MASES: Mobility And Slack Enhanced Scheduling For Latency-Optimized Pipelined Dataflow Graphs

Wenxiao Yu*
Google
Mountain View, CA, USA
wenxiao@utexas.edu

Jacob Kornerup
National Instruments
Austin, TX, USA
jacob.korenrup@ni.com

Andreas Gerstlauer
The University of Texas at Austin
Austin, TX, USA
gerstl@ece.utexas.edu

ABSTRACT

Dataflow and task graph descriptions are widely used for mapping and scheduling of real-time streaming applications onto heterogeneous processing platforms. Such applications are often characterized by the need to process large-volume data streams with not only high throughput, but also low latency. Mapping such application descriptions into tightly constrained implementations requires optimization of pipelined scheduling of tasks on different processing elements. This poses the problem of finding an optimal solution across a latency-throughput objective space. In this paper, we present a novel list-scheduling based heuristic called MASES for pipelined dataflow scheduling to minimize latency under throughput and heterogeneous resource constraints. MASES explores the flexibility provided by mobility and slack of actors in a partial schedule. It can find a valid schedule if one exists even under tight throughput and resource constraints. Furthermore, MASES can improve runtime by up to 4x while achieving similar results as other latency-oriented heuristics for problems they can solve.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Embedded software*;

1 INTRODUCTION

Many real-time streaming applications are characterized by a need to process large-volume data streams with both high throughput and low latency [1–3]. Such applications are natively non-terminating and can be modeled as synchronous dataflow (SDF) or task graphs (equivalent to homogeneous SDFs). The design process for implementing these applications requires processing elements (PEs) and time resources to be allocated to actors by a scheduler. Scheduling results determine the throughput and latency of executing a graph and are crucial for satisfying optimal performance requirements.

In order to achieve optimal performance, the schedule of a dataflow or task graph should have a highly optimized execution order that maximizes throughput while minimizing latency. Pipelining can thereby fully utilize processing resources and thus effectively increase the throughput while maintaining a minimal latency. Pipelined scheduling under multiple objectives has been studied extensively in the past. Most existing approaches, however, have been focused on throughput as the primary optimization goal [3].

To target latency minimization of a single iteration of a graph, variants of list scheduling heuristics, either standalone or in combination with other (meta-)heuristics for task partitioning, are predominantly used [4]. Pure list schedulers, however, do not consider throughput goals, and their challenge stems from dealing with resource conflicts when overlapping iterations of the same schedule under tight period, precedence and mapping constraints. Such conflicts can be resolved by pushing actor instances into later iterations, which removes dependencies and increases scheduling flexibility, but also increases pipeline length and thus latency. Instead, approaches such as modulo scheduling for software pipelining used in the compiler domain [5] employ repeated rollback with partial rescheduling to resolve conflicts while minimizing latency impact, but this is expensive and does not guarantee to find a solution.

In this paper, a novel constructive heuristic for pipelined list scheduling is proposed. Our mobility and slack enhanced scheduling (MASES) improves on existing list scheduling heuristics to address the problem of minimizing latency under given throughput, resource and heterogeneous task mapping constraints. In every iteration of the list scheduler, it runs a comprehensive analysis to determine flexibility in the existing partial schedule and minimally shift actors to create a large enough space for the next target actor to place. Overall, MASES provides a constructive list scheduling approach that supports pipelining under tight throughput and resource constraints. MASES is able give out a latency-minimized result using a decidable amount of time, rather than dealing with often unpredictable limits and lack of guarantees of existing iterative re-scheduling approaches.

2 RELATED WORK

In pipelined scheduling, there is a tradeoff among throughput, latency and area requirements. Approaches that focus exclusively on throughput have been researched extensively [1, 3]. For the problem of optimizing a latency-minimized pipelined schedule, there are two popular approaches: solving an integer linear program (ILP) or using latency-aware scheduling

*This work was performed while the author was at UT Austin.

SCOPES '18, May 28–30, 2018, Sankt Goar, Germany

ACM ISBN 978-1-4503-5780-7/18/05…$15.00
https://doi.org/10.1145/3207719.3207733

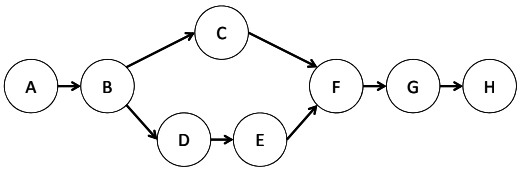


Figure 1: Example task graph.

heuristics, such as list schedulers. ILP approaches [6, 7] are optimal but exponential in complexity. By contrast, heuristics derive a result in shorter runtime, but do not guarantee optimality. Nevertheless, list schedulers specifically can often achieve close to optimal results, are flexible and can be easily combined with other objectives or algorithms to increase scope while maintaining fast execution time [3].

