Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2016, Article ID 8536432, 15 pages
http://dx.doi.org/10.1155/2016/8536432

Research Article

Performance Estimation Based Multicriteria Partitioning Approach for Dynamic Dataflow Programs

Małgorzata Michalska,[1] Nicolas Zufferey,[2] and Marco Mattavelli[1]

[1]*EPFL SCI-STI-MM, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland*
[2]*Geneva School of Economics and Management (GSEM), University of Geneva, 1211 Geneva 4, Switzerland*

Correspondence should be addressed to Małgorzata Michalska; malgorzata.michalska@epfl.ch

Received 29 January 2016; Revised 17 June 2016; Accepted 29 June 2016

Academic Editor: Maxime Pelcat

The problem of partitioning a dataflow program onto a target architecture is a difficult challenge for any application design. In general, since the problem is NP-complete, it consists of looking for high quality solutions in terms of maximizing the achievable data throughput. The difficulty is given by the exploration of the design space which results in being extremely large for parallel platforms. The paper describes a heuristic partitioning methodology applicable to dynamic dataflow programs. The methodology is based on two elements: an execution model of the dynamic dataflow program which is used as estimation of the performance for the exploration of the large design space and several partitioning algorithms competing to lead to specific high quality solutions. Experimental results are validated with executions on a virtual platform.

1. Introduction

An interesting alternative to the classical sequential programming methods for signal processing system implementations is the approach based on dataflow programming. Dataflow programs are characterized by a high analyzability and platform independence and by providing an explicit exposition of the potential parallelism. For these reasons, they can be used for exploring a variety of parallel implementation options and to provide an extensive and systematic implementation analysis [1, 2]. Hence, they have been investigated in several research works [3–5].

Dataflow programs are in general structured as, possibly, hierarchical networks of communicating computational kernels, called actors. Actors are connected by directed, lossless, order preserving point-to-point communication channels, and data exchanges are only permitted by sending data packets (called tokens) over those channels. A general dataflow *Model of Computation* (MoC) is known in the literature as “Dataflow Process Network (DPN) with firings” [6]. A DPN network evolves as a sequence of discrete steps (called firings) corresponding to the executions of actions that may consume and/or produce a finite number of tokens and modify the internal actor state. At each step, according to the current actor internal state, only one action can be fired. The processing part of the actors is thus encapsulated in the atomic executions completely abstracting from time. Figure 1 illustrates the construction of a sample dataflow program and the underlying structure of an actor.

An important property of a dataflow program is the composability of its components (actors). Porting a program onto a target architecture requires determining three settings: the partitioning (the assignment of dataflow actors to the processing units), the scheduling (the execution order inside each unit, where, depending on the internal nature of actors, a static scheduling may or may not exist), and the dimensioning of buffers (a finite size for each communication channel in the network). An essential aspect of the problem is that although the configurations are applied at the level of actors, the program execution (i.e., in terms of data dependencies) is described at the level of action firings.

Depending on the architecture and on the dataflow program itself, the size of the solution space of admissible partitioning and scheduling configurations can be, in general, extremely large. Therefore, an important design challenge is to find a set of configurations that optimizes the desired

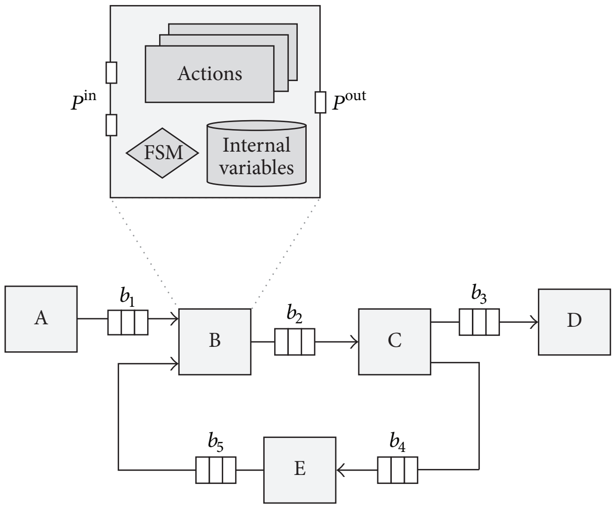

Figure 1: Construction of a sample dataflow program.

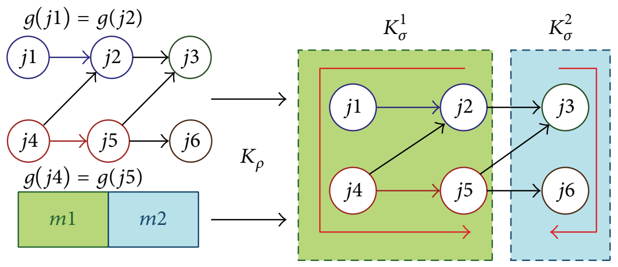

Figure 2: Partitioning and scheduling illustration.

objective function. This problem has been proven to be NP-complete even for the case of only two processors [7]. Among different objective functions that might be considered, this work focuses on the maximization of the data throughput of a dataflow program. This particular objective function is often an appropriate choice for stream programs, since in many cases it contributes also to an optimization of other criteria, such as, for instance, resources utilization and energy consumption [8].

An efficient exploration of the multidimensional design space has two important applications. First, exploration of feasible regions leads to determining a close-to-optimal set of configurations according to the desired objective function. Second, it enables the identification of unreachable regions of the design space that could become reachable by applying refactorization stages to the considered design. For instance, a different implementation of an algorithm might be required for obtaining higher performances if its current exposed parallelism is lower than the potential parallelism offered by the processing platform.

Among different dimensions of the design space, more attention is usually paid to the partitioning, since its impact on the overall performance is considered to be dominant. Furthermore, the room for improvement left, for instance, to scheduling strongly depends on the quality of the applied partitioning (i.e., low-quality partitioning configurations cannot be improved much by applying efficient scheduling). This property has been confirmed also for other, nondataflow domains [9, 10]. The purpose of this work is to describe a new partitioning methodology to perform the design space exploration of dataflow program implementations.

The original contribution of this paper includes different stages of the partitioning process: starting from presenting and discussing different aspects of the problem, possible impact of a considered architecture, through the modeling approach, up to the description of several partitioning solution methods with different quality and complexity. An important property of the methodology is the construction of partitioning algorithms based on highly accurate models of the program executions. The execution model is used in order to evaluate each solution and to extract some execution properties which are further considered as optimization criteria. The approach can be implemented with limited memory requirements (i.e., it can be implemented on a standard PC also for complex designs) and provides solutions according to quasi-constant evaluation times for different configurations considered. Unlike most of the existing approaches, the methodology can be used for analyzing static as well as fully dynamic dataflow programs.

The paper has the following structure. First, Section 2 presents and formulates the partitioning and scheduling problem explicitly for the case of dynamic dataflow programs. In this context, the related work is discussed in Section 3. Then, Section 4 describes the execution model used to derive the estimations and Section 5 reports in detail the proposed partitioning algorithms built upon this model. The algorithms are analyzed for complexity, quality of solution, and usability, based on the experimental results presented in Section 6. Finally, conclusions and future works are discussed in Section 7.

2. Dataflow Partitioning and Scheduling

The design problem considered here can be applied to any dataflow MoC.

2.1. Problem Presentation. Following the terminology commonly used in the production field [11], an objective is to find an assignment of n jobs (representing *action firings*) to a set of m parallel machines (corresponding to *processors*). The objective function to minimize is the overall makespan (i.e., the completion time of the last performed job among all processors). Assuming that the processors do not allow parallel execution, only one job can be performed at a time on each machine. Therefore, when all jobs are assigned to the machines, it must be decided in which order the jobs are going to be performed on each machine. The literature terminology often calls the first problem as mapping in the spatial domain (binding) and the second as mapping in the temporal domain (scheduling) [12]. In this work, partitioning (K_ρ) and scheduling (K_σ) refer to these problems, respectively, and are illustrated in Figure 2.

