Technical Report

MASES: Mobility And Slack Enhanced Scheduler For Synchronous Dataflow Graphs

Wenxiao Yu and Andreas Gerstlauer

UT-CERC-15-01

May 8, 2015

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

201 E. 24th St., Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967

http://www.cerc.utexas.edu

THE UNIVERSITY OF TEXAS AT AUSTIN
Electrical and Computer Engineering

Abstract

MASES: Mobility And Slack Enhanced Scheduler For Synchronous Dataflow Graphs

Wenxiao Yu, Andreas Gerstlauer

The University of Texas at Austin, 2015

Nowadays, real-time streaming and digital signal processing applications create an increased demand for embedded systems with better capability to process large-volume data streams with low latency and large throughput. Synchronous dataflow (SDF) graphs allow for static analysis and optimization, and are widely used for modeling real-time streaming applications. To reach a better performance, mapping of SDF descriptions into tightly resource-constrained real-time implementations requires optimization of pipelined scheduling of tasks on different processing elements (PEs). This poses the problem of finding the optimal solution across a multi-dimensional latency-throughput-area objective space.

In this report, we address the problem of pipelined scheduling of SDF graphs on heterogeneous multi-processor platforms. Integer linear programming (ILP) models have been applied to solve this problem, providing theoretically optimal results. However, the execution time of solving an ILP model is exponential in the size of the input SDF graph, limiting the usage of ILPs. By contrast, list scheduling heuristics have been proposed to solve the pipelined scheduling problem in polynomial time, but existing approaches do not guarantee the optimality of the result and fail to find a valid schedule when the period

constraint of the SDF graph is tight. This report contributes a heuristic called MASES that improves the performance of list scheduling. MASES explores the flexibility in a partial schedule by moving already-placed actors on the timeline such that enough space is created for actors placed next. Different from heuristics based on backtracking, MASES finds a valid actor assignment without the need for un- and re-scheduling of already-placed actors. In contrast to backtracking heuristics, MASES guarantees to find a valid schedule if one exists. In our experiments with randomly generated SDF graphs of varying size, MASES was able to find valid schedules for all test cases whereas backtracking only solved 47.2% of cases on average. Furthermore, for test cases that succeeded for MASES and backtracking, MASES on average reduces execution time by 11% and increases latency by 11% compared to backtracking.

Table of Contents

Abstract ii

List of Tables v

List of Figures vi

Chapter 1 Introduction 1

1.1 Synchronous Dataflow Graph 1

1.2 Pipelined Scheduling 2

1.3 Report Outline 5

Chapter 2 Related Work 6

2.1 ILP Model 6

2.2 List Scheduling 7

2.2 Backtracking 9

Chapter 3 Mobility And Slack Enhanced Scheduling 12

3.1 Mobility Analysis 13

3.2 Slack Analysis 20

3.3 Gap Selection 25

3.4 MASES Algorithm 33

Chapter 4 Experiments And Results 34

4.1 Random SDF graphs 34

4.2 H.263 decoder 39

Chapter 5 Summary 41

Acknowledgements 42

References 43

List of Tables

Table 4.1: Improvement of MASES over backtracking 38

List of Figures

Figure 1.1: Synchronous Dataflow Graph 3

Figure 1.2: Modulo Reservation Table for SDF graph 3

Figure 2.1: Flow chart of list scheduler 8

Figure 2.2: Flow chart of list scheduler with backtracking 10

Figure 2.3: Actor F cannot be scheduled 11

Figure 2.4: Actor A and its successors are unscheduled 11

Figure 2.5: Unscheduled actors are put back 11

Figure 3.1: Actor F cannot be scheduled on PE0 12

Figure 3.2: MRT is adjusted and actor F founds its place 13

Figure 3.3: Example SDF graph 14

Figure 3.4: Mobility analysis 15

Figure 3.5: Example SDF graph 15

Figure 3.6: Actor C is blocked by E and cannot be moved 16

Figure 3.7: Example SDF graph 16

Figure 3.8: Actor G should not be moved 17

Figure 3.9: Flow chart of mobility computation 18

Figure 3.10: Flow chart of recursive Mob(X) function 19

Figure 3.11: Example SDF graph 21

Figure 3.12: Slack analysis 21

Figure 3.13: Example SDF graph 22

Figure 3.14: Actor G should not be moved 23

Figure 3.15: Flow chart of slack computation 23

Figure 3.16: Flow chart of recursive Slack(X) function 24

Figure 3.17: Example SDF graph 26
Figure 3.18: Squeezing Gap(C,E) 26
Figure 3.19: Squeezing Gap(A,C) 27
Figure 3.20: Example SDF graph 27
Figure 3.21: Precedential violation caused by squeezing 28
Figure 3.22: Squeezing Gap(E,G) will not cause precedential violation 28
Figure 3.23: Flow chart of gap selection 30
Figure 3.24: Flow chart of list scheduler with MASES 32
Figure 4.1: Size of graph vs. number of tests succeeded for backtracking 35
Figure 4.2: Execution time of backtracking and MASES 36
Figure 4.3: Comparison of normalized latency 37
Figure 4.4: H.263 decoder 49
Figure 4.5: MRT of pipelined schedule of H.263 decoder 40
Figure 4.6: Result schedule generated by MASES 40

CHAPTER 1: INTRODUCTION

Real-time streaming applications are characterized by a need to process large-volume data streams with low latency and high throughput. Such applications are natively non-terminating and can be modeled and analyzed using synchronous dataflow (SDF) graphs [1,2]. Given the current trend towards heterogeneous multiprocessor systems-on-chips (MPSoC) platforms, the design process for implementing these applications requires processing elements (PEs) and time resources to be allocated to actors by a scheduler. Since scheduling patterns determine the throughput and latency of executing an SDF graph, the requirement of optimal performance, including high throughput and low latency, presents a big challenge to the design of scheduling algorithms.

1.1 SYNCHRONOUS DATAFLOW GRAPH

Synchronous data flow (SDF) is a model of computation (MoC) consisting of synchronous actors and directed edges. An actor can be invoked when there are enough input tokens on its input edges, and after the actor finishes its execution, it will put tokens on its output edges. Edges are represented as FIFO channels between actors. The number of tokens consumed and produced by every actor is constant, and will not change during the execution [4,5]. This characteristic allows us to perform static analysis of SDF graphs to compute the number of repetitive execution times of every actor during an iteration and optimize resource allocation. For these reasons, SDF graphs have been widely used to model applications with fixed data production and consumption rates [1].

Homogeneous synchronous data flow (HSDF) is a subset of SDF, where every actor consumes one token and produces one token per firing, and each actor will fire once

per iteration. Every SDF graph can be transformed into an equivalent HSDF graph using the algorithm described in [8]. In this report, the default example SDF graphs used for illustrative purposes are HSDF graphs.

1.2 PIPELINED SCHEDULING

In order to achieve real-time performance, the schedule of an SDF graph should have a highly optimized execution order that minimizes latency. Latency denotes the time cost of executing an iteration of the SDF graph, from the beginning of firing the first actor to the end of finishing the last actor. If a schedule is not pipelined, the execution of the next iteration cannot start before finishing the previous iteration, and the latency will be equivalent to the period of execution, i.e. the reciprocal of throughput. Similar to the concept of modulo scheduling approaches for software pipelining used in compiler domain [3], the scheduling of actors in SDF graphs can equally be pipelined, such that the next iteration will start early before the current iteration finishes its execution. A pipelined schedule can fully utilize processing elements and effectively increase the throughput while maintaining a minimal latency.

In the description of pipelined scheduling, the concept of Modulo Reservation Table (MRT) is commonly used. A MRT is a two-dimensional table for the modeling of resource constraints. In this table, the rows represent processing elements of the MPSoC model, and the number of columns is equal to the number of time slots in the period for execution of the SDF graph [2]. Fig. 1.1 shows an example use of a MRT: actor A is mapped on PE0 at time 0, and the execution time of actor A is 1. Therefore, the MRT will be updated by allocating MRT entry (PE0, 0) to actor A, making this time slot unavailable to the other actors.

The schedule shown in Fig. 1.2 is a pipelined schedule, since the actor F is scheduled between actor A and C. In each period, the actor F will consume tokens from the previous period and fire, while the actor A, B, C, D and E are sequentially fired, producing tokens for actor F in the next period.

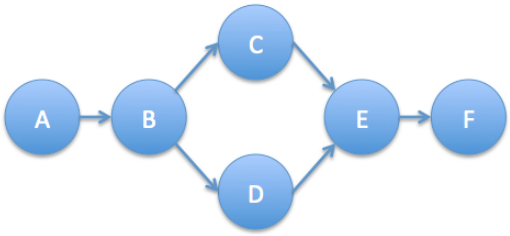

Fig. 1.1 Synchronous Dataflow Graph

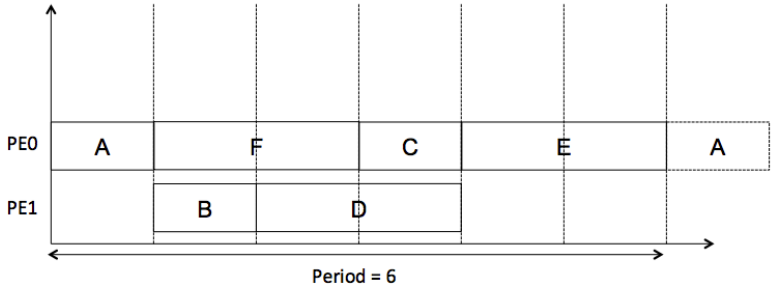

Fig. 1.2 Modulo Reservation Table for SDF graph

In the process of pipelined scheduling, there is always a tradeoff among throughput, latency and area requirements. Optimization of these objectives creates a complete set of Pareto-optimal solutions, and designers can customize an objective function to find an optimal result [4,7]. For the problem of optimizing a pipelined schedule, there are two typical approaches: one is building an integer linear programming (ILP) model for the SDF graph and MPSoC platform, and using an ILP solver to solve this model and produce the schedule [4,5]; the other one is using list scheduling algorithms to allocate time and processor resources to actors greedily [5].

The first approach can guarantee the optimality of the result, since the ILP solver will traverse the entire design space of the ILP model. The only drawback is the problem of complexity. A linear growth of the size of the SDF graph will result in exponential growth of the ILP design space. The second approach, on the other hand, uses heuristics to derive a result in polynomial time and avoids design space explosion. However, it does not guarantee the optimality of the result.

Despite the accuracy and optimality of ILP, their exponential complexity makes them infeasible for practical use. Instead, list schedulers are widely used in practice. A list scheduler's complexity is linear in the number of actor instances to schedule. Furthermore, list schedulers are flexible and can be easily combined with other algorithms or heuristics to increase optimality while maintaining a reasonable amount of execution time. In [10], a list scheduler is combined with a backtracking heuristic, which unschedules and then reschedules blocking actors when there is not enough space to place the next actor. Backtracking improves the rate of successfully finding a valid solution, but its performance relies on the search depth limit, which in turn significantly affects execution time performance. Backtracking also does not provide any guarantee of finding a valid scheduling solution if it exists.

In this report, a new heuristic to resolve pipelined scheduling problems is proposed. Our mobility and slack enhanced scheduling (MASES) heuristic is designed to improve on existing list scheduling heuristics. In every iteration of the list scheduler, it runs a comprehensive analysis on the partial schedule and reschedules actors on the MRT to create a large enough space for the target actor to place. Using MASES, a list scheduler approach is extended to be able to schedule the target actor and give out a result using a decidable amount of time, rather than dealing with unpredictable limits of backtracking.

1.3 Report Outline

The rest of this report is organized as follows: Chapter 2 introduces the related work of solving pipelined scheduling problems for SDF graphs, including ILP and list scheduling algorithms, as well as the backtracking heuristic that helps increasing the performance. Chapter 3 introduces our MASES heuristic in detail. Chapter 4 presents benchmarks and results of experiments on MASES in comparison to backtracking heuristics. Finally, Chapter 5 summarizes the report and states the conclusions.

CHAPTER 2: RELATED WORK

2.1 ILP model

Pipelined scheduling optimization problems of SDF graphs can be modeled using ILPs defined with input parameters, decision variables, constraints and an objective function. [2,3] Solving ILP models can provide theoretically optimal results. However, the size of ILP models as well as the execution time of ILP solvers grows exponentially when the size of SDF graph increases. As such, it is infeasible to process a large SDF graph using ILP models.

An ILP model for pipelined scheduling of SDF graphs is introduced in [4]. There, the decision variables includes $S_i(t)$ and $E_i(t)$, which accumulate the number of started and ended executions of actor i up to time t. $MRT_j(t)$ denotes the PE resource allocation, and is also adopted in decision variables of the ILP model.

