An Efficient and Complete Approach for Throughput-maximal SDF Allocation and Scheduling on Multi-Core Platforms

Alessio Bonfietti, Luca Benini, Michele Lombardi and Michela Milano
DEIS - Universitá di Bologna

***Abstract*—Our work focuses on allocating and scheduling a synchronous data-flow (SDF) graph onto a multi-core platform subject to a minimum throughput requirement. This problem has traditionally be tackled by incomplete approaches based on problem decomposition and local search, which could not guarantee optimality. Exact algorithms used to be considered reasonable only for small problem instances. We propose a complete algorithm based on Constraint Programming which solves the allocation and scheduling problem as a whole. We introduce a number of search acceleration techniques that significantly reduce run-time by aggressively pruning the search space without compromising optimality. The solver has been tested on a number of non-trivial instances and demonstrated promising run-times on SDFGs of practical size and one order of magnitude speed-up w.r.t. the fastest known complete approach.**

I. Introduction

Synchronous Dataflow Graphs (SDFGs) [2] are used to model streaming applications with explicit data communication. They have a precisely defined execution semantics and several key properties of SDFG execution (e.g. bounded buffering) can be verified efficiently. The interest in SDF and related models of computations has picked up momentum in the last few years, as they match well the specification style used in many embedded computing domains (signal processing, multimedia), and even more importantly, they can be mapped efficiently on up-coming multi-core platforms for embedded computing [1].

In this work we focus on throughput maximization and throughput-constrained SDFG mapping on multi-core platforms. This is a practically relevant allocation and scheduling problem, as in most stream computing applications a hard or soft requirement on the minimum acceptable throughput is specified, e.g. in term of samples-per-second or frame-rate. A number of researchers have addressed this problem in the last two decades [2], [3], with a flurry of recent studies triggered by the "multi-core revolution" [5], [6]. The NP-Hardness proof of SDFG mapping was given by E. Lee [2] and since then this problem has been addressed by incomplete (sometimes called heuristic) algorithms which provide "good" solutions, with no guarantee of optimality, and often quite loose bounds on optimality gap (i.e. the difference between the max-throughput solution found by the solver and the maximum throughput achievable).

Most incomplete solution strategies decompose the mapping problem into an allocation step followed by a scheduling step. In allocation, SDFG nodes, corresponding to atomic units of computation, are assigned to processors for execution, scheduling orders these executions in time, trying to match throughput constraints. Decomposition leads by itself to optimality loss. In addition, even the two sub-problems (which remain NP-Hard) are solved via incomplete search. Very recently, a new complete approach based on constraint programming (CP) was proposed [7], which could solve realistic instances. The present work moves several steps further in the same direction: we start from the same CP formulation as [7] and we address its main computational bottlenecks, namely the throughput bound computation and the search tree exploration strategy.

We introduce several novel techniques to speed-up the computation of throughput bounds and to enhance the efficiency of the search, by jump-starting with a high-quality bound and rapidly tightening it, thereby pruning huge fractions of the search space. Experimental results show that the combined effect of these two enhancement leads to an order-of-magnitude reduction in solution time, and also provides effective incomplete search options for problems which still cannot be solved within the allotted time budget. In essence, our work demonstrates for the first time that complete search techniques for optimal SDFG mapping are practically viable.

II. Related Work

We address the problem of SDFG mapping (actor allocation and scheduling) on a target platform. This problem has been studied extensively in the past. However, researchers have mostly focused on incomplete mapping algorithms for the allocation and scheduling problem (see [5], [6]). The reason for the use of incomplete approaches is that both computing an optimal allocation and an optimal schedule is NP-hard.