Each job j has an associated processing time (weight) p_j and a group (actor) g_j. There are k possible groups, and each one can be divided into subgroups where all jobs have the same processing time. This division can be easily identified with actions and their executions (firings). Between some pairs $\{j, j'\}$ of incompatible jobs (i.e., with $g_j \neq g_{j'}$) is associated a communication time $w_{jj'}$. The communication time is subject to a fixed quantity $q_{jj'}$ of information (or the number of tokens) that needs to be transferred. The size of this data is fixed for any subgroup (i.e., an action always produces/consumes the same amount of data). Due to the structure of dataflow programs, the partitioning and scheduling must take into account the following constraints:

(i) Group constraint: all jobs belonging to the same group have to be processed on the same machine (i.e., an actor must be entirely assigned to one processing unit). A fixed relative order is decided within each group. It can be assumed that this order is established based on the program's input data.

(ii) Precedence constraint: (j, j') means that a job j must be completed before job j' is allowed to start.

(iii) Setup constraint: it requires that, for each existing connection (j, j') involving jobs from different groups, a setup (or communication) time $w_{jj'}$ occurs.

(iv) Communication channel capacity constraint: the size of a communication channel (buffer) through which the information (tokens) is being transmitted is bounded by B. That is, the sum of $q_{jj'}$'s assigned to this buffer (q_b) cannot exceed B. If it occurs, it might affect the executability of j and j' and introduce serious delays in the overall makespan.

2.2. Architecture Impact. Executing the dataflow program on a given architecture requires introducing the notion of time to its originally time-abstracting definition. It is achieved by assigning a concrete value to the weight p_j of each job and to the communication time $w_{jj'}$. The types and ranges of values for p_j's and $w_{jj'}$'s fully depend on the properties of the targeted architecture. For the case of homogeneous platforms, p_j is constant no matter how a group (actor) is actually partitioned. It is not the case for heterogeneous platforms, where this value can vary according to the processor family that the processing unit belongs to (i.e., software or hardware).

Assuming that job j is assigned to the machine ρ, the actual value of communication time $w_{jj'}$ is a product of two elements: the number of tokens $q_{jj'}$ and the variable time $c_{jj'}(\rho_j, \rho_{j'})$ needed to transfer a single unit of information from ρ_j to $\rho_{j'}$. For two given jobs j and j', the largest $c_{jj'}$ can be significantly larger than the smallest $c_{jj'}$. In theory, every connection (j, j') can have as many different $c_{jj'}$'s as the number of different possible assignments to the machines. But in practice, this number can be usually reduced to few different values of latency, depending on the memory level serving a given communication (see Chapter 4 of [13]).

The case of heterogeneous platforms (typically hardware or software families of processors) may introduce some further constraints to the problem, such as the following:

(i) Eligibility constraint: not all jobs can be processed on each machine (i.e., each machine has a collection $d(\rho_i)$ of not supported operations).

(ii) Capacity constraint: the memory size is limited for a given family fl of processors. It involves that the memory requirements of all jobs assigned to processors of this family cannot exceed the given size.

Different figures of merit can be also introduced for communication time, especially for communication occurring between jobs partitioned to machines belonging to different families.

2.3. Problem Formulation. Following different aspects of the problem discussed in Sections 2.1 and 2.2, for each job it is required to find a partitioning configuration K_ρ and for each partition a scheduling configuration K_σ, so that the total execution time is minimized. The constraints include group, setup/communication, buffer capacity, eligibility, and capacity. These aspects can be summarized in the problem formulation provided below.

Decision Variables are as follows: $\forall_j K_\rho(j) = \rho_i$, $\forall_\rho K_\sigma = \{j, j', \ldots\}$.

Objective Function is $\min(T_{\text{end}}(j_{\text{last}}))$.

Constraints are as follows:

(i) $g_j = g_{j'} \Rightarrow K_\rho(j) = K_\rho(j')$,

(ii) $j \prec j' \Rightarrow T_{\text{start}}(j') \geq T_{\text{end}}(j)$,

(iii) $j_\sigma = j \Rightarrow \Sigma_{q_b} + \text{tokens}(j) \leq B$,

(iv) $K_\rho(j) = \rho_i \Rightarrow j \notin d(\rho_i)$,

(v) $\Sigma_{\rho \in \text{fl}} \Sigma_{g \in \rho} \Sigma_{j \in g} \text{mem}(j) \leq \text{size}(f)$.

3. Related Work

Due to the NP-completeness of the partitioning problem for realistic instances, it is only possible to develop methods providing close-to-optimal solutions [14]. They can be obtained by applying constructive heuristics, where a solution is generated from scratch by sequentially adding components to the current partial solution according to some criteria until the solution is complete [15]. Another possibility is metaheuristics, formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space [16]. Metaheuristics (e.g., simulated annealing, tabu search, variable neighborhood search, and guided local search) can usually lead to solutions of higher quality, but in general they require much longer computing times [17, 18].

There are several examples of the approaches based on metaheuristics used for partitioning or, more generally, for the design space exploration of dataflow programs. In [19], simulated annealing is employed for estimating the bounds of the partitioning program. Various optimization stages (including the selection of a target architecture, partitioning, scheduling, and designing space exploration) are applied in [20] in order to identify feasible solutions. The optimizations are performed using an evolutionary algorithm. Multiobjective evolutionary algorithms used for performing an automatic design space exploration are also an objective of work discussed in [21]. An interesting transition from simple heuristics to advanced metaheuristics (such as genetic algorithms) is also described in [22], where more advanced methods act as a refinement to the less advanced ones.

An important aspect of metaheuristic approaches for design space exploration, which requires explicitly providing an execution model, is performance estimation that can be used as an evaluation of a solution. The level of details considered in the model usually determines the accuracy of estimation as well as the evaluation time. The more detailed is the model, the more accurate estimation can be obtained, but employing a much longer time [23–25]. There are multiple frameworks which use the concept of performance estimation in order to build the optimization methods on them.

An interesting example, demonstrating some similarities with the methodology described in this paper, is the MPSoCs *Application Programming Studio* (MAPS) [26]. This design space exploration framework performs an estimation of a program execution, based on the analysis of the execution trace. Chapter 6 of [27] introduces a *trace replay module* employing a discrete-event simulator in order to simulate the scheduling of the segments of the trace. Several optimization heuristics are built on the estimation, with an emphasis on the advantages of light-weight heuristics over evolutionary methods, and the composability analysis for the purpose of executing simultaneously multiple applications on a given platform without interference [4]. In terms of modeling differences, the framework addresses only KPN dataflow programs and supports a limited set of dependencies. Another framework, supporting only the KPN programs, is *Sesame system-level simulation framework* [28]. Similar to the work presented in this paper, it introduces a concept of an application model which is independent from the architecture model and the partitioning configuration. The objective of the framework is to find the most suitable and efficient target MPSoC platform for a given program. Hence, the exploration process involves simulating and analyzing the candidate architectures.

Unlike the two approaches mentioned earlier, *Metropolis* is a framework that abandons the imposition of a specific language or flow model to the design [29]. It introduces a design description at different levels of abstraction and provides an infrastructure based on meta-modeling that remains generic enough to support existing MoCs, whereas it can also accommodate new MoCs. The meta-model allows capturing the architecture, the functionality, and the mapping between different abstraction levels. The simulation and formal analysis allow the user to determine how well an implementation satisfies the specified requirements. With its generic features, the framework responds only partially to the demands of DPN, because it does not support a simultaneous analysis at the level of partitioning, possibly dynamic scheduling and buffer dimensioning.

In the mentioned examples, the partitioning methodologies have been designed explicitly for the purpose of dataflow partitioning, which is a specific instance of a graph partitioning problem. The research field of graph partitioning is thoroughly covered by different algorithms proposed in the literature [30] as well as some software packages, such as METIS [31] or SCOTCH [32]. Such general purpose partitioning algorithms cannot be, however, easily applied for the case of dataflow programs, since they are not aware of the semantics related to the nodes and edges of a dataflow graph. An attempt of applying the METIS, mentioned earlier, for the purpose of run-time actor mapping of dataflow programs has been made in [33]. This approach explores the results of profiling and extracts some optimization criteria (connectivity between actors, critical path). It is, however, difficult to evaluate the obtained solutions in terms of being close-to-optimal, or point to possible optimizations in the design, since no execution model is provided. The considered partitioning graph is the program network itself and not the execution trace which can provide elements and measures of the execution properties of the dataflow program. Furthermore, such combinatorial approach, which might operate quite effectively for small instances of the problem, cannot be successfully applied to the exploration of design problems of larger size.