The input parameters include d_{ij}, $p_{i1,i2}$, $c_{i1,i2}$, $o_{i1,i2}$, n_i, A_{ij}, period, buffer and time window. d_{ij} denotes the execution time of actor i on PE_j. p_{i1i2} and c_{i1i2} denotes the number of tokens produced and consumed on the edge between actor i_1 and i_2. o_{i1i2} denotes the number of initial tokens on this edge. n_i denotes the repetition count of actor i in one iteration. A_{ij} refers to the allocation of actor i on PE_j. Period places limit on the length of periodic schedule. Buffer limits the maximum number of tokens on every edge. Time window is a length of time on MRT in which the pipelined schedule is constructed. It consists of a startup phase and a stable and periodic phase.

Constraints in the ILP model include execution precedence, execution times, sequential execution, and periodicity in the stable phase. Execution precedence guarantees that actors will only execute when there are enough tokens on input edges. Execution time indicates the relationship between starting time and ending time of every

actor. Sequential execution signifies that one PE can only be allocated to at most one actor at any time.

The objective function in the ILP model for this problem is to minimize the latency of a schedule under given constraints.

2.2 List scheduling

List scheduling heuristics are widely used for solving scheduling problems and optimizing latency. A list scheduler prioritizes every actor in a SDF graph according to the maximum execution time along every possible path from the actor to the sink actors (actors that have no outgoing edges to any other actor). After assigning priority to all actors, the list scheduler greedily puts actors on PEs as soon as they are ready to execute and there is enough space on the MRT. If every actor is put on the MRT, the latency will be computed and returned. When an actor cannot be scheduled due to lack of MRT space, the list scheduler returns a latency of ∞, indicating that there is no possible schedule under the given constraints for this SDF graph. Fig. 2.1 shows the flow chart of such a basic list scheduler.

List scheduling heuristics do not guarantee the optimality of the scheduling result. Every time when an actor is scheduled on a PE, it will take a certain amount of consecutive time slots on the MRT. When the desired throughput is higher, the period will be smaller and it is harder to find enough space on the MRT for actors. Theoretically, the maximum throughput is reached when the period is so small that every time slot on at least one PE is utilized. In other words, the highest achievable theoretical throughput of a pipelined SDF graph is defined by the total execution time of all actors mapped to the

most critical, i.e. most highly utilized PE. In reality, however, such strict period constraints will keep list scheduler from finding a solution.

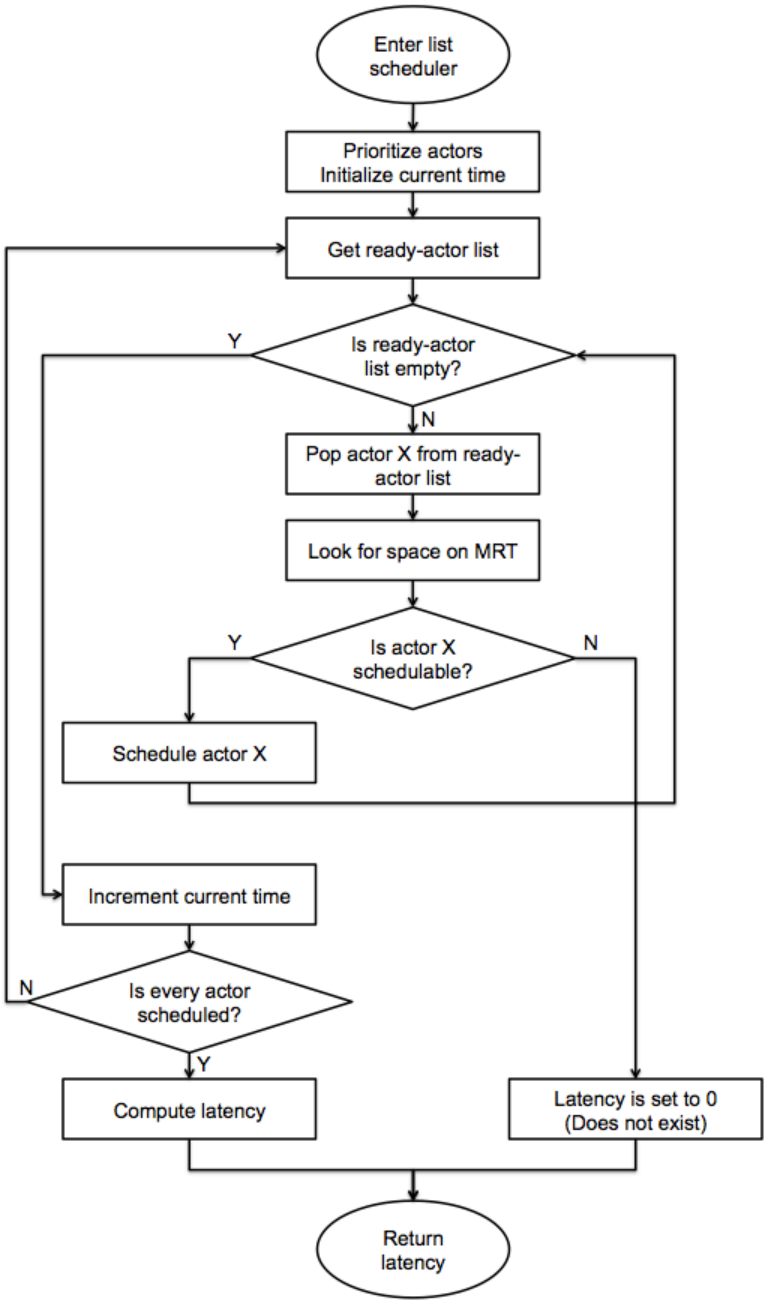

Fig. 2.1 Flow chart of list scheduler

2.3 Backtracking

Backtracking is a heuristic proposed to help the list scheduler to go back a few steps from its current partial schedule when it can no longer schedule any more actors [10]. Fig. 2.2 shows the flow chart of a list scheduler with backtracking included. The backtracking module is highlighted in the figure, and it will keep working until either finding a valid actor allocation or reaching a limit of backtracking attempts.

Ideally, exhaustive backtracking could cover the overall design space of pipelined schedules and find an optimal solution. An exhaustive search cannot avoid exponential design space explosion, however. Backtracking heuristics to solve the problem in acceptable time have been proposed. The problem is, however, that applying heuristics in backtracking might end up with an endless loop, where the list scheduler keeps scheduling and unscheduling actors. One heuristic for backtracking is proposed in [10], where the blocking actors and their successors will be first unscheduled, the actor to schedule will be placed, and the previously unscheduled actors will then be rescheduled. For the example in Fig. 1.1, backtracking heuristic starts from the state in Fig. 2.3, where it attempts to schedule actor F but no consecutive space is available. To place actor F, it then unschedules actor A as well as all other actors depending on A, as show in Fig. 2.4. Finally, in Fig. 2.5, previously unscheduled actors are rescheduled.

To keep actors from displacing each other over and over, there is a limit on the number of times the backtracking subroutine is called. Increasing the limit increases the odds of finding a solution, but will incur a higher cost in execution time.

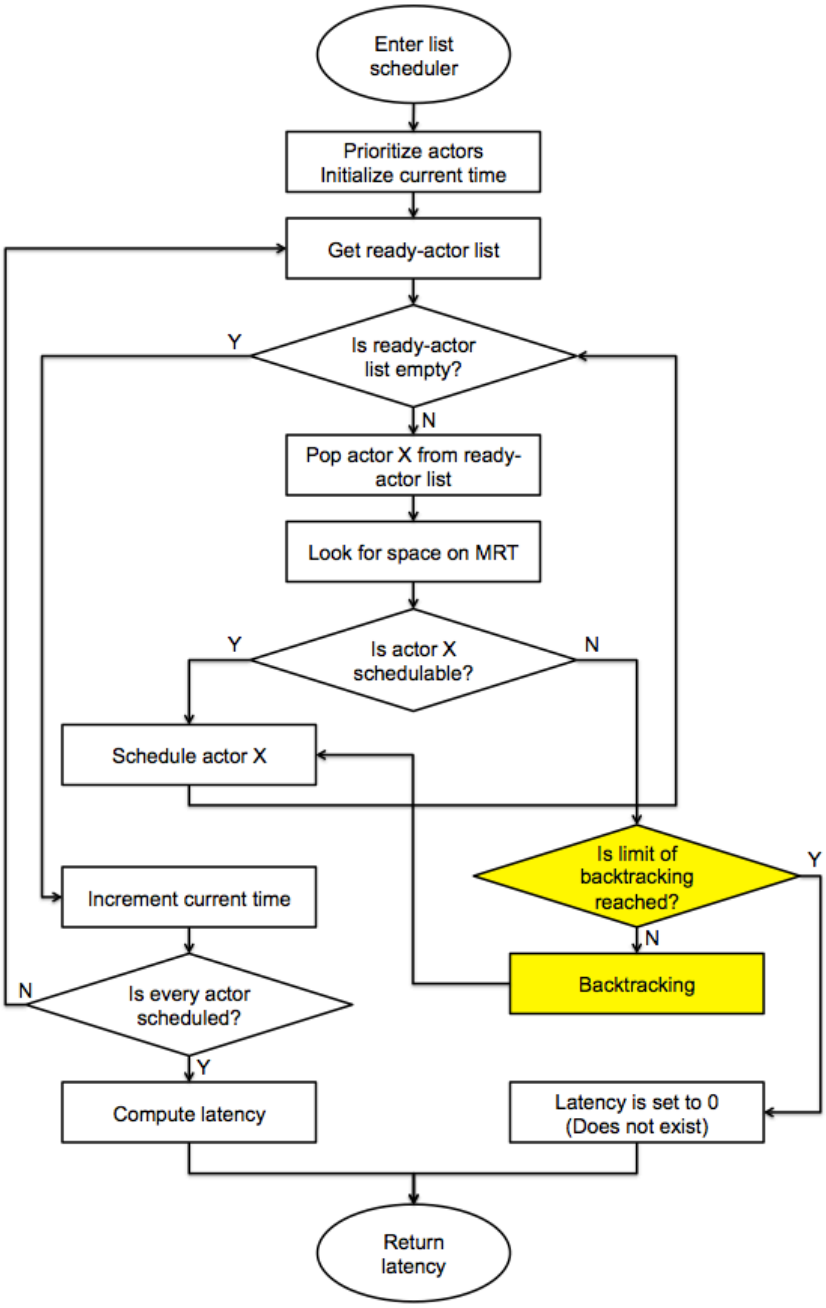

Fig. 2.2 Flow chart of list scheduler with backtracking

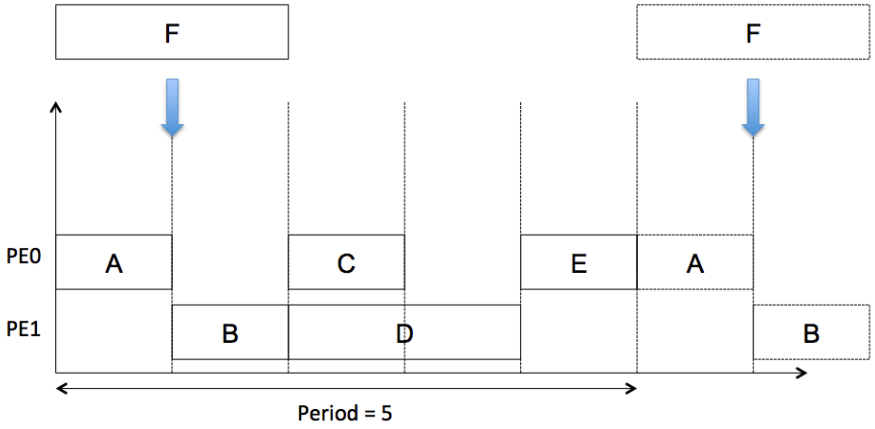

Fig. 2.3 Actor F cannot be scheduled

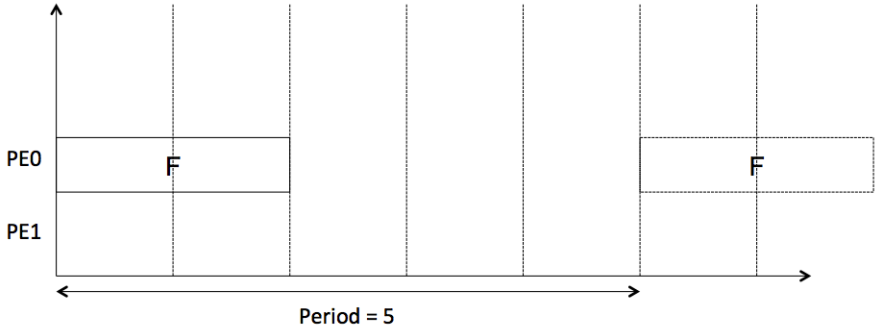

Fig. 2.4 Actor A and its successors are unscheduled

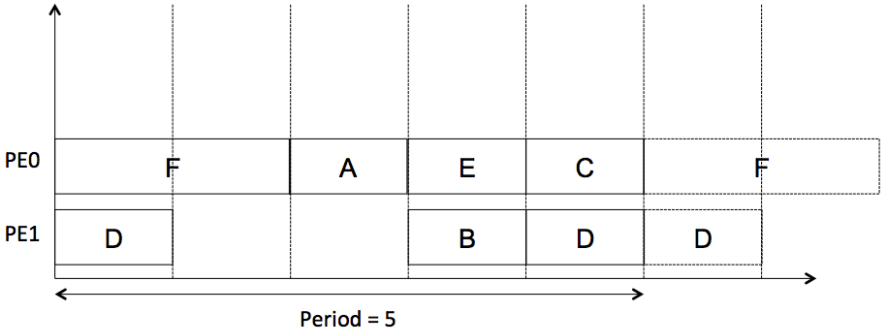

Fig. 2.5 Unscheduled actors are put back

CHAPTER 3: MOBILITY AND SLACK ENHANCED SCHEDULING

MASES is a heuristic that makes adjustment on the MRT of a partial schedule when a list scheduler cannot find a solution. Compared to backtracking heuristics, MASES does not unschedule any actor from the MRT, therefore resolving the problem of potentially endless loops in which actors keep displacing each other.