Among throughput- and latency-aware pipelining approaches, the authors in [8] discuss heuristics for latency-minimized scheduling of SDF graphs under throughput goals, but they do not consider resource constraints. The work in [9, 10] applies iterative task replication and clustering to meet both throughput and resource constraints while minimizing latency, but they only target homogeneous mapping problems. Several approaches combine list scheduler variants with different heterogeneous partitioning heuristics [11–15]. None of these approaches aggressively optimize latency under strict throughput constraints, however. In all cases, these algorithms resort to trying either a larger period or a larger number of periods, i.e. significantly increased latency if they fail to find a valid schedule. In [5], a pipelined list scheduler is combined with a backtracking heuristic to improve the rate of successfully finding a valid and latency-optimized solution even under tight throughput and resource constraints. However, its performance relies on the search depth limit, which in turn significantly affects execution time. Furthermore, backtracking does not provide any guarantee of finding a valid schedule. By contrast, MASES works with a fixed period and never unschedules any previous placed actors, therefore resolving potentially endless backtracking loops in which actors keep displacing each other. This significantly improves success and reduces runtime in finding a valid and optimized schedule (if one exists).

3 MOBILITY AND SLACK ENHANCED SCHEDULING

In pipelined scheduling, multiple iterations of the graph are allowed to overlap within one period. In the description of pipelined scheduling, the concept of a Modulo Reservation Table (MRT) is commonly used [5]. A MRT is a two-dimensional table for the modeling of resource constraints. In this table, the rows represent the PEs and the number of columns is equal to the number of time slots in the period for execution of the graph. MASES makes use of MRTs and adjusts a partial schedule when the list scheduler cannot find a solution.

In MASES, we use *mobility* and *slack* concepts to describe the flexibility in a partial schedule. Note that we adopt these terms from other domains, but our use case differs, e.g., compared to high-level synthesis in which mobility is defined

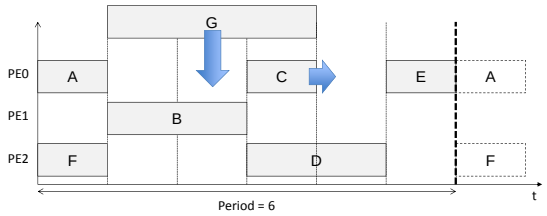


Figure 2: Using mobility of actor C to allocate G.

as the difference in operation start times between resource-unconstrained ASAP and ALAP schedules. By contrast, we define an actor to have mobility or slack if it can be moved in the MRT without violating resource constraints in an already existing partial schedule, thus creating a larger space for other actors. When there is not enough space for an actor, a comprehensive mobility and slack analysis of every actor on the partial schedule is performed. In the following discussions, we use the term *gap* to refer to a number of consecutive available time slots on the MRT.

For the example in Fig. 1 and Fig 2, actor G is supposed to be scheduled on PE0, and is requiring 3 consecutive time slots to execute. A list scheduler will return failure when it attempts to schedule G. Due to its greedy strategy, available time slots are divided and cannot be utilized. Such gaps can be avoided and actors freely scheduled by assigning dependent actors on different PEs to different periods/iterations. E.g., postponing C by one period will allow it to be scheduled right after A, which will avoid fragmentation. However, this increases schedule length by one period. Instead, actor C has mobility and can be moved without affecting the rest of the schedule. MASES will move C forward for 1 time slot and create enough space for actor G. Compared to conservative (re-)scheduling approaches, no additional period overhead will be incurred. Comparing to backtracking, MASES exploits such actor mobility to adjust the MRT and find a valid scheduling pattern without unscheduling any actor.

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 do not have any scheduled successor have no mobility.