Summarizing, a comparison with existing approaches puts in evidence that there are some important differences that should be emphasized. First of all, the approach can be applied to all dataflow models of execution including fully dynamic dataflow programs [34]. Hence, it includes also other, less expressive dataflow variants such as, for instance, SDF and CSDF [35]. Furthermore, due to the demands of the dynamic programs, such as dynamic scheduling influenced by the applied buffer configuration, the methodology can be used directly for optimization of other design configurations, without imposing any particular order of optimizations. Finally, the long-term objective of the methodology is to find a close-to-optimal (close-to-the-potential-parallelism) design configuration on a given platform, in order to assess the maximum performance of a dataflow program, identify its bottlenecks, and hence identify new regions of the design space to be reached after applying some refactorization to the design.

4. Dataflow Program Execution Modeling

A general dataflow MoC, DPN with firings, is considered. It allows implementation of fully dynamic programs efficiently capturing all classes of signal processing applications (e.g., video and audio codecs, packet switching in communication networks). For this reason, a representation of a dataflow program must capture its entire dynamic behavior. Such a representation can be built by generating a directed, acyclic graph G, called *Execution Trace Graph* (ETG), where nodes of the graph represent a collection of action executions, called *firings*. Such representation has been already extensively studied to solve some optimization problems of dynamic dataflow implementations [36].

It is possible to explicitly characterize various intrinsic dependencies between two firings, which are used to describe the program execution. They can be grouped into two types. The first type includes *internal dependencies* related to FSM, internal variables, and ports ($p^{\text{in}}/p^{\text{out}}$ in Figure 1) and describes the relations between two *firings* of the same actor. The second type (*token dependencies*) describes the relations between two *firings* of different actors that, respectively, produce and consume at least one token. Defining dependencies between executed firings establishes precedence orders: if firing f_2 depends on firing f_1, then f_1 has to be executed and completed (including the communication, if applicable) before f_2 can be started.

An ETG has to consider a sufficiently large and statistically meaningful set of input stimuli in order to cover the whole span of dynamic behavior of the application considered. The size of G varies according to the type of application and to the size of the input stimuli set. In fact, for some applications, it can be large. For instance, a very complex algorithm, such as HEVC video compression operating on an input signal of a few tenths of frames, may result in G containing a few billions of nodes. Still, it can be processed in reasonable time using standard PC platforms, as demonstrated by the examples described in Chapter 9 of [36].

Considering the program execution on a given architecture, the ETG is weighted at the nodes and at the edges, and the weights correspond to the values of p_j and $w_{jj'}$, respectively. These values are normally obtained by profiling the program on the target architecture. A weighted ETG allows simulating the program execution for a given partitioning, scheduling, and buffer dimensioning configuration. This task is accomplished by an event-driven performance estimation tool that processes the firings following the constraints and properties, as presented in Section 2.

5. Partitioning Algorithms

This section presents the algorithmic details of different partitioning solution methods, namely, greedy constructive procedures, the decent local search heuristics, and the tabu search metaheuristic. Each method takes the weighted ETG as input data.

5.1. Greedy Constructive Procedures. In order to construct a solution, the greedy procedures require specifying only the target number of processors. The solution generation succeeds in a negligible time frame, since no performance estimation needs to be performed.

5.1.1. Workload Balance (WB). The concept of balancing the workload in order to minimize the bottleneck of the program and hence maximize the throughput has been already successfully employed for partitioning purposes of systems of different types [37]. Inspired by such approaches, the very first constructive heuristic has been designed. The algorithm starts from calculating the total workload of each group throughout the program execution. It is expressed as the sum of the p_j's for all jobs (firings) belonging to one group (actor) g. The actors are then sorted decreasingly by the sum of weights (workload) $\mathrm{wl}(g) = \sum_{j\in g} p_j$. The partitioning decision is based on the sum of workloads of actors partitioned already in one processor ρ: $\mathrm{wl}(\rho) = \sum_{g\in\rho} \mathrm{wl}(g)$. The next actor on the list is always partitioned on the processor with the smallest sum of workloads $\mathrm{wl}(\rho)$. In this way, a balance of overall workload of each partition should be achieved and the bottleneck (understood as the workload of the most occupied processor) is likely to be minimized.

5.1.2. Balanced Pipeline (BP). The algorithm starts from giving each actor a dedicated processor. Next, the processors are being iteratively reduced and the members of the least occupied processors are attached to the remaining processors. The optimization criteria of the algorithm involve equalizing the average preceding workload (understood as the maximal sum of weights of each firing of each actor that precedes the given actor in the network in terms of topological order) and maximizing the number of common predecessors (understood as the number of actors appearing on the topological list of predecessors for a given pair of actors) within each partition. The simulation of the execution time might be incorporated to the algorithm in order to estimate the optimal number of processors from the perspective of processors utilization. The *Balanced Pipeline* algorithm, extended with some additional optimization procedures for communication volume and idle time, has been shown to outperform existing partitioning algorithms for dataflow programs. For a more detailed description of the algorithm, its metrics, and its results, the reader is referred to [38].

5.2. Decent Local Search Heuristics (DLS). As described in [39], a local search starts from an initial solution and then explores the solution space by moving from the current solution to a neighbor solution. A *neighbor* solution is usually obtained by making a slight modification of the current solution, called a *move*. The *neighborhood* $N(s)$ of a solution s is the set of solutions obtained from s by performing each possible move. In a *descent local search* (DLS), the best solution (according to the considered objective function f) of $s' \in N(s)$ is generated at each iteration. The main drawback of this method is that it stops in the first local optimum. Two DLS approaches are proposed below: the *Idle DLS* and the *Communication Frequency DLS*.

5.2.1. Idle DLS (IDLS). Representing the program execution with the ETG, and simulating it for a given partitioning, scheduling, and buffer dimensioning configuration using the performance estimation tool, may provide important information related to actor states throughout the execution. The following states may occur for an actor that is currently not processing and has not yet terminated.

(i) *Blocked reading* considers the situation where an actor has not yet received the required input tokens and therefore cannot be executed.

(ii) *Blocked writing* takes into account the situation where the buffer an actor is expecting to write to is full, so it has to wait for the available space.

(iii) *Idle* corresponds to the situation where although an actor has the necessary tokens and required space in the buffers, it cannot be fired because another actor is currently processing in the same processor (as previously mentioned, only one job can be executed on each processor at a time).

Assuming a design space exploration in the dimension of partitioning and scheduling, it is particularly important to minimize the occurrences of the idle state. In order to achieve that, in IDLS, all actors are sorted according to their idle times in decreasing order (*idle time list*). A newly created solution s is generated by moving a single actor to the most idle partition, where the *idleness* of a partition is defined as the overall time during the execution when none of its actors could be executed due to being blocked reading/writing or terminating already. In each iteration, the possible moves are prioritized according to the position of the considered actor on the *idle time list*. A move is evaluated by estimating the makespan of the new solution. For the case of a successful move, the statistics on the idle times of the actors and the corresponding *idle time list* are being regenerated. Since the moves are prioritized, there is a risk that if there is a move with a high priority that does not improve the solution, it will be unnecessarily repeated in each iteration. To prevent that from happening, a simple *release* mechanism is implemented: a (once unsuccessful) move, understood as an actor-partition pair, may be repeated only if the content of the target partition has been modified by applying another move.

5.2.2. Communication Frequency DLS (CFDLS). Another information that can be extracted from the ETG is the number of token dependencies between the firings of different actors. Accumulating these numbers for all firings leads to creating an actor-actor communication frequency map. This map is independent from the partitioning configuration, but taking the partitioning into consideration, it can be easily transformed into an actor-partition map. Indeed, this map is taken as an optimization criterion by another local search. For each actor, the algorithm calculates the internal communication frequency (token exchange with actors partitioned to the same processor) and external communication frequency (token exchange with actors partitioned to different processors). As described previously in Section 2.2, the partitioning of actors may strongly influence the values of communication cost and therefore the makespan.

