finds the optimal solution if enough time is given. Incomplete approaches cannot offer such a guarantee. The method tackles the mapping and scheduling problem as a whole, avoiding any sub-optimality due to decomposition; to the best of the authors' knowledge this is the first complete approach which can handle realistic and general SDF Graphs.

Our method is based on Constraint Programming (CP) [38], a declarative programming paradigm based on constraint propagation and search (for details see Appendix). The efficiency of the approach hinges on a global throughput constraint, maintaining a tight bound on the maximum achievable throughput based on the current state of the search; search nodes are pruned whenever such bound becomes lower than an input throughput requirement, or than the best solution found so far. We propose in the paper two versions of the throughput constraint: the non-incremental and the incremental version, the second reaching one order of magnitude speed-up with respect to the former with very significant benefits on scalability.

The paper is organized as follows. Sections 2 and 3 discuss respectively the background and related works in the area. Section 4 details the Constraint Programming model and search strategy, while Section 5 illustrates in detail the throughput filtering algorithm. In Section 6 is about the experimental evaluation. Finally, Section 7 concludes the paper.

2. Background: Synchronous Data-Flow Graphs

Synchronous Dataflow Graphs (SDFGs) [30] are used to model periodic applications that must be bound to a Multiprocessor System on Chip. They allow the modeling of both pipelined streaming and cyclic dependencies between tasks. To assess the performances of an application on a platform, one important parameter is the throughput. In the following we provide some preliminary notions on synchronous data flow graphs used in this paper.

Definition. A Synchronous Data-flow Graphs is a tuple $\mathbb{G} = \langle \mathbb{A}, \mathbb{D} \rangle$ consisting of a finite set $\mathbb{A}$ of actors (*nodes*) and a finite set $\mathbb{D}$ of *dependency edges*. A dependency edge $d = (a, b, p, q, t)$ denotes a dependency of actor b on a, with $a, b \in \mathbb{A}$. When a executes, it produces p tokens on d and when b executes, it consumes q tokens from d. Edges may contain initial tokens $tok(d) = tok(a, b) = t$.

Actor execution is defined in terms of firings. An essential property of SDFGs is that every time an actor fires it consumes a given and fixed amount of tokens from its input edges and produces a known and fixed amount of tokens on its output edges. These amounts are called *rates*. The rates determine how often actors have to fire w.r.t. each other such that the distribution of tokens over all edges is not changed. This property is captured by the repetition vector. Fig. 1 represents a simple Synchronous Data-Flow graph with 4 nodes; the execution times are $[A = 2, B = 5, C = 2, D = 1]$. This graph will be used as an example throughout the whole paper.

Definition. A repetition vector of an SDFG $\mathbb{G}$ is a function $\gamma : \mathbb{A} \to \mathbb{N}$ such that for every edge $(a, b, p, q, t) \in \mathbb{D}$ from $a \in \mathbb{A}$ to $b \in \mathbb{A}$, $p \cdot \gamma(a) = q \cdot \gamma(b)$. A repetition vector q is called non-trivial if $\forall a \in A, \gamma(a) > 0$.

The smallest non-trivial repetition vector is usually referred to as *the* repetition vector. We say the SDFG completes an *iteration* whenever each actor a has fired exactly $\gamma(a)$ times. We refer to each actor firing within an iteration as *repetition*. For instance, the repetition vector of the graph described in Fig. 1 is [1, 2, 2, 3] (the numbers on the arcs are the *rates* and the dots represent the tokens).

SDFGs in which all rates equal 1 are called Homogeneous Synchronous Data Flow Graphs (HSDFGs, [30]). Every SDFG $\mathbb{G}$ can be converted to an equivalent HSDFG $\mathbb{G}'$, by using the conversion algorithm in [7]. The transformation procedure produces an homogeneous graph that has a node for any repetition of each actor of the original SDF graph (i.e. $\gamma(a)$ nodes for each actor a); as a consequence the homogeneous graph is usually larger than the related SDF.

In Fig. 2 we report the HSDFG corresponding to the SDFG in Fig. 1. Note that, for example, actors B_1 and B_2 of the HSDFG correspond to the actor B of the SDFG that has a repetition vector $\gamma(B) = 2$. In the figure the (unary) rates are omitted.

2.1. Throughput

Throughput is an important design constraint for embedded multimedia systems. The throughput of an SDFG refers to how often an actor produces tokens. To compute throughput, a notion of time must be associated with the firing of each actor (i.e., each actor has a duration also called *execution time*) and an execution scheme must be defined. We consider as execution scheme the **self-timed execution** of actors: each actor fires as soon as all of its input data are available (see [42] for details). In a real platform self-timed execution is implemented by assigning to each processor a sequence of actors to be fired in fixed order: the exact firing times are determined by synchronizing with other processors at run time.

Working with Synchronous Data-Flow models of computation, it becomes natural to adopt a scheduling strategy which defines only the allocation and let the run-time scheduler to decide the

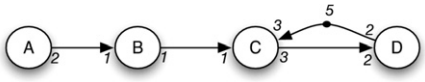

Fig. 1. Synchronous data-flow graph.

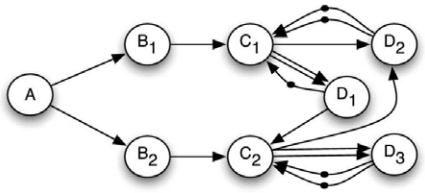

Fig. 2. Homogeneous SDF.

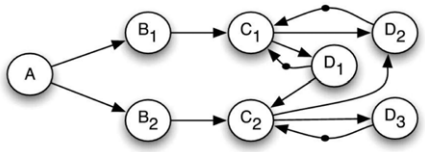

Fig. 3. Filtered homogeneous SDF.

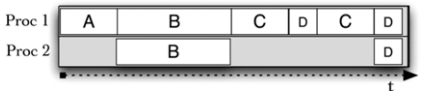

Fig. 4. Single-iteration self-time execution.

start times. Considering the graph in Fig. 1, its single-iteration self-timed execution can be expressed by the Gantt chart of Fig. 4. First actor A is executed, it produces two tokens on (A, B) since the *out*-rate of actor A on the edge is 2. The *in*-rate of actor B on the same edge is 1; then B can fire twice concurrently. After both executions of B, C can start. Its execution consumes 1 token on (B, C) and 3 on (D, C) and produces 3 tokens on (C, D). Then only the actor D could fire, because the actor C is constrained by the presence of only 2 tokens on (D, C). Actor D produces 2 tokens on (D, C) and enables the firing of C whose execution enables the concurrent execution of two instances of D that terminate the single-iteration self-timed execution of the graph.

Note that turning an SDFG into the equivalent homogeneous graph may produce multiple arcs between pairs of nodes (see edges from D_2 to C_1 in Fig. 2). Therefore the homogeneous graph should be simplified before throughput computation removing multiple edges between two nodes. Fig. 3 shows the filtered graph corresponding to the one in Fig. 2.

3. Related work

Data-Flow graphs are an extension of *computational graphs*, defined, for the first time, in 1966 by Karp and Miller [27]. Their studies focused on determinacy property and on termination conditions. The problem of mapping and scheduling task graphs has been widely studied (see, for instance, [34,39,46]), but the limited descriptive power of task graphs as models of computation has lead to the development of graph models with a richer execution semantic.

The Synchronous Data-Flow Model of Computation (SDF MoC) has been proposed by Lee and Messerschmitt [30] to represent digital signal processing (DSP) applications. This Data-Flow MoC has been adopted in wide-ranging areas such as networking, multimedia, high-definition digital TV and wireless base stations; it can efficiently represent streaming applications such as mp3 playback [47], DAB channel decoding [8] and Software Defined Radio [33].

Much work has been published on the scheduling of data-flow graphs with real-time requirements. Researchers have mostly focused on incomplete (also called heuristic) mapping algorithms for SDF allocation and scheduling (see [32,43]). The motivation for the use of incomplete approaches is that both computing an optimal allocation and an optimal schedule are NP-hard [18]. The interested reader is referred to [7] for an excellent, in-depth survey of the topic.

Here we briefly describe state-of-the-art approaches to mapping and scheduling synchronous data-flow graphs that are classifiable onto two separate sets: complete and heuristic (incomplete) methods applied to Homogeneous SDFG, and heuristic methods applied directly on SDF graphs.

HSDF Scheduling. The first class of approaches, pioneered by the group lead by E. Lee [42], and extensively explored by other researchers [7], can be summarized as follows. A SDFG specification is first checked for consistency, and its non-null iteration vector is computed. The SDFG is then transformed, using the algorithm, described in [7] into an Homogeneous SDF graph (HSDF). The HSDFG is then mapped onto the target platform in two phases. First, an allocation of HSDFG nodes onto processors is computed, then a static-order schedule is found for each processor. The overall goal is to maximize throughput, given platform resource constraints. Unfortunately, throughput depends on *both* allocation and scheduling. However, the combination of possible mapping and scheduling decisions leads to an exponential blow-up of the solution space.

With the widespread diffusion of multi-core processors, scheduling and allocation of data-flow applications onto parallel computing platforms has received renewed interest. Kudlur et al. described in [28] an ILP that unfolds and partitions a stream application onto MPSoC architecture. Their approach consists of two steps: a fission and partitioning step, performed through Integer Linear Programming (ILP), to ensure work balancing, and then a stage assignment step wherein each actor is assigned to a pipeline stage for execution. An enhanced version of the same work was presented in [14] by Choi et al..

Chatha and co-authors have proposed two methods to support the compilation of streaming application on multi-core processors. The first, described in [12], uses fusion and fission operations to schedule streams onto (SPM based) multi-core processors while the second one, in [13], adopts a classic retiming technique. In both works the method is not complete, therefore the optimality is not guaranteed. They adopted the StreamIt language from MIT as the input specification (see [45] for details). StreamIt is an architecture-independent language with a synchronous data-flow semantic, supplied with an efficient compiler, described in [25] and in [41].

Ostler et al. devise, in [35], an ILP model for mapping streaming applications on multi-core platforms; the approach tackles acyclic applications, takes into account limited local memory capacity and allows throughput improvement via task fission. Communications are handled via double buffering, assuming exactly one DMA channel is dedicated to each processor. Within the specified assumptions the approach is optimal; it is important to observe that, since only acyclic SDFGs are considered, computing a feasible schedule is trivial once the mapping is specified.