The first class of approaches, pioneered by the group lead by E. Lee [3] and extensively explored by many other researchers [4], 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 [4] into an Homogeneous SDF graph (HSDF). The transformation procedure is based on the iteration vector (also called *repetition vector*) and produces an homogeneous graph (with all the edge rates one) that has a node for any repetition of each actor of the original SDF graph. 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 constraints. Unfortunately, throughput depends on *both* allocation and scheduling. This is not an easy problem to solve, as throughput depends on the order of execution of actors that can be allocated to different processors, and we have an exponential blowup of the solution space. Incomplete search strategies are used, such as list scheduling, driven by a priority function related to throughput.

A different class of approaches [5] works directly upon SDF graphs, without the HSDFG transformation. This approach has the advantage to avoid the potential blow-up in the number of actors, with the disadvantage that no feasible solution is found if problem constraints are tight. 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 end scheduling information until a fixed point is reached.

The incomplete approaches summarized above obviously cannot give any proof of optimality. 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 [5].

In this work we describe a complete method that improves and extends the algorithm presented in [7]. Our approach is based on constraint programming and faces the allocation and scheduling problem as a whole.

III. Constraint Programming

Constraint Programming (CP) is a methodology used to solve hard combinatorial problems. It is currently applied with success to many domains such as planning, vehicle routing, configuration, scheduling and bioinformatics. The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them.

The key concepts of constraint programming can be abstracted into two independent concepts: *Constraint Modelling* and *Constraint Solving*. A constraint model is defined in terms of variables and constraints. Each variable has an associated domain containing the values that the variable can assume. Constraints define combination of consistent assignments. They can be seen as software components embedding a filtering algorithm whose purpose is to reduce variable domains by removing provably infeasible values. The model might have an objective function. Once the constraint model is stated, constraint solving interleaves propagation and search.

Constraint propagation is a form of inference that removes values that are not contained in any feasible solution.
As a simple example, consider the following variables and their domain:
X:[1,2,3]
Y:[1,2,3]
Z:[1,2,3]
and the following constraints:

$X < Y$
$Y < Z$

From the first constraint we can observe that variable *X* can never assume value 3 and this value is removed from the domain of the variable. The same happens to value 1 for the *Y* variable. The second constraint implies that *Y* cannot assume value 3 and *Z* value 1. Variable *Y* has one value left in its domain, while the other two variables have still two values. Since the second constraint has modified the domain of the variable *Y* which is considered in the first one, the constraint propagation needs to re-evaluate it. At this point *Y* assumes value 2 forcing the propagation to reduce the domain of *X* to 1 and of *Z* to 3.

In this example the propagation preserves the solver from the search part. For efficiency reasons constraint propagation is not usually able to remove all the infeasible values from the domains and a search phase is necessary. Removing all infeasible values would simply be as difficult as solving the problem itself.

IV. CP-based HSDFG Mapping

The approach proposed in [7] tackles the allocation and scheduling problem as a whole; hence it avoids the intrinsic loss of optimality due to the decomposition into two stages. The method described in [7] is in principle complete: it is guaranteed to find an optimal solution. However, in [7] results are limited to finding a feasible solution in case a throughput bound constraint exists in the problem formulation since the optimal algorithm exceeds the time budget in almost all instances.

The basic idea of [7] is to model the effects of mapping choices by means of HSDFG modifications: during the search process, whenever allocation and scheduling decision are taken, the graph is modified accordingly.

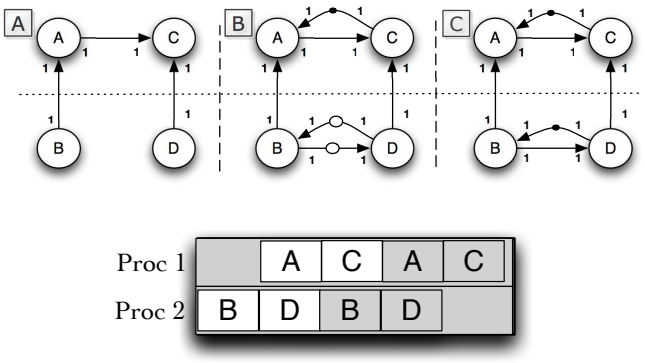

Fig. 1. The HSDFG and its scheduling

Consider for example the HSDFG in figure 1A, where all rates are assumed to be one and actors A and C are mapped on one processor, while actors B and D on another. Actors mapped on the same processor must be non overlapping in time: this is captured by adding arcs to create a cycle between each pair of independent tasks mapped on the same processor;