If for any actor the external communication frequency with one processor exceeds the internal communication frequency, this actor-partition pair is considered as a move. The moves are prioritized according to the overall communication frequency of the actors and a *release* mechanism (similarly to IDLS) is implemented. The move can be evaluated in two ways: by estimating the execution time of a new solution or by analyzing if the overall external communication frequency (calculated collectively for all partitions) has decreased. In this work, for the purpose of consistency with other solution methods, only the first method is used.

5.3. Tabu Search (TS). Tabu search, as introduced by Glover [40], is still among the most cited and used local search metaheuristics for combinatorial optimization problems. It avoids the problem of getting stuck in the first local optimum by making use of recent memory with a *tabu list*. More precisely, it forbids performing the reverse of the moves done during the last *tab* (parameter) iterations, where *tab* is called *tabu tenure*. At each iteration of TS, the neighbor solution s' is obtained from the current solution s by performing on the latter the best nontabu move (ties are broken randomly). The process stops, for instance, when a time limit T (parameter) is reached. In most TS implementations, if the neighborhood size is too big, only a proportion is explored in each iteration. This proportion can be, for instance, a random sample involving $e\%$ (parameter) of the neighbor solutions.

TS has proven to have a good balance between intensification (i.e., the capability to focus on specific regions of the solution space) and diversification (i.e., the ability to visit diverse regions of the solution space). In addition, it has a good overall behavior according to the following measures [18]: (1) quality of the obtained results (according to a given objective function f that has to be optimized); (2) quickness (time needed to get competitive results); (3) robustness (sensitivity to variations in data characteristics); (4) simplicity (facility of adaptation); and (5) flexibility (possibility to integrate properties of the considered problem). To adapt TS to the studied problem, the following elements have to be designed: the representation of any solution s, the neighborhood structure (i.e., what is a move), the tabu list structure (i.e., what type of information is forbidden), and a stopping criterion (i.e., what is the most appropriate time limit).

5.3.1. Solution Encoding and Neighborhood Structure. A solution for partitioning is represented as a map of actors and processors, where the number of processors is fixed. Each actor can be mapped to only one processor at the time, and each processor must be mapped to at least one actor. Hence, leaving empty processors is not allowed. The following basic types of moves are possible: (1) REINSERT: moving an actor to another processor; (2) SWAP: two actors belonging to two different processors. For the purpose of swapping, the term *complementary move* is introduced. Assume that a move $m(j,\rho_i,\rho_{i'})$ consists of relocating an actor j from source partition ρ_i to target partition $\rho_{i'}$. A move $m(j',\rho_{i'},\rho_i)$ is complementary to $m(j,\rho_i,\rho_{i'})$ if it involves moving any actor j' from source partition $\rho_{i'}$ to target partition ρ_i. In this work, the neighborhood structures are generated by performing REINSERT and SWAP moves according to four different criteria, presented below:

(1) $N^{(B)}$ (for balancing):

- (i) REINSERT: choose randomly an actor from the most occupied processor and move it to the least occupied processor.
- (ii) SWAP: choose randomly two actors in different partitions so that swapping the actors decreases the relative workload imbalance between the two partitions.

(2) $N^{(I)}$ (for idle):

- (i) REINSERT: for each actor which has a bigger idle time is than its processing time, find the most idle processor, different from the one currently mapped, where the definition of *idle* is as described in Section 5.2.1.

(ii) SWAP: generate a set of moves on the REINSERT basis, but allow actors to be moved to *any* partition except for the least idle one (search for complementary pairs of moves).

(3) $N^{(\mathrm{CF})}$ (for communication frequency):

(i) REINSERT: check the internal and external communication frequency of each actor and consider the moves, as described in Section 5.2.2.

(ii) SWAP: generate a set of moves on the REINSERT basis (search for complementary pairs of moves).

(4) $N^{(R)}$ (for random):

(i) REINSERT: choose randomly an actor and move it to a different processor (randomly chosen).

(ii) SWAP: generate a set of moves on the REINSERT basis (search for complementary pairs of moves).

5.3.2. Parameters. Any time an actor j is moved from a processor ρ to another processor, it is forbidden to put j back to ρ for *tab* iterations, where *tab* is an integer uniformly generated in interval $[a, b]$, and the values of parameters a and b were tuned to 5 and 15, based on the preliminary experiments. Smaller values do not allow escaping from local optima, whereas larger values do not allow intensifying the search around promising solutions. There are two other sensitive parameters that have to be tuned for TS, namely, e (the proportion of neighbor solutions explored during each iteration) and T (the time limit). Reaching the time limit T results in immediate termination of the search and returning of the best solution ever found. Usually, T is set so that the improvement potential is poor (i.e., the percentage of improvement is below a threshold during a predefined time interval) if the method is run for larger time limits. Next, the smaller is e, the more iterations are performed but the fewer neighbors are investigated in each iteration. A large value of e contributes to the intensification ability of the method (indeed, all the solutions around the current one are explored), whereas a small value plays a diversification role (indeed, no focus is put on the neighborhood of each solution). Finally, a small (resp., large) value of *tab* strengthens the intensification (resp., diversification) ability of the search.

5.3.3. Advanced Variants of Tabu Search. Since each of the used neighborhood structures relies on different properties, a more advanced version of the TS involves a consolidation of all neighborhood structures. It is applied in two different variants:

(i) *Joint Tabu Search* (JTS): at each iteration, the neighborhood structure includes moves obtained according to all types. Therefore, the used neighborhood structure is $N^{(J)} = N^{(B)} \cup N^{(I)} \cup N^{(\mathrm{CF})} \cup N^{(R)}$. This variant should have more flexibility, because it involves various types of moves. The proportion of the set sizes for different types of moves can be freely tuned.

(ii) *Probabilistic Tabu Search* (PTS): at each iteration, the search assigns a probability to the selection of each neighborhood of the set $\{N^{(B)}, N^{(I)}, N^{(\mathrm{CF})}, N^{(R)}\}$. This probability is tuned based on the history of the search during the considered run. As a result, the search is guided by the success rate of each type of move (where a success corresponds to an improvement of the current solution).

6. Experimental Results

6.1. Experimental Setup. The focus of this work is to explore the design space along the partitioning dimension. Hence, the other dimensions need to be fixed. The dynamic scheduling policy used in the experiments is *nonpreemptive*, which involves that the same actor is continuously scheduled, as long as its firing conditions (i.e., available input tokens and output space) are fulfilled. The nonsatisfaction of any of the conditions results in choosing the next actor (on the round-robin basis) among all actors with satisfied input/output conditions. Regarding the buffers, an infinite size would be, ideally, considered. Since it is not possible for the practical implementations, the buffer size must be sufficiently large, so that deadlocks are avoided and the overall *blocked writing* time of the actors, as defined in Section 5.2.1, is kept at a minimum level. In this way the influence of a buffer size which is too small remains negligible for the overall performance. For the dataflow programs considered here it has been sufficient to set all buffers to the size of 512 tokens.

6.1.1. Experimental Designs. The quality of a partitioning algorithm is, in principle, assessed by the quality of a solution it provides. For each solution it can be evaluated how close it is to the objective function and how it approaches the potential parallelism of a given dataflow program implementation, for instance, how far is a given throughput from the best throughput achievable by the dataflow program under consideration. Hence, it is essential to perform the experiments with an application that in principle can provide a sufficient level of parallelism. If this condition is not satisfied, it is likely that either none or *any* partitioning algorithm can provide any satisfactory performance. For this reason, finding appropriate dataflow programs for validating partitioning algorithms is not a trivial task. The application used in the experiments is an MPEG-4 SP decoder design that consists of 41 actors in total and provides the upper bound on the potential parallelism around 6.28. This upper bound on the potential parallelism is evaluated as a proportion of the critical executions in the overall execution time, assuming a full parallel execution and the unbounded buffer size configuration (as described in Chapter 5 of [36]).