For the example in Fig. 1.1, actor F is supposed to be scheduled on PE0, and it is taking 2 consecutive time slots on the MRT. A list scheduler without backtracking will stop at the state in Fig. 2.3 and return failure. Since it greedily scheduled previous actors, when it comes to actor F, the available time slots are split and cannot be utilized. A list scheduler with backtracking will unschedule every actor from the MRT, putting actor F in the position of actor A, and then attempting to reschedule the other actors. For MASES, it is much simpler. As shown in Fig. 3.1 and Fig. 3.2, the actor C can be moved forward for 1 time slot, creating enough space for actor F. MASES exploits such actor mobility to adjust the MRT and find a valid scheduling pattern without unscheduling any actor.

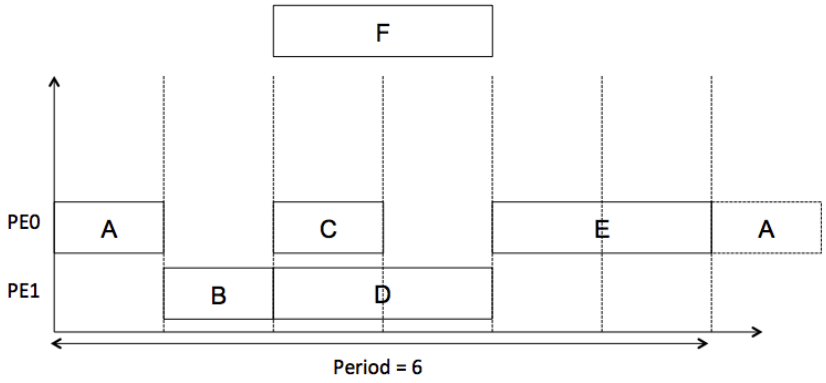

Fig. 3.1 Actor F cannot be scheduled on PE0

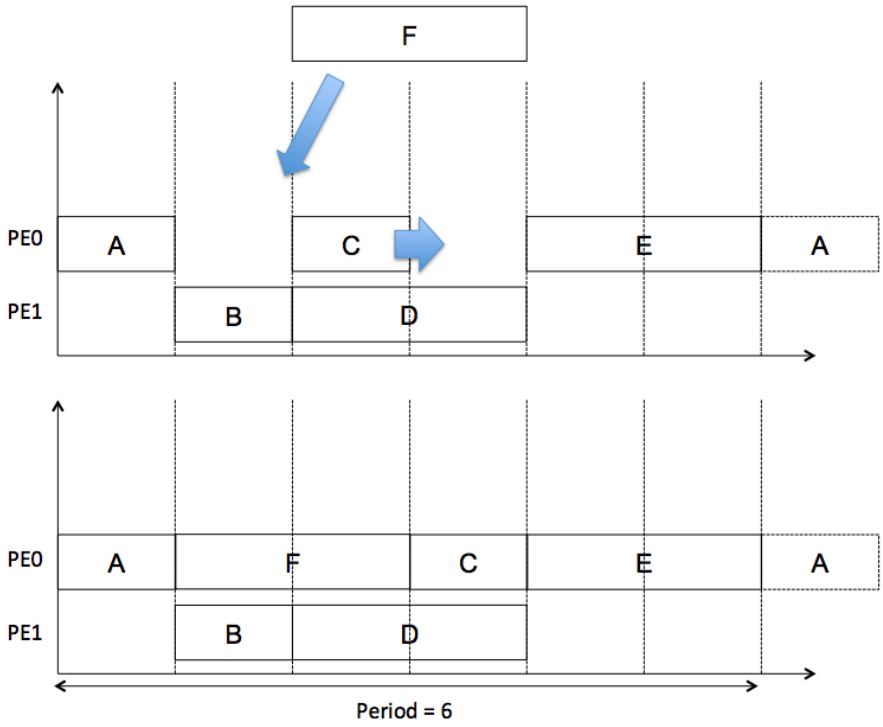

Fig. 3.2 MRT is adjusted and actor F founds its place

In MASES, we use concepts of mobility and slack to describe the flexibility in a partial schedule. If an actor has mobility or slack, it can be moved in the MRT with little side effect, thus creating a larger space for other actors. When there is not enough space for an actor, a comprehensive analysis of mobility and slack of every actor on the partial schedule is performed. To simplify the verbose description of consecutive available time slots on the MRT, we use a term called gap to refer to this concept.

3.1 Mobility analysis

The mobility of an actor is defined as the number of time slots that this actor can be moved forward without affecting the overall latency of the schedule. Actors that don't have any scheduled successor do not have mobility.

According to the definition, actor N's mobility can be computed by taking the minimum value of N's successors' mobility plus the distance between N and N's successors:

$$Mob(N) = \min_{M \in Succ(N)}\{dist(N,M) + Mob(M)\},$$

where dist(N,M) denotes the number of time slots from the end of actor N to the start of actor M:

$$dist(N,M) = (S(N)\%period - E(n)\%period + period)\%period$$

Here, S(N) and E(N) indicate the beginning and end of execution, respectively, of actor N relative to the beginning of execution of the first actor in the same iteration.

An example is shown in Fig. 3.3. According to the definition, the mobility is computed recursively. The mobility of actor C equals the mobility of actor D, which is the distance from actor D to F, a.k.a. 1 time slot. Therefore, both actor C and D have mobility of 1. The result of using mobility is shown in Fig. 3.4.

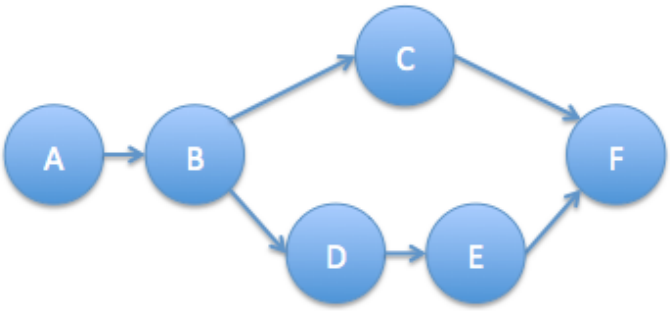

Fig. 3.3 Example SDF graph

While computing the mobility of an actor N, its successors' mobilities are taken into account. It is obvious that if all of actor N's successors have mobility, actor N can be moved as well. There is one more thing that needs to be considered when attempting to move actor N: the MRT only allows at most one actor to be scheduled on each PE at any time slot. Let Next(N) refer to the actor that is scheduled right behind actor N on the

same PE. If Next(N) is not actor N's successor, the computation of actor N's mobility should include the mobility of Next(N).

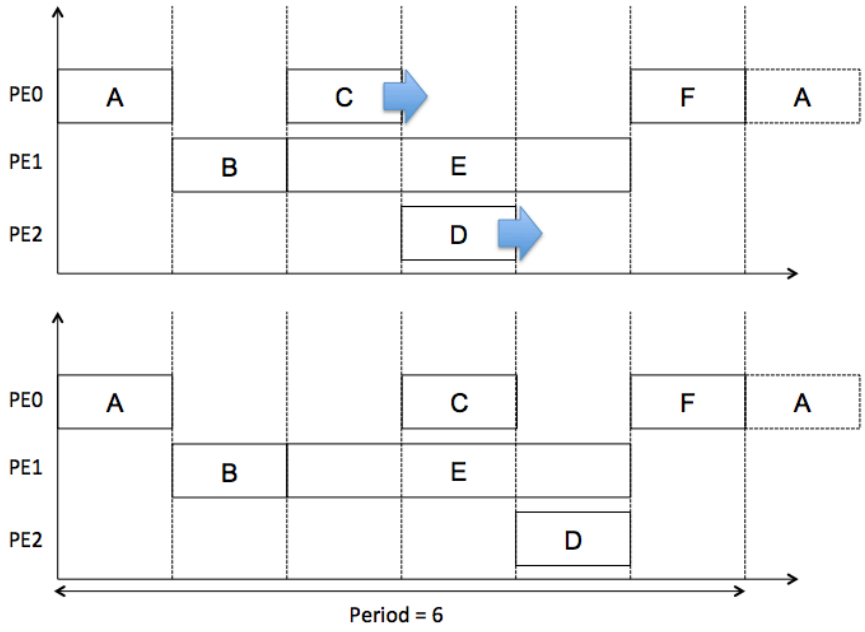

Fig. 3.4 Mobility analysis

For the example in Fig. 3.5, the distance between actor C and its immediate successor, actor F, is 1. However, actor C does not have a mobility of 1, because actor E, which is scheduled right behind it on the same PE, has no mobility and blocks actor C. In Fig. 3.6, actor C has no mobility.

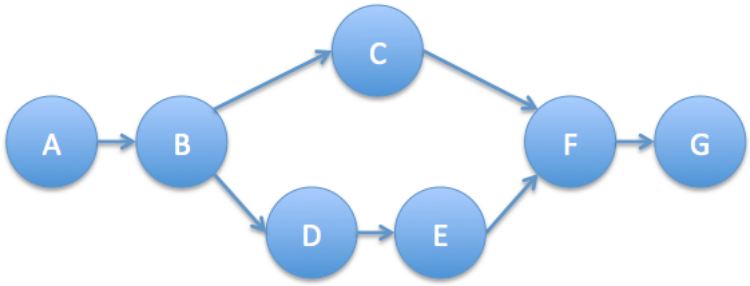

Fig. 3.5 Example SDF graph

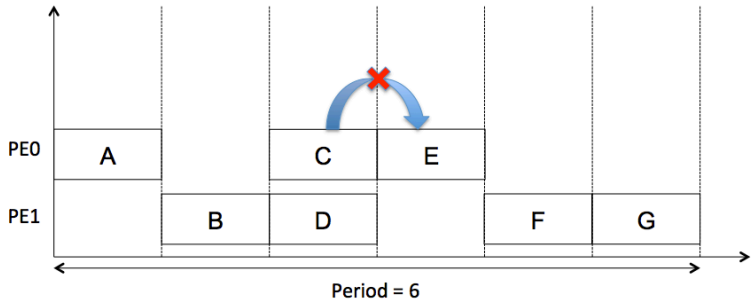

Fig. 3.6 Actor C is blocked by E and cannot be moved

For any actor N, Next(N) affects its mobility in the same way as N's successors do. With this, the computation of mobility becomes as follows:

$$Mob(N) = \begin{cases} 0 & , Succ(N) \in \phi \\ \min_{n \epsilon Succ\_Next(N)} \{Mob(n) + dist(N,n)\} & , else \end{cases},$$

where

$$Succ\_Next(N) = Succ(N) \cup \{Next(N)\}$$

At this point, the mobility of actor N denotes the possibility of moving actor N forward on the MRT. However, the motivation of exploring actor N's mobility is to expand the gap in front of actor N. Given that the gap is formed between actor M and actor N, if moving actor N requires moving actor M, the gap will be shifted instead of be expanded.