note this requires adding one arc for the (A, C) pair in figure 1B and two arcs for the (B, D) pair. To avoid deadlocks, a token must be placed for each of these cycles: while the choice is forced for the (A, C) pair, both arcs are suitable for (B, D) as hinted by the empty circles in figure 1B. Choosing the placement of those tokens implies scheduling decisions: for example in figure 1C the order B, D, C was chosen.

The main advantage of this approach is that a standard throughput computation algorithm (such as [8]) can be used almost off-the-shelf to evaluate the effect of each mapping decision. The throughput computation algorithm proposed in [7] is inspired to the one described in [8], with some notable extensions: it accounts for multiple tokens on arcs, while the original algorithm assumes maximum one token per arc. It is also applicable to Cyclic and Acyclic graphs, as opposed to strongly connected graphs.

Each time an allocation/scheduling decision is taken by the solver, throughput is computed on the modified graph. These values are upper bounds for the actual application throughput value which will be computed when all mapping decisions are taken.

As more mapping decisions are taken during search, edges are added to the graph, thereby increasing the likelihood of creating new cycles. Since throughput depends on the critical cycle of the graph, during search throughput bound values only decrease.

Each time one of these throughput upper bound falls below the current lower threshold (either the initial throughput requirement or the highest throughput found so far for a fully mapped graph), the solver fails since all solutions in the current subtree will be infeasible.

V. Algorithmic Optimizations

The basic CP-based mapper outlined in the previous section could solve small, but realistic instances in reasonable time [7]. However, its speed and scalability have ample margins of improvement. First, run-time is dominated by throughput computation, which is used as the sole bounding function. Second, the sequence of mapping decisions during search and the initial mapping choices are not optimized. This leads to initially loose bounds that prevent effective pruning of the search space.

In this section we describe several radical improvements to the solver described in [7], bringing drastic performance enhancements. In subsection V-A we describe a new search-directing function (also called "heuristic function" in CP terminology) which leads to early determination of much tighter bounds. In the remaining Subsections, we describe several algorithmic improvements to speed up throughput computation.

A. Heuristic Function

CP problems are generally solved via tree search. Constraint propagation is exploited 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. A pictorial intuition of this fact is given in Figure 2.

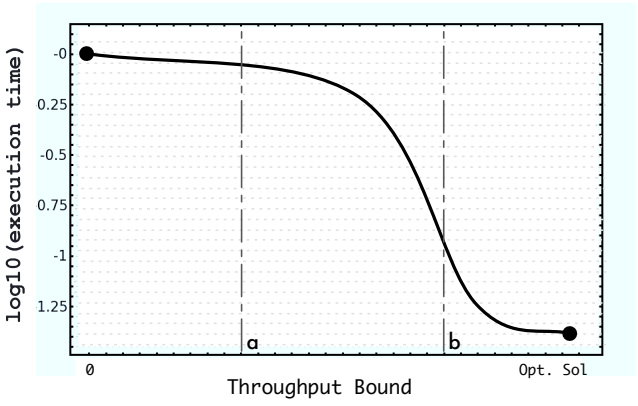

Fig. 2. Total search time over the bound value

The number of feasible solutions in the search space depends on the throughput bound tightness: the tighter the bound, the faster the search. A loose bound (dotted line a) leaves a lot of feasible solutions in the search space while with a tight bound (dotted line b) the number of feasible solutions is drastically reduced. The main purpose of the heuristic function is to quickly find a complete mapping that ensures a tight bound to drastically reduce the search space.

