



Multiprocessor Scheduling of a Multi-Mode Dataflow Graph Considering Mode Transition Delay

HANWOONG JUNG, Seoul National University

HYUNOK OH, Hanyang University

SOONHOI HA, Seoul National University

The Synchronous Data Flow (SDF) model is widely used for specifying signal processing or streaming applications. Since modern embedded applications become more complex with dynamic behavior changes at runtime, several extensions of the SDF model have been proposed to specify the dynamic behavior changes while preserving static analyzability of the SDF model. They assume that an application has a finite number of behaviors (or modes), and each behavior (mode) is represented by an SDF graph. They are classified as multi-mode dataflow models in this article. While there exist several scheduling techniques for multi-mode dataflow models, no one allows task migration between modes. By observing that the resource requirement can be additionally reduced if task migration is allowed, we propose a multiprocessor scheduling technique of a multi-mode dataflow graph considering task migration between modes. Based on a genetic algorithm, the proposed technique schedules all SDF graphs in all modes simultaneously to minimize the resource requirement. To satisfy the throughput constraint, the proposed technique calculates the actual throughput requirement of each mode and the output buffer size for tolerating throughput jitter. We compare the proposed technique with a method that analyzes SDF graphs in each execution mode separately, a method that does not allow task migration, and a method that does not allow mode-overlapped schedule for synthetic examples and five real applications: H.264 decoder, lane detection, vocoder, MP3 decoder, and printer pipeline.

CCS Concepts: • **Theory of computation** → *Streaming models*; • **Computer systems organization** → *Embedded software*;

Additional Key Words and Phrases: Synchronous dataflow, multi-mode dataflow, mode transition delay, task migration, mode-overlapped schedule, throughput requirement;

ACM Reference Format:

Hanwoong Jung, Hyunok Oh, and Soonhoi Ha. 2017. Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. *ACM Trans. Des. Autom. Electron. Syst.* 22, 2, Article 37 (January 2017), 25 pages.

DOI: <http://dx.doi.org/10.1145/2997645>

1. INTRODUCTION

Model-based design methodology is widely accepted for embedded system design since it enables us to cope with ever-increasing system complexity by maximizing the benefit of abstraction. As an algorithm specification model, this article adopts a coarse-grain

This research was supported by a grant to Bio-Mimetic Robot Research Center Funded by Defense Acquisition Program Administration and by the Agency for Defense Development (UD130070ID), Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1A2B3012662, 2013R1A1A1013384), and IT R&D program MKE/KEIT (No. 10041608, Embedded system Software for New-memory based Smart Device).

Authors' addresses: H. Jung and S. Ha (corresponding author), Department of Computer Science and Engineering, Seoul National University; emails: jhw7884@iris.snu.ac.kr, sha@snu.ac.kr; H. Oh, Department of Information System, Hanyang University; email: hoh@hanyang.ac.kr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1084-4309/2017/01-ART37 \$15.00

DOI: <http://dx.doi.org/10.1145/2997645>

dataflow model that is suitable for specifying signal processing or streaming applications. In a dataflow graph, a node presents a function module and an arc represents the flow of data samples (or tokens) through the First-In, First-Out (FIFO) channel between two end nodes. When a node is invoked, it consumes a specified number of data samples (called *sample rate*) from each input arc and produces a specified number of samples to each output arc. A node becomes executable when all input arcs have as many data samples as the specified sample rates. If the sample rate is a fixed integer number which does not change at runtime, then the dataflow graph is called a synchronous dataflow (SDF) graph [Lee and Messerschmitt 1987]. An SDF graph is called *consistent* when there exists a schedule of node executions that does not change the number of samples on all FIFO channels. Such a schedule that consists of the minimum number of node executions is called an *iteration* of the SDF graph.

For a given multiprocessor system, we need to determine the mapping of nodes to the processors and the execution order of mapped nodes on each processor. The static sample rate in the SDF model allows us to make the mapping and scheduling decision statically. From the static mapping and scheduling result for an SDF graph on a multiprocessor, we can estimate the performance and the resource requirement, which is very desirable for the design of embedded systems with tight real-time and resource constraints. If the implemented system follows the pre-determined mapping and execution order of nodes at runtime, then the system can be claimed to be “*correct by construction*.”

But the SDF model has a severe restriction to be used for modern embedded applications. It cannot express the dynamic behavior of an application, while modern embedded applications become more complex with dynamic behavior changes at runtime. For example, advanced video CODEC algorithms have several function modules that are conditionally invoked depending on the contents of the input frame. In addition, an application may have multiple implementations of the same algorithm to support various levels of quality of service.

To express such dynamic behavior of an application in the SDF model with keeping the static analysis capability, several extensions have been proposed to the SDF model, including Finite State Machine (FSM)-based scenario-aware dataflow (FSM-SADF) [Stuijk et al. 2011], parameterized SDF (PSDF) [Bhattacharya and Bhattacharyya 2001], Mode Transition Machine (MTM) SADF [Jung et al. 2014], mode-controlled dataflow (MCDF) [Moreira 2012], mode-aware dataflow (MADF) [Zhai 2015], and so on [Girault et al. 1999; Wiggers et al. 2008]. They all assume that an application has a finite number of behaviors (or modes) and each behavior (mode) can be represented by an SDF graph. We denote those Models of Computation (MoCs) as multi-mode dataflow (MMDF) graphs in this article and define a representative MMDF model that can be implemented by any specific extension.

In this article, we are interested in the multiprocessor scheduling of an MMDF graph that specifies a streaming application with a throughput constraint. In an MMDF graph, we assume that the mode transition is made at the iteration boundary while the mode switching decision can be made any time during the execution. In each mode, the associated SDF graph is scheduled statically to satisfy the throughput constraint. If the static schedule is followed at runtime, then output samples will be produced periodically. When mode switching occurs, however, we may have to pay extra overhead of mode transition, called *mode transition delay*, which affects the throughput performance of the application. Thus we propose an analytical formula to compute the mode transition delay and investigate how the mode transition delay affects the throughput performance when the schedule of each mode is given.

While there exist several techniques [Moreira 2012; Zhai 2015] to schedule an MMDF graph with considering the mode transition delay, they assume that each task is

mapped to the same processor throughout all modes. On the other hand, we propose a scheduling technique that does not have such an assumption, observing that the number of required processors can be reduced if a task can be mapped onto different processors among modes. Suppose that a task is mapped onto different processors between two modes. Then, we have to pay extra overhead to migrate the tasks, called *task migration delay*, which should be included in the computation of mode transition delay.

In summary, we propose a multiprocessor scheduling technique of an MMDF graph considering the mode transition delay conservatively. Our scheduling objective is to minimize the number of processors, while satisfying the throughput constraint. Experiment results show that the proposed scheduling approach provides better solutions than three existing approaches; the first approach is to determine the mapping and the scheduling of task graphs for each mode independently incurring excessive task migration between modes, the second is to disallow different mapping of tasks among modes, and the third is to make the mode transition in the blocking fashion meaning that the next mode starts its execution after the current mode finishes its execution in all processors.

The rest of this article is organized as follows. The next section gives a motivational example to clarify the problem addressed in this article and introduces the key idea of the proposed technique. Section 3 reviews the related work. The problem addressed in this article will be defined and formulated in Section 4. In Sections 5 and 6, the throughput requirement analysis technique and the proposed scheduling technique considering the mode transition delay are explained in detail, respectively. In Section 7, we discuss our experimental results and draw conclusions in Section 8.