MPEG-4 SP decoder network is an implementation of a full MPEG-4 4 : 2 : 0 Simple Profile decoder standard written in CAL Actor Language [41]. The main functional blocks include a parser, a reconstruction block, a 2D inverse discrete

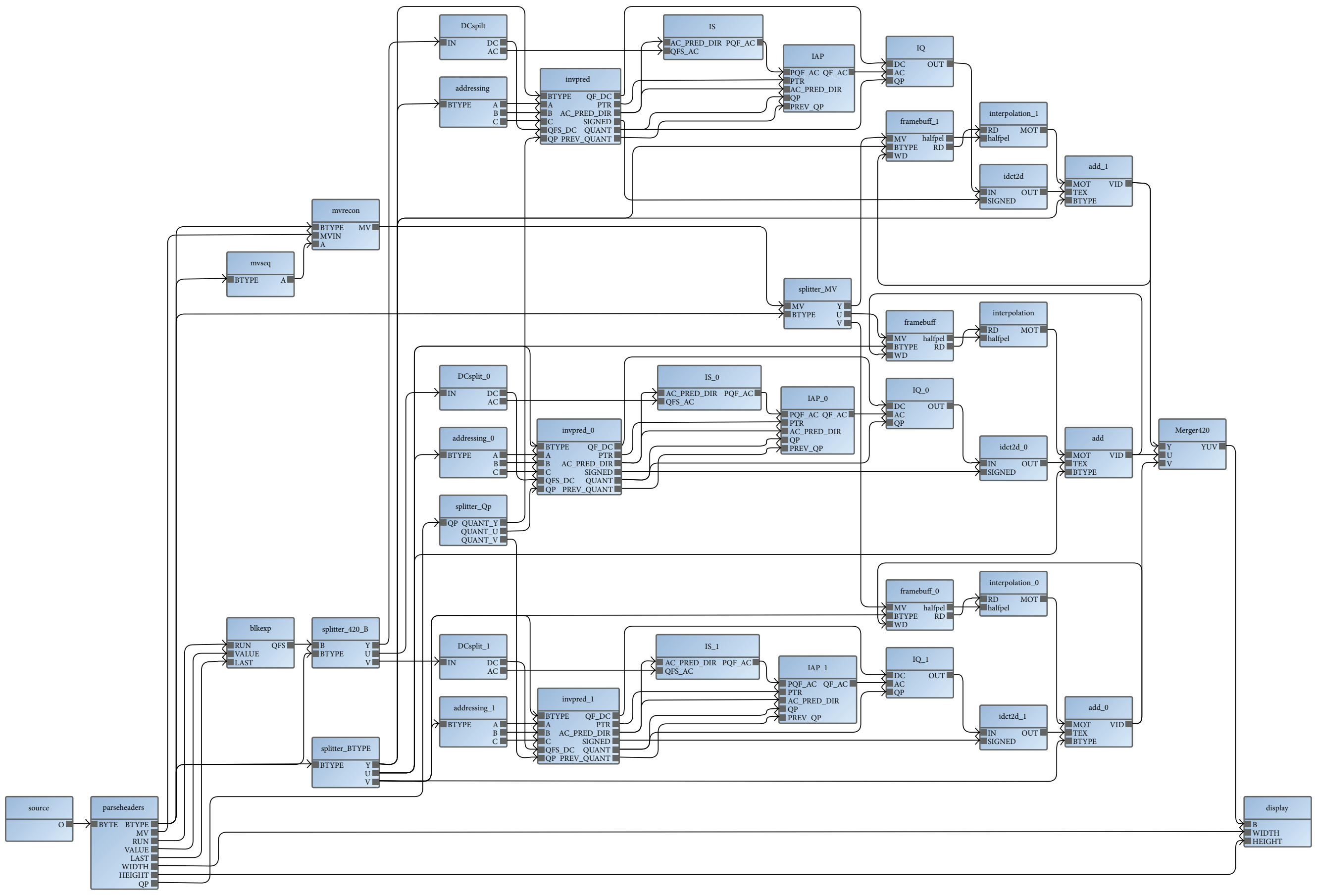

Figure 3: MPEG-4 SP decoder network.

cosine transform (IDCT) block, and a motion compensator. These functional units are hierarchical compositions of actors in themselves. The decoding starts from the parser (the most complicated actor in the network consisting of 71 actions) which extracts data from the incoming bitstream, through reconstruction blocks exploiting the correlation of pixels up to the motion compensator performing a selective adding of blocks. In order to give an overview of the complexity of the design, a schematic representation of the network is presented in Figure 3.

6.1.2. Target Platform. The platform used for the experiments is an array of *Transport Triggered Architecture* (TTA) processors [42]. There are several features of this platform that allow an easy modeling and a precise simulation for the purpose of partitioning and scheduling design space exploration. The most significant one is a cacheless, negligible interprocessor communication, and therefore an independence of the profiling results from the mapping configuration. In fact, as confirmed by our previous experiments on the platform, the results of a single profiling remain valid for multiple partitioning and scheduling configurations [43]. Hence, such a platform is a good choice for an initial validation of the effectiveness of the methodology.

6.1.3. Tuning of Parameters. As described in Section 5.3, apart from the length of the *tabu tenure*, TS is sensitive to two parameters that need to be properly tuned: the time limit T and the percentage e of explored neighbor solutions. For that purpose, 3 runs on a set of initial partitioning configurations have been performed for each: $N^{(B)}$, $N^{(I)}$, $N^{(\mathrm{CF})}$, and $N^{(R)}$. First, with a fixed value of $e = 0.5$, each TS variant was performed 3 times on the initial set of partitioning configurations. For each run, TS was stopped any time 5 minutes have elapsed without improving the best encountered solution (during the current run) by at least 1%. Parameter T has been set as the largest encountered stopping time (minus 5 minutes) among all these experiments.

Next, with the selected value of T, all TS variants were tested with different values of $e \in 0.2, 0.4, 0.6, 0.8$ in order to deduce the best value for each neighborhood type. The value of $e = 0.4$ has been chosen as the one providing the best average results among all instances in the test set. It was also observed that if a method is performed several times on the same instance, it gets similar results. This indicates the robustness of the proposed approach.

Proper tuning of parameters is important in order to reliably compare all of the iterative methods. All stages of parameters tuning in the proposed methodology have been performed automatically. The time limit T tuned for TS has been used also as a time limit for the DLS methods. Additionally, since these methods tend to get quickly stuck in local optima, a *restarting* procedure has been implemented.

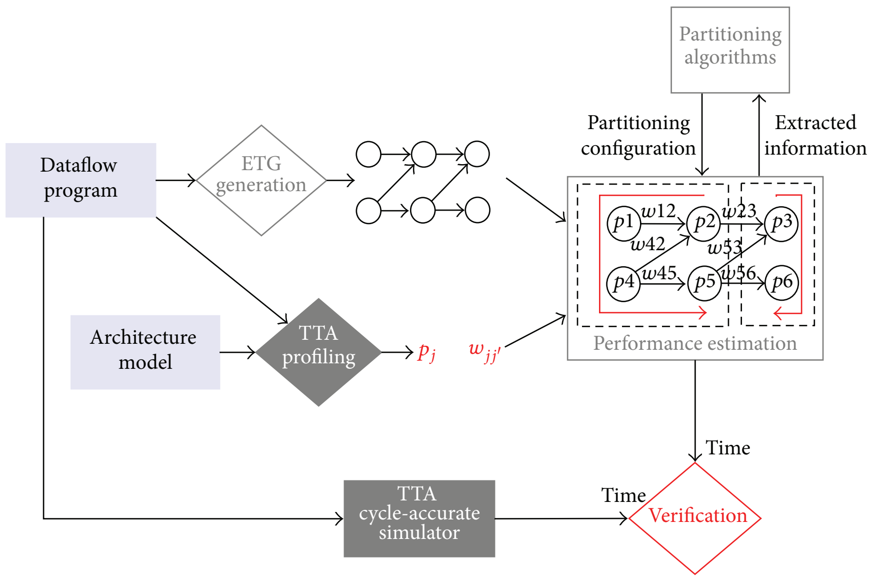

Figure 4: Experimental workflow.