Experimental tests evidenced that branching over resource allocation variables have 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:

$$\frac{\alpha * Tmax}{\tau_i} + \frac{\beta * dep_i}{Dmax}$$

  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.
  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 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:

$$\frac{\delta * WL_p}{WLmax} + \frac{\theta * con_p}{Cmax}$$

where *WL* 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 execution times of the nodes that are *non-dependent* on *p*, and *Cmax* is the highest of these numbers. The coefficients δ and θ have been experimentally tuned to 0.79 and 0.21 respectively ($\delta = 1 - \theta$).

The experimental results show (see section VI) that the described heuristics obtains one order of magnitude speed up w.r.t. blind search.

B. Total Execution Time Bound

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 processors. This is done by setting a cyclic path that orders the actor execution over each processor. Before executing the throughput algorithm, the total actor execution time on each processor is computed (Ω_p). This value represents one over the maximal throughput (MTp_p) that the processor p can achieve:

$$\Omega_p = \frac{1}{MTp_p} = \sum_{i \in I_p} \tau_i$$

where I_p is the set of actors allocated on p and τ_i is the execution time of the actor i. The highest sum must be higher than the problem throughput threshold ($TpLB$) and higher than the current best solution found (Tp). Otherwise the search is stopped and the solver backtracks.

$$\max_{p \in Proc} \Omega_p > \max\left(\frac{1}{TpLB}, \frac{1}{Tp}\right)$$

C. Strictly Connected Component Filtering

Since the throughput value is cycle dependent, nodes not belonging to any cycle are usless. A filtering algorithm has been implemented to recursively remove the non-strictly connected component from the graph. First, we recursively remove all nodes without any in-going and out-going arcs; then the graph is further filtered from Algorithm 1, described in subsection (V-D), which removes the strictly connected components (scc) involving nodes *allocated on the same processor*.

A single iteration of the algorithm is sufficient to compute the throughput of a strictly connected component; otherwise, if the graph filtered presents more than one scc, the process is repeated starting from the first untouched node, until no such node exists.

Algorithm 1 Processor Pruning of given graph G

```
1: Let int X_i the ID of the processor of the node i.
2: Let bool P_j enable the processor j (default true).
3: for all p in Processors do
4:     int in, out = 0
5:     for all i in Tasks do
6:         if X_i == p && P_p then
7:             for j! = i, j ∈ Tasks && X_j! = p do
8:                 if exists arc (i, j) then
9:                     out++
10:                end if
11:                if exists arc (j, i) then
12:                    in++
13:                end if
14:            end for
15:        end if
16:    end for
17:    if out == 0 or in == 0 then
18:        P_p = false
19:        for y, z in Tasks do
20:            if X_y = p then
21:                remove X_y from the graph
22:                remove (y, z) and (z, y) from the graph
23:            end if
24:        end for
25:    end if
26: end for
```

D. Processor Pruning

During the search process the actors of the graph are bound to a processor. Actors not yet allocated could be considered bound to an ideal (different for each node) processor. Let C be the set of all cyrcles of the SDF graph G. Then we can define two different subsets: the set MpC which contains all *multi-processor* cycles, and SpC that have all the remaining cycles (*single-processor* ones).

$$C := MpC \cup SpC$$

We have experimentally noticed that in the large majority of the instances, the application throughput bound involves nodes allocated on different processors. Thus, the critical cycle is a *multi-processor* cycle. The filter described in this paragraph, implemented again in algorithm 1, excludes from the graph the nodes that cannot be part of any *multi-processor* cycle. The algorithm iterates over all processors (line 3), and counts the number of the *ingoing* (in variable) and *outgoing* (out variable) arcs of the nodes allocated on them (lines 5-12). A processor can be part of a cycle only if its actors have at least one input and one output channel ($in, out > 0$) that connect them to actors onto other processors. The aim of the last part of the algorithm (lines 17-22) is to remove nodes whose arcs cannot form *multi-processor* cycles.

