Reducing the Complexity of Dataflow Graphs Using Slack-Based Merging

HAZEM ISMAIL ALI, CISTER Research Centre/INESC-TEC
SANDER STUIJK, Eindhoven University of Technology
BENNY AKESSON, CISTER Research Centre/INESC-TEC, Czech Technical University in Prague
LUÍS MIGUEL PINHO, CISTER Research Centre/INESC-TEC

There exist many dataflow applications with timing constraints that require real-time guarantees on safe execution without violating their deadlines. Extraction of timing parameters (offsets, deadlines, periods) from these applications enables the use of real-time scheduling and analysis techniques, and provides guarantees on satisfying timing constraints. However, existing extraction techniques require the transformation of the dataflow application from highly expressive dataflow computational models, for example, Synchronous Dataflow (SDF) and Cyclo-Static Dataflow (CSDF) to Homogeneous Synchronous Dataflow (HSDF). This transformation can lead to an exponential increase in the size of the application graph that significantly increases the runtime of the analysis.

In this article, we address this problem by proposing an offline heuristic algorithm called *slack-based merging*. The algorithm is a novel graph reduction technique that helps in speeding up the process of timing parameter extraction and finding a feasible real-time schedule, thereby reducing the overall design time of the real-time system. It uses two main concepts: (a) the difference between the worst-case execution time of the SDF graph's firings and its timing constraints (slack) to merge firings together and generate a reduced-size HSDF graph, and (b) the novel concept of merging called *safe merge*, which is a merge operation that we formally prove cannot cause a live HSDF graph to deadlock. The results show that the reduced graph (1) respects the throughput and latency constraints of the original application graph and (2) typically speeds up the process of extracting timing parameters and finding a feasible real-time schedule for real-time dataflow applications. They also show that when the throughput constraint is relaxed with respect to the maximal throughput of the graph, the merging algorithm is able to achieve a larger reduction in graph size, which in turn results in a larger speedup of the real-time scheduling algorithms.

CCS Concepts: • **Theory of computation → Streaming models**; • **Computer systems organization → Real-time system specification**; • **Software and its engineering → Data flow architectures;**

Additional Key Words and Phrases: Synchronous dataflow model, merging algorithms

This work was partially supported by Portuguese National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme Thematic Factors of Competitiveness), within project FCOMP-01-0124-FEDER-037281 (CISTER) and by FCT and EU ARTEMIS JU, within project ARTEMIS/0001/2013, JU grant no. 621429 (EMC^2), under PhD grant SFRH/BD/79872/2011, and project CATRENE CA703 OpenES. Also by the European social fund within the framework of realizing the project "Support of inter-sectoral mobility and quality enhancement of research teams at Czech Technical University in Prague," CZ.1.07/2.3.00/30.0034.
Authors' addresses: H. I. Ali, B. Akesson, and L. M. Pinho, CISTER Research Centre, ISEP - Instituto Superior de Engenharia do Porto, Rua Dr. António Bernardino de Almeida 431, 4249-015 PORTO, Portugal; S. Stuijk, Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems, Groene loper 19 (Flux 3.130) 5612 AZ, Eindhoven, The Netherlands.

DOI: http://dx.doi.org/10.1145/2956232

ACM Reference Format:
Hazem Ismail Ali, Sander Stuijk, Benny Akesson, and Luís Miguel Pinho. 2017. Reducing the complexity of dataflow graphs using slack-based merging. ACM Trans. Des. Autom. Electron. Syst. 22, 2, Article 24 (January 2017), 22 pages.
DOI: http://dx.doi.org/10.1145/2956232

1. INTRODUCTION

The Synchronous Dataflow (SDF) model of computation [Lee and Messerschmitt 1987] is widely used for representing streaming applications. This is due to its simplicity and ability to exploit parallelism in embedded streaming applications. SDF is a parallel computation model that satisfies the high processing requirements of streaming applications, for example, H.264 video decoders [Kim et al. 2010], by enabling the use of massive computational power of current multi- and many-core processors [Pankratius et al. 2009].

The SDF model can be used to analyse and derive different parameters that define a dataflow application. Examples of these parameters are throughput [Ghamarian et al. 2008; Damavandpeyma et al. 2012], latency [Ghamarian et al. 2007; Bamakhrama and Stefanov 2012], scheduling [Moreira et al. 2007], and timing parameters (offsets, deadlines, and periods) [Bamakhrama and Stefanov 2011; Saifullah et al. 2011; Ali et al. 2015; Bekooij et al. 2005; Liu et al. 2014; Hausmans et al. 2013]. Some of these analysis algorithms operate directly on SDF graphs, while many others require transformation to Homogeneous Synchronous Dataflow (HSDF) graphs prior to the analysis. This transformation can lead to an exponential increase in the size of the original SDF graph, which significantly increases the runtime of the analysis algorithm.

Timing parameter extraction algorithms for cyclic HSDF graphs (i.e., Ali et al. [2013, 2015], Moreira et al. [2007], and Hausmans et al. [2013]) are examples of algorithms where the size of the HSDF graphs significantly affects the runtime. These algorithms enable the use of a wide range of real-time scheduling and analysis techniques on dataflow applications with timing constraints. There are other techniques that have been proposed for extracting timing parameters directly from different dataflow computational models [Bamakhrama and Stefanov 2011, 2012; Liu et al. 2014]. However, these techniques are restricted to acyclic graphs. In all cases, the size of the input graphs affects the runtime of the algorithms.

The problem of extracting timing parameters is not only restricted to dataflow applications with timing constraints. It can be generalised to cover parallel applications with timing constraints. This problem has been addressed by several works [Saifullah et al. 2011; Lipari and Bini 2011; Qamhieh et al. 2013; Pinho et al. 2014], where the authors model the parallel application as a graph of communicating tasks. They propose algorithms for extracting timing parameters of these applications that allow them to apply real-time scheduling and analysis techniques that provide guarantees on safe execution without violating timing constraints. Similarly to dataflow, the runtime of these algorithms has a direct relation to the size of the input graphs. This shows the need for graph reduction techniques to speed up the process of timing parameter extraction and finding a feasible real-time schedule. The problem of generating reduced-size HSDF graphs has been tackled before in Geilen [2009]. However, the generated graph is not suitable for extracting timing parameters, as explained in detail in Section 2.

In this article, we propose a heuristic algorithm called *slack-based merging*. It is an offline graph reduction technique that aims to speed up the process of timing parameter extraction and finding a feasible real-time schedule, thereby reducing the overall design time of the real-time system. To achieve this goal, the algorithm combines two main concepts: (a) the *slack*, which is the difference between the worst-case execution time of the SDF graph's firings and its timing constraints, and (b) the *safe merge*, which

is a novel merging concept that we formally prove cannot cause a live HSDF graph to deadlock. The output is a reduced-size HSDF graph that satisfies the throughput and latency constraints of the original application graph. However, it may have a decreased maximum throughput compared to the original one. This reduction is not a problem because in real-time systems there is no need to do better than the timing constraints (throughput and latency) and our algorithm exploits this to address the scalability problem that comes from HSDF transformation. The experimental results show that the generated reduced-size HSDF graphs speed up the process of extracting their timing parameters and finding a feasible real-time schedule compared to using the original HSDF graphs. Moreover, when the throughput constraint is relaxed with respect to the maximal throughput of the application graph, the merging algorithm is able to achieve a larger reduction in graph size, which in turn results in a larger speedup in the parameter extraction and scheduling processes.

The remainder of this article is organized as follows. Section 2 provides an overview of related work. Section 3 provides the necessary preliminaries to understand the system model and the proposed algorithm. The *slack-based merging* algorithm and its complexity analysis are detailed in Section 4. Section 5 provides experimental results. Finally, we provide conclusions in Section 6.

2. RELATED WORK

This section reviews the state of the art related to timing parameter extraction algorithms and reduction techniques for dataflow graphs.

There exist several works on extracting timing parameters of dataflow applications with timing constraints. For example, Moreira et al. [2007] presents a method to calculate individual deadlines of HSDF actors. The method is based on an Integer Linear Programming (ILP) optimization problem that finds the amount of slack for each actor that makes it able to extend its execution without violating the HSDF throughput and timing constraints. However, their proposed method is restricted to strongly connected HSDF graphs. In Hausmans et al. [2013], the authors propose a temporal analysis for dataflow applications modelled as cyclic HSDF graphs under a nonstarvation-free scheduler, that is, Static-Priority Preemptive scheduling (SPP). To apply their analysis, they extract timing properties like offsets, periods, and execution times, but not deadlines. In Ali et al. [2015], the authors proposed a generalised algorithm for deriving timing parameters (offsets, deadlines, and periods) of actors of cyclic HSDF applications with multiple inputs and outputs. The proposed algorithm enables applying a wide range of real-time scheduling (static and dynamic priorities) and analysis techniques. In Bamakhrama and Stefanov [2011, 2012] and Liu et al. [2014], the authors provide an analytical framework for computing timing parameters for actors of Cyclo-Static Dataflow (CSDF) applications with a single input. Although these works use expressive computational models as input, their approaches are limited to acyclic graphs. In Bouakaz et al. [2012], the authors present a new dataflow computational model that is a superset of SDF/CSDF application graphs called Affine Dataflow (ADF). The ADF is a time-triggered dataflow model that explicitly represents each firing of each actor in a complete iteration of the graph as a so-called clock tick. These clock ticks are related to each other using firing relations called affine relations. These relations maintain precedence constraints between different firings of actors in the graph, since it ensures the correct execution order of different clock ticks. Based on this framework, they present an algorithm that computes affine schedules for these clock ticks, where it enables applying real-time scheduling algorithms, for example, earliest deadline first or rate monotonic. However, the use of clock tick representation and affine relations to represent the firing behaviour of actors does not speed up the process of finding a feasible schedule, because it indirectly transforms the ADF to an HSDF graph (using

the clock tick representation) to be able to find a feasible schedule. In addition, the presented algorithm does not support end-to-end latency constraints, since it assumes an implicit-deadline task model.