Other approaches combine off-line/on-line scheduling techniques. For instance FlexStream, presented in [22], is a runtime adaptation system that dynamically re-maps an already partitioned stream graph according to the number of processors available for heterogeneous multi-core systems.

SDF Scheduling. A different class of approaches [43] works directly upon SDF graphs using simulation techniques, without an explicit

HSDFG transformation. This approach has the advantage of avoiding the potential blow-up in the number of nodes, with the disadvantage that if problem constraints are tight, incomplete approaches do not find any feasible solution. These approaches use a heuristic function to generate a promising allocation, and then compute the actual throughput by performing state-space exploration on the SDFG with allocation and scheduling information until a fixed point is reached.

Researchers from ST-Ericsson designed a scheduling strategy that allows a heterogeneous MPSoC to handle a dynamic mix of hard-real-time jobs which can start or stop independently. To solve this problem, a combination of Time Division-Multiplex (TDM) schedule and static-order of actors per processor is applied [33].

The incomplete approaches summarized above cannot give any proof of optimality, nor guarantee to find a feasible solution; actually, if the throughput requirement of the problem is tight, an incomplete solver is likely to fail. Our work aims at addressing this limitation, and proposes a complete search strategy which can compute max-throughput mappings for realistic-size instances. Our starting point is a HSDFG, which can be obtained from a SDFG by a pseudo-polynomial transformation [7]. We develop a CP-based solver which, given an architecture and an application described through a SDF graph, finds either the optimal or a feasible mapping and scheduling.

4. Problem definition

The problem considered in this paper is the allocation and scheduling of an HSDFG on a target set of processors subject to a throughput constraint. Given a HSDFG labeled with actor durations, given a target platform defined by a set of processors, the problem is to assign each actor to a processing element and to define an ordering between actors allocated on the same processor such that an input throughput constraint is satisfied and the execution is guaranteed to be resource contention free. Our approach is complete, meaning that it is guaranteed to find a feasible solution in case it exists. Moreover, the approach can easily handle the optimization version of the problem, i.e. finding the maximum throughput allocation and scheduling of an input application on a target platform. This problem is strongly NP-hard.

The allocation and scheduling algorithm for SDFG we propose in this paper is based on Constraint Programming. Constraint Programming (CP) [17,38] is a programming paradigm used to solve hard combinatorial problems. A constraint model is defined in terms of variables and constraints. Each variable X_i has an associated domain D_i containing values that the variable can assume (the notation for linking variables and domains is $X_i :: D_i$). Constraints define combinations of consistent assignments. The model might have an objective function defining a (possibly partial) order in the solution space. Once the constraint model is stated, constraint solving is started by interleaving propagation (domains filtering procedure) and search (further details in the Appendix).

4.1. Model

We devised a two-layer CP model: on one level the model features two sets of decision variables, respectively representing allocation and scheduling/ordering decisions; on the second level we have a set of graph description variables working directly on the HSDFG by adding and removing arcs and tokens as a consequence of the allocation and scheduling decisions. For this reason, the two models are linked via *channeling constraints*.

As far as the first level is concerned, let n be the number of actors in the input HSDFG and let p be the number of processors in the platform, then the decision variables are:

$$\forall i = 0 \dots n-1 : \; P_i \in [0 \dots p-1] \tag{1}$$

$$\forall i = 0 \dots n-1 : \; N_i \in [-1 \dots n-1] \tag{2}$$

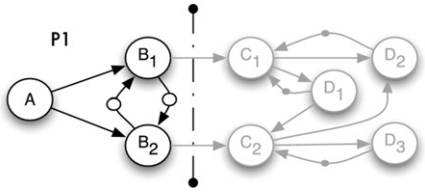

Fig. 5. Concurrent task mapped on the same resource.

where P_i represents the processor allocated to actor i and N_i represents the actor following actor i if allocated on the same processor.

P_i and N_i variables are subject to a set of constraints. First dependencies in the input SDFG cannot be violated: thus $i \prec j \Rightarrow N_j \neq i$. We say that an activity i precedes j, $i \prec j$, if there exists a path directed from i to j. Less intuitively, assuming that i and j (allocated on the same *unary* processor) cannot execute concurrently, the presence of an arc(j, i) with $tok(j, i) = 1$ in the input HSDFG implies i to fire always *before* j, and therefore, $N_j \neq i$.

Moreover, two nodes on the same resource cannot have the same successor: $P_i = P_j \Rightarrow N_i \neq N_j$. Then, a node i can be next of j only if they are on the same processor: $P_i \neq P_j \Rightarrow N_i \neq j$ and $N_j \neq i$. The -1 value is given to the last node of each (non empty) processor:

$$\forall\, proc : \quad \sum_{i=0}^{n-1} (P_i = proc) > 0 \Rightarrow \sum_{i=0}^{n-1} [(P_i = proc) \times (N_i = -1)] = 1. \tag{3}$$

Finally, the transitive closure on the actors running on a single processor is kept by posting a *nocycle* constraint [36] on the related N variables.

Note that we consider the mapping platform as an ideal architecture without any communication cost or buffer requirement.

The second model, instead, considers the (dynamically changing) graph structure and defined decision variables on it. We define a matrix of binary variables $ARC_{ij} \in [0, 1]$ such that $ARC_{ij} = 1$ iff an arc from i to j exists. Existing arcs in the input HSDFG result in some pre-filling of the ARC matrix, such that $ARC_{ij} = 1$ for each arc $(i, j, 1, 1, t)$ in the original graph. Channeling constraints link the two models, i.e., allocation and scheduling decisions and graph description variables; first observe that token positioning is implicitly defined by the N_i variables and is built on-line only at throughput computation time. As far as the P_i variables are concerned, the relation with ARC variables depends on whether a path with no tokens exists in the original graph between two nodes i, j. Let us write $i \prec j$ if such path exists; then, if $i \neq j$ and neither $i \prec j$ nor $j \prec i$ hold:

$$P_i = P_j \Rightarrow ARC_{ij} + ARC_{ji} = 2. \tag{4}$$

Constraint (4) forces two arcs to be added, if two independent nodes are mapped to the same processor (e.g. nodes B_1 and B_2 in Fig. 5).

If instead there is a path from i to j ($i \prec j$), then the following constraint is posted:

$$\left[(P_i = P_j) \wedge \sum_{k \prec i} (P_k = P_i) = 0 \wedge \sum_{j \prec k} (P_k = P_j) = 0 \right] \Rightarrow ARC_{ji} = 1. \tag{5}$$

The above constraint completes dependency cycles: considering only tasks on the same processor (first element in the constraint condition), if there is no task before i in the original graph (second element) and there is no task after j in the original graph (third element), then close the loop, by adding an arc from j to i. Fig. 6 shows that, assuming an allocation of A, B_1, C_1 on the same resource, an

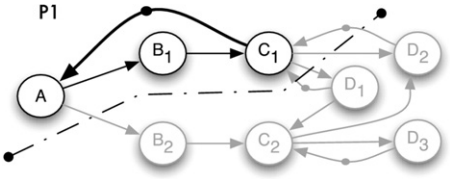

Fig. 6. Pipelined task mapped on the same resource.

edge with a token is added from C_1 to A. Finally, auto-cycles can be added to each node in a pre-processing step and are not considered here. Since we are dealing with a throughput bounded application, we need a constraint computing the throughput depending on decisions taken during search. For this purpose we have defined a novel Throughput Constraint (for details see Section 5) which is satisfied if and only if an allocation of P and N exists that defines an augmented graph with a throughput value higher than the current bound. The constraint is global and has the following signature:

$$thcst(TPUT, [P_{0\ldots n-1}], [N_{0\ldots n-1}], [ARC_{(0,0)\cdots(n-1,n-1)}], W_{0\ldots n-1})$$

where $TPUT$ is a real valued variable representing the throughput, $[P_{0\ldots n-1}]$, $[N_{0\ldots n-1}]$ and $[ARC_{(0,0)\cdots(n-1,n-1)}]$ are defined as above, W is a vector such that W_i is the computation time of actor i.

Note that with this constraint, we can easily find also throughput maximal solutions (objective function $z = \max(TPUT)$), by iteratively solving a set of throughput bounded problems with increasing values of throughput.

4.1.1. Communication buffers and latency

For the sake of simplicity, the model presented in this work is based on an ideal MPSoC architecture, where communication is considered as ideal (zero cost). However communication buffers and latencies can be modeled in different ways, depending on the target architecture. In this section we describe two approaches to model buffers and latencies for two widely adopted MPSoC architectural templates.

Tightly-Coupled Shared-Memory Cluster Architecture (e.g. P2012 Platform [5]): in this architecture all the processing units within a cluster share a fast multi-banked on-chip L1 data memory. The memory stores the buffers and the access and transfer cost (i.e. communication latency) is the same for each processor. In this case latencies' time lags can be merged within the task execution times.

Let now ω be the bandwidth of the communication channel, $\widehat{L}$ be the latency of a single token communication. The latency L of a communication depends on the bandwidth ω and the size of the transmission: i.e. the number of tokens tok the task produces

$$L = \widehat{L} \cdot \frac{tok}{\omega} \tag{6}$$

where ω has been normalized considering the size of a single token (e.g. when $\omega = 2$ the channel transmits two tokens concurrently). Hence the final execution time $\widehat{W}_i$ of a node i should be $\widehat{W}_i = W_i + L_{in} + L_{out}$ where L_{in} and L_{out} are the sum of the latencies of the ingoing and outgoing communications, respectively. Furthermore, the memory capacity (L1 size) and the buffer requirements can be modeled[1] through a global cumulative constraint [2]. The constraint is satisfied iff, for each time instant, the sum of the buffer allocated does not exceed the total capacity of the memory.

[1] One of the advantages that the use of constraint programming (see Appendix) has is that the definition of the model is loosely coupled with the search strategy adopted. Hence adding further constraints to existing models, not only is easily feasible but it could even help in making, with the constraint propagation, the search for a solution more efficient and more effective.

Non-Uniform Memory Access (NUMA) Architecture: the template in this scenario is based on a tile-based multiprocessor architecture (widely described in [16]) in which multiple tiles are connected by an interconnection network. Each tile contains a processor and a memory containing the communication buffers. The system has a Global Address Space, therefore the tasks and their communication buffers should be allocated as near as possible. Hence the model presented in this work had to be drastically modified. In fact it should consider the buffer allocation problem and the impact of the allocation choices on the communication latencies. In this case latencies should be modeled through additional nodes with variable durations depending on the allocation of the buffers (e.g. see the approach in [40]) and each local memory capacity should be modeled though a cumulative constraint [2] (avoiding resource over-usage).