While computing the mobility of actor a, its successors' mobilities are taken into account. It is obvious that if all of actor a's successors have mobility, actor a can be moved as well. According to the definition, actor a's mobility $M(a)$ can be computed as the minimum sum of a's successor's mobility plus the distance between a and its successor:

$$M(a) = \min_{s \in Succ(a)} (dist(a, s) + M(s)),$$

where $dist(a,s)$ denotes the number of time slots from the end of actor a to the start of actor s:

$$dist(a, s) = (S(s)\%P - E(a)\%P + P)\%P.$$

Here, P denotes the period, and $S(i)$ and $E(i)$ indicate the beginning and end of execution of actor i relative to the beginning of execution of the first actor in the same iteration.

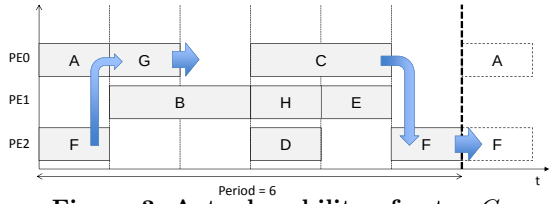


Figure 3: Actual mobility of actor C.

There is, however, one additional consideration when attempting to move actor a: the MRT may be blocked by other actors scheduled on the same PE. For example, since it is blocked by actor E, the mobility of actor C in Fig. 2 is 1 even though the distance to its immediate successor F is 2.

Let $Next(a)$ refer to the actor that is scheduled right behind actor a on the same PE, if any. If $Next(a)$ is not a successor of actor a, the computation of actor a's mobility should include the mobility of $Next(a)$. Hence, for actor a, $Next(a)$ affects its mobility in the same way as actor a's successors do:

$$M(a) = \begin{cases} 0 & Succ(a) \in \varnothing \\ \min_n(M(n) + dist(a,n)) & \text{otherwise,} \end{cases}$$

where $n \in Succ(a) \cup Next(a)$.

At this point, the mobility of actor a denotes the possibility of moving actor a forward on the MRT. However, the motivation of exploring actor a's mobility is to expand the gap in front of it. If moving actor a requires moving the previous actor on the same PE, the gap will be shifted instead of be expanded. Consider the partial schedule in Fig. 3 for the example graph from Fig. 1. The mobility of actor C is 1. According to mobility definition, actor C should be able to move forward for 1 time slot without affecting the overall latency. However, in this case, moving actor C is not a reasonable choice, since it will move actor G as well, and hence fails to create a larger gap. This phenomenon does not affect the recursive computation of mobility. It only needs to be considered when an actor is selected to be the one that is going to be moved to create a larger gap before itself.

To resolve this problem, we define the concept of *actual mobility*. Let $Prev(a)$ refer to the actor that is scheduled right before actor a on the same PE, if any. The actual mobility $M_a(n)$ refers to the number of time slots that actor a can be moved forward without moving actor $Prev(a)$. If $Prev(a)$ is scheduled later than actor a, i.e. $S(Prev(a)) > S(a)$, then moving a might affect $Prev(a)$. In this case, the actual mobility of actor a can be computed conservatively by excluding the effect of actor a's successors' and $Next(a)$'s mobilities:

$$M_a(a) = \begin{cases} 0 & Succ(a) \in \varnothing \\ \min_n(dist(a,n)) & S(Prev(a)) > S(a) \\ M(a) & \text{otherwise,} \end{cases}$$

where $n \in Succ(a) \cup Next(a)$.

Algorithm 1 shows the mobility computation of every actor on the MRT. The *mobility* of all actors is first initialized to 0. Since the computation of mobility for actor a requires the mobility of $Next(a)$, to avoid a circular dependency that may cause endless recursion, we compute the mobility iteratively. Each time iterating over all the actors, the previously computed or initialized mobility of $Next(a)$ 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.

Algorithm 1 Mobility Computation

```
1:  procedure MOBILITY(a)
2:      if a.visited then return
3:      a.visited = true
4:      a.mob = ∞
5:      a.actualMob = ∞
6:      if Succ(a) ∈ ∅ then
7:          a.mob = 0
8:          a.actualMob = 0
9:      for all s ∈ Succ(a) do
10:         MOBILITY(s)
11:         a.mob = min(a.mob, (dist(a, s) + s.mob))
12:         a.actualMob = min(a.actualMob, dist(a, s))
13:     n = Next(a)
14:     if n ∧ n ∉ Succ(a) then
15:         a.mob = min(a.mob, (dist(a, n) + n.mob))
16:         a.actualMob = min(a.actualMob, dist(a, n))
17:     if ¬Prev(a) ∨ S(Prev(a)) < S(a) then
18:         a.actualMob = a.Mob
1:  procedure MOBILITYCOMPUTATION
2:      for all a ∈ actorList do a.mob = 0
3:      for all a ∈ actorList do a.prevMob = None
4:      do
5:          loopFlag = false
6:          for all a ∈ actorList do a.visited = false
7:          for all a ∈ actorList do
8:              MOBILITY(a)
9:              if a.mob != a.prevMob then
10:                 a.prevMob = a.mob
11:                 loopFlag = true
12:     while !loopFlag
```