The problem of extracting timing parameters is not restricted to dataflow applications with timing constraints. It also extends to cover general parallel applications with timing constraints. In Lipari and Bini [2011], the authors present a deadline assignment approach called ORDER for dependent tasks composing real-time pipeline applications executing on a multicore system. The proposed approach considers the problem of scheduling a pipeline such that the end-to-end deadline is met and the amount of required resource capacity is minimal. In Saifullah et al. [2011], the authors also address the problem of scheduling periodic tasks, each consisting of subtasks forming an acyclic graph. They are assigned individual deadlines and release times such that all subtasks have equal densities. Another approach presented in Qamhieh et al. [2013] calculates offsets and deadlines for subtasks in an acyclic task graph based on computing the interference between each subtask and the higher-priority subtasks of all tasks in an acyclic graph running on the system.

In all previous work, the runtime of the proposed timing parameter extraction algorithms has a direct relation with the size of the input graph. Reducing the size of the input graph will likely have a positive effect on the algorithm runtime. This is because these algorithms will have less number of actors/tasks for which to extract timing parameters, which is the main goal of our proposed algorithm.

In Geilen [2009], the authors propose a SDF graph reduction technique based on maxplus algebra. It transforms an SDF graph into a smaller HSDF graph with equivalent maximal throughput and latency, which is faster to analyse. However, the output HSDF graph of this technique hides the actual execution behaviour of the original SDF graph, because a single firing of an SDF actor can exist in multiple actors of the output HSDF graph. This means that a single firing in the SDF graph is executed multiple times in the output HSDF graph, which complicates extracting timing parameters and finding a feasible schedule. In contrast, we propose a reduction algorithm that generates a reduced-size HSDF graph that speeds up the process of extracting timing parameters, as shown in the experiments. In addition, having a reduced-size graph speeds up the process of finding a feasible mapping and schedule for the application, since the number of tasks in the generated graph is smaller compared to the original HSDF graph. Also, the generated graph represents the actual execution behaviour of the original graph, avoiding the problem with the approach in Geilen [2009]. It also ensures that the throughput and latency constraints are met, although with a possibility of having a lower maximum throughput compared to the original graph. This is not a problem, because the main goal for real-time systems is satisfying timing constraints.

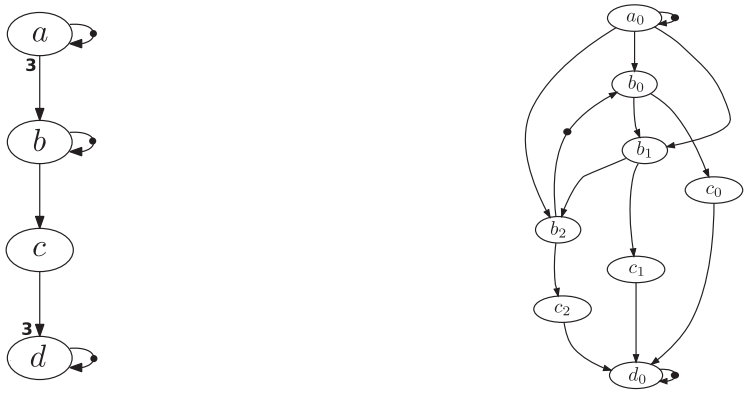
Also, having a reduced-size graph speeds up the process of finding a feasible mapping and schedule for the application, since its number of tasks is smaller compared to the original HSDF graph.

3. PRELIMINARIES

In this section, we present background material that is essential for understanding the computational model, the system model, and the proposed algorithm.

3.1. Synchronous Dataflow

The Synchronous Dataflow (SDF) model of computation [Lee and Messerschmitt 1987] is widely used in modelling and analysing streaming, Digital Signal Processing (DSP), and concurrent multimedia applications [Bhattacharyya et al. 1999; Sriram and Bhattacharyya 2000]. Its use is considered for designing applications for multi- and many-core processors [Poplavko et al. 2003; de Dinechin et al. 2013]. A synchronous



(a) SDF graph (b) HSDF graph

Fig. 1. An SDF graph and its HSDF representation.

dataflow application graph, shown in Figure 1(a), is a data-driven network of *actors* (nodes), where the same behaviour repeats in each actor every time it is executed. An actor *fires* (executes) once all its input ports have the required *tokens* (data) for consumption. Each actor has production and consumption rates associated with its ports that determine the number of input and output tokens atomically produced and consumed in the firing process. Also, they determine the number of firings of each actor in a complete graph iteration, which is called a *repetition vector*. In SDF, the network that connects the actors (*channels*) can have *initial tokens*. Every initial token represents a delay between the token that is produced and consumed at the other end of the channel. Tokens are always consumed in a First In First Out (FIFO) order.

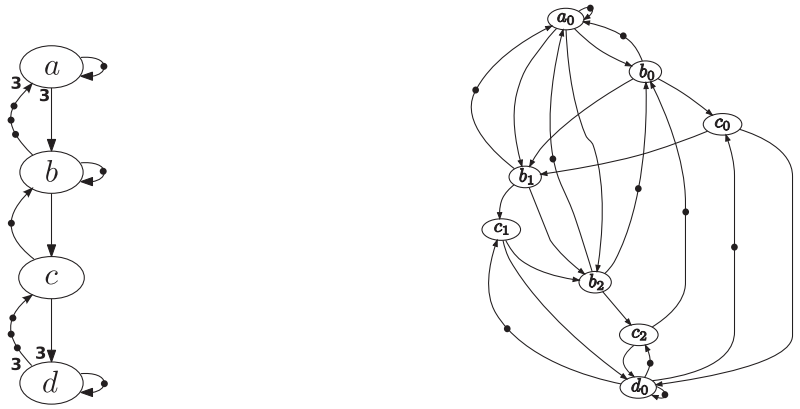
3.2. Homogeneous Synchronous Dataflow

Homogeneous Synchronous Dataflow (HSDF) [Lee and Messerschmitt 1987], as shown in Figure 1(b), is a special case of SDF graphs in which all production and consumption rates associated with actor ports are equal to 1. Therefore, when each actor is fired once, the distribution of tokens on all channels return to their initial state completing a graph iteration.

Every SDF [Lee and Messerschmitt 1987] graph can be converted to an equivalent HSDF graph. Figure 1 shows an example of an SDF graph and its equivalent HSDF graph. The conversion can be done using several algorithms, such as the one presented in Sriram and Bhattacharyya [2000]. The conversion to HSDF is fundamental, since many dataflow analysis algorithms depend on it.

3.3. Buffer Modelling in SDF Graphs

In theory, SDF channels have infinite buffer sizes. However, in practice SDF channel buffer sizes must be finite. Finite buffer sizes for channels can be modelled by adding back-edges carrying a number of initial tokens. These initial tokens on each back-edge represent the buffer size (in tokens) available to the corresponding channel. Figure 2(a) shows the example application from Figure 1(a), considering finite buffer sizes. As we can see, the channels (ab, bc, cd) have buffer sizes of $(3, 1, 3)$ tokens, respectively. These buffer sizes are modelled as back-edges (ba, cb, dc) carrying initial tokens equivalent to the corresponding channel buffer size, as shown in Figure 2(a). The modelling of buffers in an SDF graph affects its execution behaviour, because it adds extra dependencies between firings of different actors, limiting the set of possible firing sequences of the



(a) SDF graph (b) HSDF graph

Fig. 2. An SDF graph and its HSDF representation with finite-size buffers.

graph. Figure 2(b) shows an HSDF graph representation of the SDF graph shown in Figure 2(a). As we can see, firing b_1 is dependent on the three firings a_0, b_0, and c_0. However, in the infinite buffer case shown in Figure 1(b) the same firing b_1 is only dependent on firings a_0 and b_0, which gives the application the freedom to fire b_1 and c_0 in parallel.

3.4. System Model

Any SDF application can be formally represented by a Directed Cyclic Graph (DCG) $G = \langle V, E, d \rangle$, where V is the set of nodes, E is the edges connecting them, and d is the set of delays (initial tokens) on the edges of the graph. Each node in this graph is an actor v_i and each edge is a communication channel. An SDF graph G has a repetition vector $\vec{q}$ that determines the number of firings q_i of each actor $v_i \in V$ in one complete graph iteration (i.e., minimal number of firings to return to same token distribution). Each actor $v_i \in G$ has a computation time denoted by C_i. The j^{th} firing of an SDF actor v_i in G is denoted by v_{i_j} and executes for C_i time units. An SDF application has throughput and latency requirements that must be satisfied for the correct execution of the application. The throughput requirement ζ is a performance measure that determines the minimum output data rate of the application (graph iterations per time unit). The latency requirement L is an end-to-end timing constraint that defines the latest possible time a complete graph iteration of G could finish its execution. In case the end-to-end deadline constraint of G is not defined, L can be set to any value such that L is greater than or equal to the execution time of the Critical Path (CP) in G, defined as follows:

$$L \geq \sum_{\forall v_{i_j} \in CP} C_i. \tag{1}$$

Intuitively, the CP is the longest path of firings v_{i_j}, in terms of execution time C_i, from the input to the output of G.