2. MOTIVATIONAL EXAMPLE

2.1. Throughput Requirement Calculation Considering Mode Transition Delay

In a streaming application that requires the periodic output stream production, an output buffer is usually used as depicted in Figure 1(a). The display update frequency becomes the throughput constraint of the application. If the execution time of an application varies dynamically, then the time interval between output samples will vary even though the average throughput satisfies the throughput constraint. The output buffer is used to tolerate such variation to produce the periodic output stream to the display. The size of output buffer depends on the amount of variation.

Suppose that a streaming application has multiple modes of operation and each mode is specified by an SDF graph. For each mode of operation, we can find a static schedule of the associated SDF graph that satisfies the throughput requirement. If the static scheduling is followed at runtime, then the output samples will be produced periodically without variation of output intervals at each mode. When mode change occurs, however, the interval between two output samples may vary due to additional time delay during mode transition. Then the overall throughput may become smaller than the throughput constraint even if the throughput of each mode is no less than the throughput constraint, as illustrated in Figure 1(b).

Figure 1(b) shows two different schedules of an MMDF graph that consists of two different modes of operation. The arrows on an upper line represent the times when the system dequeues data from the output buffer periodically with the same rate as the throughput constraint. The arrows on a lower line tell when an MMDF application enqueues data to the output buffer. A number annotated on an arrow, $Buf.x$, denotes the number of data items in the output buffer after the access is completed. If the number becomes negative, then it means that buffer underflow occurs.

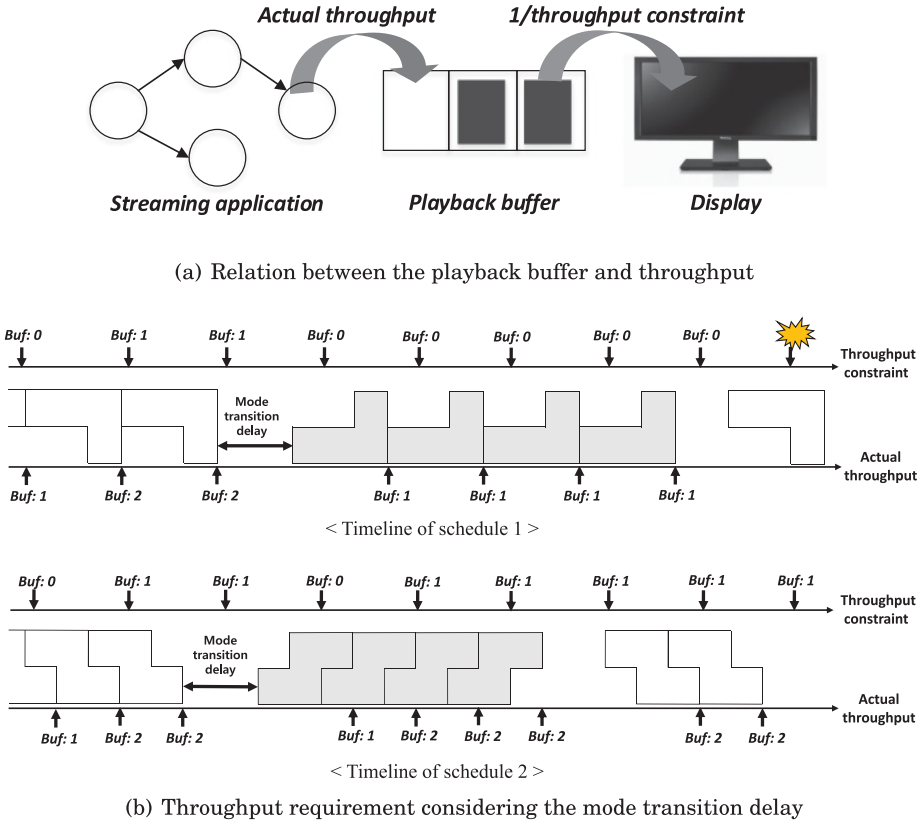


Fig. 1. Motivational example of throughput requirement calculation considering the mode transition delay.

In the case of schedule 1 in Figure 1(b), even though the schedule of each mode satisfies the throughput constraints, the throughput constraint is eventually violated since the mode transition delay is accumulated. To avoid this problem, we need to set the throughput constraint of each mode higher than that of the application as schedule 2 illustrates in the figure; it keeps the throughput constraint because it fills the output buffer faster than the throughput constraint. Therefore, we need to calculate the actual throughput requirement for each mode considering the mode transition delay in order to not violate the given constraint. Details will be discussed in Section 5.

2.2. Task Migration between Mode Transition

Figure 2(a) shows an MMDF graph example that consists of two modes: $M1$ and $M2$. We assume that the throughput constraint of the MMDF graph is given as $1/35$. Each execution mode of an MMDF graph is represented with an SDF graph. The execution time and sample rates of each node may vary depending on the execution mode. In mode $M1$, the execution times of nodes A , B , C , and D are 17, 13, 14, and 16, respectively, and, in mode $M2$, they are 12, 10, 8, and 10. The output sample rates of nodes B and C are unity in mode $M1$ while they are 3 in mode $M2$. Refer to Section 4 for the formal description of the MMDF model assumed in this article. Also, in this section, we only consider the task migration overhead as the mode transition delay to simply show the effect of task migration during mode transition.

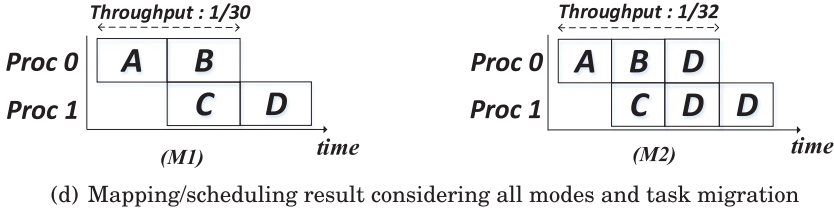
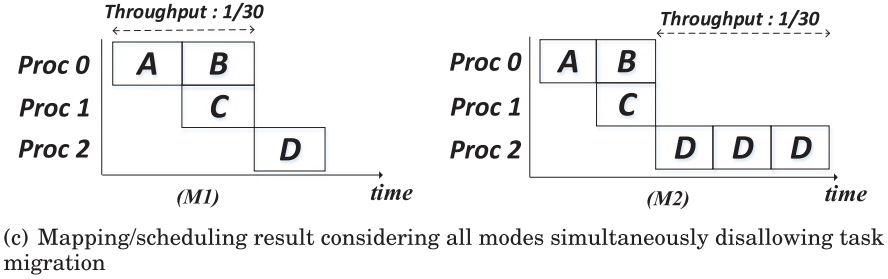
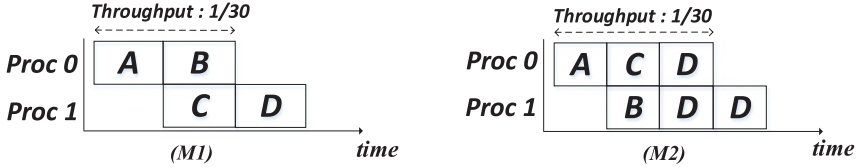
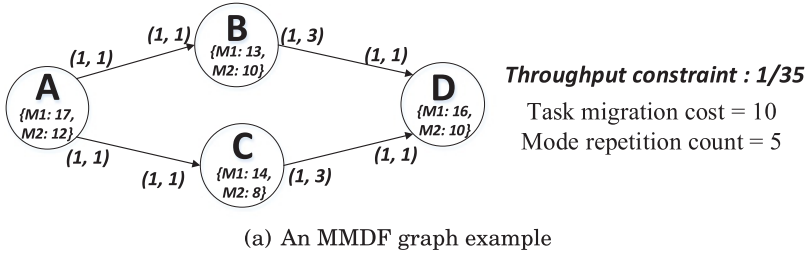


Fig. 2. Motivational example of task migration.