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 successors that have already been scheduled on the MRT. Moving actors using slack will move whole blocks of actors including ones that have no scheduled successors on the MRT. This can be used to redistribute and consolidate gaps. Exploiting slack, however, will affect possible start times of future actors and increase the overall latency of the partial schedule.

According to the definition, actor a's slack $L(a)$ can be computed as the minimum over a's successors' slacks:

$$L(a) = \min_{s \in Succ(a)}(L(a))$$

Similar to mobility, slack is computed recursively. Every actor on the MRT that does not have any scheduled successor has a maximum slack of one whole period.

However, a blocking actor $Next(a)$ also needs to be considered. If $Next(a)$ starts later than actor a, they will be moved

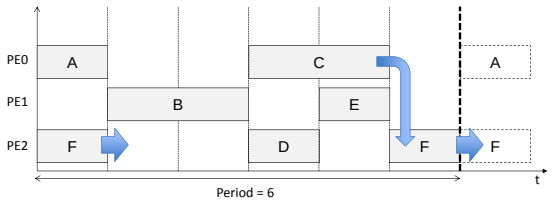


Figure 4: Slack of actor C.

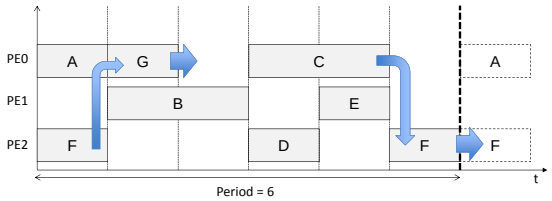


Figure 5: Actual slack of actor C.

together and the slack of $Next(a)$ will be taken into account. Otherwise, if $Next(a)$ is not going to move along with a, $dist(a, Next(a))$ will be considered. Slack is thus computed as the minimum of successor and next actor slack:

$$L(a) = \min(L_s(a), L_n(a)),$$

where

$$L_s(a) = \begin{cases} period & Succ(a) \in \varnothing \\ \min_{s \in Succ(a)}(L(s)) & \text{otherwise} \end{cases}$$

and

$$L_n(a) = \begin{cases} period & Next(a) \in \varnothing \\ L(Next(a)) & S(a) < S(Next(a)) \\ dist(a, Next(a)) & \text{otherwise.} \end{cases}$$

Consider the example schedule for the graph from Fig. 1 shown in Fig. 4. Here, the slack of actor C is 1 while its mobility is 0. Moving actor C using slack would move actor F and increase overall latency. Limited by its next blocking actor D on the same PE, the slack of F is 2. However, C is further limited by actor A blocking its way.

Similar to actual mobility, there is *actual slack*. As mentioned before, if moving actor a ends up moving actor $Prev(a)$, it will not lead to gap expansion. This is very likely to be the case if $S(Prev(a))$ is greater than $S(a)$. A conservative approach is therefore to set the actual slack of actor a to 0 in this case:

$$L_a(a) = \begin{cases} 0 & S(Prev(a)) > S(a) \\ L(a) & \text{otherwise.} \end{cases}$$

For the example in Fig. 5, the slack of actor C remains 1. In comparison, its actual slack is 0, since $S(G) > S(C)$.

Overall slack computation is performed similar to *MobilityComputation()* described earlier. A *SlackComputation()* function calls *Slack()* for every actor, which recursively computes slack. Actors on the MRT are first initialized with fixed slack values. For actors that do not have successors, the initial slack is equal to the number of available time slots after them up to a whole period. The slack of other actors is initialized to 0.