In this model, we assume that all the SDF applications have periodic input sources and all actors computation time C_i are equal to the Worst Case Execution Time (WCET), which can be determined using methods and tools detailed in Wilhelm et al. [2008]. Therefore, each firing v_{i_j} of an actor v_i in any SDF application can be considered a periodic task with an execution time C_i equal to WCET. The choice of WCET is safe,

because the dataflow model of computation is monotonic, which means faster execution of actors does not result in a worse performance.

4. PROPOSED ALGORITHM

In this section, we present the *slack-based merging* algorithm intended to reduce the size of an HSDF graph with timing constraints. In the following sections, we introduce some definitions required to specify the proposed algorithm (Section 4.1). Then, we detail the merging strategy of our algorithm (Section 4.2), as well as the conditions for guaranteeing a valid merge (Section 4.3). Finally, we present the *slack-based merging* algorithm (Section 4.4) followed by its complexity analysis (Section 4.5) and an example illustrating how it works (Section 4.6).

4.1. Definitions

In this section, we define parameters and concepts essential to our proposed algorithm. They are (1) the *earliest start time of a firing* v_{i_j}, (2) the *latest finish time of a firing* v_{i_j}, (3) a *topologically ordered set of actors*, (4) the concept of *dependent/independent firings*, and (5) the *safe merge* concept, which is fundamental for understanding the *slack-based merging* algorithm. We also prove that a *safe merge* is a deadlock-free operation.

Firstly, the *earliest start time of a firing* parameter defines the earliest possible time instance a firing v_{i_j} can start its execution. It is defined as follows:

Definition 4.1 (*Earliest Start Time of a Firing*). In an SDF application G, the earliest start time of the j^{th} firing v_{i_j} of an actor v_i occurs once all of its input ports have the required input tokens. The required input tokens are available when the latest firing in the set of predecessor firings $\Omega(v_{i_j})$ occur, which contains all the firings that must execute before to enable the firing v_{i_j}. The set of predecessor firings $\Omega(v_{i_j})$ represents the set of precedence constraints that must be satisfied before the v_{i_j} firing. Therefore, the earliest start time ϑ_{i_j} of a firing v_{i_j} is expressed as follows:

$$\vartheta_{i_j} = \begin{cases} 0 & \text{if } \Omega(v_{i_j}) = \varnothing \\ \max_{\forall v_{l_k} \in \Omega(v_{i_j})} (\vartheta_{l_k} + C_l) & \text{if } \Omega(v_{i_j}) \neq \varnothing, \end{cases} \tag{2}$$

where C_l is the WCET of actor v_l and $\varnothing$ is the empty set.

Secondly, the *latest finish time of a firing* parameter defines the latest possible time instance a firing v_{i_j} can finish its execution. It is defined as follows:

Definition 4.2 (*Latest Finish Time of a Firing*). The latest finish time of the j^{th} firing v_{i_j} of an actor v_i in an SDF graph G defines the latest possible time it finishes its execution such that the latency constraint L of the graph G is satisfied. Therefore, θ_{i_j} is expressed as follows:

$$\theta_{i_j} = \begin{cases} L & \text{if } \Phi(v_{i_j}) = \varnothing \\ \min_{\forall v_{l_k} \in \Phi(v_{i_j})} (\theta_{l_k} - C_l) & \text{if } \Phi(v_{i_j}) \neq \varnothing, \end{cases} \tag{3}$$

where $\Phi(v_{i_j})$ is the set of successor firings, which contains all the firings (dependencies) that cannot execute before the firing v_{i_j}.

Thirdly, a topologically ordered set of actors defines the order in which firings are selected for a merge. It is defined as follows:

Definition 4.3 (*Topologically Ordered Set of Actors*). The topologically ordered set of actors $\hat{V}$ is a set in which the actor set V is sorted in a breadth-first traversal

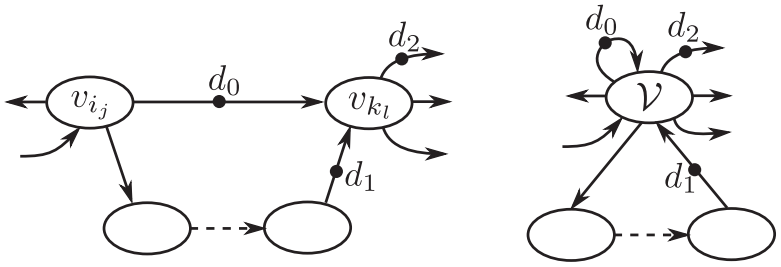


Fig. 3. A *safe merge* operation of two *independent* firings (v_{i_j}, v_{k_l}) into a new cluster $\mathcal{V}$.

sequence, where the input actors (parents) are in the beginning of the set followed by their successor actors (children). To get that ordered set $\hat{V}$, we deal with the SDF graph as an acyclic graph by ignoring all the back edges carrying initial tokens. Then, the SDF graph is traversed in a breadth-first fashion, listing the parent actors followed by their successors. In the case in which a group of actors are on the same level in the graph, they are listed in $\hat{V}$ in arbitrary order. The only order considered in $\hat{V}$ is parents followed by children.

For example, in the case of the graph shown in Figure 2(a), the topological ordered set of actors $\hat{V}$ is (v_a, v_b, v_c, v_d).

Fourthly, the *dependent / independent firings* is a term that describes the connectivity relation between two firings, which helps in deciding whether a merge is safe or not. It is defined as follows:

Definition 4.4 (*Dependent / independent Firings*). Two firings are *dependent* if and only if there is a sequence of edges (not a single edge) connecting them carrying zero initial tokens. Otherwise, they are *independent* firings.

For example, the firings v_{b_0} and v_{b_1} of actor v_b in the cases with infinite and finite buffers shown in Figures 1(b) and 2(b), respectively. In the case of infinite buffers, these firings are *independent*, since there is no path between them other than the direct edge (e_{b_0,b_1}), as shown in Figure 1(b). However, in the case of finite buffers, they are considered *dependent* firings due to the existence of a path between the firings v_{b_0} and v_{b_1} that consists of the firings $(v_{b_0}, v_{c_0}, v_{b_1})$ connected by the sequence of edges $(e_{b_0,c_0}, e_{c_0,b_1})$ that have zero initial tokens, as shown in Figure 2(b).

Lastly, a *safe merge* is a merging operation of any two firings that is defined as follows:

Definition 4.5 (*Safe Merge*). A *safe merge* operation is an act of combining two *independent* firings (v_{i_j}, v_{k_l}) creating a new cluster $\mathcal{V}$ with an execution time equal to the sum of execution time of both firings. The new cluster $\mathcal{V}$ has the same input/output ports and channels of both firings except the ports and channels are carrying zero initial tokens between both firings (v_{i_j}, v_{k_l}). A *safe merge* operation keeps all the initial tokens in the graph distributed on the same edges without change.

Figure 3 shows a merging operation between two *independent* firings (v_{i_j}, v_{k_l}) into a new cluster $\mathcal{V}$. The two firings are *independent* according to Definition 4.4, because the only path connecting them (other than the direct edge that carries the initial token d_0) consists of a sequence of edges that carry the initial token d_1. As we can see, the *safe merge* operation kept the distribution of the initial tokens (d_0, d_1, d_2) the same after the merge.

Applying *safe merge* operations on the graph ensures that the resulting graph is deadlock-free, as stated by the following theorem (the proof is in Appendix A):

THEOREM 4.6. *A **safe merge** operation on a consistent and live HSDF graph results in a new consistent and live HSDF graph.*

However, a *safe merge* operation may cause timing constraints to be violated. Therefore, the slack-based merging algorithm has an additional method to ensure that timing constraints are satisfied called *valid merge*, which is detailed in Section 4.3.

4.2. Merging Strategy

The proposed algorithm combines two ideas: (1) *slack-based merging* and (2) merging firings of the same actor. Before introducing the complete algorithm, we will first discuss the idea of slack-based merging. For this purpose, we formalize the definition of slack.

Definition 4.7 (*Slack*). The *slack* of a firing j of actor i, v_{i_j}, is the difference between its latest finish time θ_{i_j} and its earliest start time ϑ_{i_j} minus its computation time C_i. It is defined as follows:

$$\sigma_{i_j} = \theta_{i_j} - \vartheta_{i_j} - C_i. \tag{4}$$

For example, consider two firings v_{i_j} and v_{i_l} of an actor v_i. If v_{i_j} has σ_{i_j} greater than or equal to the computation time of v_{i_l} ($\sigma_{i_j} \geq C_i$) and the reverse ($\sigma_{i_l} \geq C_i$), the algorithm can merge both firings together in one cluster. This strategy allows having a reduced-size graph without elongating the Critical Path (CP) larger than L, satisfying the graphs end-to-end latency constraint. However, this is not the only condition to have a valid merge. Section 4.3 lists all the conditions in detail.

The second strategy aims to merge the firings v_{i_j} of the same actor v_i together in the minimum number of clusters. This helps in generating a reduced-size graph that is suitable for mapping on a message-passing multicore architecture, because the firings v_{i_j} of the same actor v_i will be mapped on the minimum number of cores. This results in a smaller memory footprint on the platform and less communication overhead.