Single-processor cycles are computed during the Total Execution Time Bound, described in section V-B. The throughput value of the graph (Thp) is given by the longest cycle found over all *single/multi-procossor cycles*.

$$Thp = \min\left(\min_{c \in SpC} \frac{1}{\sum_{i \in c} \tau_i}, \min_{c \in MpC} \frac{1}{\sum_{i \in c} \tau_i}\right)$$

E. Example

Algorithm 1 filters the graph by removing nodes that do not contribute to the throughput computation. For example the HSDFG reported in figure 3 has eight actors allocated

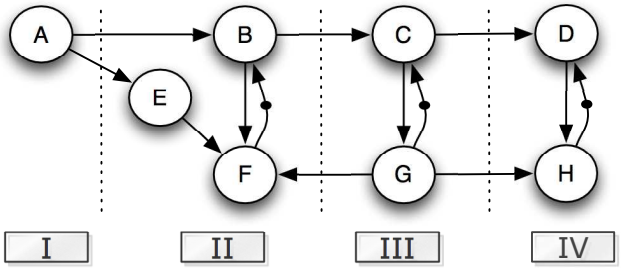

Fig. 3. An HSDFG allocation example

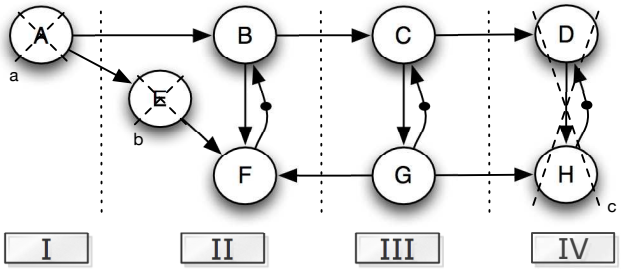

Fig. 4. An HSDFG allocated and optimized

onto four different processors (*I...IV*). First, the Strictly Connected Component Filter described in section V-C recursively removes the actor *A* and all arcs starting/ending in A, namely arc *(A,E)*. The actor *E* is removed at the second iteration. The graph remains with six nodes allocated onto tree different processors.

Then the application graph is filtered by the Processor Pruning filter. Figure 4 shows the pruned actors of the graph of figure 3. Since the actors allocated on the processor *IV* have only ingoing arcs they cannot be part of a multiprocessor cycle. Thus they are removed from the graph. The algorithm, in this example, computes the bound over the subgraph composed by actors *B,C,F,G* reducing the overall computation time of the throughput constraint.

VI. Experimental Results

We have evaluated our model and the scalability of our code on various sets of realistic instances. The graphs belong to three distinct classes of Synchronous Data-Flow Graph, built by means of the generator provided in the SDF3 framework [9]. Graph classes include Cyclic, Acyclic and Strongly Connected (SC) graphs. Clearly, if a graph contains cycles, it has an implicit throughput upper bound defined by the longest cycle in the graph. In contrast, acyclic graphs have no implicit bound and expose the highest parallelism: this makes them the most challenging instances.

We computed optimal mappings on a set of target multicores featuring 2, 4 and 8 processors, respectively. All tests were run on a Core2Duo machine with 2GB of RAM. Furthermore, 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. The second type of allocation, that we define $constrained$, is typically obtained by approaches working directly on the SDFG, without a preliminary transformation into HSDFG [5], [6].

First, we tested the run-time improvement given by the new filtering algorithm on groups of homogeneous graphs with 10 to 250 nodes. Figure 5 shows that the average run-time of the throughput computation algorithm presented in [7] (continuous line) grows rapidly with the size of the instance, while the algorithm proposed in this paper (dashed line) scales much better. Since the throughput computation algorithm is triggered each time the graph is modified, our faster throughput computation has a major beneficial impact on overall run-time.