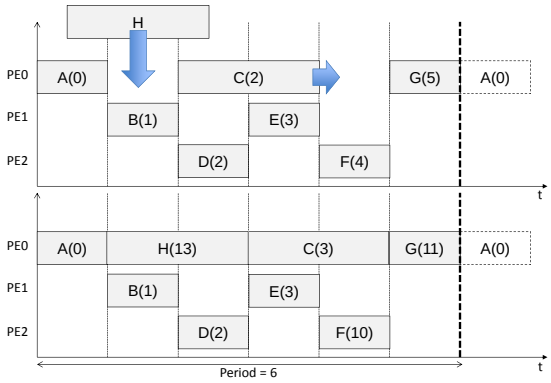


Figure 6: Gap selection and squeezing.

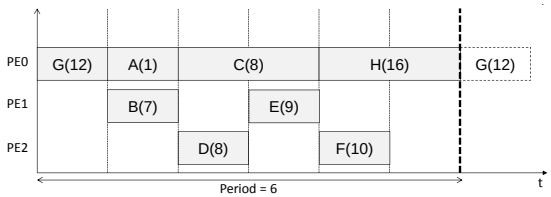


Figure 7: Alternative gap selection result.

3.3 Gap Selection

After mobility and slack computation has finished, we put all available gaps on the target PE under consideration into a gap list, and then decide which gap to expand for placing the new actor. However, mobility and slack are not always sufficient for expanding and finding a gap. If we cannot use mobility nor slack to make a large enough gap, we have to borrow space and expand a gap by squeezing other gaps. This will require moving actors while violating dependencies, which are resolved by pushing successors into later periods and thus significantly increasing latency. However, resources on the MRT are utilized more effectively, leaving less unused time slots. Because of the latency cost, gap squeezing should only happen when other mobility and slack options are exhausted. In this case, latency is traded off for throughput.

Consider the example shown in Fig. 6. Numbers in parentheses indicate actor start times $S(a)$. There is no mobility or slack for any actor on the MRT. Hence, we have two choices for placing actor H: expanding the (A, C) gap by squeezing the gap between C and G, or expanding (C, G) by squeezing (A, C). Choosing gap (A, C) requires moving actor C without moving its successors, which results in a dependency violation pushing F, G and H out by one period. The resulting latency is 15. Alternatively (Fig. 7), placing H in gap (C, G) would require moving G and A by one slot, violating the $A \mapsto B$ dependency and thus pushing all successors of A into a later period (with a total latency of 17).

Squeezing gaps is able to provide more space when needed. However, some gaps can be non-eligible for squeezing due to backward edges. A backward edge is an edge that forms a cycle and contains initial tokens. The sink actor of a backward edge is thereby scheduled before the source actor. If there are not enough initial tokens to support two consecutive firings of the sink actor before the source actor finishes,

Algorithm 2 Gap Selection

```
1: procedure GapSelection(a,t_start)
2:     d = a.len
3:     for all g ∈ gapList do
4:         if g.len + g.actualMob + g.actualSlack ≥ d then
5:             slack = max(d − g.len − g.actualMob, 0)
6:             g.cost = dist(t_start + slack, S(g)) + slack
7:             continue
8:         l = g.len
9:         bias = 0
10:        for all h ∈ gaps after g do
11:            if squeezable(h) then
12:                l = l + h.len
13:                n = number of precedence violations
14:                if h.start == E(first actor) then
15:                    bias = bias + h.len
16:                if l + h.actualMob + h.actualSlack ≥ d then
17:                    s = max(d − l − h.actualMob, 0)
18:                    g.cost = dist(t_start + s, S(g))
19:                    g.cost = g.cost + s + n·P - bias
20:                    break
21:    return gap with smallest cost
```

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. Similarly, buffer space constraints can make a gap non-squeezable. Essentially, buffer space constraints impose an implicit backward edge between the involved actors, where larger buffer space constraints correspond to more initial tokens.

The gap selection process is shown in Algorithm 2. The input is a target actor a, the earliest starting time of a known as t_{start}, and a list of gaps. In this algorithm, *g.start* refers to the first slot of a gap g, *g.len* denotes the length of the gap, and *g.actualMob* and *g.actualSlack* are defined as the actual mobility and actual slack of the actor at the right boundary of g. The goal is to choose and expand a gap in order to create a large enough space for the target actor's execution delay d. During this process, the cost of choosing each gap is precisely evaluated. Different gaps will result in different schedules. The objective is to find the best gap minimizing overall latency. For each gap g, we first try to use mobility and slack for expansion. If using mobility and slack achieve the goal, the gap is not further evaluated and the latency cost is measured as the distance between t_{start} and *g.start* adjusted by the used *slack*, if any. Note that using any slack will shift all successor actors and hence increase both latency and t_{start} accordingly. Finally, when using both mobility and slack does not succeed, gaps to the right of g will be evaluated for squeezing one by one until the accumulated length l meets the delay requirement d. 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 bias has to be compensated for in the evaluation of cost.