A trivial solution in these scenarios could be to force the allocation of all the HSDFG nodes corresponding to repetitions of the original SDFG nodes on the same processor (thus in NUMA architectures, buffers could be allocated locally). However the experiments show that without this constrained hypothesis forcing the allocation, the search found much better solutions (see Section 7.3).

5. Throughput constraint

The relation between decision variables and the throughput value is captured in the proposed model by means of a novel global throughput constraint, whose signature is:

$$thcst(TPUT, [P_{0\ldots n-1}], [N_{0\ldots n-1}], [ARC_{(0,0)\cdots(n-1,n-1)}], W_{0\ldots n-1})$$

where $TPUT$ is a real valued variable representing the throughput, $[P_{0\ldots n-1}]$, $[N_{0\ldots n-1}]$ and $[ARC_{(0,0)\ldots(n-1,n-1)}]$ are stated in Section 4.1, W is a vector such that W_i is the computation time of actor i.

We devised a filtering algorithm consistently updating an upper bound on $TPUT$ (this is sufficient for a throughput maximization problem).

Each time the graph is modified, by fixing an ARC variable or taking an ordering decision, the constraint receives a new description of the graph, and computes the throughput value over it.

During search the throughput variable is constrained to be within a lower and an upper bound. Initially the upper bound is set to the intrinsic iteration bound of the starting graph. This value always decreases during search. In fact, the application throughput depends on the inverse of the *longest* cycle whose value increases as search decisions are taken. On the other hand, the lower bound is set to the throughput requirement of the application, if any; in case we want to maximize the throughput value, the lower bound is updated with the best solution found so far. Since the optimal solution is found by iteratively improving feasible solutions, the lower bound increases during search.

At any time during the solution process, if the upper bound becomes lower than the lower bound, the search fails and backtracking is forced.

5.1. Throughput algorithm

As stated in the Appendix, global constraints comprise efficient filtering algorithms.

The filtering algorithm we propose extends the Maximum Cycle Mean (MCM) algorithm [23] and [15], which in turn is based on Karp's algorithm [26]. The MCM algorithm is based on a recursive formula which computes, starting from a source node, the weight of each path (execution times of the considered nodes) of

the graph. As soon as a cycle c is found, the throughput Th_c is computed. The final throughput value is the lowest found, that is the weightiest cycle.

$$Th = \min_{c \in Cycles} Th_c. \tag{7}$$

The algorithm is based on two three-dimensional matrices:

- $D_{(k,i,t)}$ that stores the weight of the path. In particular each element (k, i, t) is the maximum weight of a path of length k from a node source s to i; the number of tokens in the path is described with t. If $D_{(3,2,1)} = 13.4$ means that at level $k = 3$ (i.e., three nodes far from the source s) there exists a path that connects s to $i = 2$ with one token over its edges; if no such path exists, then $D_{(k,i,t)} = -\infty$.
- $\Pi_{(k,i,t)}$ that saves the location of the predecessor of task i at level k. In particular, such location consists of two coordinates: the index of the task and its token number; note that the predecessor level is $k - 1$. For instance $\Pi_{(k,i,t)} = (3, 2)$ means that the actor i at level k has node 3 as predecessor (referred to as $idx(\Pi_{(k,i,t)})$); the number of tokens on the path from the source (referred to as $tok(\Pi_{(k,i,t)})$) is 2.

If n is the number of the tasks and Γ the number of tokens of the original graph, both D and Π are $(n + 1) \times n \times (\Gamma + n)$ matrices.

The algorithm is divided in three phases:

Step 1: *Building the input graph*

The input for the throughput algorithm is a "minimal" graph built by adding arcs to the original HSDFG based on the current state of the model. More precisely, an arc is assumed to exist between actors i and j iff $ARC_{ij} = 1$; unbound ARC variables are therefore treated as if they were set to 0. Note that the computation of a lower bound for the throughput would require to fix values for unbound ARC variables as well. Let $V_{i,j}[0, 1]$ (*Vertex matrix*) be a matrix which defines for each couple of actors the presence of an arc ($V_{i,j} = 1$ if $ARC_{ij} = 1$ exists, 0 otherwise).

Step 2: *Token positioning*

Next we construct a dependency graph DG with the same nodes as the original HSDF graph G, and such that an arc (i, j) exists in DG iff either an arc $(i, j, 1, 1, 0)$ exists in G (detected since $ARC_{ij} = 1$ and $tok(i, j) = 0$) or $N_i = j$. Note that a DG graph is a Direct Acyclic Graph (DAG) augmented with the scheduling information of the partial solution.

A token matrix TK is then built, according to the following rules:

$$ARC_{ij} = 0 \Rightarrow TK_{ij} = 0 \tag{8}$$

$$ARC_{ij} = 1 \Rightarrow \begin{cases} TK_{ij} = 0 & \text{if } i \prec^{DG} j \\ TK_{ij} = 1 & \text{otherwise} \end{cases} \tag{9}$$

where we write $i \prec^{DG} j$ if there is path from i to j in DG. The rules above ensure the number of tokens is over-estimated, until all N and P are fixed. In the actual implementation, the dependency check is performed without building any graph, while the token matrix is actually stored in the constraint. By considering the graph described in Fig. 3 and a hypothetical allocation of actors B_1, B_2 on the same processor, the modified graph is shown in Fig. 5. Assuming that in the DG graph both nodes are independent the resulting associated values of the Token Matrix are $TK_{B_1,B_2} = TK_{B_2,B_1} = 1$. This is clearly an over-estimation of the number of tokens. Whenever an ordering decision is taken, for example $B_1 \prec B_2$, the token matrix is changed with the following values: $TK_{B_1,B_2} = 0$, $TK_{B_2,B_1} = 1$. Finally note that the token positioned are only used in the throughput computation. They could be considered as *fake* tokens as they do not represent real packets of data.

Step 3: *Throughput computation*

For a HSDFG, the throughput equals the inverse of a quantity known as the iteration period of the graph and is denoted as $\lambda(HSDFG)$; formally:

$$\frac{1}{th} = \lambda(HSDFG) = \max_{C \in HSDFG} \frac{W(C)}{T(C)}$$

where C is a cycle in the HSDFG, $W(C) = \sum_{i \in C} \mathcal{W}_i$ is the sum of the execution time of all actors in C and $T(C) = \sum_{(i,j) \in C} TK_{ij}$ is the total number of tokens on the arcs of C. The quantity $\max_{C \in HSDFG} \frac{W(C)}{T(C)}$ is also called *maximum cycle ratio* (strictly related to *maximum cycle mean*, see [23,15,26]) of the graph.

In [23] it is shown how to compute the iteration period as the *maximum cycle mean* of an opportunely derived *delay graph*; Karp's algorithm [26] is used for the computation. In general *cycle mean* algorithms cannot be used to compute the throughput directly on a Homogeneous SDF. In fact, it is necessary to transform the graph into a weighted directed graph. Unfortunately it has been experimentally proven that this transformation is very time consuming [19]. Here, we show that the transformation can be avoided by using proper data structure; this enables a *maximum cycle mean* algorithm to be used to compute the iteration period directly on a HSDFG. This is done by exploiting the third dimension (token dimension) of the matrices D and Π of the data structure, in the sense that they can store paths with different numbers of tokens. Karp's algorithm works on a set of two-dimensional matrices; in fact, the *mcm* algorithm considers a single token on each edge. We introduce a third matrix dimension to keep track of the number of tokens on the paths.

The basic idea is that, according to Karp's theorem, the critical loop constraining the iteration period can be found by analyzing cycles on the worst case k-length paths starting from an arbitrary source. Since no cycle can involve more than n nodes, considering k-length paths with k up to n is sufficient. Starting from a source node, we traverse the graph, storing for each node the critical path in the Matrices D and Π. The critical path is the path with maximum cycle ratio; namely, assuming the same number of tokens, the path with greater execution time. Each time a cycle is detected, the throughput bound is updated.

Algorithm 1: Throughput computation - build D table

```
   Data: Let s be the source node
   Data: Let all D_(k,i,t) = −∞, Π_(k,i,t) = NIL
 1 begin
 2     Q_0^+ = {(s, 0)};
 3     Q^− = ∅;
 4     D_(0,s,0) = 0;
 5     Π_(0,s,0) = −1;
 6     for k = 0 to n do
 7         forall the (i, t) ∈ Q^+ do
 8             forall the j ∈ A^+(i) do
 9                 cycle = false;
10                 t^nx = t + TK_ij;
11                 currPos = (k, i, t);
12                 nextPos = (k + 1, j, t^nx);
13                 WP_j = D_nextPos;
14                 WP_i = D_currPos + W_j;
15                 if WP_i > WP_j then
16                     Q^− = Q^− ∪ {i};
17                     D_nextPos = WP_i;
18                     Π_nextPos = (i, t);
19                     if WP_i > Bnd then
20                         Find loops on level k;
21                 if not cycle then
22                     Q^+_{k+1} = Q^+_{k+1} ∪ {(j, t^nx)};
```

non-trivial. The pseudo code for the throughput computation is reported in Algorithm 1, where $A^+(i)$ denotes the set of direct

successors of i. Q^- is the set of nodes visited while Q_k^+ store, for each level k the set of nodes to visit and their token level. Once the table is initialized, a source node s is chosen. The choice has no influence on the correctness of the method, but a strong impact on its performance, hence choosing an arbitrary node is not recommended. We face the problem by reordering the actors with a heuristic function. The function is based on scores computed using the following expression:

$$score_i = \sum_{0 \le j \le task} Dep_{j,i} \tag{10}$$

where $Dep_{j,i}$ is 1 if there exists a path without tokens that connects i to j, 0 otherwise. This structure can be easily computed from matrices V and T.

Next, the procedure is initialized by setting $D_{(0,s,0)}$ to 0 (line 4, 5) and adding s to the list of nodes to visit Q_0^+ (line 2). For each node i in Q each successor j is considered (lines 7, 8), and, if necessary, the corresponding cells in D and Π are updated to store the k-length path from s to j (lines 15 to 18). Once a cell is updated, if the weight of the path is higher than the current bound Bnd, loops are detected as described in Algorithm 2. If the node j does not close a cycle (line 21), it is added to the Q_{k+1}^+ queue and then we move to the next k value. A single iteration of the algorithm is sufficient to compute the throughput of a strictly connected graph; otherwise, the process is repeated starting from the first never touched node, until no such node exists.