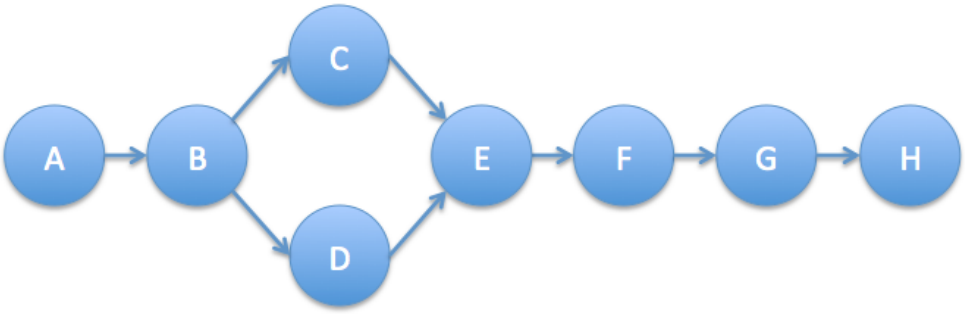

Fig. 3.7 Example SDF graph

Consider the example shown in Figs. 3.7 and 3.8, where the mobility of actor G is 1. Through recursive propagation, the mobility of actor F, E and C is also 1. According to definition of mobility, actor C can be moved forward for 1 time slot without affecting the overall latency. However, moving actor C is not beneficial, since it will not create a larger gap between actor G and C.

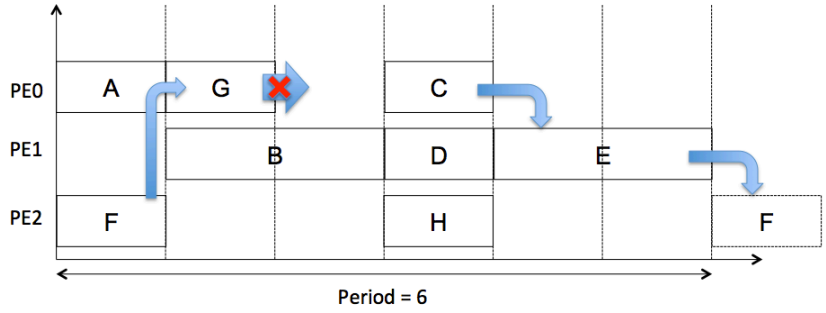

Fig. 3.8 Actor G should not be moved

This phenomenon does not affect the recursive computation of mobility of every actor. It only needs to be considered when an actor is selected to be the one that is actually going to be moved to create a larger gap before this actor. In case of this situation, we define another concept: *actual mobility*.

Let Prev(N) refer to the actor that is scheduled right before actor N on the same PE. The actual mobility refers to the number of time slots that this actor can be moved forward without moving actor Prev(N). If Prev(N) is scheduled later than N, i.e. S(Prev(N)) > S(N), moving actor N might affect Prev(N). In this case, the actual mobility of actor N can be computed conservatively by excluding the effect of actor N's successors' and Next(N)'s mobility:

$$Actual\_Mob(N) = \begin{cases} Mob(N) & ,S\big(Prev(N)\big) < S(N) \\ \min\limits_{n \epsilon Succ\_Next(N)} \{dist(N,n)\} & ,else \end{cases}$$

Fig. 3.9 and Fig. 3.10 shows the workflow of the algorithm that computes mobility of every actor on the MRT. It includes two nested functions: the outer function

calls the inner function for every actor on the MRT and returns the mobility information of every actor; the inner function recursively computes the mobility of the given actor.

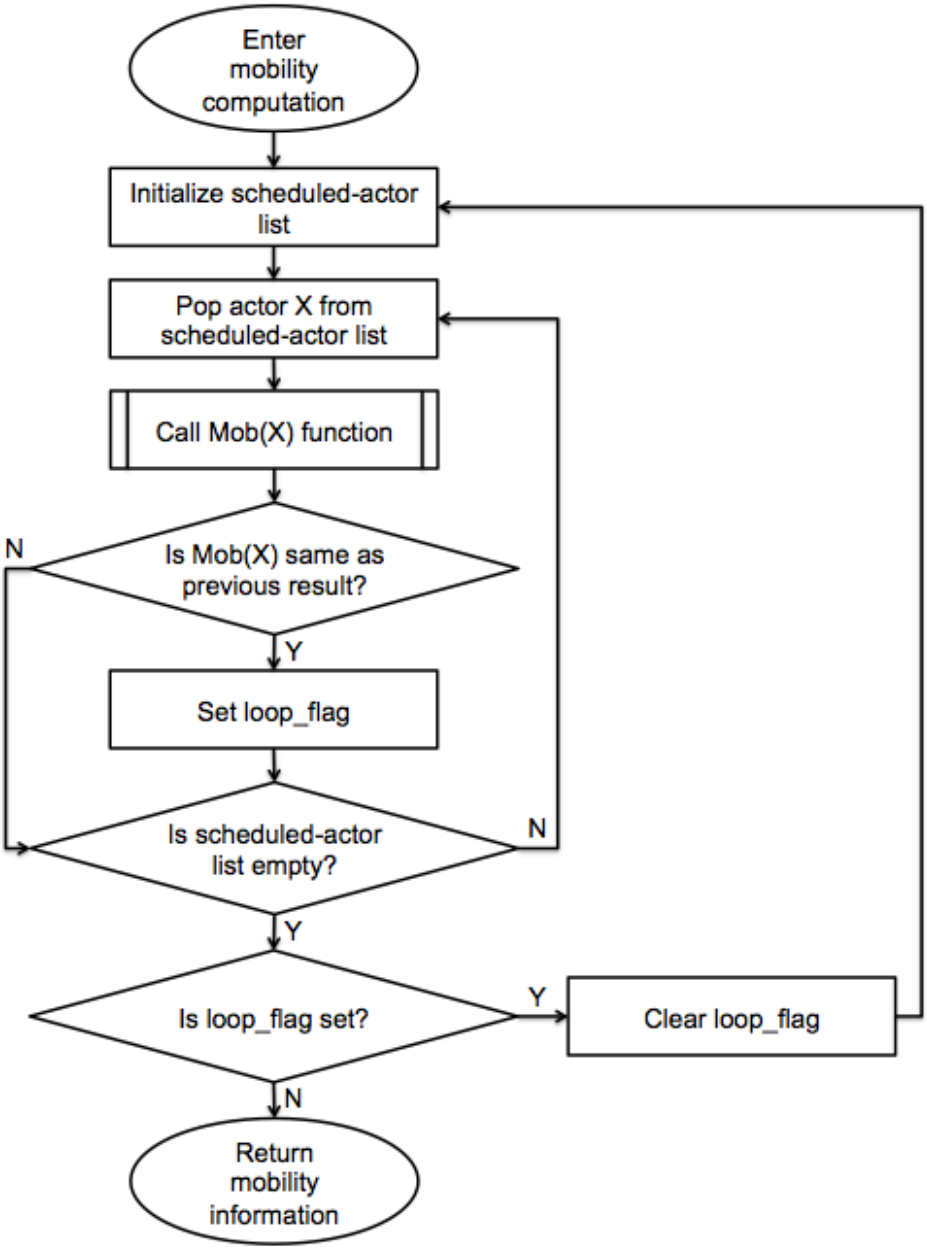

Fig. 3.9 Flow chart of mobility computation

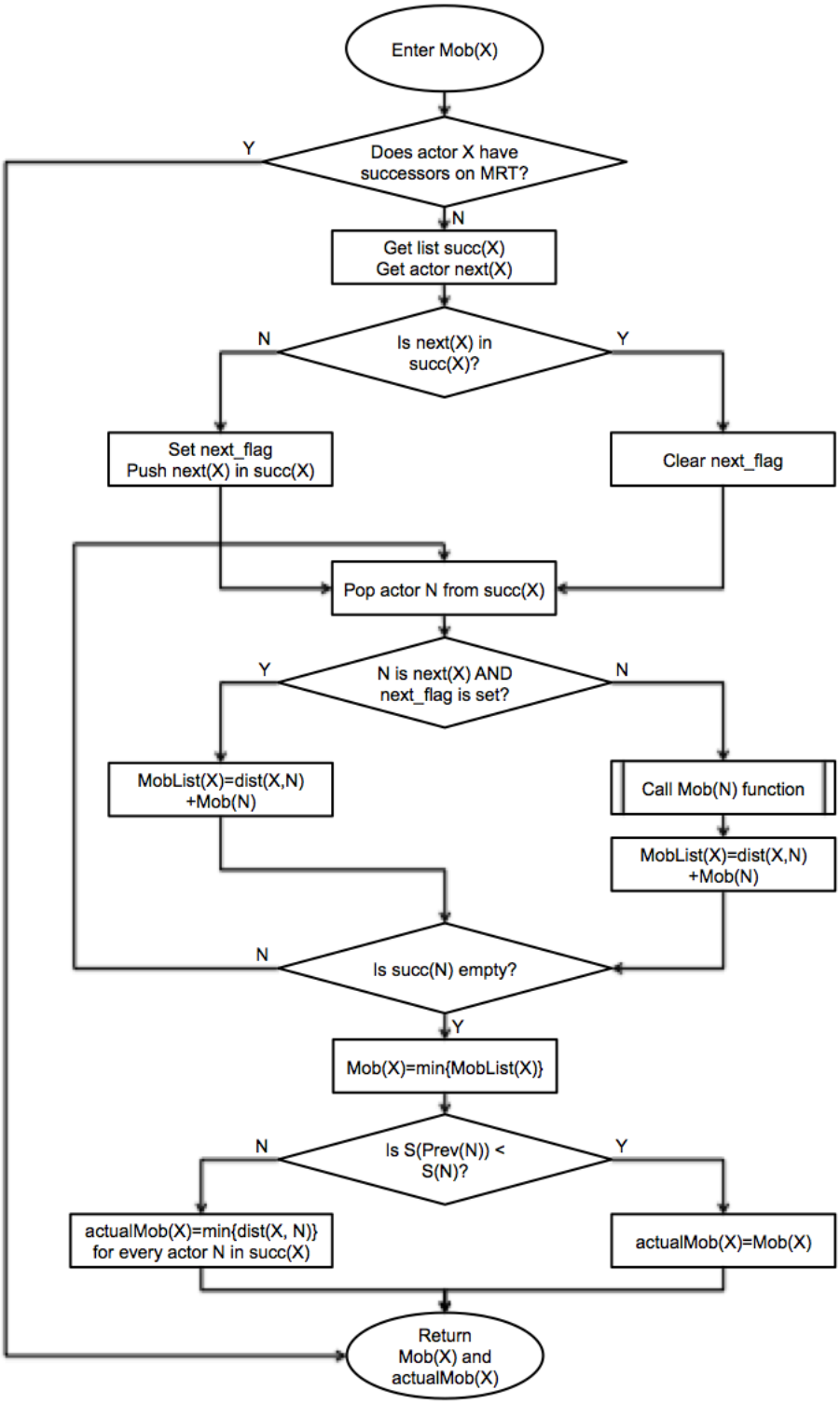

Fig. 3.10 Flow chart of recursive Mob(X) function

During this process, the mobility of all actors on the MRT is first initialized to 0. After that, mobility computation is performed on every actor on the MRT. Since mobility computation for an actor X involves the mobility of Next(X), to avoid a circular dependency that may cause endless recursion, we compute the mobility of Next(X) iteratively. Each time iterating over all the actors, the previously computed or initialized mobility of Next(X) is used. After each such iteration, the computed mobility information is compared with the one collected in the previous iteration. Mobility computation finishes when the mobility information no longer changes between two iterations.

3.2 Slack analysis

The slack of an actor is defined as the number of time slots that this actor can be moved forward along with its successors that have already been scheduled on the MRT. Moving actors using slack will move the actors that have no scheduled successors on the MRT, hence increase the overall latency of this partial schedule.

According to the definition, actor N's slack can be computed by taking the minimum value of N's successors' slack:

$$Slack(N) = \min_{M \in Succ(N)} \{Slack(M)\}$$

Similar to mobility, slack is also computed recursively. Since moving an actor using slack means moving several actors, including sinks, every actor on the MRT that does not have any successor scheduled on the MRT must have a slack of dist(N, Next(N)).

An example is shown in Fig. 3.11. From Fig. 3.12, we can find that none of the actors have mobility, but the slack of actor C is 2. If we move actor C using the slack, actor D and F will be moved simultaneously, and the overall latency is increased.