A naive approach to schedule an MMDF graph is to schedule an SDF graph in each mode independently with multiple objectives of resource minimization and throughput maximization. For example, for the given throughput constraint, we find an optimal mapping/scheduling result in each execution mode as shown in Figure 2(b). Since it does not consider mapping results in the other modes, a node may be mapped onto different processors between modes. Therefore, the mapping result requires task migration when the mode changes. In Figure 2(b), nodes *B*, *C*, and *D* will be migrated to other processors when the mode transition occurs.

Another approach to schedule an MMDF graph is to consider all modes simultaneously disallowing task migration [Stuijk et al. 2010; Moreira 2012]. Since the mapping is constrained in these approaches, the scheduling results generally require more processors than those that allow task migration. For instance, three processors are required to meet the given throughput constraint for the mapping/scheduling result

Table I. Comparison of Various MoCs Supporting Dynamic Behavior Specification

MoC	Multiprocessor scheduling	Mode transition delay	Task migration
FSM-SADF	Static mapping/scheduling	System reconfiguration overhead	Not allowed
MTM-SADF	Hybrid mapping (static/dynamic)	Run-time overhead of the RTM	Allowed
PSDF	Not supported	Not considered	Not allowed
MCDF	Static mapping/scheduling	Time interval between modes	Not allowed
VPDF	Not supported	Not consider	Not allowed
MADF	Real-time scheduling (fixed mapping)	Time interval between modes	Not allowed
BPDF	Not supported	Not considered	Not allowed
HDF	Not supported	Not considered	Not allowed
Proposed	Static mapping/scheduling	Time interval considering task migration	Allowed

without task migration as shown in Figure 2(c), while two processors are enough for the scheduling result with task migration in Figure 2(b). Since the objective of this article is to minimize the resource requirement under a given throughput constraint, the proposed approach allows task migration. Their approach is used as a reference technique for comparison with the proposed technique in experiments.

Consider the former approach that allows task migration in Figure 2(b). If the mode transition occurs frequently and the task migration overhead is non-negligible, then the given throughput constraint may not be satisfied. For instance, assume that the mode transition occurs every five iterations and the task migration overhead of each node is 10. In Figure 2(b), for every five iterations, a migration delay of 30 is additionally required because nodes *B*, *C*, and *D* should be migrated for mode transition. Then, the output buffer will be eventually empty, because the average throughput performance of the MMDF graph becomes lower than the throughput constraint.

Therefore, in this article, we propose another approach that schedules the SDF graphs of all modes simultaneously, allowing task migration among execution modes. Figure 2(d) shows a mapping and scheduling result produced by the proposed technique. It requires two processors and only 10 additional time units for task migration, which may satisfy the throughput requirement with proper output buffering. Throughput analysis considering task migration overhead will be discussed in Section 5.

3. RELATED WORK

There exist several extensions of the synchronous dataflow model that have been proposed to express the dynamic behavior of an application. Table I summarizes well-known dataflow-based extensions supporting dynamic behavior specification, compared with the proposed technique.

One of the most representative multi-mode dataflow models is the FSM-based SADF model [Stuijk et al. 2008] or FSM-SADF. In the FSM-SADF model, an application consists of multiple scenarios (modes) and each scenario is specified by an SDF graph. To specify multiple scenarios and their transitions, it defines a special control task called *detector* that has an FSM inside. The *detector* task sends the control information to the normal computation tasks that may change its behavior. For the FSM-SADF model, several techniques to statically analyze the timing behavior such as worst-case latency and throughput [Geilen and Stuijk 2010] have been proposed. Also, a binding-aware scenario graph [Stuijk et al. 2010] has been proposed to take into account the

resource constraint. And, in Damavandpeyma et al. [2013], it considers reconfiguration overhead for Dynamic Voltage Frequency Scaling (DVFS) as the mode transition delay. However, it only considers the worst-case performance analysis of the FSM-SADF graph for the given task mapping and requires inherently exponential time complexity for exact analysis. The authors of Stuijk et al. [2010] proposed a design flow that finds Pareto solutions among resource requirements under a given throughput constraint. But it does not allow task migration between different modes, and it constructs a static schedule for each mode independently.

As a similar model to the FSM-SADF, an MTM-SADF [Jung et al. 2014] has been proposed to specify application-level dynamism based on an SDF. Instead of an FSM, it uses a MTM, which is a simplified form of the FSM, to represent the mode transition. It proposes a hybrid task mapping technique with minimizing the overall energy consumption under the throughput constraints. Even though it allows task migration and considers task migration cost to evaluate the energy consumption of the runtime manager, it analyzes each SDF graph independently and does not consider how the mode transition delay has an effect on the throughput performance of an application.

PSDF [Bhattacharya and Bhattacharyya 2001] proposes a meta-modeling technique for runtime adaptation of parameters in a structured way. In the PSDF model, the dynamic behavior of a task is modeled by parameters, and the task behavior can change at the iteration boundary at runtime. Since the PSDF becomes an SDF graph at each iteration, the PSDF can be regarded as a multi-mode dataflow graph that may change modes every iteration. To the best of our knowledge, however, there is no research published for multiprocessor scheduling of the PSDF model.

Mode-Controlled Data Flow (MCDF) [Moreira 2012] is another extended model that can express the data-dependent functional behavior. The authors proposes temporal analysis techniques for a given sequence of mode changes. In addition, they propose a quasi-static multiprocessor scheduling technique under the given constraints. Based on static scheduling information of each mode, they model the *mode transition interval*, which denotes a time interval between two consecutive schedules. While the modeling method is similar to our proposed technique, it does not allow task migration between modes. Thus the mode transition delay in their work does not take into account the task migration delay.

Variable-Rate Data Flow (VRDF) [Wiggers et al. 2008] is proposed to allow variable port rates within a specified range, and Variable-Rate Phase Data Flow (VPDF) [Wiggers et al. 2011] is proposed to combine characteristics of VRDF and CSDF where each actor has a sequence of phases, and, for every phase, the number of firings can be parameterized. For these MoCs, buffer size analysis technique is proposed to satisfy the given timing and resource constraints, but it does not consider the scheduling problem.

Mode-Aware Data Flow (MADF) [Zhai 2015] has been proposed to support hard real-time scheduling for a multi-mode Cyclo-Static Data Flow (CSDF) model [Bilsen et al. 1995]. It combines the advantages of SADF and VPDF to specify application-level dynamism. Also, it proposes a Maximum-Overlap Offset (MOO) mode transition protocol to derive an efficient analysis for hard real-time scheduling of an MADF graph, allowing overlapped execution between modes, similarly to the proposed technique of Moreira [2012]. With this mode transition protocol, the temporal behavior of individual modes and during mode transitions can be analyzed independently. But it assumes that the mapping of tasks is given and fixed and does not allow task migration between modes.