The loop finding procedure (Algorithm 2) is started when a cell in D at a specific level (let this be k) is updated. The algorithm moves backward along the predecessor chain ($\Pi(k+1, j, t^{nx})$ is the predecessor of current node) until a second occurrence of the starting node j is detected ($a' = j$ in line 5) and a cycle is found. If this loop constrains the iteration period more than the last one found so far (line 11), this is set as a critical cycle. The algorithm also stops when the start of D is reached (in line 4).

Algorithm 2: Throughput computation - finding loops

```
   Data: Let i be the node considered and t its tokens lvl
   Data: Let j be the successor and t^nx = t + TK_ij its tokens lvl
   Data: Let D_nextPos be the cell updated, nextPos = (k+1, j, t^nx)
1  begin
2      define a' = i;
3      define t' = t^nx;;
4      for z = k to 1 do
5          if j == Π_{z,a',t'} then
6              define Π' = Π_{z,a',t'};
7              define backPos = (z − 1, idx(Π'), tok(Π'));
8              define W_Thp = D_nextPos − D_backPos;
9              define T_Thp = t^nx − tok(Π');
10             cycle = true;
11             if W_Thp / T_Thp > Bnd then
12                 Bnd = W_Thp / T_Thp;
13             return;;
14         define temp = a';
15         a' = idx(Π_{z,temp,t'});
16         t' = tok(Π_{z,temp,t'});
```

Example. Fig. 8 presents matrices D and Π with regard to the subgraph composed by actors C and D (5 nodes) of Fig. 3 with execution time respectively 2 and 1. The subgraph is reported in Fig. 7A. Assuming the source node is C_1, Fig. 8 reports the sub-matrices $D_{i,j,0}$, $\Pi_{i,j,0}$, $D_{i,j,1}$ and $\Pi_{i,j,1}$.

Node C_1 has two outgoing edges: (C_1, D_1) and (C_1, D_2) stored in the matrix in cell $D_{1,2,0}$ and $D_{1,3,0}$. Let us now consider level $k = 1$: the only non-negative entries are the ones for D_1 and D_2. D_1 has

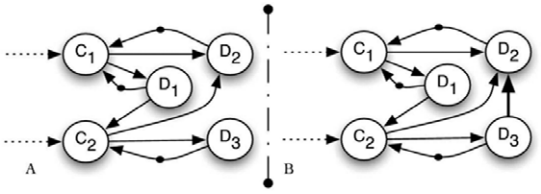

Fig. 7. Sub-graphs of Fig. 2.

	C1	C2	D1	D2	D3		C1	C2	D1	D2	D3
$D_{i,j,0}$	0	1	2	3	4	Π	0	1	2	3	4
0	**0**	-∞	-∞	-∞	-∞	0	**-1**	-∞	-∞	-∞	-∞
1	-∞	-∞	**2**	2	-∞	1	-∞	-∞	**0,0**	0,0	-∞
2	-∞	**3**	-∞	-∞	-∞	2	-∞	**2,0**	-∞	-∞	-∞
3	-∞	-∞	-∞	**5**	5	3	-∞	-∞	-∞	**1,0**	1,0
4	-∞	-∞	-∞	-∞	-∞	4	-∞	-∞	-∞	-∞	-∞
5	-∞	-∞	-∞	-∞	-∞	5	-∞	-∞	-∞	-∞	-∞
$D_{i,j,1}$	0	1	2	3	4	Π	0	1	2	3	4
0	-∞	-∞	-∞	-∞	-∞	0	-∞	-∞	-∞	-∞	-∞
1	-∞	-∞	-∞	-∞	-∞	1	-∞	-∞	-∞	-∞	-∞
2	3	-∞	-∞	-∞	-∞	2	2,0	-∞	-∞	-∞	-∞
3	-∞	-∞	-∞	-∞	-∞	3	-∞	-∞	-∞	-∞	-∞
4	**6**	6	-∞	-∞	-∞	4	**3,0**	4,0	-∞	-∞	-∞
5	-∞	-∞	-∞	-∞	-∞	5	-∞	-∞	-∞	-∞	-∞

Fig. 8. Matrices $D_{i,j,0}$, $\Pi_{i,j,0}$, $D_{i,j,1}$ and $\Pi_{i,j,1}$.

an outgoing edge that enters in C_1, that is stored in the cell $D_{2,0,1}$; at run-time when C_1 entry is processed, the "find loop" procedure detects the cycle $C_1 \to D_1 \to C_1$. The path is deduced by using the information stored in matrix Π: the cell $\Pi_{2,0,1} = 2, 0$ (actor C_1) refers to cell $D_{1,2,0} = 0, 0$ (actor D_1) that points to cell $D_{0,0,0}$ (actor C_1 again). By finding a loop, the algorithm infers a new bound over the throughput, namely one over the sum of the execution times: $\frac{1}{2+1} = 0,333$; the value is computed based on the starting and ending cell: $\frac{tok(D_{2,0,1}) - tok(D_{0,0,0})}{D_{2,0,1} - D_{0,0,0}}$.

Then the algorithm can proceed by considering edges (D_1, C_2) and (D_2, C_1): the former updates the cell $D_{2,1,0} = 3$ while the latter updates the cell $D_{2,0,1}$ with value 3. However, since the current value of the same cell is 3, no change is performed. This means that there exist two different paths (namely $C_1 \to D_1 \to C_1$ and $C_1 \to D_2 \to C_1$) with the same length (2 steps) that connect the source actor with the same end node; since they have the same weight (computation time), they are equivalent, and only one is stored. The algorithm, then, computes all the remaining paths of the subgraph considered and finds, as expected, the iteration bound; this corresponds to the cycle ($C_1 \to D_1 \to C_2 \to D_2 \to C_1$) that refers to the throughput of the graph, that is $\frac{1}{6}\left(\frac{tok(D_{4,0,1}) - tok(D_{0,0,0})}{D_{4,0,1} - D_{0,0,0}}\right)$. Note that by finding new longer loops, the upper bound always decreases; hence, if at any step a cycle is found such that the resulting throughput is lower than the minimum value of the *TPUT* variable, then the constraint fails. Moreover, it is easy to prove that no more than 1 token can be collected by traversing a sequence of nodes on a single processor: the filtering algorithm exploits this property to improve the computed bound at early stages of the search, where the number of tokens is strongly overestimated (see Section 5.4).

Although the throughput computation is rather efficient, experimental tests show that its computational time takes more than the 70% of the total search time. In fact, every time the constraint is considered, it has to re-compute the throughput on the entire modified graph. Therefore, we propose an incremental version of the constraint that avoids the re-computation of the throughput starting from scratch.

5.2. Incremental algorithm

In this section we describe an incremental filtering algorithm that enables to achieve over one order of magnitude speed up w.r.t. the non-incremental version, therefore increasing scalability and enabling the solution of harder and larger problems.

Note that the state of the constraint, referred to as Υ, is defined by the data structures $\Upsilon \equiv \langle D, \Pi, V, TK \rangle$. During search such data structures are modified at each search node and restored on backtracking. In detail, the data structure contains:

- Matrix $D_{k,i,t}$: it stores the maximum weight of the k-arc path with t tokens from a source node s to actor i.
- Matrix $\Pi_{k,i,t}$: it stores the predecessor of the corresponding element of D.
- Matrix $V_{i,j}[0, 1]$ (*Vertex matrix*) which defines for each couple of actors the presence of an arc ($V_{i,j} = 1$ if $i \rightarrow j$ exists, 0 otherwise)
- Matrix $TK_{i,j}[0, \text{inf}]$ (*Token matrix*) which defines the number of tokens on the edge between nodes i and j.

  Clearly, $TK_{i,j} > 0$ only if $V_{i,j} = 1$, that is one or more tokens can exist between two nodes if and only if there is a corresponding edge.
- Throughput value of the longest cycle.

At the root node, the data structures are initialized from the original graph, getting state Υ_0. The iteration bound of the graph is computed and used to shrink the throughput variable domain. At every search node, the state Υ and the throughput value are updated on the basis of graph modifications.

In particular, during search the graph is modified either by

- Adding arcs (arc append operation)
- Adding tokens (token append operation)
- Removing tokens (token remove operation).

Edges are removed only in backtracking. Therefore they are not considered as possible graph modifications. Edges and tokens are added (tokens are also removed) in the graph for ordering the execution between actors allocated on the same processing element, as explained in Section 4.1.

At each invocation of the constraint, a new state Υ_{new} is computed starting from the previous one Υ_{old}; the procedure requires one to know the current (modified) graph G, described by its *Vertex* and *Token matrices*. The update procedure consists of two main phases:

- **Gathering changes**: in this phase the current graph structure is compared to the previous one. Differences are stored in a proper data structure called *UPDATES* consisting of a set of dynamic queues *UPDATES*(k) (one for each level in the D matrix but the last one). Each queue stores triples (i, j, t), where i and j are respectively the source and the destination nodes of the arc (i, j) to be re-computed and t is the number of tokens collected along the path to i. Note that, joining triples (i, j, t) and the index k of the structure *UPDATES*, we compute the coordinates, in D and Π, of the source and destination cells: in fact, (k, i, t) refers to the starting node while $(k + 1, j, t + TK_{i,j})$ is the destination node.
- **Updating the values**: in this phase, arcs in the *UPDATES*(k) queues are processed and the corresponding elements of matrices D and Π are re-computed, possibly identifying new cycles.

In the following, we describe in detail the algorithmic steps performed in each phase for the three possible types of graph modifications (arc append, token append, token remove) and the update phase.

Algorithm 3: Arc Append Operation

```
   Data: Let e = (i, j) be an arc appended from node i to j
   Data: Let n be the number of the graph nodes
   Data: Let Γ be the sum of the original graph tokens
1  begin
2      for level k in 1..n and level t in 1..Γ + n do
3          if D_(k,i,t) ≥ 0 then
4              UPDATES(k) → push(i, j, t);;
```

5.2.1. Gathering changes for an arc append operation

Let $e = (i, j)$ be an arc added from node i to j (see Algorithm 3). Intuitively, adding an arc creates new paths containing node i; such paths may possibly cover (in terms of weight) existing ones and thus update the D matrix cells referring to i. In this step we want to collect all matrix cells that need to be modified.