Algorithm 3 MASES Algorithm

```
1: procedure MASES
2:     Compute heights for all actors
3:     t = 0
4:     while !all actors scheduled do
5:         for all a ∈ ReadyActors(t) do
6:             slot = FindSlot(a,MRT,t)
7:             if ¬slot then
8:                 MobilityComputation()
9:                 SlackComputation()
10:                gap = GapSelection(a,t)
11:                if ¬gap then return Infeasible
12:                slot = S(gap)
13:            actor.start = slot
14:        t++
15:    latency = t - startTime
```

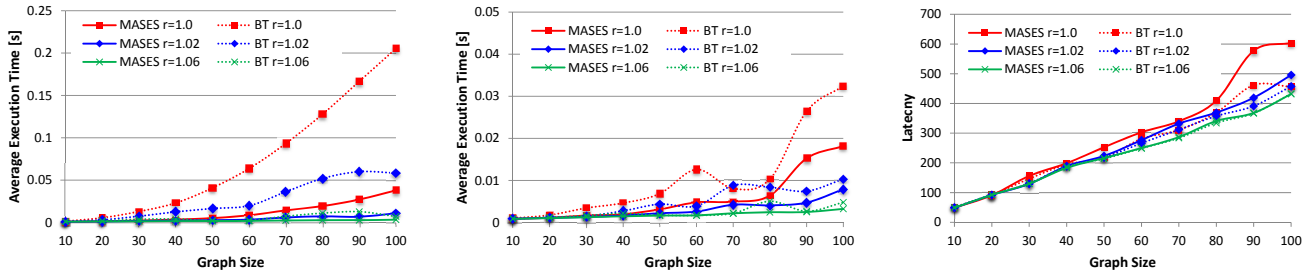
After going through the gap list, a list of gaps sorted by cost is created. The cost is defined as the distance from t_{start} to the start of the gap, plus the number of periods added. The algorithm then selects the gap with the minimal cost. After putting the new actor in the selected gap, the absolute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

The complete MASES algorithm (Algorithm 3) extends a basic list scheduler with mobility computation, slack computation and gap selection. When the list scheduler cannot move forward, mobility and slack of every actor will be computed and gap selection will be performed as described previously.

4 EXPERIMENTS AND RESULTS

We evaluated and compared MASES against the backtracking-based list scheduler from [5] that also targets pipelining under latency goals and throughput constraints. Both schedulers were implemented in C++, and all experiments were performed on a 3.5GHz Intel i7-4771 quad core workstation. Using SDF3 [16], we generated 1000 random acylic SDF graphs of different sizes, i.e. number of actor instances, ranging from 10 to 100 at steps of 10, with 100 random graphs of each size. Graphs were constrained to have half as many nodes as instances. The number of PEs is set to 3. Other attributes, such as repetition vectors, execution time and unique PE assignment were randomly generated. Period constraints were varied in a range $P = rP_{min}, 1.0 \leq r \leq 1.06$, where P_{min} is the theoretical minimum period defined by the sum of execution times of actors mapped to the most critical, i.e. most highly utilized PE on which every available time slot is used. Buffer space is assumed to be unlimited.

For the backtracking heuristic, a search depth limit is required to avoid endless re-scheduling loops. The depth limit is usually proportional to the size of the input graph [5]. 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, even an infinite search limit does not eliminate failed cases, and doing so would make execution times unaffordable. Based on [5], the depth limit for a SDF graph that has N instances can vary from N to $6N$, and the authors of [5] suggest a limit



(a) Runtime for all cases (b) Runtime for backtracking success cases (c) Latency

Figure 8: Backtracking (BT) versus MASES runtime and latency.

Table 1: Tests succeeded for backtracking.

	Graph Size									
r	10	20	30	40	50	60	70	80	90	100
1.0	86%	63%	48%	48%	44%	41%	38%	34%	32%	38%
1.02	91%	83%	74%	75%	81%	84%	79%	78%	77%	83%
1.06	96%	95%	91%	95%	99%	99%	95%	97%	96%	99%

of $2N$, which balances execution time and success rate of finding a solution. In the following experiments, we set the default backtracking depth limit to $2N$.