4.3. Valid Merge

In this section, we present the concept of a *valid merge* that is used by the *slack-based merging* algorithm (Section 4.4) to decide whether to accept or reject a merging operation. It is defined as follows:

Definition 4.8 (*Valid Merge*). A *valid merge* is a *safe merge* operation between two firings v_{i_j} and v_{i_l} of the same actor $v_i \in G$, resulting in a new graph G_m that satisfies the following two constraints:

(1) the throughput constraint ζ such that,

$$\zeta_m \geq \zeta; \tag{5}$$

(2) the end-to-end latency constraint L such that,

$$L \geq \sum_{\forall v_{i_j} \in CP \in G_m} C_i. \tag{6}$$

To satisfy the throughput constraint, G_m must fulfil two conditions:

(a) G_m must be live, that is, deadlock-free, defined as follows:

$$\zeta_m \neq 0; \tag{7}$$

(b) the execution time of each cycle $\mathcal{C}_k \in G_m$ and each merged cluster $\mathcal{V}_o \in G_m$ must not exceed the period constraint T, which is equal to the inverse of the throughput constraint ζ, $T = 1/\zeta$. This is defined as follows:

$$(\forall \mathcal{C}_k \in G_m) \wedge (\forall \mathcal{V} \in G_m),\ T \geq \sum_{\forall v_{i_j} \in \mathcal{C}_k} C_i,\ T \geq \sum_{\forall v_{i_j} \in \mathcal{V}_o} C_i. \tag{8}$$

The first condition is satisfied by the *safe merge* operation (Theorem 4.6). It ensures that the merge operation does not create a cycle without an initial token in the generated graph G_m (a deadlock situation). Therefore, we implemented a function that searches for a path between the two firings about to be merged, other than the direct edge connecting them. The function searches for a path that consists of firings connected by edges carrying zero initial tokens (*dependent firings*). If a path is found, then the merge is not valid, because the merging process will create an extra illegal cycle that does not have an initial token and leads to deadlock in the application graph. Otherwise, the graph G_m is live. Consider as an example the scenarios in which we would like to merge the firings v_{b_1} and v_{b_2} of actor v_b in the cases with infinite and finite buffers shown in Figures 1(b) and 2(b), respectively. In the case of infinite buffers, merging the firings v_{b_1} and v_{b_2} satisfies the first condition (*independent firings*), since there is no path between them other than the direct edge (e_{b_1,b_2}), as shown in Figure 1(b). Contrarily, in the case of finite buffers, this merge does not satisfy the first condition (*dependent firings*), because it will create an illegal cycle without an initial token. This is due to the existence of a path between the firings v_{b_1} and v_{b_2} that consists of the firings ($v_{b_1}, v_{c_1}, v_{b_2}$) connected by the edges (e_{b_1,c_1}, e_{c_1,b_2}) that have zero initial tokens, as shown in Figure 2(b). In this case, the merge between (v_{b_1}, v_{b_2}) into a single cluster $\mathcal{V}_{b_1,b_2}$ creates an illegal cycle without an initial token between the cluster $\mathcal{V}_{b_1,b_2}$ and the firing v_{c_1}, which would result in deadlock.

The second condition is ensured by implementing a function that checks that both the execution time of each cycle $\mathcal{C}_k$ and each merged cluster $\mathcal{V}_o$ is not exceeding the application period constraint T. The algorithm identifies all cycles in the application graph and saves them in a lookup table. Each entry in the lookup table contains the cycle and its total execution time. When merging any actor involved in a cycle, the cycle is updated by replacing the actors with the new cluster and calculating the new execution time of the cycle. If the execution time of the cycle exceeds the period of the application, the merge is not valid. Otherwise, the merge is approved. In the case of merged clusters, the algorithm checks the execution time of every merged cluster and guarantees that it does not exceed the application period.

The *slack-based merging* algorithm merges as long as each firing v_{i_j} of every actor $v_i \in G$ has positive slack ($\sigma_{i_j} \geq 0$). This means that the execution time of the critical path of the application cannot exceed the application end-to-end latency constraint L. This guarantees that the second constraint is satisfied.

4.4. Slack-Based Merging Algorithm

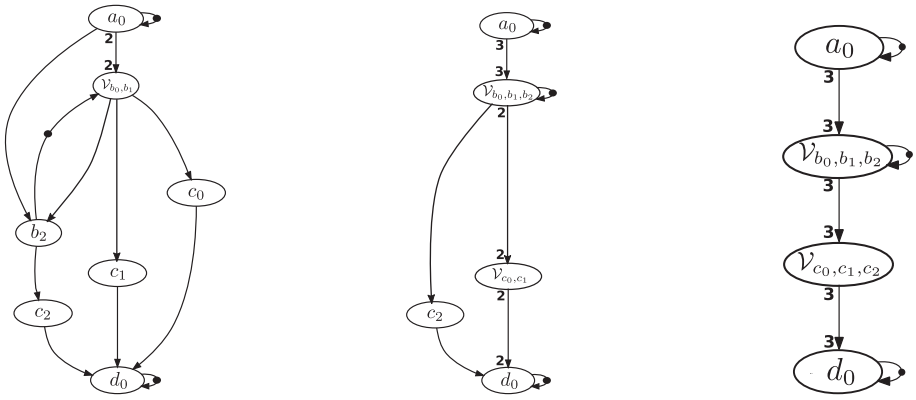
The *slack-based merging* algorithm, shown in Algorithm 1, aims to generate a simpler, smaller size graph G_m that reduces the runtime of its analysis. The proposed algorithm starts by calculating the *earliest start time* ϑ_{i_j} and the *latest finish time* θ_{i_j} for each firing v_{i_j} in the SDF graph G using Equations (2) and (3), respectively. Then, it computes the slack σ_{i_j} for each firing using Equation (4). If all the firings v_{i_j} in G have slack σ_{i_j} greater than or equal to zero ($\forall v_{i_j} \in G, \sigma_{i_j} \geq 0$), a merging operation can possibly be applied. Otherwise, the merging algorithm terminates. When all firings have positive slack, the algorithm needs to determine which firings to merge. An optimal algorithm would try all possible combinations of firings from the same actor, for each actor,

ALGORITHM 1: *Slack-Based Merging* Algorithm

Input:
G: SDF application graph, $G = \langle V, E, d \rangle$.
Output:
G_m: merged HSDF application graph.
Variables:
n: number of actors in G.
V: set of SDF actors, $V = \{v_1, v_2, \ldots, v_n\}$.
$\hat{V}$: breadth-first topologically ordered set of actors.
$\vec{q}$: repetition vector for G, $\vec{q} = \{q_1, q_2, \ldots, q_n\}$, where q_i is the corresponding number of firings of v_i.
v_{ij}: is the j^{th} firing of actor v_i, where $\{j : j \in \mathbb{Z}, j \in [1, q_i]\}$.
G_{hsdf}: HSDF graph representation of G, where $G_{hsdf} = \langle V_h, E_h, d \rangle$ and $v_{ij} \in V_h$.

begin
  Convert G to G_{hsdf}.
  Calculate ϑ_{ij}, $\{\vartheta_{ij} : \forall v_{ij} \in G, \textit{Equation}\ (2)\}$.
  Calculate θ_{ij}, $\{\theta_{ij} : \forall v_{ij} \in G, \textit{Equation}\ (3)\}$.
  $\{\sigma_{ij} : \forall v_{ij} \in G, \sigma_{ij} = \theta_{ij} - \vartheta_{ij} - C_i\}$.
  $G_m = G$.
  if $(\forall v_{ij} \in G_m, \sigma_{ij} \geq 0)$ **then**
    foreach v_i **in** $\hat{V}$ **do**
      $\{v_{ij},\ v_{il} : j \neq l, \sigma_{ij} \geq C_i \textit{ and } \sigma_{il} \geq C_i\}$.
      if $(\textit{valid\_merge}(v_{ij},\ v_{il}))$ **then**
        merge v_{ij} and v_{il}.
        Calculate ϑ_{ij}, $\{\vartheta_{ij} : \forall v_{ij} \in G_m, \textit{Equation}\ (2)\}$.
        Calculate θ_{ij}, $\{\theta_{ij} : \forall v_{ij} \in G_m, \textit{Equation}\ (3)\}$.
        $\{\sigma_{ij} : \forall v_{ij} \in G_m, \sigma_{ij} = \theta_{ij} - \vartheta_{ij} - C_i\}$.
        if $(\forall v_{ij} \in G_m, \sigma_{ij} \geq 0)$ **then**
          $G = G_m$
        else
          $G_m = G$
        end
      else
        `// No Merge`
      end
    end
  else
    `// Stop Merge`
  end
end

although this approach does not scale to applications of realistic complexity. Instead, our heuristic algorithm picks the actors v_i in sequence from the topologically ordered set $\hat{V}$ to begin merging different firings. This particular way of selection of firings to be merged is not formally proven to be better than others, but we have experimentally determined that it works rather well. For each actor v_i, the algorithm tries each possible combination of two firings $(v_{ij},\ v_{il})$ for merging, such that $\sigma_{ij} \geq C_i$ and $\sigma_{il} \geq C_i$, and generates a new graph G_m. After merging them, the algorithm checks the validity of the merging operation of $(v_{ij},\ v_{il})$ using the *valid_merge()* function previously explained in Section 4.3. If all the conditions of a *valid merge* are satisfied, the merge operation is valid. Otherwise, the algorithm will undo the last merging operation and pick up two new firings for merging.



(a) Merging of v_{b_0} and v_{b_1} (b) Merging of v_{c_0} and v_{c_1} (c) G_m

Fig. 4. Example of slack-based merging.