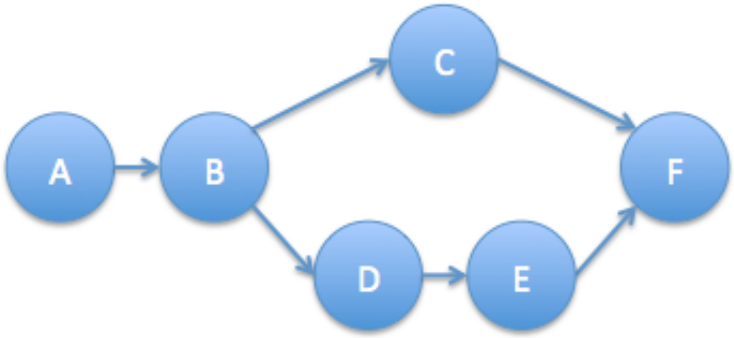

Fig. 3.11 Example SDF graph

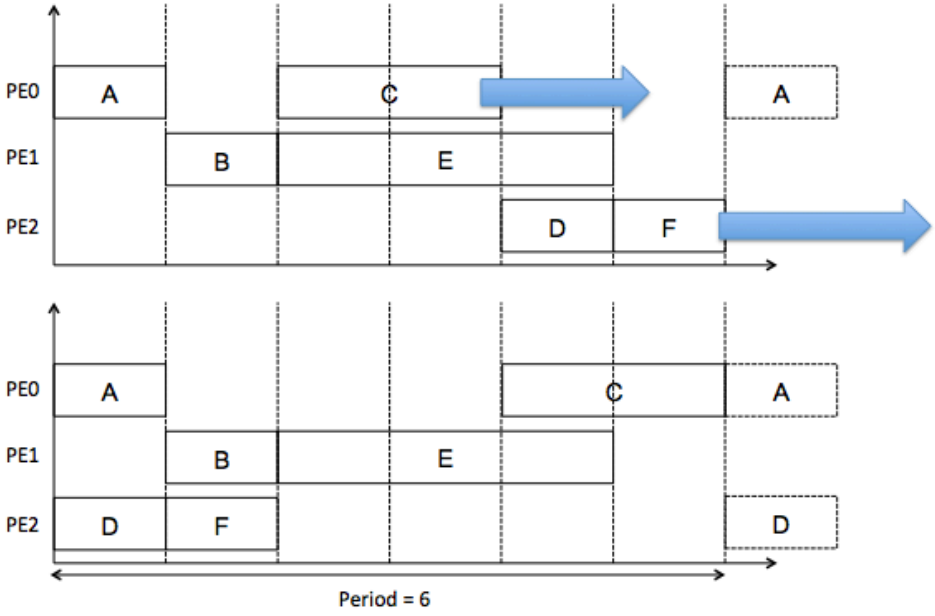

Fig. 3.12 Slack analysis

In slack analysis, actor Next(N) also needs to be considered when computing the slack of actor N. If Next(N) starts later than N, they will be moved together and Slack(Next(N)) will be taken into account. Otherwise, if Next(N) is not going to move along with actor N, dist(N, Next(N)) will be considered in slack analysis. In Fig. 3.12, actor C cannot move more than 2 steps because actor A blocks its way.

The computation of slack can be concluded as follows:

$$Slack(N) = \min\{Succ\_slack(N), PE\_slack(N)\},$$

where

$$Succ\_slack(N) = \begin{cases} period & ,Succ(N) \in \phi \\ \min\limits_{n \in Succ(N)}\{Slack(n)\} & ,else \end{cases}$$

$$PE\_slack(N) = \begin{cases} Slack(M) & ,S(N) < S(M) \\ dist(N,M) & ,else \end{cases}$$

and

$$M = Next(N).$$

Similar to actual mobility, there is a concept of actual slack. As mentioned before, if moving actor N will move Prev(N) in the end, then it is not going to expand the gap. If Prev(N) is scheduled later than actor N, it is very likely to be part of actor N's successors and their next actors. A conservative approach is setting the actual slack of actor N to constant 0 if S(Prev(N)) is greater than S(N):

$$Actual\_Slack(N) = \begin{cases} 0 & ,S\big(Prev(N)\big) > S(N) \\ Slack(N) & ,else \end{cases}$$

From the example in Fig. 3.13, we can find the difference between slack and actual slack. Since S(G) > S(C), the actual slack of actor C is 0. Moving actor C using slack will move actor G, as shown in Fig. 3.14. Similar to actual mobility, actual slack does not affect the slack propagation.

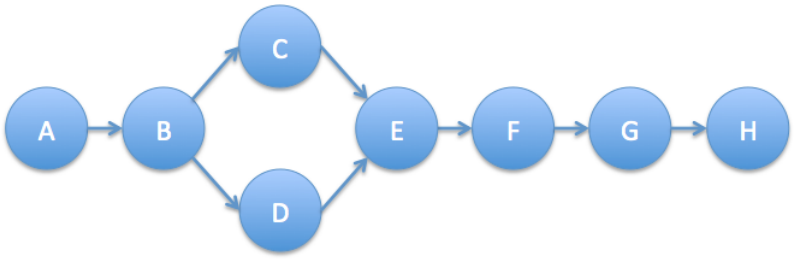

Fig. 3.13 Example SDF graph

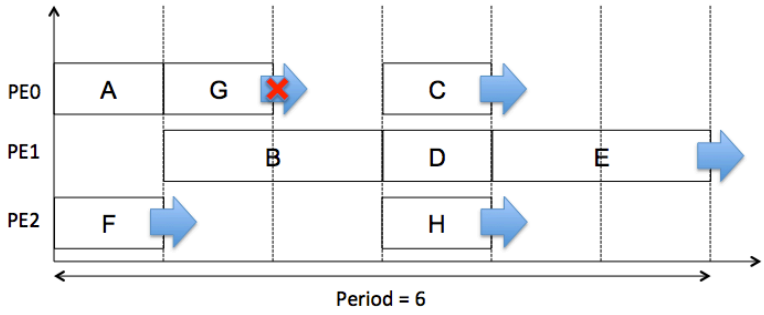

Fig. 3.14 Actor G should not be moved

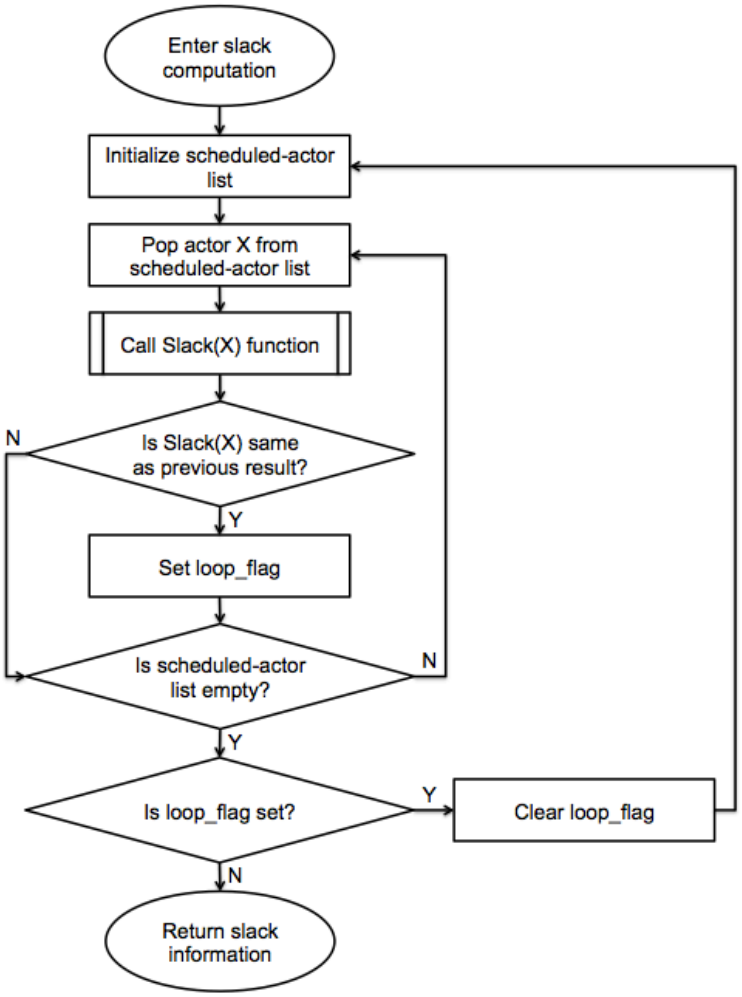

Fig. 3.15 Flow chart of slack computation

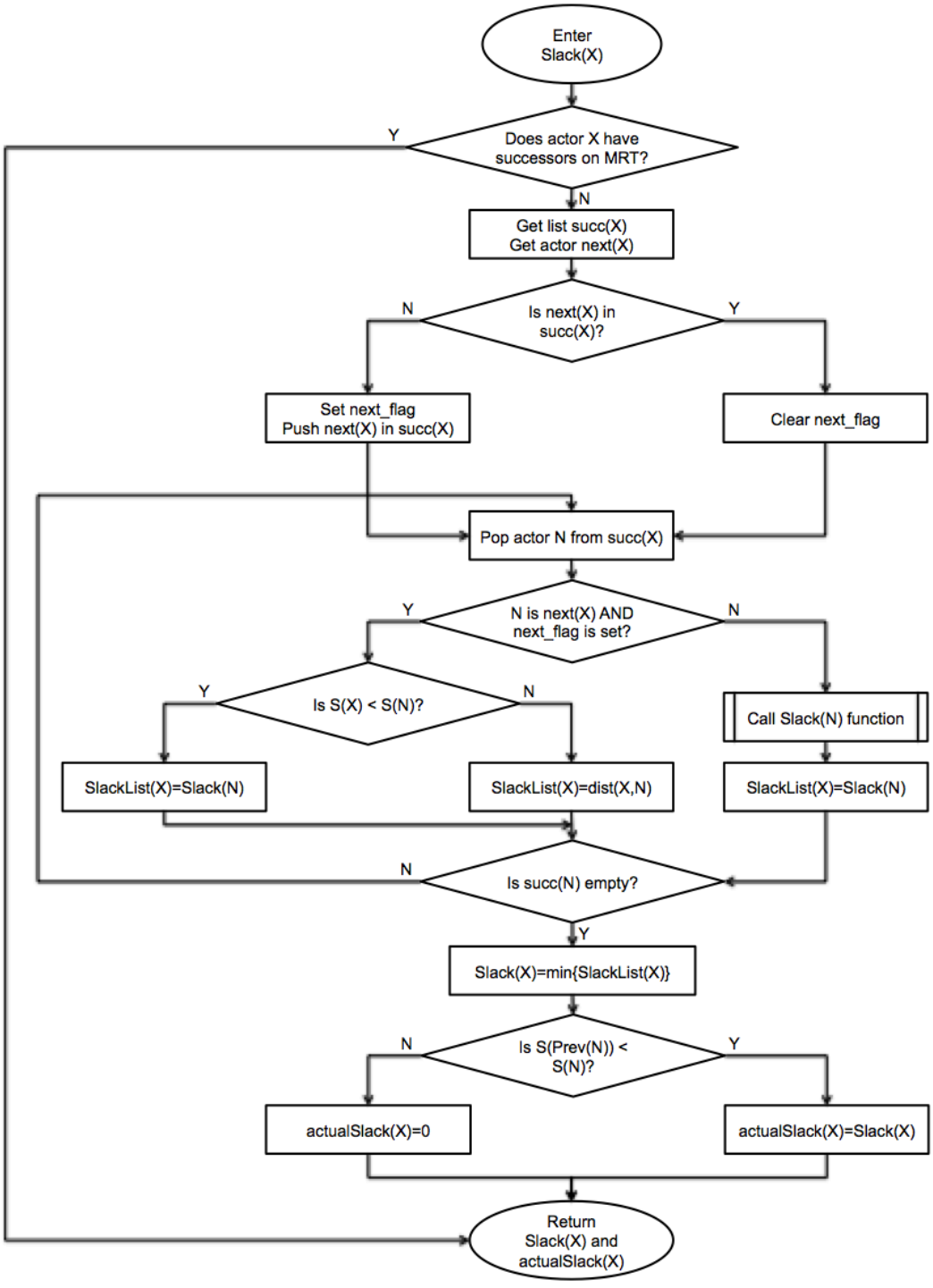

Fig. 3.16 Flow chart of recursive Slack(X) function

Fig. 3.15 and Fig. 3.16 shows the workflow of the algorithm that computes slack of every actor on the MRT. The way we compute slack resembles what we did for mobility computation. The slack computation requires two nested functions: the outer function calls the inner function for every actor on the MRT and returns the slack information of every actor, while the inner function recursively computes the slack of the given actor.

Similar to mobility computation, actors on the MRT are first initialized with fixed values of slack. For actors that don't have successors on the MRT, their slack is equal to the number of available time slots after them. The slack of other actors is initialized to 0. The recursive slack computation and comparison with previous slack information are otherwise similar to mobility computation.

3.3 Gap selection

After mobility and slack computation has finished and we have the mobility and slack information for every actor, we put all available gaps on the selected PE into a gap list, and decide which gap should be expanded for placing the new actor.

However, mobility and slack alone do not always work for expanding a gap. If we cannot use mobility or slack to make a large enough gap, we have to borrow spaces from other gaps by squeezing them.

For the example in Fig. 3.17, there is no mobility or slack for any actor on the MRT. If we hope to expand the gap between actor A and C, the only choice is squeezing the gap between actor C and E. Doing so requires moving actor C without moving its successor, actor D, and will break the precedence relationship of actor D firing before actor C finishes its execution. To solve this problem, actor D and all of its successors will

have to be moved forward for one period. In Fig. 3.18 and Fig. 3.19, we find that these actors will keep their positions on the MRT, but their starting times will be increased by one period. We can similarly squeeze the gap between actor A and C to expand the gap between C and E.