We remind that a path crossing node i at level (k, i, t), necessarily has $D_{(k,i,t)} \geq 0$, since $D_{(k,i,t)}$ is the maximum weight of a path of length k from a source node s to i. Therefore, we should identify in the matrix D all the elements with $D_{k,i,t} \geq 0$ (line 3) and insert the triple (i, j, t) into *UPDATES*(k) (line 4); this will trigger a re-computation of cells $D_{(k+1),j,t+TK_{i,j}}$ in the update phase.

5.2.2. Gathering changes for a token append operation

Algorithm 4: Token Append Operation

```
   Data: Let e = (i, j) be an arc appended from node i to j
   Data: Let n be the number of the graph nodes
   Data: Let Γ be the sum of the original graph tokens
1  begin
2      for level k in 1..n and level t in 1..Γ + n do
3          if D_(k,i,t) ≥ 0 then
4              UPDATES(k) → push(i, j, t);;
5              if idx(Π_(k+1,j,t)) == i then
6                  for level u' in n do
7                      if (u' != i) && (u' != j) && (V_(u',j) == 1) then
8                          for level k' ≤ k do
9                              if (D_(k,u',k') ≥ 0) && (k' + TK_(u',j) == t) then
10                                 UPDATES(k) → push(u', j, k');;
```

Let $e = (i, j)$ be the arc where we add a token. If the modification involves the insertion of both one arc and one token, the token modification is not considered and the only procedure run is that for the arc append. Otherwise, if the edge already exists (see Algorithm 4), the added token results in the modification of an *existent* path; the modified path may (a) cover other paths in D and (b) uncover previously covered ones.

Detecting situation (a) requires to process the arc e in exactly the same fashion as Section 5.2.1. The only difference is that, for each cell $D_{k,i,t} \geq 0$ (line 3) the triple (i, j, t) (line 4) will trigger a re-computation of $D_{(k+1),j,(t+1)}$ in the update phase.

Detecting whether the modified path uncovers existing ones (b) deserves a more detailed explanation. In particular, each cell in $D_{k,i,t} \geq 0$ corresponds to a path with k accumulated tokens and including node i. The addition of the new token uncovers other paths in D if, at the next level $k + 1$:

1. there is a node j having i as predecessor
2. the node i is the predecessor of j on a path with t accumulated tokens, as the arc $e = (i, j)$ previously had no token.

Formally, a re-computation of the cell $D_{(k+1),j,t}$ (referring to the j node) is required if $\Pi_{(k+1),j,t} = i$ (line 5). Since node j has *lost* its former predecessor i, performing the update requires to consider all paths ending in j at level $k + 1$. In practice this is done by reconsidering all arcs from nodes u' to j (lines 6–8), such that at

level k it holds $D_{k,u',t'} \geq 0$ (i.e. they are part of a path at level k). Hence, we have to append into $UPDATES(k)$ all triples (u', j, t') such that $D_{k,u',t'} > 0$ (for every $t' \leq t$) (lines 9, 10); this will trigger the re-computation of $D_{(k+1),j,t}$ in the update phase.

5.2.3. Gathering changes for a token remove operation

Algorithm 5: Token Remove Operation

```
Data: Let n be the number of the nodes
Data: Let Γ be the sum of the original tokens
Data: Let TK_(i,j) be the number of tokens of the arc (i, j)
Data: Let e = (i, j) be an arc appended from node i to j
1  begin
2      for level k in 1..n do
3          if D_{k,i,t} ≥ 0 then
4              UPDATES(k) → push(i, j, t);;
5              if Π_{(k+1),j,(t+1)} == i then
6                  for level u' in n do
7                      if (u'! = i) && (u'! = j) && (V_{u',j} == 1) then
8                          for level k' ≤ t do
9                              if (D_{k,u',k'} ≥ 0) && (k' + TK_{(u',j)} == t) then
10                                 UPDATES(k) → push(u', j, k');;
```

This is the dual of the previous case (see Algorithm 5). Simply at point (a) one has to re-compute cell $D_{(k+1),j,t}$ instead of $D_{(k+1),j,(t+1)}$. At point (b), if $\Pi_{(k+1),j,(t+1)} = i$ (note the $t+1$ index), then cell $D_{(k+1),j,(t+1)}$ needs to be re-computed (line 9); this requires to reconsider arcs (u', j) for each $D_{k,u',t'} \geq 0$ (with $t' \leq t$) (lines 10–14).

5.3. Updating the values of $D_{i,j,k}$

In this phase the algorithm processes the D matrix, by increasing values of the k index. At each level k, all triples in $UPDATES(k)$ are extracted; based on the (i, j, t) values in the triple, the proper cell of the D matrix is reconsidered (namely $D_{(k+1),j,(t+TK_{i,j})}$). If the computed value is higher than the current value of the cell, the D and Π matrices are updated if

$$D_{(k+1),j,(t+TK_{i,j})} < D_{k,i,t} + W_j \tag{11}$$

where W_j is the execution time of actor j and $TK_{i,j}$ is the number of tokens of the arc (i, j).

Next, the performed update has to be propagated recursively: this is done by inserting into $UPDATES(k+1)$ a triple $(j, j', t+TK_{i,j})$ for each outgoing arc having j as source (successors).

During this phase, new and weightier cycles can be found. The weightiest one is the critical path that impacts the throughput value of the graph.

Example. Let us consider again the subgraph shown in Fig. 9A. The current state of the matrices D and Π is described in Fig. 8. Assume now that the solver modifies the graph adding the edge (D_3, D_2) as reported in Fig. 9B. When *gathering changes*, the incremental algorithm detects that the actor D_3 (the source of the modification) has been considered only in the cell $D_{(3,4,0)}$: as consequence the triple (4, 3, 0), that stands for $(D_3, D_2, 0)$, is pushed in $UPDATES(3)$.

Next, the triple is extracted and evaluated in the *updating phase*: note that the edge (D_3, D_2) now "points" to the cell $D_{(4,3,0)} = -\infty$. Then, Inequality (11) is checked ($-\infty < D_{(3,4,0)} + W_{D_3} = 5 + 1 = 6$, with W_{D_3} execution time of D_3) and the cells $D_{(4,3,0)}$, $\Pi_{(4,3,0)}$ are updated with values $D_{(4,3,0)} = 6$ and $\Pi_{(4,3,0)} = (4, 0)$. Since some cell has been updated, the algorithm has to propagate the changes. This is done by pushing into $UPDATES(4)$ the triple that refers to the successor C_1 of the node D_2: the triple is (3, 0, 0). When this triple is pulled from the vector $UPDATES(4)$, Inequality (11) is evaluated and the cells $D_{(5,0,1)} = 7$, $\Pi_{(5,0,1)} = (3, 0)$ are updated.

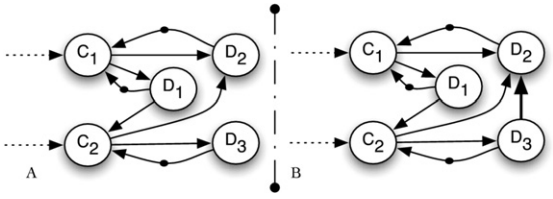

Fig. 9. Sub-graphs of Fig. 2.

	C1	C2	D1	D2	D3		C1	C2	D1	D2	D3
$D_{i,j,0}$	0	1	2	3	4	Π	0	1	2	3	4
0	0	-∞	-∞	-∞	-∞	0	-1	-∞	-∞	-∞	-∞
1	-∞	-∞	2	2	-∞	1	-∞	-∞	0,0	0,0	-∞
2	-∞	3	-∞	-∞	-∞	2	-∞	2,0	-∞	-∞	-∞
3	-∞	-∞	-∞	5	**5**	3	-∞	-∞	-∞	1,0	**1,0**
4	-∞	-∞	**6**	-∞	-∞	4	-∞	-∞	**4,0**	-∞	-∞
5	-∞	-∞	-∞	-∞	-∞	5	-∞	-∞	-∞	-∞	-∞
$D_{i,j,1}$	0	1	2	3	4	Π	0	1	2	3	4
0	-∞	-∞	-∞	-∞	-∞	0	-∞	-∞	-∞	-∞	-∞
1	-∞	-∞	-∞	-∞	-∞	1	-∞	-∞	-∞	-∞	-∞
2	3	-∞	-∞	-∞	-∞	2	2,0	-∞	-∞	-∞	-∞
3	-∞	-∞	-∞	-∞	-∞	3	-∞	-∞	-∞	-∞	-∞
4	6	6	-∞	-∞	-∞	4	3,0	4,0	-∞	-∞	-∞
5	**7**	-∞	-∞	-∞	-∞	5	**3,0**	-∞	-∞	-∞	-∞

Fig. 10. Matrices $D_{i,j,0}$, $\Pi_{i,j,0}$, $D_{i,j,1}$ and $\Pi_{i,j,1}$.

Moreover, the "loop find" procedure finds a new critical cycle that impact on the throughput upper bound. The new cycle is $C_1 \rightarrow D_1 \rightarrow C_2 \rightarrow D_3 \rightarrow D_2 \rightarrow C_1$) that refers to the throughput of the graph that corresponds to throughput value $\frac{1}{7}$ $\left(\frac{tok(D_{5,0,1}) - tok(D_{0,0,0})}{D_{5,0,1} - D_{0,0,0}}\right)$. The modified matrices D and Π are reported in Fig. 10.

The theoretical worst-case complexity of the incremental algorithm is $O(n^4)$, the same as the non-incremental version. However, in practice the number of performed operations is much lower, as pointed out by the experimental results (see Section 7). In fact, the average complexity of the incremental algorithm depends on the number of graph modifications, and this value is rarely high.

5.4. Further optimizations

In this section we describe several improvements to the global throughput constraint described in Section 5 that speed up the throughput computation. Optimization concerns three aspects: first, the non-strictly connected components are removed as they do not contribute to the throughput computation, the cycles are partitioned into multiprocessor and single processor cycles and considered separately. In the following we detail the optimization's performance:

5.4.1. Removing the non-strictly connected components

Since the throughput value is cycle dependent, nodes not belonging to any cycle are useless. A filtering algorithm has been implemented to recursively remove the non-strictly connected components from the graph. The result is a graph composed by a set of strictly connected sub-parts.