When the merge operation is considered a *valid merge*, the algorithm recalculates the *earliest start time* ϑ_{i_j}, the *latest finish time* θ_{i_j}, and the slack σ_{i_j} for each firing v_{i_j} in the new output merged graph G_m. If the slack of all firings in the G_m are greater than or equal to zero ($\forall v_{i_j} \in G_m, \sigma_{i_j} \geq 0$), the merge operation of (v_{i_j}, v_{i_l}) is approved and the algorithm continues to try merging different firings. Otherwise, the algorithm will undo the last merging operation and move forward by picking up two new firings for merging. The algorithm iterates until no possible merges can be done. Reaching that stage, it generates a new small size compact HSDF graph G_m that reduces the analysis time, as shown in Section 5.

4.5. Complexity Analysis

In this section, we provide a complexity analysis for the slack-based merging algorithm, previously presented in Algorithm 1. The algorithm starts by calculating earliest start time ϑ_{i_j} and latest finish time θ_{i_j} of all firings, each having a complexity of $O(|V_h|+|E_h|)$, since they are based on a Breadth First Search (BFS) [Lynch 1996]. Then, it continues with the calculation of the slack σ_{i_j}, which has a complexity of $O(|V_h|)$. The next part of the algorithm is a loop (**foreach** statement) that runs $|V_h|$ times (in the worst case) and contains earliest start time ϑ_{i_j}, latest finish time θ_{i_j}, and slack σ_{i_j} calculations, with the previously stated complexities. Therefore, the complexity of the loop is equivalent to $O(|V_h| \cdot ((|V_h| + |E_h|) + (|V_h| + |E_h|) + (|V_h|))) = O(3|V_h|^2 + 2|V_h||E_h|)$. Hence, the final complexity of the *slack-based merging* algorithm is $O(|V_h|^2 + |V_h||E_h|)$, which is polynomial and depends on both $|V_h|$ and $|E_h|$.

4.6. Example

In this section, we present an example that illustrates how to apply the *slack-based merging* algorithm on an SDF/HSDF graph, shown in Figure 1, until reaching the reduced-size HSDF graph G_m, shown in Figure 4(c). Here, we demonstrate the algorithm for a single iteration for brevity, because it is a repeated process and it takes several iterations to reach the final output graph G_m. The following paragraphs explain this in detail.

Consider the SDF graph and its HSDF representation shown in Figure 1. Let us assume all the execution times of all actors equal to 1, the throughput requirement $\zeta = {}^1/_3$, and the end-to-end latency constraint $L = 8$. The period T of this graph is equal to 3 and the total execution time of its CP $(v_{a_0}, v_{b_0}, v_{b_1}, v_{b_2}, v_{c_2}, v_{d_0})$ is equal to 6. Calculating

Table I. SDF^3 Benchmark Applications

Application	Number of Actors	Number of Channels	
		Infinite Buffer	Finite Buffer
h263decoder	1,190	2,378	4,160
h263encoder	201	399	785
modem	48	109	170
samplerate	612	1,633	2,654
satellite	4,515	11,619	18,723
mp3playback	10,000	32,237	32,237

$\langle \vartheta_{i_j}, \theta_{i_j}, \sigma_{i_j} \rangle$ for every firing v_{i_j} in the graph results in $v_{a_0} = \langle 0, 3, 2 \rangle$, $v_{b_0} = \langle 1, 4, 2 \rangle$, $v_{b_1} = \langle 2, 5, 2 \rangle$, $v_{b_2} = \langle 3, 6, 2 \rangle$, $v_{c_0} = \langle 2, 7, 4 \rangle$, $v_{c_1} = \langle 3, 7, 3 \rangle$, $v_{c_2} = \langle 4, 7, 2 \rangle$, $v_{d_0} = \langle 5, 8, 2 \rangle$. As we see, every firing v_{i_j} has positive slack σ_{i_j}, which allows going forward in the merging process. From Figure 1(a), we can get the topologically ordered set $\hat{V} = \{v_a, v_b, v_c, v_d\}$. The algorithm will skip actor v_a and move on to actor v_b, because v_a consists of a single firing v_{a_0}. It picks up the two firings (v_{b_0}, v_{b_1}), because they have positive slack that satisfy the two conditions $\sigma_{b_0} \geq C_b$ and $\sigma_{b_1} \geq C_b$. Then, it merges them into a single cluster $\mathcal{V}_{b_0,b_1}$ with execution time $C_{b_0,b_1} = 2$, as shown in Figure 4(a). This merging operation is a valid merge, because it satisfies the throughput ζ and the end-to-end latency L constraints defined by Equations (5) and (6), respectively. The throughput constraint ζ is satisfied, because the total execution time of the maximum cycle in the graph ($\mathcal{V}_{b_0,b_1}, v_{b_2}$) is equal to 3, which means that ζ_m of the resulting graph, shown in Figure 4(a), did not change ($\zeta_m = {}^1/_3$). Also, the end-to-end latency L constraint is satisfied, because the total execution time of the CP of the resulting graph did not change (equal to 6). Then, the algorithm recalculates $\langle \vartheta_{i_j}, \theta_{i_j}, \sigma_{i_j} \rangle$ for every firing v_{i_j} and repeats the process again. Figure 4(b) shows the output of a late step of the merging algorithm, while Figure 4(c) shows the final output HSDF graph G_m of the merging algorithm.

The final output HSDF graph G_m consists of four actors ($v_{a_0}, \mathcal{V}_{b_0,b_1,b_2}, \mathcal{V}_{c_0,c_1,c_2}, v_{d_0}$) with execution times (1, 3, 3, 1), respectively. Its throughput ζ_m is equal to ${}^1/_3$, while the total execution time of its CP ($v_{a_0}, \mathcal{V}_{b_0,b_1,b_2}, \mathcal{V}_{c_0,c_1,c_2}, v_{d_0}$) is equal to 8. Therefore, G_m satisfies the throughput ζ and the end-to-end latency L constraints of the original SDF/HSDF graph. As we see, G_m has a single path ($v_{a_0}, \mathcal{V}_{b_0,b_1,b_2}, \mathcal{V}_{c_0,c_1,c_2}, v_{d_0}$) compared to the original HSDF graph, shown in Figure 1(b). This speeds up the timing parameter extraction process since it depends on the number of paths that exist in the graph. We experimentally demonstrate this in Section 5.

5. EVALUATION AND RESULTS

In this section, we evaluate the slack-based merging algorithm using two experiments. The first experiment evaluates the runtime of the algorithm with several applications and the effect of different buffer sizes on the performance of the algorithm. The second experiment measures the runtime of the Timing Parameter Extraction (TPE) algorithm proposed in Ali et al. [2015] with merged and nonmerged graphs as inputs and compares their runtimes. Also, it shows the effect of changing the application throughput constraint and the buffer sizes on the size of the merged output graph.

5.1. Evaluation of Slack-Based Merging

In this experiment, we evaluated our proposed algorithm on SDF applications from the SDF^3 benchmarks [Stuijk et al. 2006], shown in Table I. The main goal is to evaluate its runtime with SDF graphs of different sizes, but also to show the impact of different buffer sizes on the performance of the slack-based merging algorithm. The buffer sizes

Table II. Runtime (Seconds) of the Algorithm

Application	Runtime (sec)		
	Infinite Buffer Sizes	Minimum Buffer Sizes	
		ζ_{max}	ζ_{min}
h263decoder	264	495	11,824
h263encoder	0.55	8.9	11.13
modem	0.215	0.47	0.65
samplerate	38	51	53
satellite	14,390	20,917	26,334
mp3playback	5 (days)	∞	∞

Table III. Number of Actors Before and After Merging

Application	Number of Actors			
	Before Merge	After Merge		
		Infinite Buffer Sizes	Minimum Buffer Sizes	
			ζ_{max}	ζ_{min}
h263decoder	1,190	4	71	300
h263encoder	201	5	11	181
modem	48	16	31	31
samplerate	612	6	127	263
satellite	4,515	22	988	1,972
mp3playback	10,000	5,000	N/A	N/A

used in this experiment are infinite buffers, minimum buffers for maximum throughput ζ_{max}, and minimum buffers for minimum throughput ζ_{min}. The throughput constraint ζ for the input applications is set to the minimum ζ_{min}, $\zeta = \zeta_{min}$, while their latency constraint L is set to the inverse of their throughput constraint, $L = 1/\zeta = 1/\zeta_{min}$. This choice is made to provide enough slack for the applications while we study the effect of changing other parameters, that is, throughput and buffer sizes, as shown in the experiment.

Tables II and III show the summary of the results. In most cases, the algorithm succeeds in generating a reduced-size graph in reasonable time. However, for some cases, for example, *mp3playback*, the runtime varies from seconds to days depending on the complexity of the graph. This result is in line with our expectations because the original graph before merge is huge and consists of 10,000 firings. The algorithm achieves large reduction rates, as shown in Table III, ranging from 2× in the case of *mp3playback* up to 300× (approximately) in the case of *h263decoder*, in the case of infinite buffers. In the case of finite buffers, the reduction rates are less compared to the infinite case. It ranges from 2× up to 17× (approximately) depending on the buffer sizes and the throughput constraint. Also, we notice that the slack-based merging algorithm's runtime and output graph size have an inverse relation with the buffer size of the application. The reason is that small buffer sizes add extra dependencies in the graph that prevent further merging and makes the algorithm spend more time exploring every combination of actors that could be merged. The ∞ and N/A entries imply that the merging algorithm spends unreasonable time (>1 week) without generating any output.

From these results, we can conclude that the slack-based merging algorithm typically succeeds in achieving large reduction rates in the size of the output graphs. This result reflects positively on the TPE algorithm, as shown in the next experiment.