Although this will significantly increase the latency, the resource on the MRT are utilized more effectively, leaving less unused time slots on the MRT. Because of the latency cost, gap squeezing should only happen when the period is very small. In this case, latency is traded off for throughput.

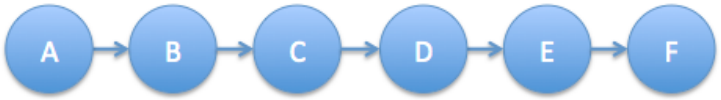

Fig. 3.17 Example SDF graph

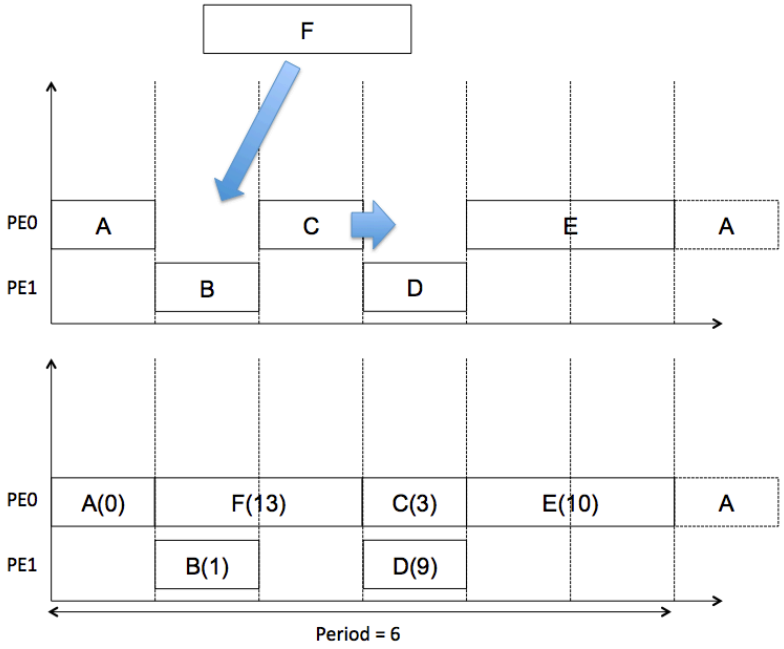

Fig. 3.18 Squeezing Gap(C, E)

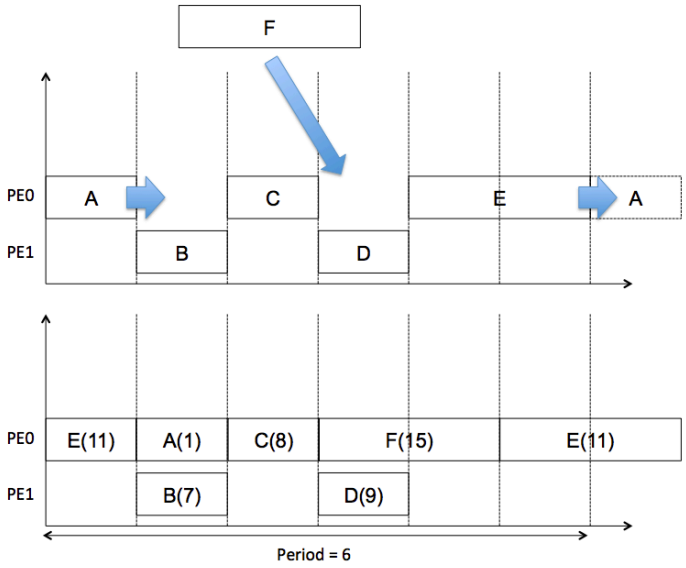

Fig. 3.19 Squeezing Gap(A, C)

Squeezing gaps helps a lot when we need more space. However, some gap can be non-eligible for squeezing due to backward edges or buffer space constraints.

A backward edge is an edge that forms a cycle and contains initial tokens. The source actor of a backward edge has lower priority, and the sink actor has higher priority. During the scheduling, the sink actor will be scheduled first by consuming initial tokens, and the source actor will be scheduled later, providing tokens for the sink actor in the next iteration. If there are not enough initial tokens to support two consecutive firings of the sink actor before the source actor finishes its execution, there will be a precedence violation. In light of this, the sink actor cannot be moved into a subsequent period. Doing so would add one period to every actor that follows the sink actor, including the source actor.

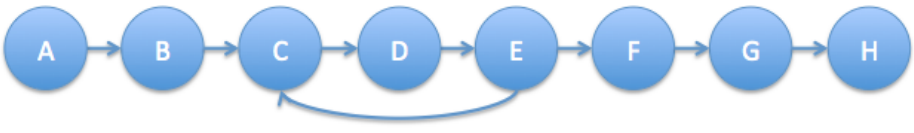

Fig. 3.20 Example SDF graph

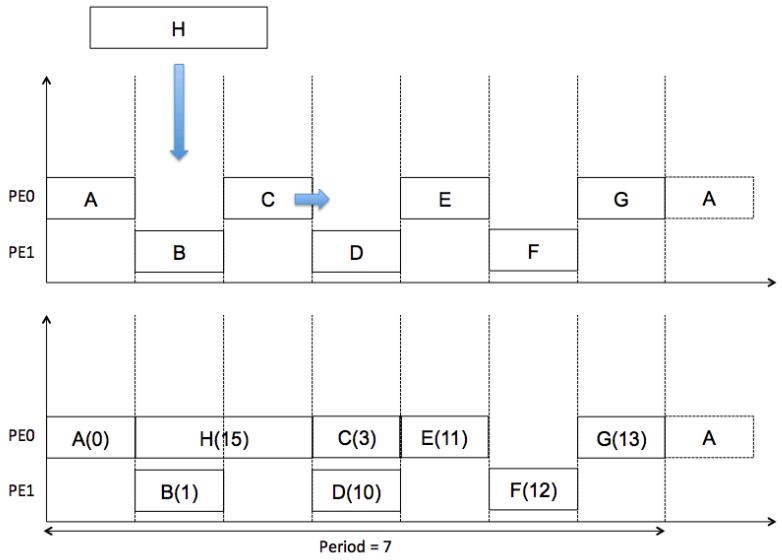

Fig. 3.21 Precedential violation caused by squeezing

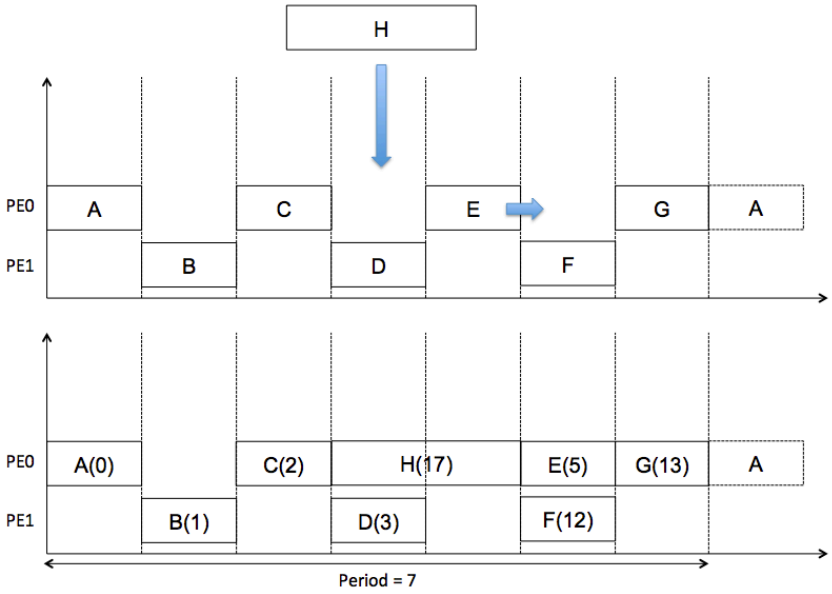

Fig. 3.22 Squeezing Gap(E, G) will not cause precedential violation

For the example in Fig. 3.20, actor H is going to be scheduled on PE0 and it will take two time slots. There are three gaps on PE0, and squeezing any of them would create

enough space for scheduling actor H. However, if the initial tokens on the backward edge from actor E to C only support one firing of actor C, squeezing gap (C, E) would push actor E forward for one period, as shown in Fig. 3.21. When the actor C starts firing in the next iteration, actor E in the current iteration has not fired yet. Therefore, gap (C, E) cannot be squeezed, and Fig. 3.32 should be the correct result of gap selection instead.

Buffer space constraints are another issue that can make a gap non-squeezable. Essentially, buffer space constraints on an edge impose an implicit backward edge between the actors of this edge, and larger buffer space constraints correspond to more initial tokens.

During the process of gap selection, the cost of choosing each gap is precisely evaluated. The goal of this phase is to find the best gap to place the new actor, while minimizing the overall latency. The workflow of gap selection is shown in Fig. 3.23. In the flow chart, start(gap) refers to the first slot of the gap, len(gap) denotes the length of the gap, mob(gap) and slack(gap) are defined as the actual mobility and actual slack of the right end actor of the gap.

At the beginning of this phase, we have a list of gaps and the target actor. The goal is to choose and expand a gap in order to create a large enough space for the target actor, such that the required length is exactly the execution time of the target actor. The earliest starting time of the target actor is known as t_start.

Choosing a different gap will generate different schedules. It is obvious that the best choice would result in the smallest latency. The latency penalty can be measured by computing the distance between t_start and the first time slot of gap.

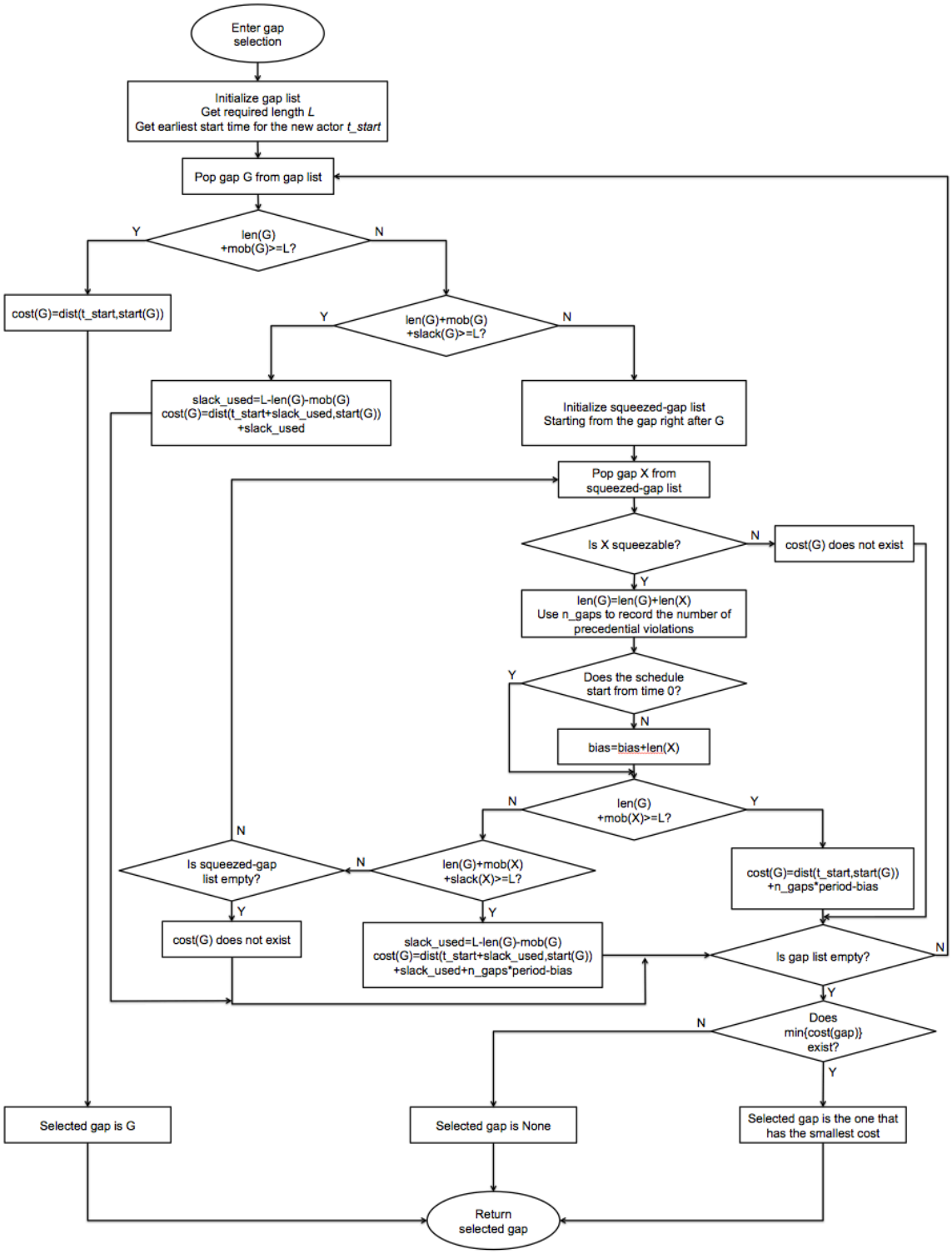

Fig. 3.23 Flow chart of gap selection

Given the gap list, we start from the gap that is closest to t_start and iterate the list. For each gap, we will first try to use mobility for expansion. If using mobility can achieve the goal, the current gap will be chosen. Since the gap list is sorted by the distance between t_start and every gap, finding a gap that allows expansion using mobility means that we do not need to look into any other gap.