The speedup is more pronounced on acyclic graph structures, which are the most computationally challenging . Figure 6 shows, in logarithmic scale, the improvement of the solver proposed in this paper (the continuous lines pair) over the one presented in [7] (dashed lines). For both solvers we compared the optimal $constrained$ solution for instances with 10 to 12 nodes over two different architectures with 2 (the lower line in each pair) and 8 cores. The optimized solver is on average one order of magnitude faster.

Figure 7 shows that with the non-optimized solver only few more than ten (10-12 nodes) acyclic instances were solved within the time bound (considering a 20 minutes time limit), while the heuristic and the throughput computation optimizations enable our solver to find rapidly the $constrained$ solution.

To get a deeper insight in the nature of the speedup, consider that the heuristic is very effective in pruning the search space. In medium size applications (10-15 actors), the explored search space could be reduced up to 99%. This reduction drastically increases the solver efficiency. For larger graphs, we have experimentally observed that search run-times tend to

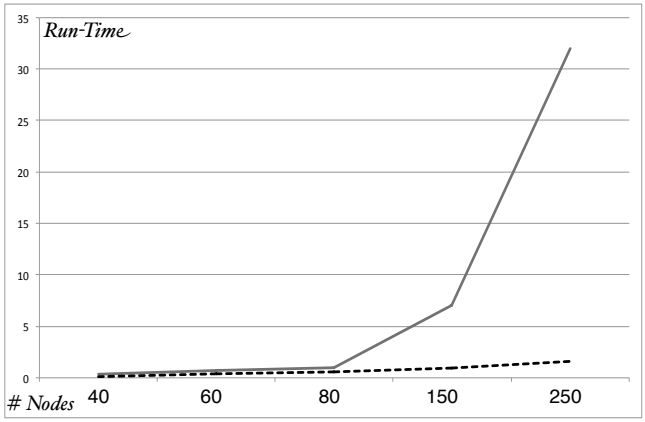

Fig. 5. Run-Times (in seconds) of throughput algorithms

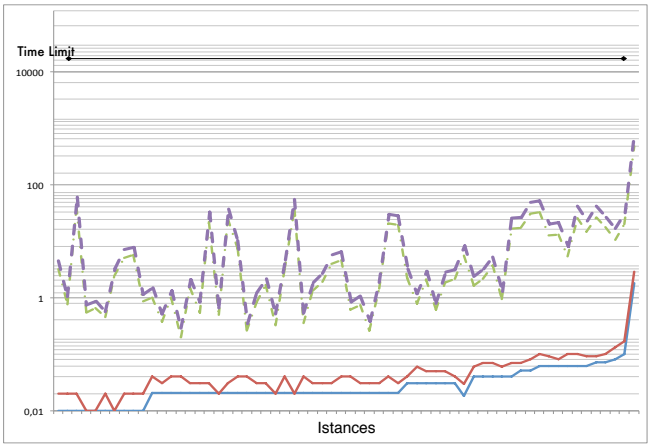

Fig. 6. Optimal Solution Run-Times (10 to 15 Nodes) on 2 and 8 processors

Istance Size	*Cost.* sol.	First sol.	First *Cost.* sol.
10 Nodes	82,66%	78,48%	75,86%
12-15 Nodes	77,44%	78,18%	66,72%

TABLE I
OPTIMALITY GAPS OF INCOMPLETE SEARCHES.

blow up with instances with more than 15 nodes. The solver find quickly a quasi-optimal allocation, but then the proof of optimality is slow. The improvement of this step is a topic for future algorithmic developments.

In Table I we provide comparisons among the throughput achievable by different mapping assumption with respect to the best throughput achieved by mapping all HSDFG nodes in an unconstrained fashion, and running complete search. We give the optimality gap of the the *constrained* solution, the first feasible solution and the first feasible *constrained* solution. The second column refers to the *constrained* solution. The optimality gap widens as the number of nodes increases: the optimal throughput value is about 20% higher on medium-size instances than that of 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.