5.2. Evaluation of TPE Performance using Merged Graphs

In Ali et al. [2015], an algorithm was proposed for TPE of HSDF applications, enabling them to be scheduled and analysed using traditional real-time analysis techniques. This algorithm requires conversion from an SDF graph to an HSDF graph, which may result in large graphs and hence long runtimes of the algorithm. In the previous experiment, we evaluated the runtime of the slack-based merging algorithm for different applications and buffer sizes. The current experiment evaluates the runtime of the TPE algorithm with HSDF graphs obtained using the classical conversion algorithm from Sriram and Bhattacharyya [2000] (G_{hsdf}) and the slack-based merging algorithm (G_m) proposed in this article. This experiment will show that spending this extra time running the merging algorithm to generate a graph G_m typically results in a reduction in the runtime of the TPE algorithm, thereby reducing the overall runtime of the complete process.

5.2.1. Experimental Setup. This experiment uses the same settings as the previous one. We change the throughput of the tested applications from the minimum throughput constraint (denoted by 0%) to the maximum throughput (denoted by 100%) in a stepwise fashion in increments of 20%. The latency L of each application is set to the inverse of the minimum throughput constraint of the application, $L = 1/\zeta_{min}$. At each throughput step, we apply our merging algorithm on G to generate a reduced-size HSDF graph G_m. Then, both types of graphs (G_{hsdf} and G_m) are provided as inputs to the TPE algorithm to compare and record their runtime.

5.2.2. Experimental Results. The experiment is on applications with two types of buffer sizes: infinite buffers and minimum buffers for maximum throughput (finite buffers). In the case of applications with infinite buffers, the results show that the proposed algorithm succeeds in generating a reduced-size compact graph G_m at the maximum throughput (100%) in most of the cases, as shown in Figures 6(a), 7(a) and 8(a). This is reflected in the large speed-up in the runtime of slack-based merging added to the TPE algorithm compared to the runtime of the TPE algorithm on the original G_{hsdf} graphs, as shown in Figures 6(b), 7(b), and 8(b). Also, the results show that having a reduced-size graph G_m at the maximum throughput is not always possible in the case of infinite buffers. The *h263encoder* application results, shown in Figure 5, show that there are cases where the ability to generate a reduced-size graph decreases with increasing application throughput (see Figure 5(a)). This is natural, because a higher throughput requirement restricts the ability to merge parallel firings, which results in larger output graphs. This is reflected in the increase in the total runtime of slack-based merging and TPE algorithm following the increase in throughput constraint due to the larger G_m graph size, as shown in Figure 5(b).

In the case of applications with minimum buffers for maximum throughput (finite buffers), the results show that when the throughput constraint is relaxed with respect to the maximum throughput of the application, the proposed algorithm is able to achieve a larger reduction in the application graph size, as shown in Figures 5(a), 6(a), 7(a), and 8(a). This significantly reduces the total runtime of slack-based merging and the TPE algorithm at relaxed throughput constraints. This effect gradually decreases when approaching the maximum throughput of the graph, as shown in Figures 5(b), 6(b), 7(b), and 8(b). Moreover, in some finite buffer cases, that is, *h263encoder* and *h263decoder*, when approaching the maximum throughput, the total runtime of slack-based merging and the TPE algorithm exceeds the runtime of applying TPE directly on G_{hsdf}, as shown in Figures 5(b) and 6(b). This is due to the increase in the throughput constraint that decreases the ability of merging parallel firings. Also, the minimum buffers introduce more dependencies in the graph compared

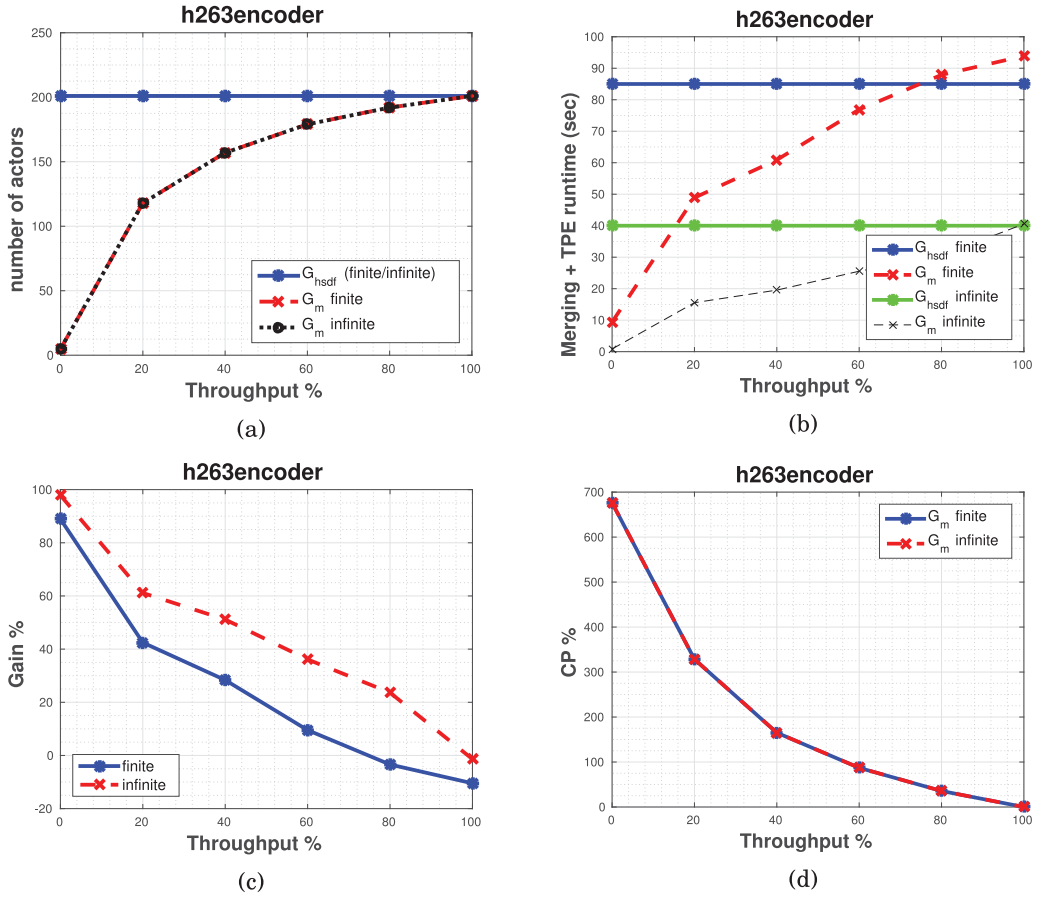


Fig. 5. *h263encoder* results.

to the infinite buffer case, which reduces the ability to achieve a large reduction in the graph size. For the *mp3playback*, the output graph G_m takes infinite time for extracting its timing parameters. This is due to the fact that the size of the output graph G_m is still huge (5,000 actors), although it has been reduced to 50% of its size.

Figures 5(c), 6(d), 7(c), and 8(c) are derived from Figures 5(b), 6(b), 7(b), and 8(b), respectively. They show the amount of gain in reducing the overall design time in percentage in the cases of finite and infinite buffers. Figures 5(d), 6(d), 7(d), and 8(d) show a decrease in the percentage of the total execution time of the CP of the applications (0% means an execution time of CP is equal to the CP of G_{hsdf}) with the increase of the throughput constraint for a fixed end-to-end latency constraint L. This means that the remaining slack (after generating the reduced-size graph G_m) increases along with the increase in the throughput constraint. The interpretation of this phenomena is, when the throughput constraint increases, a merging decision could be rejected despite the availability of enough slack, because it could result in a violation of the throughput constraint by increasing the period of the application. This conforms with the previous result, which states that the increase in the throughput constraint limits the ability of merging parallel firings.

From these results, we can conclude that our merging algorithm typically succeeds in generating reduced-size graphs, in particular for applications that do not need to