One of the main advantages of MASES is that it does not rely on any iterative search. Whenever there is no cycle in the graph, MASES can guarantee the existence of a solution. This makes MASES advantageous when the period constraint is small. Table 1 shows the number of failed tests using a backtracking heuristic versus the size of the SDF graph. As the size of graph increases, the possibility of the backtracking algorithm finding a valid schedule 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.

In Figs. 8(a) and 8(b), the execution times of backtracking and MASES are compared. In this experiment, we run both algorithms on all test cases and only on the ones for which backtracking succeeds. Results show that MASES has significantly 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 for small periods when backtracking often fails. 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 (Fig. 8(c)). Only test cases that are guaranteed to pass both schedulers are selected. Results show that for the cases where it succeeds, back-tracking performs slightly better in terms of minimizing latency. Backtracking repeatedly un- and re-schedules actors, which is a form of exhaustive search. By contrast, MASES relies heavily on gap squeezing, which is one of the most important features that guarantee a 100% success rate. However, squeezing gaps is a greedy process that can be suboptimal.

5 SUMMARY AND CONCLUSIONS

This paper contributed a new approach for latency-aware, pipelined scheduling of dataflow graphs. We propose mobility and slack enhanced scheduling (MASES), a heuristic that improves on existing latency-oriented pipelined list-scheduling heuristics when the period constraint is extremely strict. Mobility and slack analysis utilizes the flexibility in the existing schedule to minimize latency under a throughput goal. These features allow us to make adjustment on the scheduled actors instead of rolling back and redoing the scheduling process. The main benefit of MASES over existing work is that it is able to find a schedule if it exists even under tight throughput and resource constraints. Furthermore, it can have better complexity at similar quality of results compared to other heuristics for problems they can solve.

REFERENCES

[1] J. Javaid, S. Parameswaran, *Pipelined Multiprocessor System-on-Chip for Multimedia*, Springer, 2014.
[2] S. Francis, *et al.*, "A Reactive and Adaptive Data Flow Model For Network-of-System Specification," *IEEE ESL*, 9(4), 2017.
[3] A. Benoit, *et al.*, "A survey of pipelined workflow scheduling: models and algorithms," *ACM Computing Surveys*, 45(4), 2013.
[4] L.-C. Canon, *et al.*, "Comparative evaluation of the robustness of DAG scheduling heuristics," in *Grid Computing*, Springer, 2008.
[5] B. Rao, "Iterative modulo scheduling," *IJPP*, 24(1):3-64, 1996.
[6] J. Lin, *et al.*, "Communication-aware heterogeneous multiprocessor mapping for real-time streaming systems," *JSP*, 69(3), 2011.
[7] M. Kudlur, *et al.*, "Orchestrating the execution of stream programs on multicore platforms," in *ACM SIGPLAN*, 2008.
[8] A.H. Ghamarian, *et al.*, "Latency minimization for synchronous data flow graphs," in *DSD*, 2007.
[9] N. Vydyanathan, *et al.*, "Optimizing latency and throughput of application workflows on clusters," *Parallel Computing*, 37(10-11):694-712, 2010.
[10] C.-S. Lin, *at al.*, "Multi-objective exploitation of pipeline parallelism using clustering, replication and duplication in embedded multi-core systems," *JSA*, 59(10):10831094, 2013.
[11] S. Ranaweera, *et al.*, "Scheduling of periodic time critical applications for pipelined execution on heterogeneous systems," in *ICPP*, 2001.
[12] W. Kim, *et al.*, "Pipelined scheduling of functional HW/SW modules for platform-based SoC design," *ETRI Journal*, 27, 2005.
[13] H. Yang, *et al.*, "Pipelined data parallel task mapping/scheduling technique for MPSoC," in *DATE*, 2009.
[14] K. Chatha, *et al.*, "Hardware-software partitioning and pipelined scheduling of transformative applications," *IEEE TVLSI*, 10(3):193-208, 2002.
[15] Y. S. Chiu, *et al.*, "Pipeline schedule synthesis for real time streaming tasks with inter/intra instance precedence constraints," in *DATE*, 2011.
[16] S. Stuijk, "SDF3: SDF For Free," in *Application of Concurrency to System Design*, 2006.