If mobility cannot support the expansion, we will try to use both mobility and slack to expand current gap. Using slack will shift a block of actors and will increase t_start. As such, we have to search through all gaps even if a gap is found that can be expanded to meet the requirement using mobility and slack.

Finally, when using both mobility and slack does not work, a squeezed gap list is generated. Gaps in this list will be squeezed one by one until the accumulated length meets the requirement. Squeezing gaps will significantly affect the latency of the schedule. For every actor that is moved during the squeezing process, at least one extra period will be imposed on the start time of all of its successors that otherwise result in a precedence violation. Since squeezing gaps might change the start time of the overall schedule, i.e. move the first actor of the schedule, this shift has to be compensated for in the evaluation of cost.

After going through the gap list, a list of gaps sorted by cost is created. The cost is defined as the distance from t_start to the start of the gap, plus the number of periods added. The algorithm then selects the gap with the minimal cost. After putting the new actor in the selected gap, the absolute starting time of every actor will be updated and broken precedence relationships are rebuilt.

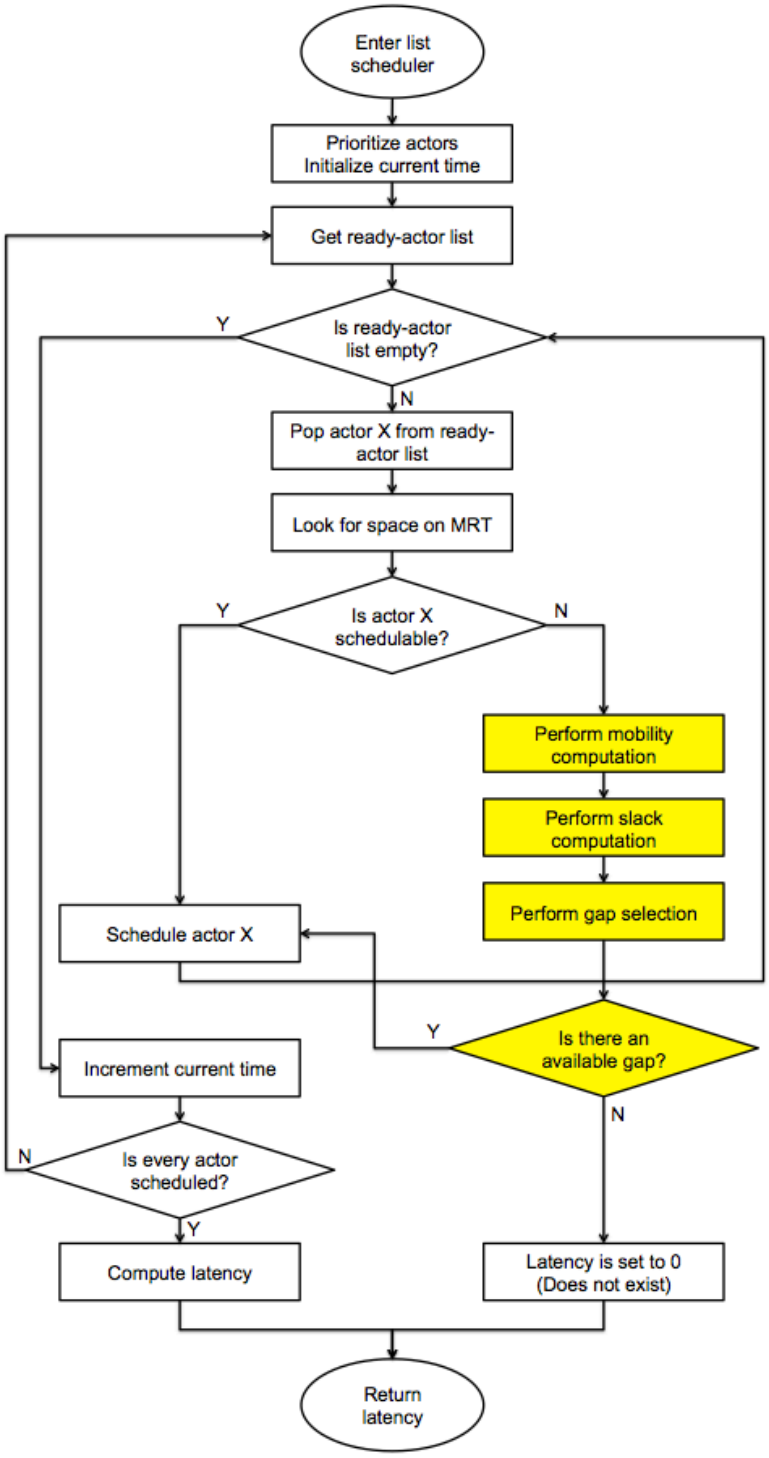

Fig. 3.24 Flow chart of list scheduler with MASES

3.4 MASES Algorithm

The complete MASES algorithm extends a basic list scheduler with three modules: mobility computation, slack computation and gap selection. The flow chart of the MASES algorithm is shown in Fig. 3.24. When the list scheduler cannot move forward, mobility and slack computations as well as final gap selection will be performed as described in previous sections.

CHAPTER 4: EXPERIMENTS AND RESULTS

We conducted a variety of experiments to evaluate list schedulers combined with backtracking and with our MASES heuristic. A set of randomly generated SDF graphs are fed into these two list schedulers and the results are compared from the perspective of execution time and optimality of results.

The SDF graphs for testing were obtained using SDF3's [9] random graph generator. We generated 1000 random SDF graphs that cover different sizes, i.e. number of actor instances, ranging from 10 to 100 at steps of 10 with 100 random graphs of each size. The number of PEs is set to 3. The other attributes of SDF graphs, such as repetition vector, execution time and unique PE assignment are randomly generated. We also introduce a real example from SDF3 [9], the H.263 decoder, to verify the functionality of MASES. In order to gain maximum throughput, the period constraint is always set to theoretical minimum value, and the buffer space is unlimited.

The list schedulers are implemented using C++, and all experiments were performed on a 3.5GHz Intel i7-4771 quad core workstation.

4.1 Random SDF graphs

As mentioned in Chapter 2, a backtracking-based list scheduler will call the backtracking subroutine when it cannot keep on scheduling new actors on the MRT. In a first experiment, we evaluated the backtracking-based list scheduler using random SDF graphs.

For the backtracking heuristic, a search depth limit is required to avoid endless loops of unscheduling and rescheduling of actors. The depth limit is usually proportional to the size of the input graph [10]. This is because larger graphs are more likely to fail

and thus require more attempts. Increasing the depth limit improves the success rate of finding a solution. However, indefinitely increasing the depth limit still does not eliminate failed cases, and doing so would make the cost of execution time unaffordable. Based on the data in [10], the depth limit for a SDF graph that has N instances can vary from N to 6*N, and the authors of [10] suggest a limit of 2*N, which balances execution time and success rate of finding a solution. In the following experiments, we set the default depth limit to 2*N.

From Chapter 3, we can conclude that one of the greatest advantages of the MASES algorithm is that it does not rely on any search depth. Whenever there is no cycle or buffer constraint in the SDF graph, MASES can guarantee the existence of a valid solution. This advantage makes MASES extremely useful when the period constraint is extraordinarily small.

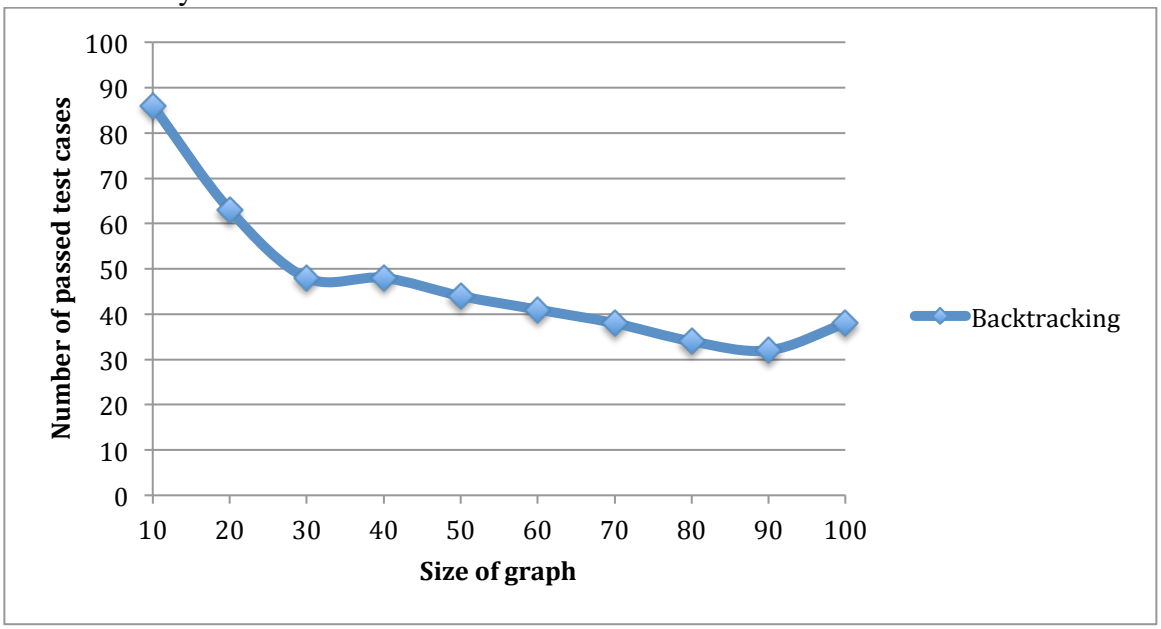

Fig. 4.1 Size of graph vs. number of tests succeeded for backtracking

Fig. 4.1 shows the relationship between size of the SDF graph and number of failed tests using a backtracking heuristic. As the size of graph increases, the possibility

of the backtracking algorithm finding a valid scheduling pattern quickly decreases. When the size of SDF graphs is 10, there are 86 out of 100 test cases that pass the backtracking heuristic. This number shrinks to 38 when the size of SDF graphs is 100. In comparison, the MASES algorithm never fails to find a solution. No matter how large the size of SDF graphs is, MASES can always find a valid scheduling pattern that satisfies the given period constraint if it exists.

The execution times of backtracking and MASES algorithms are also compared. Since there are a lot of test cases that did not succeed using backtracking, the comparison is performed in two different ways: (1) taking every test case into account, including cases for which backtracking failed, and (2) choosing only test cases that succeeded for backtracking and MASES. The results are presented in Fig. 4.2.

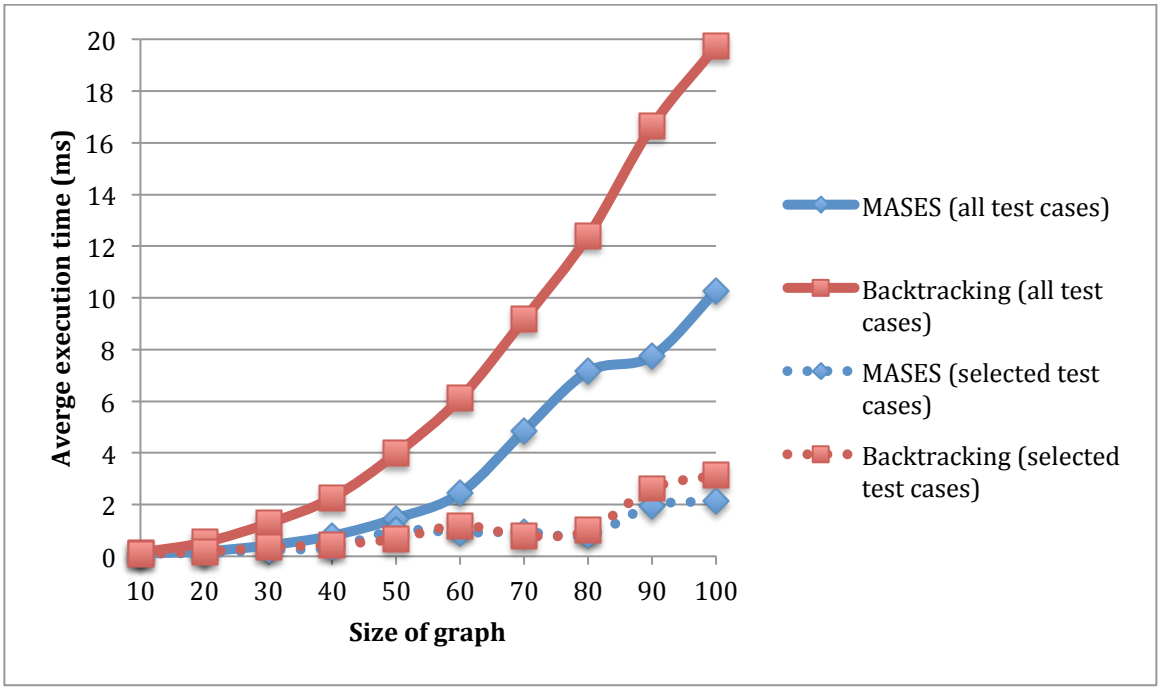

Fig. 4.2 Execution time of backtracking and MASES

The results show that MASES generally has a better execution time cost compared to backtracking. Since backtracking has to spend a lot of time on searching a schedule before reaching the depth limit, the time expense is particularly notable when including test cases for which backtracking fails. In comparison, the time expense for MASES is overall much smaller than backtracking. Even when only considering test cases that passed both MASES and backtracking, MASES still runs faster than backtracking for most of the test cases.