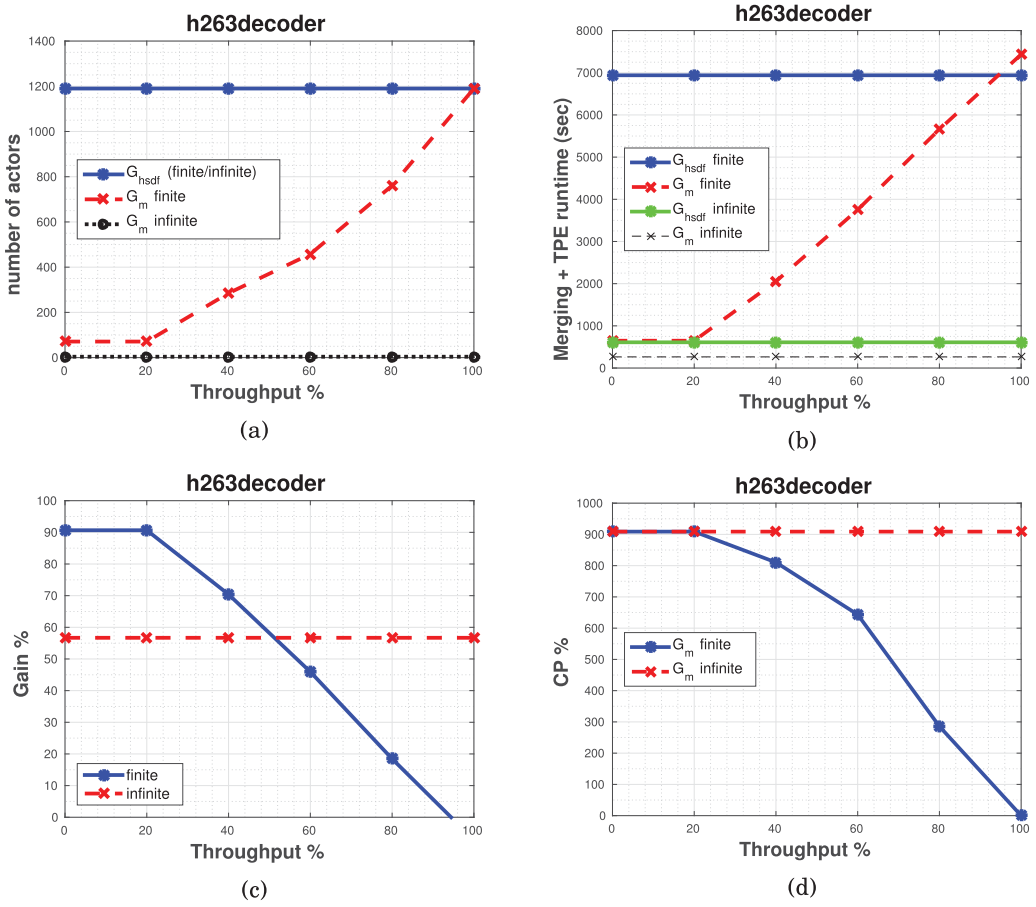


Fig. 6. *h263decoder* results.

execute at maximum throughput, which helps in speeding up the derivation of the timing parameters.

5.3. Evaluation of Quality

The previous experiments have shown the capability of our heuristic to generate reduced-size graphs that satisfy the timing constraints of the application and speed up the timing parameter extraction process, thereby reducing the overall design time of the real-time system. However, it does not address the quality of our solution. In this experiment, we assess the quality of our heuristic algorithm by determining how far it is from the optimal solution. Also, to take the opportunity to open a discussion about the trade-off between getting the optimal solution and runtime overhead added to the overall design time of the real-time system.

To evaluate the quality of our heuristic algorithm, we have to define optimality in the context of our problem. Based on that definition, we can determine a method to obtain it, then compare both solutions. *The optimal solution is defined as the reduced-size graph that satisfies the timing constraints and minimizes the TPE runtime*. To obtain this solution, we implemented an exhaustive enumerative algorithm that searches the solution space of all possible combinations of merging operations. These merging

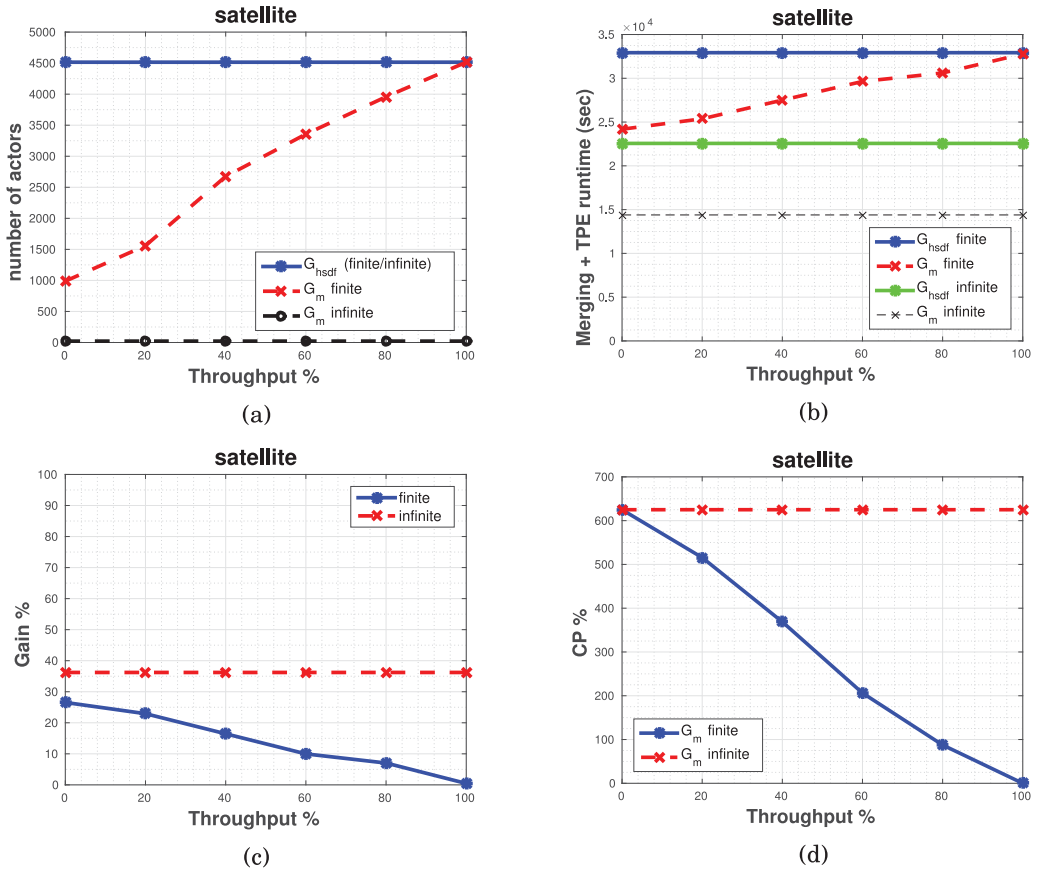


Fig. 7. *satellite* results.

operations are *valid merges* (defined in Section 4.3) constrained by our heuristic *merging strategy* (merging firings of the same actor, defined in Section 4.2) for a fair comparison with the heuristic algorithm.

We have implemented a tool that incorporates an exhaustive enumerative algorithm that searches for the optimal solution. However, such an algorithm cannot scale beyond small synthetic examples due to its exhaustive nature. For example, assuming enough slack, if we applied that on the smallest size dataflow graph *modem* used in our experiments, which has 48 firings (two actors have 16 firings each, two actors have 2 firings each, and the rest have one firing each). This means that the exhaustive enumerative algorithm has to investigate a solution space of order 10^{27} merging trials to find the optimal solution. This requires a massive runtime compared to our proposed heuristic and creates a large overhead on the overall design time. This because our proposed heuristic stands by a merging operation once it is valid without change terminating the algorithm very quickly. However, the exhaustive enumerative algorithm will try every possible merging combination to reach the optimal solution. This limitation is the main reason for selecting small synthetic SDF graphs as an input for the experiment.

We set up an experiment that randomly generates an input set of 100 small synthetic SDF graphs using the SDF^3 benchmark *generate* tool. From our experimental experience, each SDF graph can have a maximum of 12 firings in total, by controlling

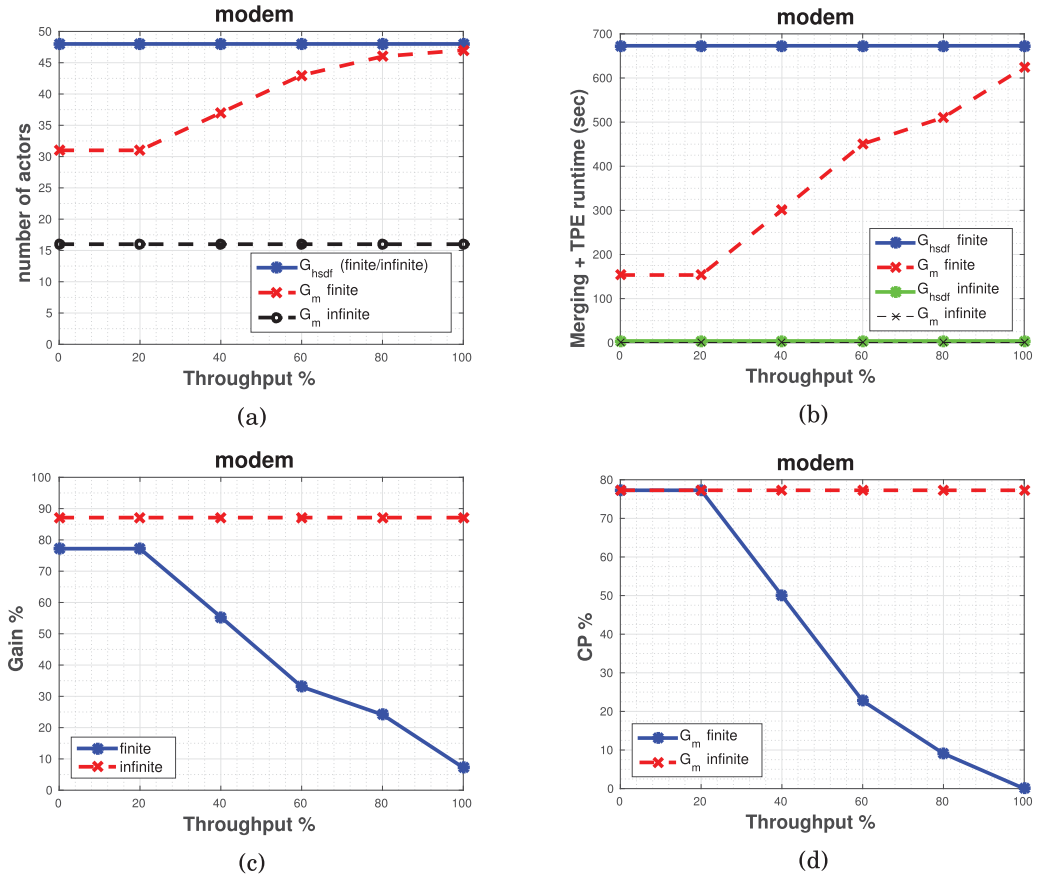


Fig. 8. *modem* results.

the value of the parameter *repetitionVectorSum* in the settings file passed to the *generate* tool. Larger graphs result in an exponential explosion in the solution space preventing the algorithm from terminating in reasonable time. For each graph, the tool investigates all possible combinations of valid merges searching for the reduced-size graph with minimum TPE runtime using the exhaustive enumerative algorithm. Also, the experiment applies the heuristic slack-based merging algorithm on the same input set to enable comparison of the final results. The experiment investigates both heuristic and optimal solutions at different throughput constraints, as described in the experimental setup in Section 5.2.1.