5.4.2. Single-processor execution time bound

A first very trivial bound on the throughput value can be computed by considering cycles on each processor. During search, as actors are allocated, edges and tokens are added to the graph to guarantee the non overlapping execution of the nodes over the

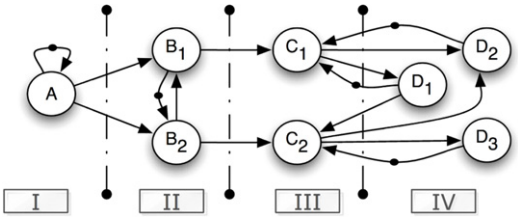

Fig. 11. An HSDFG allocation example.

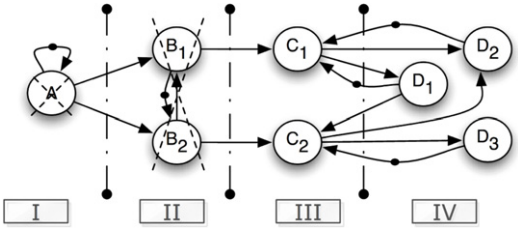

Fig. 12. An HSDFG allocated and optimized.

processors. This is done by setting a cyclic path that orders the actor execution over each processor.

Let us call Ω_p the sum of the execution times of each actor allocated on processor p. Ω_p is the inverse of the maximal throughput (Tp_p) that the processor p can achieve: $\Omega_p = \frac{1}{Tp_p} = \sum_{i \in I_p} W_i$, where I_p is the set of actors allocated on p and W_i is the execution time of the actor i.

$Tp_{min} = \frac{1}{\max_{p \in Proc} \Omega_p}$ is a throughput upper bound that must be higher than the current lower bound otherwise the search is stopped and the solver backtracks.

5.4.3. Single-processor cycle pruning

The key idea is that a processor p can be part of a *multiprocessor* cycle if and only if its actors have at least one input and one output edge that connect them to actors onto other processors. We can now remove every remaining *single-processor* cycle, since its impact over the throughput has been considered by computing TP_{min}. The result is a reduced graph which consists of actors allocated on processors that communicate with each other.

These optimizations filter the graph by removing nodes that do not contribute to the throughput computation.

Consider for example the HSDFG reported in Fig. 11; it has eight actors allocated onto four different processors (I…IV). It is composed only by strictly connected components, so the first optimization (Section 5.4.1) is not employed. Then, assuming that the computation of Ω_p^{max} does not force a backtrack on the search process (see Section 5.4.2), the latter optimization is executed (see Section 5.4.3).

Since the actor A, allocated on processor I, has only outgoing arcs, it cannot be part of a multiprocessor cycle. Thus it is removed from the graph. As a consequence, actors allocated on II (B_1 and B_2) "lose" their ingoing edges. For this reason they are recursively removed.

The algorithm, in this example, computes the bound over the subgraph composed by actors C_1, C_2, D_1, D_2, D_3 reducing the overall computation time of the throughput constraint (see Fig. 12).

6. Search

CP problems are generally solved via tree search. Constraint propagation is used to narrow the search space, but many branching choices still have to be explored during search. Hence, the efficiency of CP solvers heavily depends on good heuristics to prioritize branching decisions. The main purpose of a search heuristic is to quickly find a solution that ensures a tight bound to drastically reduce the search space.

Search Heuristics. Experimental tests evidenced that branching over resource allocation variables has far-reaching implications over the throughput values, therefore our heuristic function evaluates these variables first. Focusing on the more "decisive variables" first is more likely to lead to good solutions.

The heuristic function we propose is divided into two components:

- the variable selection heuristic intuitively gives priority to actors whose execution has more impact on the throughput value. This is achieved by giving higher rank (low value) to tasks with longer execution time and also giving priority to actors whose execution enables the execution of other nodes. The node i chosen by the heuristic is the one with minimal value of the following expression:

$$\frac{\alpha \cdot Tmax}{\tau_i} + \frac{\beta \cdot dep_i}{Dmax} \tag{12}$$

  where $Tmax$ corresponds to the maximal node execution time, τ_i is the execution time of the node i. dep_i corresponds to the number of nodes which precede actor i, and $Dmax$ the maximum over these values (a node with a low $\frac{dep_i}{Dmax}$ means that it is execution depends on few other nodes, therefore it could execute earlier than a node with a higher $\frac{dep_i}{Dmax}$, i.e. whose execution depends on more nodes). The heuristic function combines two distinct components, with relative weight set by two coefficients. The coefficients α and β have been defined experimentally, and their values are respectively 0.68 and 0.32 ($\alpha = 1 - \beta$).
- The value (resource) selection heuristic beside balancing the load, tends to allocate on the same processor actors that are tightly linked by precedence constraints. This function tries to reduce the number of dependencies between tasks on different processors. This is achieved by selecting first the processor p that minimizes the following expression:

$$\frac{\delta \cdot WL_p}{WLmax} + \frac{\theta \cdot con_p}{Cmax} \tag{13}$$

  where WL_p corresponds to the actual processor workload, i.e., the total execution time of the actors allocated on it. $WLmax$ is the highest workload over all processors. The value con_p is the total duration of the nodes that are *non-dependent* on p, and $Cmax$ is the highest of these numbers. The coefficients δ and θ are 0.79 and 0.21 respectively.

Note that the coefficients of the heuristic functions have been experimentally tuned: 1000 heterogeneous instances were solved with 20 different combination of coefficient values. The values of the best average solution quality were chosen.

The experimental results show (see Section 7) that the described heuristics obtain one order of magnitude speed up w.r.t. search using lexicographic ordering. Finally, symmetry due to homogeneous processors are broken at search time; namely, whenever an allocation decision has to be taken, if there is more than one free processor, the one with the lowest index is chosen.

7. Experimental results

We have extensively evaluated our approach for assessing three aspects: (1) the performance of the incremental throughput algorithm in comparison with the non-incremental version, (2) the scalability of the allocation and scheduling SDFG framework and (3) the quality of the solution found. The synthetic instances were

Table 1
Search and constraint execution times and speed-up.

Type	Node	SrcNInc	SrcInc	SrcSpUP	CstNInc	CstInc	CstSpUP
Cyclic	10	2.022	0.64	**2.174**	1.443	0.11	**11.88**
	12	35.86	2.72	**12.18**	27.86	0.88	**30.66**
	15	3504.43	37.64	**92.11**	2980.28	9.43	**315.04**
Strictly connected	10	3.063	0.89	**2.421**	2.264	0.15	**14.12**
	12	58.29	4.32	**12.49**	46.99	1.3	**35.15**
	15	4231.02	52.64	**79.18**	3546.98	10.92	**323.81**
Acyclic	10	3.88	1.213	**2.19**	2.77	0.16	**16.31**
	12	105.48	14.23	**6.41**	82.87	1.94	**41.72**
	15	5968.58	143.58	**40.57**	5106.05	18.89	**269.31**

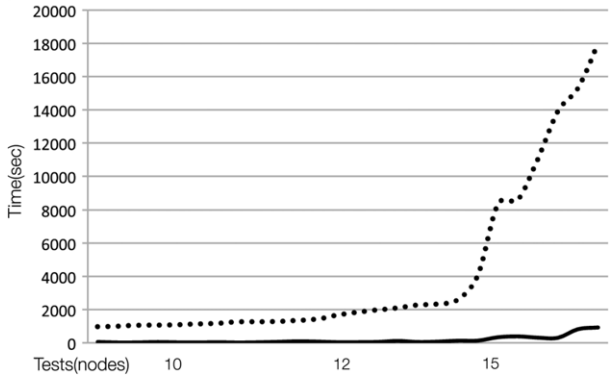

Fig. 13. Graphical representation of the speed-UP.

Table 2
Search execution times.

Node	CMedian	CMax	AMedian	AMax
10	0.12	11.84	0.53	13.44
12	0.96	36.84	1.98	47.77
14	12.67	146.75	27.15	275.56
16	63.33	446.24	96.08	659.32
18	187.64	837.32	269.43	1134.37

Node	% Non-Incr	% Incr
10	72.24	15.35
12	79.01	15.99
15	82.91	19.32

Fig. 14. Relative algorithms computation time.

built by means of the `sdf3` (see [44]) task-graph generator, designed to produce graphs with realistic structure and parameters.[2] The instances were solved using a workstation with a 3.3 GHz Core 2 Duo processor and 8 GB of RAM. The system described so far was implemented on top of ILOG Solver 6.3.

7.1. Incremental algorithm evaluation

The following section proves the effectiveness of the proposed incremental algorithm (see Section 5) by comparing its computational time with respect to its non-incremental version on a set of 4500 instances. We have generated three sets of realistic task graph instances featuring 10, 12 and 15 nodes. Each set includes cyclic, acyclic and strictly connected graphs. For these experiments we assume that two homogeneous processors are available.

In Table 1 the first two columns (SrcNInc, SrcInc) refer to the Total Search Time for finding the solution with maximum throughput with the non-incremental and incremental algorithm version. We can see that the solver with the incremental algorithm runs up to 90 times faster (see the speed-up column, SrcSpUP). The values in Table 1 represent the average over 500 instances. The remaining three columns (CstNInc, CstInc, CstSpUP) report respectively the computational times of the throughput filtering algorithm and the corresponding speed-up. The speed-up column shows that the incremental filtering algorithm gains over one order of magnitude speed-up w.r.t. its non-incremental version. Moreover the speed-up tends to increase with the dimension of the problem instance. The acyclic graphs are the most tough to solve, as they feature relatively fewer arcs compared to the cyclic and to the strictly connected ones; this results in a higher number of possible scheduling choices and a larger search tree.

The problem faced is NP-hard and clearly the computational time grows exponentially in the instance dimension. However, the reduced time for constraint computations in the incremental solver increases scalability and enables the solution of harder and larger problems.

This is clear in Fig. 13, where the two reported lines represent the total time for the throughput constraint; the x axis has an entry for each instance. The dotted line refers to the non-incremental solver, the solid line to the incremental one. Instances are sorted according to the number of nodes. We can notice that the throughput computation time for the new incremental algorithm grows much more slowly.

The table in Fig. 14 shows the relative amount of time that the algorithm computation absorbs during the search. It is evident that the new algorithm is definitely faster and lighter. Its impact on the search time is lower than 20% of the total time while the non-incremental version time exceeds 70%. Further experiments show the average impact on the search time using the non-incremental algorithm with the optimization described in Section 5.4 is 57.3%.