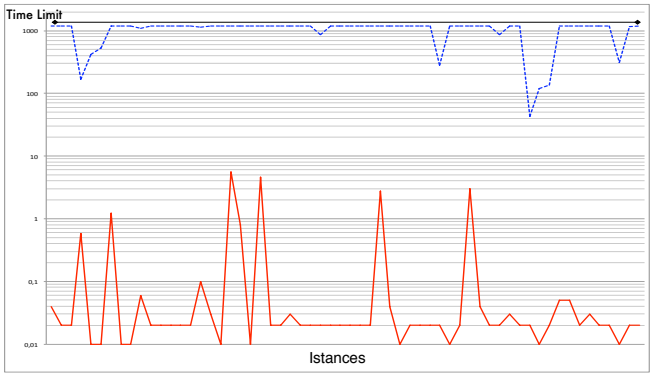

Fig. 7. Run-time comparison for medium-size instances (*constrained sol.*)

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 of the solution obtained by stopping the search with the throughput obtained with the first depth first traversal of the search tree driven by our heuristic function. 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, before entering the potentially exponential complete search phase. Thus, our strategy is quite effective even when used as a fast, incomplete search.

As expected, the *incomplete costrained* solution, reported in the third column has the largest optimality gap, with a 30% loss in throughput. This is the quality of results that could be expected by incomplete algorithms which map directly the SDFG. This result gives a clear indication that our algorithm provides a significant quality-of-results improvement with respect to previously presented incomplete algorithms.

VII. CONCLUSION

We developed and optimized a CP based method to solve the allocation and scheduling of SDF graphs on MPSoC. The solver features fast throughput computation and an effective heuristic function. This optimized algorithm is on average one order of magnitude faster than the best (to our knowledge) previously published complete solver based on CP.

An interesting topic for future research is the investigation of a transformation that almost maintains the degrees of freedom in mapping of the Homogeneous SDF without the blow up of the number of the nodes caused by the conversion of a SDFG into a HSDFG.

ACKNOWLEDGEMENT

The work described in this publication was supported by the PREDATOR Project funded by the European Community's 7th Framework Programme, Contract FP7-ICT-216008.

REFERENCES

[1] E. Flamand,Strategic Directions towards Multicore Application specific computing, Design Automation and Test in Europre 2009, pp. 1266
[2] E. Lee and D. Messerschmitt. Synchronous dataflow. Proceedings of the IEEE, 75(9),pp.1235–1245, September 1987.
[3] S. Sriram and E. Lee, "Determining the order of processor transactions in statically scheduled multiprocessors", Journal of VLSI Signal Processing, 15:207, 220, 1996.
[4] S. Sriram and S. Bhattacharyya. Embedded Multiprocessors Scheduling and Synchronization. Marcel Dekker, Inc, 2000.
[5] S. Stuijk T. Basten, A. Moonen, M. Bekooij, B. Theelen, M. Mousavi, A. Ghamarian , M. Geilen, "Throughput analysis of synchronous data flow graphs", Application of Concurrency to System Design, 2006.
[6] O.Moreira, J.D.Mol, M.Bekooij, J.van Meerbergen, "Multiprocessor Resource Allocation for Hard-Real Time Streaming with a Dynamic Job-Mix", IEEE Real Time on Embedded Technology and Applications Symposium, pp.332-341, 2005.
[7] A. Bonfietti, M. Lombardi, M. Milano L. Benini, "A Throughput constraint for Synchronous Data Flow Graphs", CPAIOR, pp.26-40, 2009.
[8] A. Dasdan, R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis", IEEE Trans. on CAD of Integrated Circuits and Systems, 1998, 17:10, 889-899
[9] S. Stuijk, M. Geilen, T. Basten, "Sdf^3: sdf for free", Acsd 2006, 276-278