If DLS finishes before elapsing the given time limit, it is restarted with a new random solution. At the end, the best found solution (among the restarts) is returned.

6.2. Methodology of Experiments. Most of the tools used for the experiments are the components of Turnus codesign framework [44]. They include the generation of an ETG for a given statistically meaningful input stimulus, the performance estimation tool exploiting the ETG, and the results of the platform profiling and the generation of partitioning configurations using different algorithms. Complementary units in this workflow are the profiling of the TTA architecture and a TTA cycle-accurate simulator [45] that allows a verification of the estimated results in terms of a real execution time obtained on the platform. The complete workflow is presented in Figure 4.

The partitioning configurations were generated using each of the described algorithm for the number of processors between 2 and 8. Considering the choice of application, 8 units should already approach their potential parallelism. For the local search methods that require specifying an initial solution, in each case, two sets were tested: the random one and the one generated by the WB algorithm. Such a choice was made in order to provide the algorithms with possibly good, as well as bad, initial configurations and also observe their sensitivity to the quality of an initial solution. The evaluation of the solutions generated by each algorithm was accomplished by the performance estimation tool that calculated the total execution time expressed in clock-cycles. Based on those values, the speed-up versus the mono-core execution was calculated in each case. The results presented in this section target the calculated values of speed-up in order to relate them easily to the potential parallelism of the application. Finally, the results obtained by estimation were also consistently verified by the platform execution.

6.3. Greedy and Decent Local Search Heuristics. Table 1 contains the speed-up values of partitioning configurations generated with the WB and BP algorithms along with the values estimated for a random set of configurations. Tables 2 and 3 contain the results obtained for the IDLS and CFDLS heuristics for the two sets of initial partitioning configurations.

Table 1: Speed-up: greedy constructive procedures.

Proc.	Workload balance	Balanced pipeline	Random
1	1.00	1.00	1.00
2	1.83	1.79	1.75
3	2.66	2.20	2.08
4	3.07	2.84	2.77
5	3.38	3.10	2.34
6	4.07	3.09	2.61
7	5.40	3.10	2.48
8	5.76	3.10	3.14

Since the purpose of a greedy constructive method is to build a solution from scratch, an important property is the scalability of the performance. In this case, both algorithms scale; however, the BP achieves a saturation already around 5 processors, unlike the WB that scales further. The maximal speed-up obtained for BP configurations is similar to the random configurations but is achieved on a smaller number of processors (5 versus 8). Applying the IDLS and CFDLS methods in all the cases improve the initial solution, but the improvement is bigger for CFDLS. The quality of the solution provided by the DLS heuristics depends also strongly on the quality of the solution provided as a starting configuration.

6.4. Tabu Search. The first experiment aimed at confirming the most beneficial types of moves. It was performed separately for each type of neighborhood structure. In the first execution, only the REINSERT moves were allowed,

Table 2: DLS speed-up: balanced start.

Proc.	IDLS	CFDLS
1	1.00	1.00
2	1.83	1.94
3	2.67	2.68
4	3.25	3.49
5	3.48	4.20
6	4.08	4.75
7	5.47	5.63
8	5.93	6.06

Table 3: DLS speed-up: random start.

Proc.	IDLS	CFDLS
1	1.00	1.00
2	1.77	1.95
3	2.09	2.32
4	2.78	3.19
5	2.35	4.27
6	2.95	3.98
7	3.02	3.93
8	3.85	4.23

whereas in the second one, the SWAP moves were also included. SWAP moves were not considered alone, since they do not lead to any change of the initial size of each partition (resulting in a nonconnected solution space). The results of this comparison for each neighborhood type are presented in Tables 4–11. Along with admitting the SWAP moves, a significant improvement has been brought only to $N^{(B)}$. In fact, the performance of $N^{(B)}$ based on REINSERT only was very poor and a slight improvement has been introduced only for certain initial configurations. It relies on the fact that the possible space of moves is very narrowed in this case (only actors from the most occupied partition are taken into consideration) and the tabu list can be very restrictive. Since it also aims at balancing the workload, for higher number of processors, it is not rare to encounter a solution where the heaviest bottleneck actor is placed alone on the processor. Due to the solution definition described in Section 5.3.1, the algorithm cannot proceed from that point. Relative balancing of the workload between two partitions instead of an overall balancing seems to be a much more effective approach.

For the other neighborhood structures, allowing SWAP moves decreased the quality of the final solution in the vast majority of the cases. It might be due to the fact that SWAP moves unnecessarily increased the set of neighbor solutions and reduce the diversification ability of the method. Comparing the neighborhood structures, there are some conclusions that can be made. First, $N^{(I)}$ seems to outperform the other variants, including $N^{(\mathrm{CF})}$. This observation is contrary to what has been previously observed for the DLS heuristics, where the search based on the *communication frequency* outperformed the *idle* optimization. It confirms that determining a local optimization criterion is challenging, whereas employing an appropriate exploration strategy (i.e., the TS framework) is the other one. Finally, the results obtained for $N^{(I)}$ and $N^{(\mathrm{CF})}$ also prove that a *guided* choice of moves outperforms a random selection. In other words, the complete freedom of choice when choosing a move, as for $N^{(R)}$, does not necessarily lead to competitive solutions.

Table 4: Speed-up: $N^{(B)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.85	1.94
3	2.67	2.77
4	3.21	3.48
5	4.04	4.32
6	4.83	4.85
7	5.40	5.62
8	5.76	6.19

Table 5: Speed-up: $N^{(B)}$ with random start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.75	1.95
3	2.08	2.74
4	2.77	3.30
5	2.34	4.13
6	2.61	3.68
7	2.48	4.04
8	3.24	4.76

Table 6: Speed-up: $N^{(I)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.92	1.92
3	2.75	2.80
4	3.61	3.46
5	4.45	4.23
6	4.92	4.88
7	5.81	5.66
8	6.28	6.18

Table 7: Speed-up: $N^{(I)}$ with random start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.93	1.94
3	2.66	2.66
4	3.36	3.27
5	3.94	3.59
6	4.73	4.12
7	4.31	4.65
8	5.95	4.65

Table 8: Speed-up: $N^{(\mathrm{CF})}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.88	1.93
3	2.79	2.78
4	3.49	3.56
5	4.30	4.29
6	4.95	4.97
7	5.79	5.74
8	6.26	6.18

Table 9: Speed-up: $N^{(\mathrm{CF})}$ with random start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.87	1.88
3	2.65	2.68
4	3.33	3.29
5	4.23	4.36
6	4.94	3.85
7	4.62	4.36
8	4.66	4.23

Table 10: Speed-up: $N^{(R)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.92	1.94
3	2.70	2.74
4	3.47	3.38
5	4.26	4.11
6	4.91	4.76
7	5.64	5.57
8	6.27	6.18

Table 11: Speed-up: $N^{(R)}$ with random start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.94	1.90
3	2.65	2.29
4	3.30	3.16
5	4.01	3.20
6	3.65	3.53
7	4.02	3.47
8	4.56	4.36

The final part of experiments with TS involves a comparison of its two advanced variants. Taking into consideration the previous observations, the analysis involved neighborhood $N^{(B)}$ based on the SWAP moves and neighborhoods $N^{(I)}$, $N^{(\mathrm{CF})}$, and $N^{(R)}$ based on REINSERT moves. Since different types provide different sizes of the neighborhood sets, such sizes have been equalized according to the averaged values. For this reason, another parameter, namely, the *admission rate*, has been introduced for each neighborhood structure. Admission rate expresses the percentage of moves that is generated at each iteration. For $N^{(I)}$ and $N^{(\mathrm{CF})}$, a given percentage of moves is extracted according to the priorities (i.e., most idle or most communicative actors, resp.). For $N^{(B)}$ and $N^{(R)}$, since there are no priorities, the solutions are extracted randomly. The values of admission rate have been tuned as follows: 0.9 for $N^{(I)}$, 0.48 for $N^{(\mathrm{CF})}$, 0.16 for $N^{(R)}$, and 0.08 for $N^{(B)}$.