7.2. Overall solver experimental evaluation

We have evaluated the scalability of our approach on various sets of synthetic instances, designed to match structure and features of realistic applications. We considered both cyclic and acyclic Homogeneous Synchronous Data-Flow Graphs. In particular our approach tends to be more effective on cyclic graphs; in fact, if a graph contains cycles, it has an implicit throughput upper bound defined by the longest loop in the graph. In contrast, acyclic graphs have no implicit bound and expose the highest parallelism: this makes them the most challenging instances.

For this experiments we assume that four homogeneous processors are available. The generated graph have been divided according to the number of nodes (from 10 to 18). Table 2 presents the median and maximum computing time for cyclic (2nd and 3rd column) and acyclic (4th and 5th) instances. A time limit of 1200 s was set on all the experiments. As expected, the average running time grows exponentially with the size of the instances. However,

[2] The generator tends to produce tasks with high execution time variance (therefore representing the difference between the loading/storing task w.r.t. the faster executing ones) and with an average number of outgoing arcs that ranges from 1.1 to 1.3. These coefficients produce SDF graphs that, transformed into HSDFG, will resemble to applications with high data-parallelism.

Table 3
Experiments on real benchmarks.

Name	Nodes	Arcs	OPT	SMS (%)	SDF^3 (%)	5 s (%)	10 s (%)	30 s (%)	60 s (%)	300 s (%)
Sobel	5	15	**0.001**	0	0.62	**0**	–	–	–	–
JPEG2000	8	10	**0.07**	0	5.29	**0**	–	–	–	–
Motion JPEG	12	15	**105.56**	8.12	0	0	0	0	0	**0**
MPEG	12	14	**75.91**	6.45	7	10.84	10.84	10.84	10.84	**0**
MPEG-2	12	14	**53.31**	9.46	47.7	10.34	10.34	0	**0**	–

the solution time is reasonable for graphs up to 20 nodes which is a realistic size for many real world applications.

To the best of the authors knowledge, this is the first complete approach that handles cyclic/acyclic synchronous data-flow graphs; this makes any comparison with existing methods more difficult, as they are all incomplete. Incomplete approaches feature higher scalability, but provide sub-optimal solutions.

We compare our method on five real benchmark with two state-of-the-art incomplete approaches: (1) the *Swing Modulo Scheduling* (SMS) approach, used by the GCC compiler [20] and (2) the "SDF^3" tool, presented in [44]. In order to have a fair comparison, both approaches have been slightly modified. In particular (1) SMS has been re-implemented considering CPU registers as MPSoC cores while (2) "SDF^3" has been modified along with the authors by disregarding memories/latencies constraints. The focus of these experiments is to assess the effectiveness on practically significant embedded multimedia applications, such as Sobel, JPEG2000, Motion JPEG, MPEG and MPEG-2. These instances were developed as benchmarking work for the *Mapping Applications to MPSoCs 2009* workshop [4].

Results are presented in Table 3. The first three columns report the name of the application, the number of tasks and the number of arcs, respectively. The following three columns (4–6) refer to the optimal solution computation time (solver runs until the optimality was proved) and optimality gap of the solution found by both approaches. The optimality gap represents the distance of the solution *Sol* from the optimal one *Opt* and it is computed in the following mode: $\text{Gap}(\%) = 100 * \frac{Opt - Sol}{Opt}$. Note that the SMS method found all the solutions within 5 s while SDF^3 within a second. Each following column (7–11) refers to a different search time limit (5, 10, 30, 60, 300 s respectively) and presents the optimal gap of our approach. All the instances were optimally solved within 106 s. The easiest instances, *Sobel* and *JPEG2000* were solved within a second. The *Motion JPEG* solution computed within the first five seconds is the optimal solution (it is proved after 105.56 s) while the SMS solution features a 8.12% gap. For both *MPEG* and *MPEG-2* applications, the SMS approach initially found a better solution; however note that our approach computes the optimal within 75.91 and 53.31 s respectively. The "SDF^3" tool is the fastest approach, however its solutions present an average gap of 12.1% (with a peak of 47.7% in the MPEG-2 benchmark) w.r.t. the 4.8% of the SMS. This experimentation shows that in real contexts the proposed solver can compute good quality solutions in terms of seconds (and the optimal solution within a few minutes).

Then we performed a further comparison with the simulation based procedure described in [43] on synthetic graphs with a manageable size for our approach. Despite the incomplete approach being much faster then our tree search procedure, the solution provided by the incomplete method was found to be on average 20% worse than the optimal one.

Note that when the number of nodes becomes larger, the search could be stopped after a certain time limit (or a given number of feasible solutions), thus obtaining an incomplete approach. Differently from other incomplete approaches, our use of tree search and constraint propagation enables to find feasible solutions of tightly-constrained problems. We can compute a feasible solution of thousand-node graphs in terms of seconds.

Table 4
Optimality gaps of incomplete searches.

Instance size	*Const*. sol. (%)	First sol. (%)	First *Const*. sol. (%)
10 Nodes	82.66	78.48	75.86
12–15 Nodes	77.44	78.18	66.72

7.3. Solution quality evaluation

We finally designed a third set of experiments, to evaluate the search heuristics; tests were performed on a new large (1000 graphs) set of synthetic instances (see also [10]). Given an initial SDF graph, we perform mapping under two different assumptions: one allocates and schedules the derived HSDFG actors independently, while the second forces the allocation of all the HSDFG nodes corresponding to repetitions of the original SDFG nodes on the same processor. We refer to the first type of allocation as *unconstrained* and to the second type as *constrained*; the latter is typically obtained by approaches working directly on the SDFG, without a preliminary transformation into HSDFG [32,43,44].In Table 4 we provide comparisons among the throughput achievable by complete search with the *unconstrained* and *constrained* approaches. This allows to assess the solution quality loss due to the use of a more restrictive assumption. We give the optimality gap of the *constrained* solution (*Const*. sol.), the first feasible *unconstrained* solution (First sol.) and the first feasible *constrained* solution (First *Const*. sol.). The optimality gap widens as the number of nodes increases: on medium-size instances the optimal *unconstrained* throughput is about 20% higher than the *constrained* solution. This clearly demonstrates that the additional degrees of freedom enabled by mapping multiple actor iterations on different processors help in finding higher throughput solutions.

The second and third column in the table refer to incomplete versions of the search procedure, which could be used to find fast, but sub-optimal solutions. In the second column we report the optimality gap obtained by stopping the search after the first solution found by tree search driven by our heuristic functions described in Section 6. The optimality gap is around 22%, which is significant but not enormous. This implies that the solver finds a reasonably good solution in a very short time, regardless of the exponential search effort required to reach the actual optimum. Thus, our strategy is quite effective even when used as a fast, incomplete search.

The first feasible *constrained* solution provides an estimate of the quality one could expect from the solution provided by an incomplete algorithm which map directly the SDFG. As expected, it has the largest optimality gap, with a 30% loss in throughput. This result gives a clear indication that our algorithm provides a significant quality improvement with respect to previously presented incomplete algorithms.

8. Conclusions

The widespread use of multi-core processors is pushing explicitly parallel high-level programming models to the forefront. Stream computing based on a data-flow model of computation [29] is viewed by many as one of the most promising programming

paradigms for embedded multi-core computing. Our work addresses one of the key challenges in the development of programming tool-flow for stream computing, namely, the efficient mapping of synchronous data-flow graphs onto multi-core platforms.

We presented a CP-based method for allocating and scheduling HSDFGs on multiprocessor platforms; to the best of our knowledge this is the first CP-Based complete approach for the target problem. The core of the system is a global throughput constraint embedding an incremental extension of the computation procedure described in [11], which proved to be crucial for the performance. Our method obtains promising results on realistic size graphs.

We are developing an environment to support the design flow of streaming application for multi-core platforms, from the early programming stages to the actual validation on real architectures. This will require us to take in account communication costs and buffer size limits.

Acknowledgment

The work described in this publication was supported by the SMECY Project, JTI ARTEMIS, Grant agreement no.: 100230.

Appendix. Constraint programming

The allocation and scheduling algorithm for SDFG we propose in this paper is based on Constraint Programming. Constraint Programming (CP) [17,38] is a programming paradigm used to solve hard combinatorial problems. It is currently applied with success to many domains such as planning, vehicle routing, configuration, scheduling and bioinformatics [1–3,6].

The key concept of constraint programming is the clear separation between *constraint modeling* and *constraint solving*.

A constraint model is defined in terms of variables and constraints. Each variable X_i has an associated domain D_i containing values that the variable can assume (the notation for linking variables and domains is $X_i :: D_i$). Constraints define combinations of consistent assignments (i.e., a subset of the Cartesian product of the variable domains). The model might have an objective function defining a (possibly partial) order in the solution space.

Once the constraint model is stated, constraint solving is started by interleaving propagation and search. The search process enumerates all possible variable-value assignments (possibly guided by a proper variable and value selection heuristics), until we find a solution or we prove that none exists. To reduce the exponential number of variable-value pairs in the search tree, domain filtering and constraint propagation are applied at each node of the search tree. Domain filtering operates on individual constraints and removes provably inconsistent domain values. Since variables are involved in several constraints, domain updates are propagated to the other constraints whose filtering algorithms are triggered and possibly remove other domain values.

As domain filtering is local to each constraint, it is a common practice in Constraint Programming to define the so called global constraints,[3] that compactly represent combination of elementary constraints, but embed more powerful filtering algorithms exploiting a global view.

Constraint propagation is not complete. This means that if a value is removed by a filtering algorithm it is proved to be infeasible. Instead, if a value is left in the domain of a variable, it can happen that it does not belong to any consistent solution. For this reason, tree search is employed to explore the values left in the domain. At each node of the search tree, constraint propagation is triggered thus interleaving propagation and search. As far as search is concerned two main factors affect the solution process: the early evaluation of a partial solution and the variable-value selection strategy. The former is usually performed via an (upper/lower) bound computation, the latter is a heuristic function that guides the search.

[3] As an example consider the *AllDiff* ($[X_1 \ldots X_n]$) constraint [37]. Declaratively it is equivalent to a set of pairwise inequalities ($X_i \neq X_j, \forall i \neq j$). However, by reasoning globally, it infers more deletions in general. Consider the following variables and their domain: $X :: [1, 2, 3]$ $Y :: [1, 2]$ $Z :: [1, 2]$ and the following constraint: *AllDiff* (X, Y, Z). By considering the set of elementary constraints ($X \neq Y, Z \neq Y, X \neq Z$) the propagation cannot remove any value, while the AllDiff global constraint removes values $[1, 2]$ from X as they should be assigned (no matter how) to Y and Z. The *AllDiff* constraint leverages network flow algorithms to perform the described filtering in polynomial time [37].