Boolean Parametric Data Flow (BPDF) [Bebelis et al. 2013] supports change of port rates and graph topology at runtime using integer and Boolean parameters. In the BPDF model, integer parameters are used to change port rates at each iteration, and Boolean parameters are used for activation and deactivation of edges to change graph

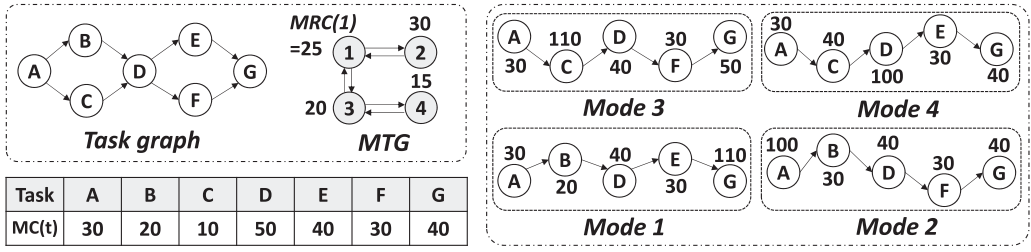


Fig. 3. An MMDF graph example.

topology. It constructs a parallel As Soon As Possible (ASAP) scheduling on a many-core platform in compile time. But it assumes that each task is mapped onto a separate processing element of the many-core.

Heterogeneous Data Flow (HDF) (or *-chart) [Girault et al. 1999] supports multi-mode applications through an FSM that executes an iteration of an SDF graph in each state. So an application is specified with a set of different SDF graphs combined with an FSM. To the best of our knowledge, there is no research result published for multiprocessor scheduling of the HDF model.

While various analysis and scheduling techniques have been proposed for those MoCs, no one considers task migration between modes in the application level. Jung et al. [2014] only supports task migration for the system-level task remapping due to workload variation or processor failures at runtime. In Lee et al. [2013], task migration is considered in the failure-aware task scheduling technique where an SDF graph is scheduled multiple times with different number of processors allocated, aiming to maximize the throughput with the allocated number of processors. When a processor fails in the middle of execution, it changes the schedule that uses the reduced number of processors by one. Then task migration occurs between two different schedules before and after processor failure. They try to minimize the migration cost between two SDF schedules. This method is similar to the base method that will be used for comparison in this article: schedule each mode separately and find the best processor-to-processor mapping (or processor renaming) in order to minimize the migration cost.

In summary, to the best of our knowledge, this article is the first work that proposes a multiprocessor scheduling technique of an MMDF graph allowing task migration between modes and analyzes the throughput requirement considering the mode transition delay.

4. PROBLEM DEFINITION

The MMDF model assumed in this article is not a specific model but a generic model encompassing existing similar models such as FSM-SADF [Stuijk et al. 2008] and MTM-SADF [Jung et al. 2014]. In those models, mode transition is specified by an FSM and all modes are integrated into a single graph with varying configuration parameters. Figure 3 shows an MMDF graph example. We first define the MMDF model and the problem formally.

Application model: An MMDF graph is specified by a combination of a task graph and a mode transition graph (MTG), or $(T, C, D) \times MTG$, where

- MTG is specified by a tuple $(Mode, Trans)$, where $Mode$ is a finite set of modes and $Trans$ is a finite set of transitions. $Trans$ is specified as follows: $Trans = \{(m_p, m_n) | m_p \in Mode, m_n \in Mode\}$, where m_p denotes a previous mode and m_n denotes a next mode.

- T is a finite set of computational tasks, and T_m denotes a subset of T that contains tasks executed in mode m . Each task $t \in T$ has a set of ports P_t to send/receive data to/from other adjacent tasks. $P_t = IP_t \cup OP_t$, where IP_t is a set of input ports and OP_t is a set of output ports. For each port $\rho \in P_t$, it is assigned a fixed rate, $Rate(\rho, mode)$, in each execution mode. If a port rate is one for all modes, then it is omitted for simple illustration of figures. Then the graph becomes an SDF graph for each mode.
- C is a finite set of FIFO channels. A channel defines a one-to-one connection between two end ports. For each channel $c \in C$, $srcTask(c)$ and $dstTask(c)$ denote a source task and a destination task of channel c , respectively.
- D is a set of the number of initial tokens in all channels. $d_c(m) \in \{0\} \cup \mathbb{N}$ for $\forall d_c \in D$ is the number of initially stored tokens in the channel c in mode m .

Architecture model: a target architecture consists of a set of processing elements.

- PE is a set of processing elements. For each $p \in PE$ and $m \in Mode$, $Map(m, p) = \{t | t \in T \text{ where } t \text{ is mapped onto a processor } p \text{ in mode } m\}$.

Note that even though the proposed technique is applicable to heterogeneous multiprocessor systems, this article assumes a homogeneous multiprocessor system for simple explanation and implementation.

To analyze the scheduling performance of an MMDF graph, we assume profiling information is available as follows:

Profiling information

- Worst-case execution time ($WCET$) for each task $t \in T$ and $m \in Mode$ is given as $WCET(t, m, p)$ for each processing element $p \in PE$ of the target architecture. In Figure 3, the $WCET$ of a node is annotated in each mode. For example, $WCET$ of node A is 30 in modes 1, 3, and 4, and 100 in mode 2.
- For each $m \in Mode$, we are given a minimum number of iterations that the application stays at the mode, which is denoted by $MRC(m)$, where MRC stands for the minimum repetition count. As MRC becomes smaller, the mode transition occurs more frequently. A mode is associated with an MRC value as shown in Figure 3 where MRC is 25, 30, 20, and 15 in modes 1, 2, 3, and 4, respectively.
- For each $t \in T$, task migration cost is given by $MC(t)$. If the system is a distributed memory system, then the migration cost will include the time overhead of moving the code and the context of a task between two processors. If it is a shared memory system, then the migration cost will be small as cache miss penalty for the first reference (called *cold miss penalty*) of the task. A table in Figure 3 shows the MC value of each task. For instance, the MC of node A is 30.

Execution semantic

- (*Intra-mode operation*) In each iteration, the task graph starts from the beginning. For the consistent execution, there will be no change in the channel buffer state before and after an iteration.
- (*Mode switching decision*) To make a mode switching decision, a special type of task is usually designated in existing approaches (e.g., detector actor in Stuijk et al. [2008], control actor in Moreira [2012]). Similarly, we designate a specific task, called the *mode decision task* (denoted by $t_{mode} \in T$), that determines the current mode of operation in the MMDF graph. It implies that the set of predecessor tasks of the mode decision task is common to all modes. There is at most one mode decision task in the MMDF graph. Note that the mode switching decision can be made any time during an iteration depending on the scheduling order of tasks. If t_{mode} is not

specified, then the mode changing decision is assumed to be made before the start of an iteration.

- (*Mode transition*) Because we allow overlapped execution of two modes, we assume that mode transition is made at the iteration boundary: When a mode change is requested, the previous iteration of the SDF graph continues to finish its execution. In the case where the mode decision task is executed more than once in an iteration, we assume that the mode switch decision is made at the last instance of the mode decision task.
- (*Inter-mode consistency*) The channel buffer state is preserved when a mode transition occurs. Initial tokens of a channel in the current mode are transferred to the initial tokens of a channel in the next mode. It incurs additional scheduling dependency between task executions in two modes: In case there exist initial tokens in channels, the start of the next mode should be delayed until the same number of tokens are produced in the current mode.

We assume that similar models introduced in Section 3 can be transformed to the MMDF model if it can keep the aforementioned construction rules and execution semantics. With those application and architecture models, profiling information and execution semantic, the problem addressed in this article is summarized as follows:

PROBLEM: Find a mapping and scheduling result of an MMDF graph that satisfies the given throughput constraint

minimize. the number of required processors

subject to. the overall throughput performance of the MMDF graph should be higher than the given throughput constraint.