Table 12: PTS and JTS speed-up: balanced start.

Proc.	PTS	JTS
1	1.00	1.00
2	1.93	1.96
3	2.83	2.80
4	3.58	3.57
5	4.44	4.37
6	5.13	5.03
7	5.81	5.72
8	6.22	6.22

Table 13: PTS and JTS speed-up: random start.

Proc.	PTS	JTS
1	1.00	1.00
2	1.96	1.93
3	2.77	2.71
4	3.62	3.39
5	4.42	4.26
6	4.98	4.31
7	5.10	4.86
8	5.72	5.12

Tables 12 and 13 contain the results of the analysis of the advanced variants of TS. In almost all cases, PTS performed better than JTS and provided the results that, considering the previously mentioned upper bound on the potential parallelism of an application, can be considered as close-to-optimal. PTS and JTS were also less sensitive to the quality of the initial configuration. In fact, in few cases, a random initial solution leads to better results than a balanced initial configuration.

6.5. Consistency Verification. In order to verify the consistency of the obtained estimated results and their relation to the platform execution, a comparison of execution times was performed on a representative fraction of generated partitioning configurations spanned on each of the considered numbers of processors. Figure 5 presents a chart indicating the consistency of the estimation and platform execution results. Finally, Table 14 summarizes the best solutions obtained with PTS for each number of processors. Apart from the values of execution times expressed in clock-cycles and the speed-up, the distance between the execution

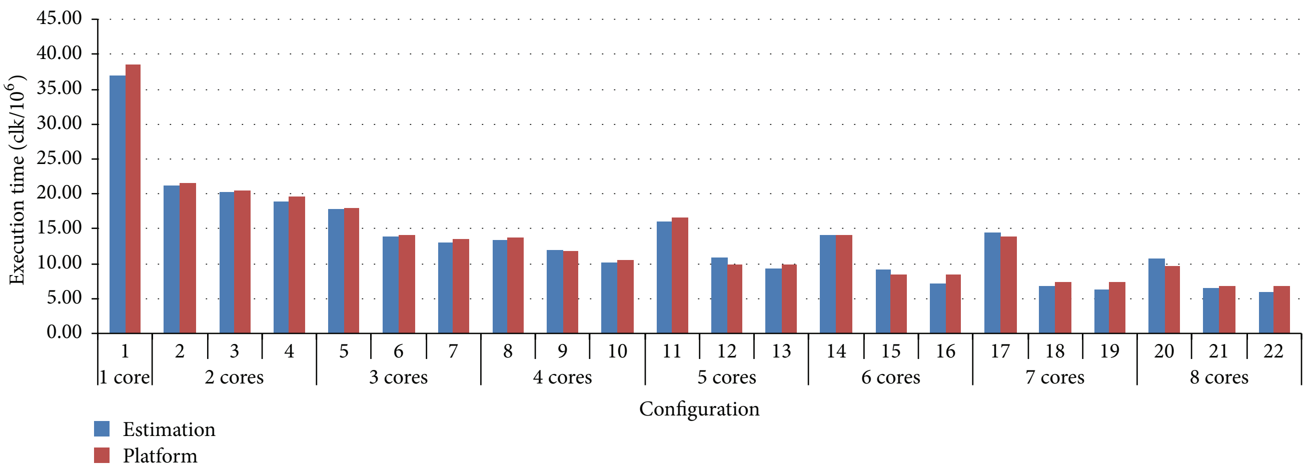

Figure 5: Consistency verification.

Table 14: Improvement summary.

Proc.	Time	Speed-up	CP dist [%]	Diff [%]
1	36938764	1.00	528	4.06
2	18839134	1.96	220	12.64
3	13045976	2.83	122	7.35
4	10200033	3.62	73	3.62
5	8321995	4.44	41	14.73
6	7194547	5.13	22	13.8
7	6354158	5.81	8	14.28
8	5941632	6.22	1	11.96

Table 15: Averaged time of the final improvement.

Algorithm	Time
WB	N/A
BP	N/A
IDLS	73 min
CFDLS	58 min
$N^{(B)}$	13 min
$N^{(I)}$	44 min
$N^{(\mathrm{CF})}$	84 min
$N^{(R)}$	318 min
JTS	308 min
PTS	276 min

time and the length of the critical path expressed in % is also highlighted. This value indicates how far a given solution is from the maximal parallelism of the application. It might be a particularly precious information for the application designer in terms of possible tracking the approach to the potential parallelism of an application. The last column contains the value of estimation discrepancy for this particular solution.

6.6. Discussion. Comparing the results obtained for all implemented algorithms, the first observation is that according to the decreasing quality of the output solutions, the algorithms can be ordered as follows: advanced TS variants, TS, DLS, and the greedy constructive procedures. This ranking is consistent, as a more refined approach outperforms a simpler one. It highlights that the specific ingredients belonging to a more refined method are relevant. Additionally, finding a good partitioning configuration for a small number of processors (i.e., 2 or 3) is relatively easy and the differences between the solutions provided by different algorithms are minor. For instance, for the case of two processors, the difference between the solutions provided by the best and the worst algorithms is less than 6%. With the increasing number of processors, the differences become more significant. For the case of the WB algorithm, the biggest difference of 30% with respect to PTS can be observed at around 5 processors, whereas for the BP algorithm, on 8 processors, the difference goes up to 100%.

The comparison of different variants of TS leads to a conclusion that the resulting solution might benefit from varying the definition of the neighborhood. In fact, both JTS and PTS outperformed the variants where only one type of neighborhood was taken into consideration. Among the advanced variants, the success of PTS over JTS might rely on two factors: (1) using the history of local search, which allows an adaptation of the search to the properties of the test case and (2) much smaller size of the neighborhood in each iteration that contributes to a diversification of the search.

An important aspect that must be also taken into account for evaluating the algorithms is the time required for their completion. It involves the evaluation time for all considered solutions in all iterations, extraction of the optimization criteria, and computation of new solutions. For DLS and TS, the upper bound on the time is defined by the user. However, for each algorithm, it was observed when the last improving move (before termination at the specified point) was performed. The averaged values among different instances are summarized in Table 15. For the TS, a big difference is visible between $N^{(R)}$ and the other variants. In fact, in this work, it is the time for $N^{(R)}$ that enforces the time limit for all other algorithms, but in the case of this particular variant, it does not necessarily correspond to the quality of the final solution. A promising observation can be made for the advanced variants of TS, since PTS not only provides with the best results, but also succeeds in ca. 10% shorter time than JTS. In all cases, the most impacting factor is the number of iterations performed, since the performance estimation and, at the same time, extraction of optimization criteria much outstrip the cost of computing the new solution.

Since the described partitioning methodology relies on an estimation of the actual performance, an important question is the estimation quality in terms of precision. This specific issue occurs in various fields [46]. For the case of experiments performed in this work, the average estimation discrepancy is between 3.35% (in total) and 10.31% (for the subset of best solutions). These values can be considered acceptable, since they still allow DLS and TS algorithms to iteratively improve an initial solution. In order to explain the reasons of this discrepancy, the first factor that must be taken into account is the general uncertainty of profiling [47]. The estimation of the execution time must rely on the limited set of measurements coming from the platform that can be burdened with some errors. On the other hand, there are some properties of the program execution not present in the proposed execution model, because of their hard tractability. A good example is the overhead introduced by a partition scheduler [48] that might depend on multiple factors, such as the number of actors in one partition, the properties of a scheduling policy, the number of conditions to be checked before an actor is executed, or even the order of appearance of actors on the list representing each partition.

7. Conclusion

This paper presents a partitioning methodology for dynamic dataflow programs that is based on a program execution model and uses multiple solution methods belonging to different classes of optimization algorithms: greedy constructive procedures, decent local search, and tabu search metaheuristic. The algorithms differ in terms of the time needed for generating a solution, the quality of the final solution, and the way they explore different properties of a dataflow application and its execution on the target platform. The best performing algorithms have been verified to approach the full potential parallelism of dataflow programs and hence to be capable of efficiently exploring the design space in all the partitioning dimensions.