References

[1] D. Baatar, N. Boland, S. Brand, P.J. Stuckey, Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches, in: Proc. of CPAIOR '07, pp. 1–15.
[2] P. Baptiste, C. Le Pape, W. Nuijten, Constraint-Based Scheduling, Springer, 2001.
[3] R. Bartak, M. Salido, Constraint satisfaction for planning and scheduling problems, Constraints 16 (2011) 223–227. 10.1007/s10601-011-9109-4.
[4] http://www.artist-embedded.org/artist/benchmarks.html, 2009.
[5] L. Benini, E. Flamand, D. Fuin, D. Melpignano, P2012: building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator, in: Proc of. DATE 2012, pp. 983–987.
[6] R. Bent, P.V. Hentenryck, Randomized adaptive spatial decoupling for large-scale vehicle routing with time windows, in: Proc. of AAAI '07, pp. 173–178.
[7] S.S. Bhattacharyya, S. Sriram, Embedded Multiprocessors—Scheduling and Synchronization (Signal Processing and Communications), second ed., CRC Press, 2009.
[8] T. Bijlsma, M. Bekooij, P. Jansen, G. Smit, Communication between nested loop programs via circular buffers in an embedded multiprocessor system, in: Proceedings of the 11th International Workshop on Software Compilers for Embedded Systems, SCOPES '08, pp. 33–42.
[9] G. Blake, R. Dreslinski, T. Mudge, A survey of multicore processors, IEEE Signal Process. Mag. 26 (2009) 26–37.
[10] A. Bonfietti, L. Benini, M. Lombardi, M. Milano, An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms, in: Proc. of DATE '10, pp. 897–902.
[11] A. Bonfietti, M. Lombardi, M. Milano, L. Benini, Throughput constraint for synchronous data flow graphs, in: Proc. of CPAIOR '09, pp. 26–40.
[12] W. Che, K.S. Chatha, Compilation of stream programs for multicore processors that incorporate scratchpad memories, in: Design, Automation and Test in Europe Conference, DATE, DATE '10.
[13] W. Che, K.S. Chatha, Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming, in: Design Automation Conference, DAC, DAC '11.
[14] Y. Choi, Y. Lin, N. Chong, S. Mahlke, T. Mudge, Stream compilation for real-time embedded multicore systems, in: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 210–220.
[15] A. Dasdan, R. Gupta, Faster maximum and minimum mean cycle algorithms for system-performance analysis, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 17 (1998) 889–899.
[16] A.G. David, E. Culler, Jaswinder Pal Singh, Parallel Computer Architecture: A Hardware/Software Approach, Gulf Professional Publishing, 1999.
[17] R. Dechter, Constraint Processing, Morgan Kaufmann, 2003.
[18] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.
[19] A.H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B.D. Theelen, M.R. Mousavi, A.J.M. Moonen, M.J. Bekooij, Throughput analysis of synchronous data flow graphs, in: Proc. of ACSD '06, pp. 25–36.
[20] M. Hagog, A. Zaks, Swing Modulo Scheduling for GCC, Technical Report, 2004.
[21] W. Haid, K. Huang, I. Bacivarov, L. Thiele, Multiprocessor SoC software design flows, IEEE Signal Process. Mag. 26 (2009) 64–71.
[22] A.H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, S. Mahlke, Flextream: adaptive compilation of streaming applications for heterogeneous architectures, in: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 214–223.
[23] K. Ito, K.K. Parhi, Determining the minimum iteration period of an algorithm, J. VLSI Signal Process. 11 (1995) 229–244.
[24] L. Karam, I. Alkamal, A. Gatherer, G. Frantz, D. Anderson, B. Evans, Trends in multicore DSP platforms, IEEE Signal Process. Mag. 26 (2009) 38–49.
[25] M. Karczmarek, W. Thies, S. Amarasinghe, Phased scheduling of stream programs, in: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, in: LCTES '03, ACM, 2003, pp. 103–112.

[26] R. Karp, A characterization of the minimum cycle mean in a digraph, Discrete Math. 23 (1978) 309–311.
[27] R.M. Karp, R.E. Miller, Properties of a model for parallel computations: determinancy, termination, queueing, SIAM J. Appl. Math. 14 (1966) 1390–1411.
[28] M. Kudlur, S. Mahlke, Orchestrating the execution of stream programs on multicore platforms, in: Proc. of PLDI '08, vol. 43, pp. 114–124.
[29] E.A. Lee, S.S. Bhattacharyya, P.K. Murthy, Software Synthesis from Data Flow Graphs, Kluwer Academic Press, 1996.
[30] E.A. Lee, D.G. Messerschmitt, Synchronous data flow, Proc. IEEE 75 (1987) 1235–1245.
[31] E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. 36 (1987) 24–35.
[32] O. Moreira, J.D. Mol, M.J. Bekooij, J. van Meerbergen, Multiprocessor resource allocation for hard–real-time streaming with a dynamic job-mix, in: 11th IEEE Real Time and Embedded Technology and Applications Symposium, RTAS'05, IEEE, 2005, pp. 332–341.
[33] O. Moreira, F. Valente, M. Bekooij, Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor, in: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, in: EMSOFT '07, ACM, New York, NY, USA, 2007, pp. 57–66.
[34] F.A. Omara, M.M. Arafa, Genetic algorithms for task scheduling problem, J. Parallel Distrib. Comput. 70 (2010) 13–22.
[35] C. Ostler, K.S. Chatha, V. Ramamurthi, K. Srinivasan, ILP and heuristic techniques for system-level design on network processor architectures, ACM Trans. Des. Autom. Electron. Syst. 12 (2007) 48-es.
[36] G. Pesant, M. Gendreau, J.y. Potvin, J.m Rousseau, An exact constraint logic programming algorithm for the traveling salesman problem with time windows, Transp. Sci. 32 (1996) 12–29.
[37] J.C. Régin, A filtering algorithm for constraints of difference in CSPs, in: Proc. of AAAI '94, pp. 362–367.
[38] F. Rossi, P. Van Beek, T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.
[39] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, L. Benini, A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness, Int. J. Parallel Program. 36 (2008) 3–36.
[40] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, L. Benini, A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness, Int. J. Parallel Program. 36 (2008) 3–36. 10.1007/s10766-007-0032-7.
[41] J. Sermulins, W. Thies, R. Rabbah, S. Amarasinghe, Cache aware optimization of stream programs, in: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '05, ACM, 2005, pp. 115–126.
[42] S. Sriram, E.A. Lee, Determining the order of processor transactions in statically scheduled multiprocessors, J. VLSI Signal Process. 15 (1997) 207–220.
[43] S. Stuijk, T. Basten, M. Geilen, H. Corporaal, Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs, in: Proc. of DAC '07, IEEE, 2007, pp. 777–782.
[44] S. Stuijk, M. Geilen, T. Basten, SDF$\hat{3}$: SDF For Free, in: Proc. of ACSD '06, IEEE, 2006, pp. 276–278.
[45] W. Thies, M. Karczmarek, S.P. Amarasinghe, Streamit: a language for streaming applications, in: Proceedings of the 11th International Conference on Compiler Construction, in: CC'02, Springer-Verlag, London, UK, 2002, pp. 179–196.
[46] B. Ucar, C. Aykanat, K. Kaya, M. Ikinci, Task assignment in heterogeneous computing systems, J. Parallel Distrib. Comput. 66 (2006) 32–46.
[47] M.H. Wiggers, M.J.G. Bekooij, G.J.M. Smit, Efficient computation of buffer capacities for cyclo-static dataflow graphs, in: Proceedings of the 44th Annual Design Automation Conference, DAC '07, pp. 658–663.

Alessio Bonfietti is currently a Ph.D. candidate working at the Department of Electrical Engineering and Computer Science (DEIS), University of Bologna, Italy. He received his Bachelors Degree in 2005 and his Masters Degree in 2007, both in Computer Science from the University of Bologna. Since his graduation, his research interests are in the areas of mapping and scheduling of periodic applications on multi-core architectures and Constraint Programming. He has published papers and performed reviews for international conferences and journals.

Michele Lombardi is a Post-Doctoral Fellow at DEIS, University of Bologna; his research activity is related to Constraint Programming and its integration with Integer Programming and Artificial Intelligence techniques; in particular, his focus is on resource allocation and scheduling problems. Michele Lombardi obtained a Ph.D. in Computer Engineering at the University of Bologna, performing internships at EPFL Lausanne (CH) and Cornell University (NY). He has published papers and performed reviews for international conferences and journals and he is in the program committees of IJCAI 2011 and CPAIOR 2011.

Michela Milano has a Ph.D. in Computer Science and is an Associate Professor at the University of Bologna since 2001. Her research interest is in the area of Constraint Programming and its integration with Integer Linear Programming. In this field Michela Milano has achieved international visibility. She is Program Chair of CPAIOR 2005 and CPAIOR 2010. She has published more than 100 papers on peer reviewed international journals and conferences and edited two books on Hybrid Optimization. She is Area Editor of the INFORMS Journal on Computing, Constraint Programming Letters and a Member of the Editorial Board of the Constraint International Journal. She is Project Coordinator of the EU-FP7 project e-POLICY: Engineering the Policy Making life cycle, that fosters the use of optimization and decision support techniques for aiding political decision making and impact assessment.

Luca Benini is Full Professor at the Department of Electrical Engineering and Computer Science (DEIS) of the University of Bologna. He also holds a visiting faculty position at the Ecole Polytechnique Federale de Lausanne (EPFL) and he is currently serving as Chief Architect for the Platform 2012 project at STmicroelectronics, Grenoble. He received a Ph.D. degree in Electrical Engineering from Stanford University in 1997. Dr. Benini's research interests are in energy-efficient system design and Multi-Core SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks for biomedical and ambient intelligence applications. He has published more than 500 papers in peer-reviewed international journals and conferences, four books and several book chapters. He is a Fellow of the IEEE and a Member of Academia Europea, and of the steering board of the ARTEMISIA European Association on Advanced Research & Technology for Embedded Intelligence and Systems.