The proposed MMDF scheduling framework is based on a genetic algorithm. So, it needs to evaluate all candidate solutions in every iteration. How to evaluate whether a given mapping and scheduling result of an MMDF graph satisfies the given throughput constraint will be explained in the next section.

5. THROUGHPUT REQUIREMENT ANALYSIS

Figure 4(a) shows a simple MMDF graph example that consists of two modes of operation. For each mode, a static schedule that satisfies the given throughput constraint is constructed as shown in Figure 4(b). If a mode transition does not occur, then the schedule of the current mode will be repeated and the output samples will be produced periodically. The period is equal to the inverse of the throughput performance, which is denoted as the initiation interval (II) in the figure. If a mode transition occurs, then production of the next output sample will be delayed because of interference between two different schedules of the previous mode and the next mode. Figure 4(c) shows a timeline of the execution of the MMDF graph.

Even though the static schedule of each mode satisfies the given throughput constraint, the overall throughput performance of the MMDF graph may not satisfy the throughput constraint because of the mode transition delay. To satisfy the throughput constraint, we may need to tighten the throughput requirement for each mode of operation. To understand how the mode transition delay affects the throughput requirement, we first explain how to compute the mode transition delay.

5.1. Mode Transition Delay

The mode transition delay between two modes is defined how the time interval between the last output production time of the previous mode (t_1 in Figure 4(c)) and the first output production time of the next mode (t_2 in Figure 4(c)) is larger than the initiation

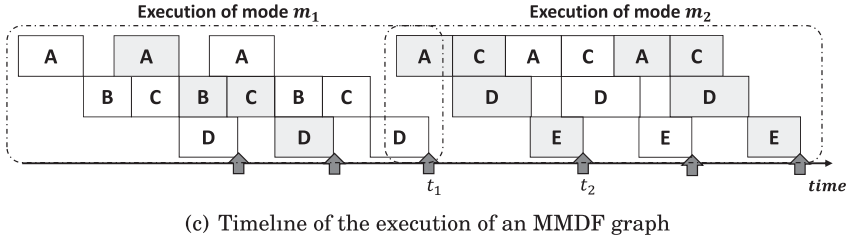
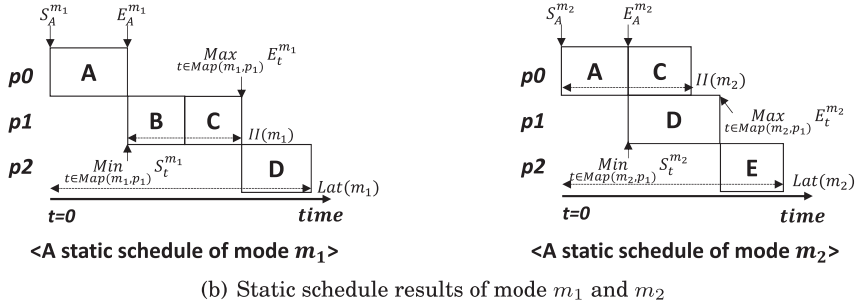
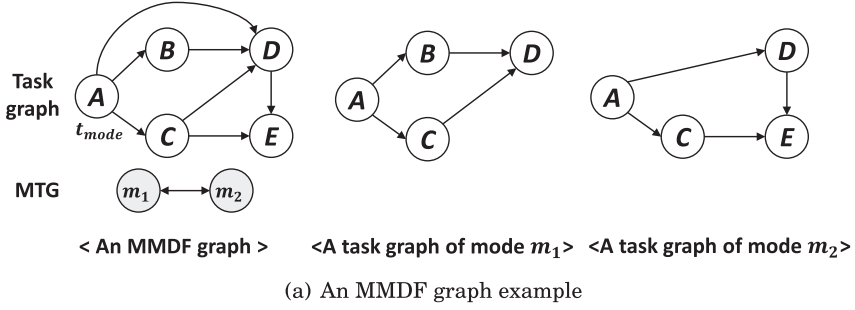
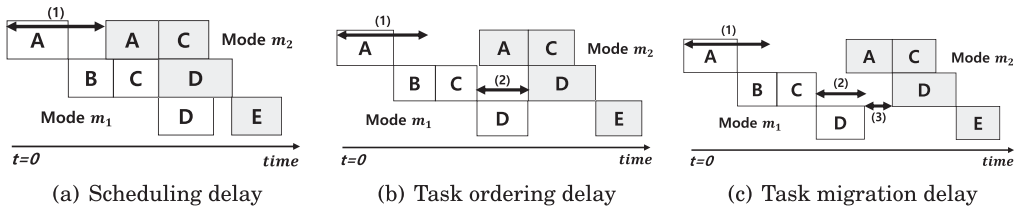


Fig. 4. An MMDF graph example and static scheduling result of each mode.


 Fig. 5. Mode transition delay between static schedules of mode m_1 and m_2 in Figure 4(b).

interval of the next mode. Suppose that the last iteration of the previous mode is started at $t = 0$. First we formulate the start offset (χ) of the first iteration of the next mode. The start offset (χ) is determined by the following three factors:

(1) *Scheduling delay* (D_{sched}): To guarantee consistent execution of the MMDF graph, we need to shift the start time of the subsequent mode. The time interval, denoted by (1) in Figure 5(a), illustrates the scheduling delay between modes m_1 and m_2 of Figure 4.

There are three factors that determine the start time of the next mode. The first factor is the time delay (D_{proc}) to keep the temporal property of the static schedule of

the next mode. The second factor (D_{mode}) is the finish time of the last instance of the mode decision task in the previous mode since the next mode may start afterwards. The third factor (D_{data}) accounts for the data dependency between two modes. In case there exist initial tokens in channels, the start of the next mode should be delayed until the same number of tokens are produced in the previous mode. The scheduling delay can be calculated as follows:

Definition 5.1 (Scheduling Delay from Mode m_i to Mode m_j).

$$D_{sched}^{m_i m_j} = \max(D_{proc}^{m_i m_j}, D_{mode}^{m_i m_j}, D_{data}^{m_i m_j}), \quad (1)$$

$$D_{proc}^{m_i m_j} = \max_{\substack{p \in PE, \\ Map(m_i, p) \neq \emptyset \\ \wedge Map(m_j, p) \neq \emptyset}} \begin{cases} \max_{t \in Map(m_i, p)} E_t^{m_i} - \min_{t \in Map(m_j, p)} S_t^{m_j} & \text{if } \max_{t \in Map(m_i, p)} E_t^{m_i} - \min_{t \in Map(m_j, p)} S_t^{m_j} \geq 0, \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$D_{mode}^{m_i m_j} = E_{t_{mode}}^{m_i}, \quad (3)$$

$$D_{data}^{m_i m_j} = \max_{\substack{c \in C, \\ d_c(m_j) > 0}} \begin{cases} E_{srcTask(c)}^{m_i} - S_{dstTask(c)}^{m_j} & \text{if } E_{srcTask(c)}^{m_i} - S_{dstTask(c)}^{m_j} > 0, \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

where $E_t^{m_i}$ denotes the end time of task t in mode m_i and $S_t^{m_j}$ denotes the start time of task t in mode m_j . In Figure 4(b), the start time and the end time of a task are depicted.