The algorithms are based on the performance estimation of a program execution on a target platform. The estimation takes into account the partitioning, scheduling, and buffer size configuration and is capable of evaluating performances in terms of the total execution time. During the evaluation, the execution properties are also tracked and extracted in order to provide optimization criteria to the algorithms. The estimation has been experimentally verified to be highly accurate and consistent with real executions on the considered platform.

A direction for future improvements is to investigate the opportunities of extending the model and the methodology by further properties in order to minimize the estimation discrepancy. The methods might include tracking the variance for different executions of the same action in order to provide the model with individual weights for each job and profiling of scheduling cost that might be used for accurate scheduling and buffer size optimization. Further study is also required in order to determine the level of influence of these configurations on each other in order to effectively explore the design space in multiple dimensions. After validating the methodology on a simple platform with small level of uncertainty, an important objective is to consider more performing platforms which could turn to be more difficult to model. Studying the methodology of profiling communication cost occurring on such platform is currently an ongoing work.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work is supported by the *Fonds National Suisse pour la Recherche Scientifique*, under Grant 200021.138214.

References

[1] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, vol. 19 of *Lecture Notes in Computer Science*, pp. 362–376, Springer, Berlin, Germany, 1974.

[2] G. Kahn, *The Semantics of Simple Language for Parallel Programming*, IFIP Congress, 1974.

[3] S. Bhattacharyya, P. Murthy, and E. Lee, *Software Synthesis from Dataflow Graphs*, vol. 360, Springer Science & Business Media, 2012.

[4] J. Castrillon, R. Velasquez, A. Stulova et al., "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '10)*, pp. 753–758, Dresden, Germany, March 2010.

[5] M. Mattavelli, "MPEG reconfigurable video representation," in *The MPEG Representation of Digital Media*, L. Chiariglione, Ed., pp. 231–247, Springer, New York, NY, USA, 2012.

[6] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[7] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384–393, 1975.

[8] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, article 46, 2014.

[9] L. Benini, M. Lombardi, M. Milano, and M. Ruggiero, "Optimal resource allocation and scheduling for the CELL BE platform," *Annals of Operations Research*, vol. 184, pp. 51–77, 2011.

[10] S. Casale Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. W. Janneck, "Partitioning and optimization of high level stream applications for multi clock domain architectures," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS '13)*, pp. 177–182, October 2013.

[11] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, New York, NY, USA, 2008.

[12] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD '07)*, pp. 29–40, IEEE, Bratislava, Slovakia, July 2007.

[13] M. Selva, *Performance monitoring of throughput constrained data flow programs executed on shared-memory multi-core architectures [Ph.D. thesis]*, INSA, Lyon, France, 2015.

[14] M. Garey and D. Johnson, *A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

[15] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, 2003.

[16] I. H. Osman and G. Laporte, "Metaheuristics: a bibliography," *Annals of Operations Research*, vol. 63, pp. 513–623, 1996.

[17] M. Gendreau and J.-Y. Potvin, *Handbook of Metaheuristics*, Springer, Berlin, Germany, 2010.

[18] N. Zufferey, "Metaheuristics: some principles for an efficient design," *Computer Technology and Application*, vol. 3, pp. 446–462, 2012.

[19] M. A. Arslan, J. W. Janneck, and K. Kuchcinski, "Partitioning and mapping dynamic dataflow programs," in *Proceedings of the 46th Asilomar Conference on Signals, Systems and Computers (ASILOMAR '12)*, pp. 1452–1456, IEEE, Pacific Grove, Calif, USA, November 2012.

[20] J. Teich, T. Blickle, and L. Thiele, "An evolutionary approach to system-level synthesis," in *Proceedings of the 5th International Workshop on Hardware/Software Codesign (Codes/CASHE '97)*, pp. 167–171, Braunschweig, Germany, March 1997.

[21] T. Schlichter, M. Lukasiewycz, C. Haubelt, and J. Teich, "Improving system level design space exploration by incorporating sat-solvers into multi-objective evolutionary algorithms," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '06)*, pp. 309–316, Karlsruhe, Germany, March 2006.

[22] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. Nezan, "An open framework for rapid prototyping of signal processing applications," *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 598529, 2009.

[23] M. Obaidat and G. Papadimitriou, *Applied System Simulation: Methodologies and Applications*, Springer, New York, NY, USA, 2013.

[24] P. Bose and T. M. Conte, "Performance analysis and its impact on design," *Computer*, vol. 31, no. 5, pp. 41–49, 1998.

[25] S. Pllana, I. Brandic, and S. Benkner, "Performance modeling and prediction of parallel and distributed computing systems: a survey of the state of the art," in *Proceedings of the 1st International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '07)*, pp. 279–284, Vienna, Austria, April 2007.

[26] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, 2013.

[27] J. Castrillon, *Programming Heterogeneous MPSoCs*, Springer, Berlin, Germany, 2013.

[28] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–111, 2006.

[29] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[30] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering: Selected Results and Surveys*, vol. 9220 of *Lecture Notes in Computer Science*, 2015.

[31] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[32] F. Pellegrini and J. Roman, "A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking: International Conference and Exhibition HPCN EUROPE 1996 Brussels, Belgium, April 15–19, 1996 Proceedings*, vol. 1067 of *Lecture Notes in Computer Science*, pp. 493–498, Springer, Berlin, Germany, 1996.

[33] H. Yviquel, E. Casseau, M. Raulet, P. Jaaskelainen, and J. Takala, "Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms," in *Proceedings of the 8th International Symposium on Image and Signal Processing and Analysis (ISPA '13)*, pp. 732–737, September 2013.

[34] S. Bhattacharyya, E. Deprettere, and B. Theelen, "Dynamic dataflow graphs," in *Handbook of Signal Processing Systems*, pp. 905–944, Springer, Berlin, Germany, 2013.

[35] T. Parks, J. Pino, and E. A. Lee, "A comparison of synchronous and cyclo-static dataflow," in *Proceedings of the Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, Calif, USA, November 1995.

[36] S. Casale-Brunet, *Analysis and optimization of dynamic dataflow programs [Ph.D. thesis]*, EPFL, Lausanne, Switzerland, 2015.

[37] T. Tabirca, S. Tabirca, L. Freeman, and L. T. Yang, "A static workload balance scheduling algorithm," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW '02)*, pp. 235–239, August 2002.

[38] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, "Execution trace graph based multi-criteria partitioning of stream programs," *Procedia Computer Science*, vol. 51, pp. 1443–1452, 2015.

[39] S. Thevenin, N. Zufferey, and M. Widmer, "Metaheuristics for a scheduling problem with rejection and tardiness penalties," *Journal of Scheduling*, vol. 18, no. 1, pp. 89–105, 2015.

[40] F. Glover, "Tabu search—part I," *ORSA Journal on Computing*, vol. 1, pp. 190–205, 1989.

[41] J. W. Janneck and J. Eker, "CAL language report," Tech. Memo UCB/ERL M03/48, UC Berkeley, 2003.

[42] TTA-Based Co-Design Environment, http://tce.cs.tut.fi/tta.html.

[43] M. Michalska, J. Boutellier, and M. Mattavelli, "A methodology for profiling and partitioning stream programs on many-core architectures," *Procedia Computer Science*, vol. 51, pp. 2962–2966, 2015.

[44] S. Casale-Brunet, C. Alberti, M. Mattavelli, and J. W. Janneck, "Turnus: a unified dataflow design space exploration framework for heterogeneous parallel systems," in *Proceedings of the 7th Conference on Design and Architectures for Signal and Image Processing (DASIP '13)*, pp. 47–54, Cagliari, Italy, October 2013.

[45] H. Yviquel, A. Sanchez, P. Jääskeläinen, J. Takala, M. Raulet, and E. Casseau, "Embedded multi-core systems dedicated to dynamic dataflow programs," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 121–136, 2014.

[46] E. A. Silver and N. Zufferey, "Inventory control of an item with a probabilistic replenishment lead time and a known supplier shutdown period," *International Journal of Production Research*, vol. 49, no. 4, pp. 923–947, 2011.

[47] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*, pp. 215–224, Austin, Tex, USA, April 2013.

[48] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, article 35, 2011.