Figure 9 summarizes the results regarding the quality of our heuristic algorithm. At relaxed throughput constraints, the heuristic-based solution is exactly the same as the optimal solution. As we notice, the average TPE runtime per graph is the same for both solutions up to 40% of the maximum throughput constraint. The reason for this is the availability of enough slack that allows merging as much firings without violating the latency constraint L. Also, an equally important reason is the relaxed throughput constraint ζ, which means a large period for the application. This allows merging of parallel firings without violating the period of the application, as demonstrated in the previous experiment. Once the throughput constraint becomes tighter (>40%), a slight deviation of maximum 10% in average TPE runtime appears between both solutions.

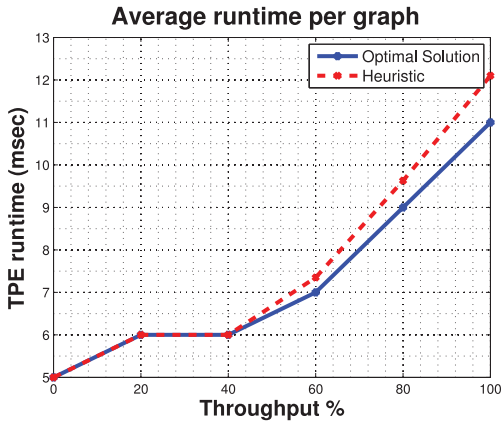


Fig. 9. Exhaustive enumeration vs. heuristic algorithm.

This is because the tighter throughput constraints means a tighter period for the application. This limits the merging ability of parallel firings despite the availability of enough slack, as detailed in the previous experiment. Therefore, the final solution our heuristic reaches depends on its first merging decision.

6. CONCLUSIONS

In this work, we presented a new heuristic reduction algorithm for synchronous dataflow graphs called slack-based merging. The proposed algorithm generates reduced-size HSDF graphs that satisfy the throughput and latency constraints of the original application graph. The generated reduced-size graphs typically enable faster extraction of timing parameters and finding a feasible real-time schedule compared to using the original larger HSDF graphs. Moreover, the experimental results with real application models from the SDF^3 benchmark show that when the throughput constraint is relaxed with respect to the maximal throughput of the application graph, the merging algorithm is able to achieve a larger reduction in graph size and hence a larger speedup in algorithms for extracting timing parameters. Also, we showed that the slack-based merging heuristic gives results that are near optimal with a maximum deviation of 10% and small overhead compared to an exhaustive optimal approach for small graphs.

REFERENCES

Hazem Ismail Ali, Luís Miguel Pinho, and Benny Akesson. 2013. Critical-path-first based allocation of real-time streaming applications on 2D mesh-type multi-cores. In *Proceedings of the 2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 201–208. DOI:http://dx.doi.org/10.1109/RTCSA.2013.6732220

Hazem Ismail Ali, Luís Miguel Pinho, and Benny Akesson. 2015. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Proceedings of the 2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 701–710. DOI:http://dx.doi.org/10.1109/PDP.2015.57

Mohamed Bamakhrama and Todor Stefanov. 2011. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT'11)*. ACM, New York, NY, 195–204. DOI:http://dx.doi.org/10.1145/2038642.2038672

Mohamed Bamakhrama and Todor Stefanov. 2012. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12)*. ACM, New York, NY, 83–92. DOI:http://dx.doi.org/10.1145/2380445.2380464

Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, JanDavid Mol, Sander Stuijk, Valentin Gheorghita, and Jef van Meerbergen. 2005. Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and Robust Streaming In and Between Connected Consumer-Electronic Devices*, Peter Stok (Ed.). Philips Research Book Series, Vol. 3. Springer, Netherlands, 81–108. http://dx.doi.org/10.1007/1-4020-3454-7_4

Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. 1999. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 21, 2 (1999), 151–166. DOI:http://dx.doi.org/10.1023/A:1008052406396

Adnan Bouakaz, Jean-Pierre Talpin, and Jan Vitek. 2012. Affine data-flow graphs for the synthesis of hard real-time applications. In *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design (ACSD'12)*. IEEE Computer Society, Washington, DC, 183–192. DOI:http://dx.doi.org/10.1109/ACSD.2012.16

Morteza Damavandpeyma, Sander Stuijk, Marc Geilen, Twan Basten, and Henk Corporaal. 2012. Parametric throughput analysis of scenario-aware dataflow graphs. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD)*. 219–226. DOI:http://dx.doi.org/10.1109/ICCD.2012.6378644

Benot Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, Franois Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frdric Riss, and Thierry Strudel. 2013. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of the 2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. DOI:http://dx.doi.org/10.1109/HPEC.2013.6670342

Marc Geilen. 2009. Reduction techniques for synchronous dataflow graphs. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC'09)*. 911–916.

Amir Hossein Ghamarian, Marc Geilen, Twan Basten, and Sander Stuijk. 2008. Parametric throughput analysis of synchronous data flow graphs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. 116–121. DOI:http://dx.doi.org/10.1109/DATE.2008.4484672

Amir Hossein Ghamarian, Marc Geilen, Twan Basten, Bart D. Theelen, Mohammad Reza Mousavi, and Sander Stuijk. 2006. Liveness and boundedness of synchronous data flow graphs. In *Formal Methods in Computer Aided Design (FMCAD'06)*. 68–75. DOI:http://dx.doi.org/10.1109/FMCAD.2006.20

Amir Hossein Ghamarian, Sander Stuijk, Twan Basten, Marc Geilen, and Bart D. Theelen. 2007. Latency minimization for synchronous data flow graphs. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD'07)*. 189–196. DOI:http://dx.doi.org/10.1109/DSD.2007.4341468

Joost P. H. M. Hausmans, Maarten H. Wiggers, Stefan J. Geuns, and Marco J. G. Bekooij. 2013. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (M-SCOPES'13)*. ACM, New York, NY, 13–22. DOI:http://dx.doi.org/10.1145/2463596.2463603

Minsoo Kim, Joonho Song, Dohyung Kim, and Shihwa Lee. 2010. H.264 decoder on embedded dual core with dynamically load-balanced functional paritioning. In *Proceedings of the 2010 17th IEEE International Conference on Image Processing (ICIP)*. 3749–3752. DOI:http://dx.doi.org/10.1109/ICIP.2010.5653439

Edward A. Lee. 1991. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems* 2, 2 (April 1991), 223–235. DOI:http://dx.doi.org/10.1109/71.89067

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of the IEEE* 75, 9 (Sept. 1987), 1235–1245. DOI:http://dx.doi.org/10.1109/PROC.1987.13876

Giuseppe Lipari and Enrico Bini. 2011. On the problem of allocating multicore resources to real-time task pipelines. In *Proceedings of the 2011 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*.

Di Liu, Jelena Spasic, Jiali Teddy Zhai, Todor Stefanov, and Gang Chen. 2014. Resource optimization for CSDF-modeled streaming applications with latency constraints. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. 1–6. DOI:http://dx.doi.org/10.7873/DATE.2014.201

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Orlando Moreira, Frederico Valente, and Marco Bekooij. 2007. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM/EEE International Conference on Embedded Software (EMSOFT'07)*. ACM, New York, NY, 57–66. DOI:http://dx.doi.org/10.1145/1289927.1289941

Victor Pankratius, Ali Jannesari, and Walter F. Tichy. 2009. Parallelizing Bzip2: A case study in multicore software engineering. *IEEE Software* 26, 6 (Nov. 2009), 70–77. DOI:http://dx.doi.org/10.1109/MS.2009.183

Luís Miguel Pinho, Eduardo Quinones, Marko Bertogna, Andrea Marongiu, Jorge Pereira Carlos, Claudio Scordino, and Michele Ramponi. 2014. P-SOCRATES: A parallel software framework for time-critical many-core systems. In *Proceedings of the 2014 17th Euromicro Conference on Digital System Design (DSD)*. 214–221. DOI: http://dx.doi.org/10.1109/DSD.2014.94

Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. 2003. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. ACM, New York, NY, 63–72. DOI: http://dx.doi.org/10.1145/951710.951721

Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. 2013. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS'13)*. ACM, New York, NY, 287–296. DOI: http://dx.doi.org/10.1145/2516821.2516836

Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2011. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium (RTS'11)*, 217–226. DOI: http://dx.doi.org/10.1109/RTSS.2011.27

Sundararajan Sriram and Shuvra S. Bhattacharyya. 2000. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006. SDF^3: SDF for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06)*. 276–278. DOI: http://dx.doi.org/ 10.1109/ACSD.2006.23

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem-Overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems* 7, 3, Article 36 (May 2008), 53 pages. DOI: http://dx.doi.org/10.1145/1347375.1347389

Received September 2015; revised May 2016; accepted June 2016