(2) *Task ordering delay (D_{order})*: Because the proposed technique allows task migration between modes, a task can be mapped onto different processors in each mode. So two consecutive executions of the same task should not be overlapped or inverted during mode change. Therefore, we need to guarantee that a task in the next mode can start only after the task finishes its execution in the current mode. In Figure 5(a), two executions of task D are overlapped between modes. Thus the execution of the next mode should be delayed by the task ordering delay denoted by (2) in Figure 5(b). In case of task C , even though it is mapped onto different processors in each mode, no overlapping occurs as shown in Figure 5(a). So task C does not incur additional time delay. The task ordering delay (D_{order}) can be formulated as follows:

Definition 5.2 (Task Ordering Delay from Mode m_i to Mode m_j).

$$D_{order}^{m_i m_j} = \begin{cases} \max_{t \in T_{m_i} \cap T_{m_j}} E_t^{m_i} - (S_t^{m_j} + D_{sched}^{m_i m_j}) & \text{if } \max_{t \in T_{m_i} \cap T_{m_j}} E_t^{m_i} - (S_t^{m_j} + D_{sched}^{m_i m_j}) > 0. \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

(3) *Task migration delay (D_{mig})*: Tasks that are mapped onto different processors between modes should be migrated during the time interval between the end time in the previous mode and the start time in the next mode. If the time interval is not long enough to migrate the task, then additional time delay will be required. In Figure 5(b), task D should be migrated to other processor after the end of execution in the previous mode. However, there is not enough of a time interval for task migration. So additional time delay is needed, which is the task migration delay denoted by (3) in Figure 5(c). In the case of task C , additional time delay is not required because there already exists enough time for task migration. The task migration delay (D_{mig}) can be formulated as follows:

Definition 5.3 (Task Migration Delay from Mode m_i to Mode m_j).

$$D_{mig}^{m_i m_j} = \max_{\substack{t \in T_{m_i} \cap T_{m_j}, \\ proc(t, m_i) \neq proc(t, m_j)}} MC(t) - (S_t^{m_j} + D_{sched}^{m_i m_j} + D_{order}^{m_i m_j} - E_t^{m_i}) \quad (6)$$

$$\text{if } S_t^{m_j} + D_{\text{sched}}^{m_i m_j} + D_{\text{order}}^{m_i m_j} - E_t^{m_i} < MC(t),$$

where, $\text{proc}(t, m_i)$ denotes a processor that task t is mapped onto in mode m_i .

Summing up all three types of delay mentioned above, we compute the start offset of the next mode as follows:

Definition 5.4 (Start Offset of Mode m_j in the Case of Mode Transition $m_i \rightarrow m_j$).

$$\chi^{m_i m_j} = D_{\text{sched}}^{m_i m_j} + D_{\text{order}}^{m_i m_j} + D_{\text{mig}}^{m_i m_j}. \quad (7)$$

Note that the mode transition delay between two modes is defined whereby the time interval between the last output production time of the previous mode and the first output production time of the next mode is larger than the initiation interval of the next mode. Since the output production time of each mode equals the latency of the static schedule, the mode transition delay can be formulated as follows:

Definition 5.5 (Mode Transition Delay from Mode m_i to Mode m_j).

$$\text{TransDelay}(m_i, m_j) = \text{Lat}(m_j) + \chi^{m_i m_j} - \text{Lat}(m_i) - II(m_j), \quad (8)$$

where $\text{Lat}(m_j)$ represents the latency of mode m_j and $II(m_j)$ represents the initiation interval of mode m_j .

Note that if $\text{Lat}(m_j) + \chi^{m_i m_j} - \text{Lat}(m_i) \leq II(m_j)$, then $\text{TransDelay}(m_i, m_j)$ will be smaller than zero. It means that the time interval of the output production times during a mode transition can be shorter than the output production time interval of the next mode ($II(m_j)$).

The mode transition delay will be used to determine the required output buffer size and throughput requirement of an MMDF graph for the given throughput constraint. Details will be explained in the following section.

5.2. Buffer Size Determination

As discussed in Section 2.1, an output buffer is adopted to produce data samples periodically. Since the mode transition delay causes the jitter of output production in an MMDF application, the output buffer should be large enough to provide the data samples during mode transitions. The required output buffer size depends on the maximum mode transition delay and the throughput difference between the input stream and the output stream in the buffer.

To determine the buffer size, we compute the arrival curves of the input and the output streams in the buffer. The arrival curve of a stream informs the number of arriving (or departing) samples (y -axis) within a time interval (x -axis) as shown in Figure 6 [Thiele et al. 2000]. For conservative estimation, we use the maximum arrival curve for the output stream and the minimum arrival curve for the input stream.

In Figure 1(a), task *Display* dequeues data from the output buffer periodically with satisfying the throughput constraint, which is depicted as the output curve (gray solid line) in Figure 6. The slope of the output curve is equal to the throughput constraint. The black solid line represents the minimum arrival curve of the input stream that presents the number of generated samples to the output buffer. The buffer size is computed based on the minimum repetition count (*MRC*), the inverse of the throughput, and the maximum mode transition delay among all possible transition scenarios to the mode. In each mode, we compute the buffer size and then choose the maximum buffer size in all modes.

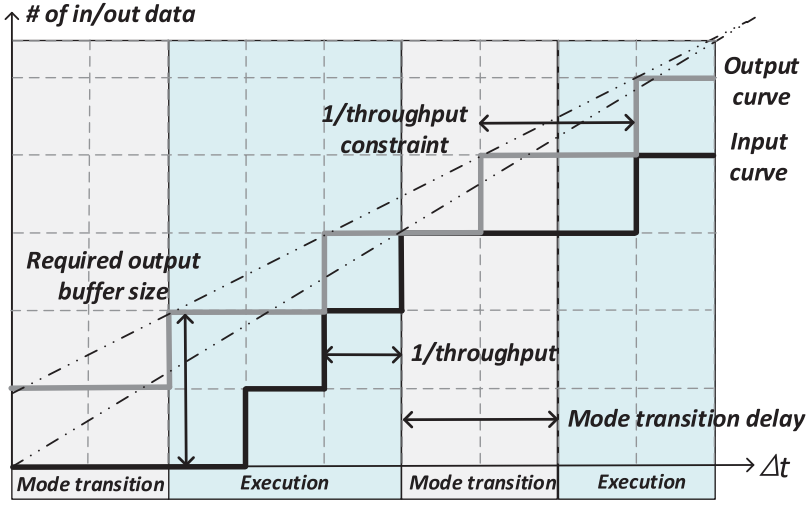


Fig. 6. Arrival curves for input and output streams in the output buffer.

The maximum mode transition delay to mode m is computed as following:

Definition 5.6 (Worst-case Mode Transition Delay to Mode m).

$$MaxTransDelay(m) = \max_{\substack{\forall (m_i, m_j) \in Trans, \\ m_j = m}} TransDelay(m_i, m_j). \quad (9)$$

Note that since the slope of the curve depends on the mode transition delay, the mode repetition count, and the throughput performance of the MMDF schedule, the buffer size is determined after constructing an MMDF schedule meeting the throughput constraints in all modes.

From the arrival curves, we obtain the minimum output buffer size, which is the maximum difference between the curves in every time interval (Δt). If the overall throughput constraint is satisfied, then the output buffer size is computed as follows.