The quality of scheduling results in terms of achieved latency is also evaluated. In order to compare the latency, only test cases that are guaranteed to pass both schedulers must be selected. To compare latency results between MASES and backtracking, we normalized all results against latency achieved by a non-pipelined list scheduler that has a large enough period to accommodate every actor. The result is shown in Fig. 4.3.

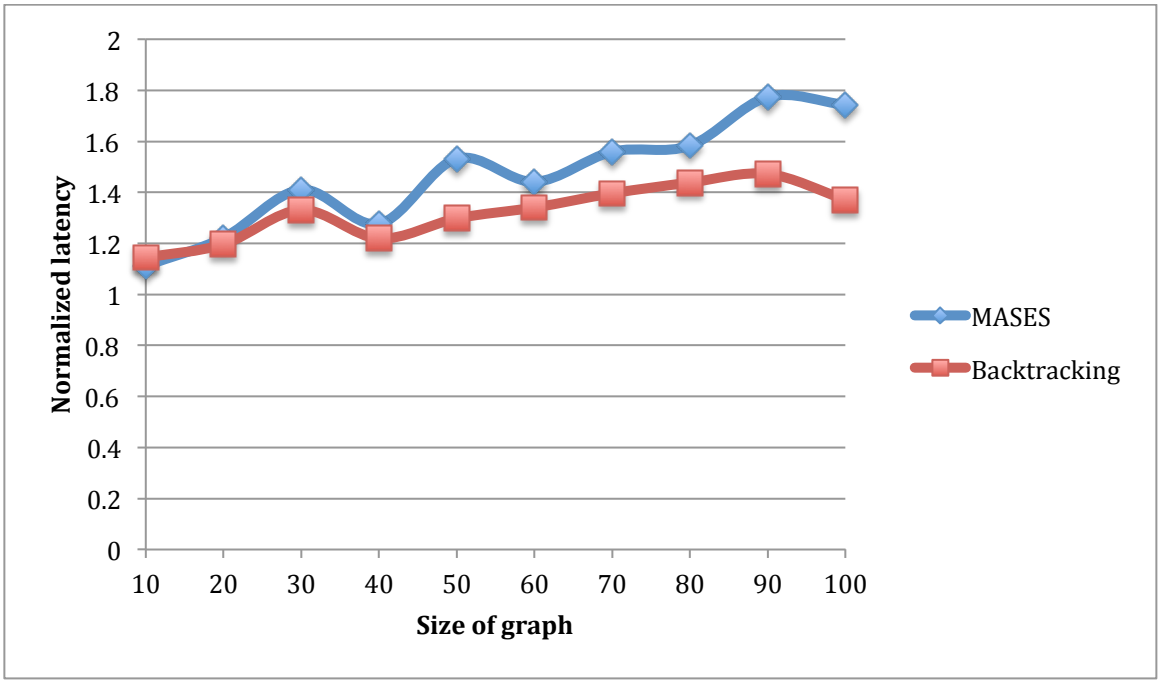

Fig. 4.3 Comparison of normalized latency

The results show that MASES does not perform as well as backtracking in terms of reducing latency. Backtracking basically repeats un- and re-scheduling actors, which is a form of exhaustive search targeted at minimizing latency. By contrast, MASES relies heavily on gap squeezing, which is one of the most important features that guarantees a 100% success rate of MASES. However, squeezing gaps can negatively affect the latency.

	Latency	Execution time
10	3%	4%
20	-2%	25%
30	-6%	27%
40	-5%	24%
50	-18%	-48%
60	-8%	24%
70	-12%	-18%
80	-10%	19%
90	-21%	26%
100	-27%	32%
Average	-11%	11%
Min	-27%	-48%
Max	3%	32%

Table 1. Improvement of MASES over backtracking

Finally, the comparison of latency and execution time improvements of MASES over backtracking for selected test cases are summarized in Table 1.

4.2 H.263 decoder

In this experiment, both MASES and backtracking algorithms were tested using the H.263 decoder example from [9]. The H.263 decoder is modeled as a SDF graph, with worst-case execution times for an ARM7TDMI core. The graph is shown in Fig. 4.4.

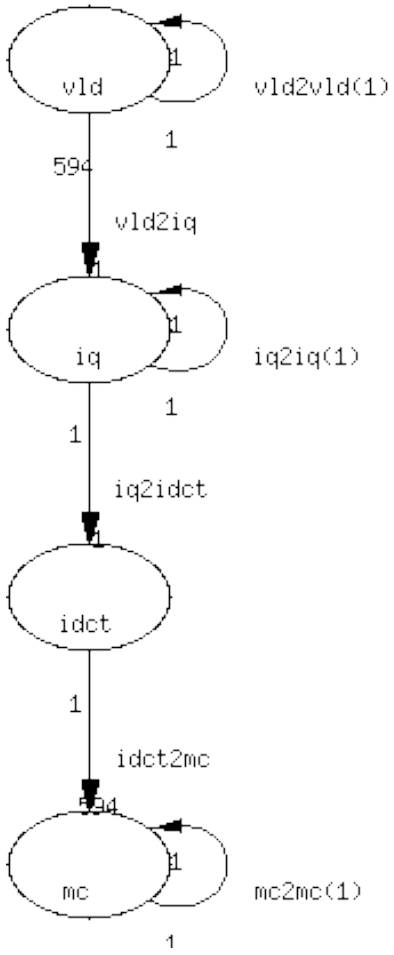

Fig. 4.4 H.263 decoder

In this example, actors vld, idct and mc are mapped on PE0, and actor iq is mapped on PE1. The execution time is 2 cycles for vld, 1 cycle for iq, 3 cycles for idct and mc. There are 1190 actor instances to schedule in this SDF graph. As shown in Fig. 4.5, we can see that when the list scheduler attempts to schedule actor mc, there are not enough consecutive available time slots. It took MASES 528.5s to solve this problem,

finding a schedule shown in Fig. 4.6 with a latency of 1792 cycles. In comparison, setting the depth limit of backtracking algorithm to 500, it still did not find a valid schedule after an execution time of 178.3s.

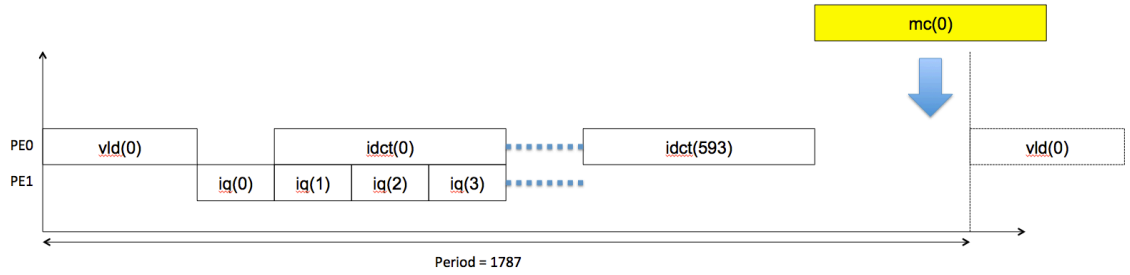

Fig. 4.5 MRT of pipelined schedule of H.263 decoder

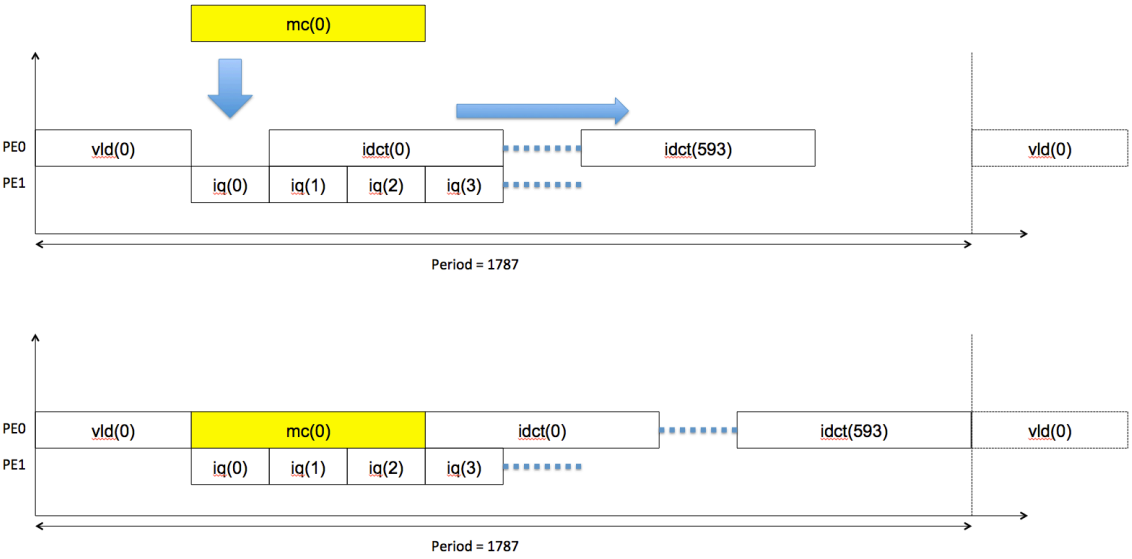

Fig. 4.6 Result schedule generated by MASES

CHAPTER 5: SUMMARY

This report contributed a new approach that solves the pipelined scheduling problem for synchronous dataflow graphs. As an alternative to backtracking heuristics in list schedulers, we propose mobility and slack enhanced scheduling (MASES), a heuristic that solves the problem of uncertainty of backtracking depth with better performance when the period constraint is extremely strict. Mobility and slack analysis utilizes the flexibility in the existing schedule for gap squeezing that trades off latency for throughput. These features allow us to make adjustment on the scheduled actors instead of removing them from the MRT and redoing the scheduling process, eliminating the possibility of actors keep displacing each other. Tested on a large set of randomly generated SDF graphs, the results of experiment prove the validity and benefits of the MASES heuristics.

ACKNOWLEDGEMENTS

This project was partially supported by National Instruments. I would like to thank Dr. Jacob Kornerup from National Instruments, Austin, for his support and active involvement in this project. I would also like to thank Dr. Brian Evans for providing his suggestions and constant encouragement in this project.

References

[1] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, 100.1 (1987): 24-35.

[2] M. Lam. "Software pipelining: an effective scheduling technique for VLIW machines," in *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988), 318-327.

[3] D. Gajski, S. Abdi, A. Gerstlauer and G. Schirner, "Embedded System Design: Modeling, Synthesis and Verification", ISBN 978-1-4419-0503-1, Springer, 2009.

[4] A. Sinha, "Multi-Objective Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs", Master's thesis, University of Texas at Austin, 2012

[5] J. Lin, et al., "Heterogeneous multiprocessor mapping for real-time streaming systems" in *IEEE International Conference on Acoustics, Speech and Signal Processing* (ICASSP), 2011.

[6] S. Bhattacharyya et al., "Synthesis of embedded software from synchronous dataflow specification," *Journal on VLSI Signal Process. Syst.* 21, 2 (1999), p. 151-166.

[7] S. Ritz et al. "Scheduling for optimum data memory compaction in block diagram oriented software synthesis" in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (ICASSP), 1995.

[8] S. Sriram and S. Bhattacharyya. "Embedded Multiprocessors Scheduling and Synchronization", Marcel Dekker, Inc, 2000

[9] S. Stuijk, "SDF3: SDF For Free" in *Application of Concurrency to System Design*, 2006.

[10] B. Rao, "Iterative Modulo Scheduling," *The International Journal of Parallel Processing*, Volume 24, 1996