THEOREM 5.7 (OUTPUT BUFFER SIZE). *The minimum size of the output buffer to satisfy the given throughput constraint ($ThrConst$) is decided by the following equation:*

$$Output\ buffer\ size = \lceil MaxInterval_{overall} \times ThrConst \rceil \quad (10)$$

$$where\ MaxInterval_{overall} = \max_{\substack{\forall (m_i, m_j) \\ \in Trans}} TransDelay(m_i, m_j) + II(m_j).$$

PROOF. The buffer size is determined by the maximum distance between the input and the output curves, which is illustrated in Figure 6. Because the proposed technique uses a higher throughput requirement than the throughput constraint when constructing a static schedule of each mode, the slope of a tangent line of the input arrival curve during execution should be larger than that of the output curve. Consider another tangent line that connects the starting point of mode transition and the ending position of the current mode execution, which is shown with a double-dotted line in Figure 6. The slope of this tangent line cannot be smaller than that of the output curve in order to keep the buffer size finite. If it is smaller, then the gap between two tangent lines will increase unboundedly if we apply the same mode transition repeatedly. By the construction rule of the arrival curves, the distance between the output curve and the input curve decreases as the time window increases. Therefore, the maximum distance

between two curves occurs is obtained just before the first jump of the input curve. Therefore,

$$\begin{aligned} \text{Output buffer size} &= \left\lceil \text{MaxInterval}_{\text{overall}} \div \frac{1}{\text{ThrConst}} \right\rceil \\ &= \lceil \text{MaxInterval}_{\text{overall}} \times \text{ThrConst} \rceil. \quad \square \end{aligned} \quad (11)$$

5.3. Throughput Requirement Analysis

The overall throughput performance of the MMDF graph depends on the mode transition delay as well as how frequently mode transition occurs. Since mode transition is triggered by an internal/external events at runtime, it may not be possible to know the mode transition scenario at compile time. Even though the exact mode transition scenario is not known, we assume that the minimum number of iterations is given as a part of the input information. Based on this information, we draw the input arrival curve of Figure 6 and estimate the buffer size conservatively.

Now we compute the throughput requirement of each mode. For conservative estimation, the input curve should be steeper than the output curve in all modes in Figure 6. The throughput requirement in each mode can be formulated as follows:

THEOREM 5.8 (THROUGHPUT REQUIREMENT). *The throughput requirement in mode m , denoted as $\text{ThrRequire}(m)$, is formulated as follows:*

$$\text{ThrRequire}(m) = \frac{\text{ThrConst} \times \text{MRC}(m)}{\text{MRC}(m) - (\text{MaxTransDelay}(m) \times \text{ThrConst})}. \quad (12)$$

PROOF.

$$\text{The slope of input curve} = \frac{\text{MRC}(m)}{\text{MaxTransDelay}(m) + 1/\text{ThrRequire}(m) \times \text{MRC}(m)}. \quad (13)$$

$$\text{the slope of output curve} = \frac{1}{1/\text{ThrConst}} = \text{ThrConst}. \quad (14)$$

Since the slope of input curve should not be smaller than that of the output curve,

$$\frac{\text{MRC}(m)}{\text{MaxTransDelay}(m) + 1/\text{ThrRequire}(m) \times \text{MRC}(m)} \geq \text{ThrConst}, \quad (15)$$

$$\text{ThrRequire}(m) \geq \frac{\text{MRC}(m)}{\text{MRC}(m)/\text{ThrConst} - \text{MaxTransDelay}(m)}. \quad \square \quad (16)$$

If the throughput performance of each mode is higher than the throughput requirement calculated by Theorem 5.2, then the MMDF graph will satisfy the throughput constraint. Note that the throughput requirement is a conservative bound. In the case where we know the exact scenario of mode transitions, the computed buffer size and the throughput requirement will be tight bounds.

6. PROPOSED MMDF SCHEDULING FRAMEWORK

6.1. Optimization Problem

The proposed technique is to find a static schedule of each mode cooperatively in a single optimization framework based on a genetic algorithm. We aim to minimize the resource requirement of an MMDF graph while satisfying the given throughput constraint. As discussed in the previous section, the throughput requirement of each mode should be computed considering the mode transition delay. On the other hand, depending on the throughput requirement of each mode, resource requirement may vary. As the

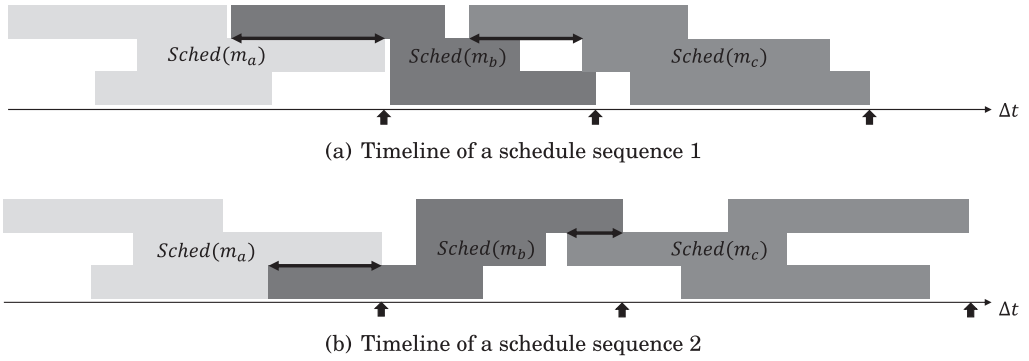


Fig. 7. Overlapped schedule sequences among different modes ($m_a \rightarrow m_b \rightarrow m_c$).

throughput requirement becomes tighter, more processing elements are likely to be required to satisfy the throughput requirement. Therefore the mode transition delay is an important factor that affects the resource requirement of the MMDF graph.

As stated in Definitions 5.4 and 5.5, the mode transition delay is influenced by three types of delay (D_{sched} , D_{order} , and D_{mig}) as well as the individual schedule of each mode. Therefore, to minimize the mode transition delay, we need to consider those delays when constructing the schedule of each mode cooperatively.

Figure 7 shows how the overlapped scheduling among different modes has an effect on the overall throughput performance of an MMDF graph. Even though the throughput performance of each mode is the same as shown in Figures 7(a) and 7(b), the overall throughput performance of the case of Figure 7(a) is better than the case of Figure 7(b), because the schedules in Figure 7(a) can be overlapped more than the case of Figure 7(b). So the static schedule of each mode should be constructed considering the schedules of the other modes.

Multiprocessor scheduling of a dataflow graph is a well-known NP-hard problem. Our MMDF scheduling problem is much harder since an MMDF graph consists of a set of modes and each mode is specified by an SDF graph. Also we need to consider the mode transition delay and compute the throughput requirement of each mode dynamically. To tackle this problem, we adopt a meta-heuristic based on a genetic algorithm to find an approximate solution.

6.2. GA Configuration

The overall GA procedure of the proposed framework is shown in Figure 8.

Initialization and Selection: Since a task (or node) can be mapped to different processors in modes, each task is regarded as a unit of mapping in each mode. The chromosome for GA is configured as shown in Figure 9. A chromosome is a set of mapping for each execution mode. Each gene of the chromosome represents to which processor a task in each execution mode is mapped. Chromosomes of the initial population are randomly generated and selected for crossover and mutation. The number of selection is a configurable parameter of the GA framework.

Crossover and Mutation: Crossover and mutation operations are applied to genes of each mode separately. As we explained in Section 4, the predecessor tasks of the mode decision task t_{mode} are common to all modes, so can be regarded as mode independent. We do not change the mapping of those tasks among all modes.

Local optimization: In order to help the convergence of evolutionary process, a local optimization step is performed before the evaluation step. For the local optimization,

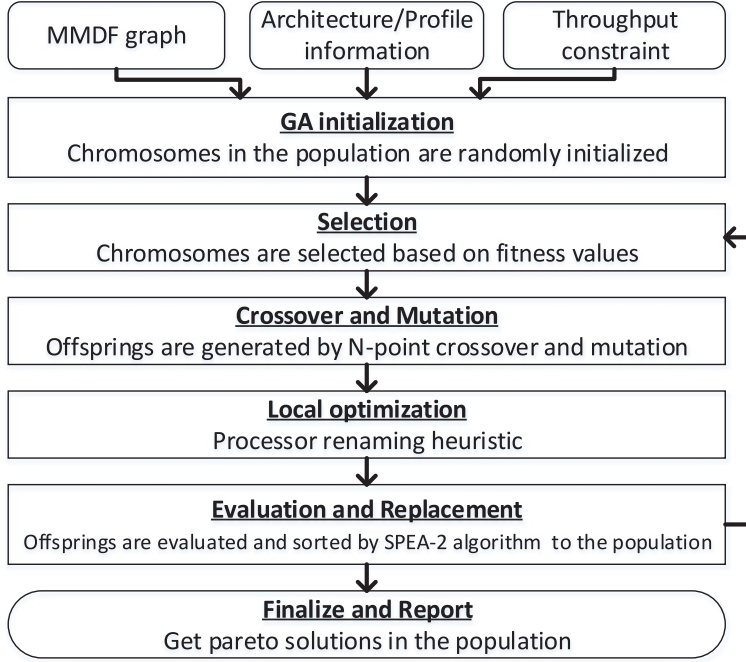


Fig. 8. The overall GA framework.

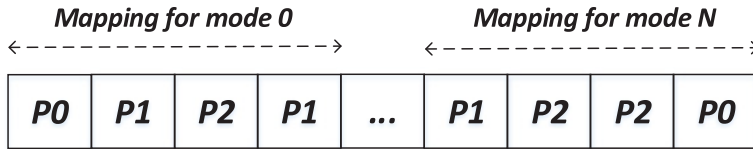


Fig. 9. Chromosome structure.

we devise a processor renaming heuristic that changes the processor id in each mode to reduce the migration cost. The details will be explained later.

Evaluation and Replacement: In this step, we apply a list scheduling heuristic to find a static task schedule in each mode, based on the mapping information given by each chromosome. Once we construct a static schedule, we evaluate the fitness value of each offspring and check whether the throughput constraint is satisfied. The fitness function will be described in the next section. Chromosomes in the population are sorted by their fitness values and poor chromosomes are eliminated.

6.3. Fitness Function

The objective of the MMDF scheduling is to minimize the number of required processors. The required number of processors is defined as the maximum number of used processors in all modes.

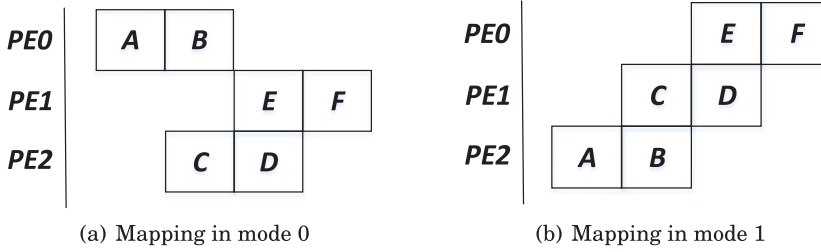


Fig. 10. Without processor renaming, every task should be migrated when mode transition occurs. If $PE0$ in mode 0 is renamed to $PE2$ in mode 1, $PE1$ to $PE0$, and $PE2$ to $PE1$, then no task migration is required.

Definition 6.1 (*The Number of Processors for an MMDF Graph*).

$$\begin{aligned} \text{The number of processor} &= \max_{m \in \text{Mode}} |\text{Proc}_m| \\ \text{where } \text{Proc}_m &= \{p \in PE \mid \text{Map}(m, p) \neq \emptyset\}. \end{aligned} \quad (17)$$

Since the large mode transition delay will degrade the throughput performance and more processors are likely to be required to meet the given throughput constraint, the mode transition delay including task migration overhead is considered to evaluate the number of required processors. And the GA framework also aims to minimize the overall task migration cost as the secondary objective. The reduction of task migration will save energy consumption of the system and reduce the network traffic in an Network On Chip (NOC) architecture. Therefore it is very desirable to reduce the total task migration cost (or delay) in an MMDF graph considering all mode transition scenarios; the total task migration cost of an MMDF graph is defined as follows:

Definition 6.2 (*Total Task Migration Cost*).

$$\begin{aligned} \text{MigCost}_{\text{total}} &= \sum_{(m_i, m_j) \in \text{Trans}} \text{MigCost}(m_i, m_j) \\ \text{where } \text{MigCost}(m_i, m_j) &= \sum_{p \in PE} \sum_{t \in \{\text{Map}(m_j, p) - \text{Map}(m_i, p)\}} \text{MC}(t). \end{aligned} \quad (18)$$

We sum up the migration cost of all possible migration scenarios that are defined by the *MTG*. For each transition in the *MTG*, we accumulate the migration cost of all tasks that are mapped to different processors after the mode transition.

6.4. Local Optimization Technique

Figure 10 shows a motivational example for local optimization, where two modes have different task mappings defined in the chromosome and a mode transition from mode 0 to mode 1 occurs. In the mapping result, all tasks should be migrated. However, since this article assumes a homogeneous multiprocessor system, it is possible to rename the processor id in each mode, which is called processor renaming. If $PE0$ in mode 0 is renamed to $PE2$ in mode 1, then tasks A and B do not need to be migrated. Similarly, if $PE1$ in mode 0 is renamed to $PE0$ in mode 1, and $PE2$ to $PE1$, then no task migration is required. Without the processor renaming technique, good solutions such as Figure 10 will be evaluated as poor solutions due to high migration delay, which seriously hinders the convergence of GA.

The time complexity of the processor renaming algorithm is given as P^M , where P denotes the number of processors and M is the number of mode transition scenarios.

ALGORITHM 1: Processor Renaming Heuristic

```

for all mode transition scenarios do
  prev  $\leftarrow$  mapping information of src mode of the transition;
  next  $\leftarrow$  mapping information of dst mode of the transition;
  for all mapping information of each processor (pId) in prev do
    for all mapping information of each processor (nId) in next do
      similarity  $\leftarrow$  check similarity between prev[pId] & next[nId];
      if (similarity  $\geq$  maxSimilarity) then
        maxSimilarity  $\leftarrow$  similarity;
        swapProcId  $\leftarrow$  nId;
      end
    end
    change mapping of next between pId & swapProcId;
  end
end

```

Table II. Configuration of the GA Framework

Population size	100
μ and λ	100
Probabilities of crossover/mutation	0.9
Maximum generations	30000

Therefore, we devise a simple greedy processor renaming heuristic as shown in Algorithm 1 to reduce the time complexity. In the proposed heuristic, the time complexity becomes $O(P^2 \times M)$. Note that processor renaming is only applicable for homogeneous processor systems.

The heuristic measures the similarity between processors. The similarity between processors is defined by how many tasks are mapped on both processors in common.

Definition 6.3 (Similarity Between Processors).

$$\text{For}(m_i, m_j) \in \text{Trans}, \text{Similarity}(p_k, p_l) = |\text{Map}(m_i, p_k) \cap \text{Map}(m_j, p_l)|. \quad (19)$$

For each mode transition, processors in the next mode are renamed to the processors in the previous mode with the maximum similarity. Even though the proposed heuristic does not consider all possible processor renaming scenarios and does not provide the optimal renaming result, it reduces the time complexity significantly while generating good quality solutions as confirmed by experimental results.

7. EXPERIMENTAL RESULTS

To prove the viability of the proposed framework, we experiment with five synthetic examples and five real applications: H.264 decoder, lane detection, vocoder [Zhai 2015], MP3 decoder [Yang 2012], and printer pipeline [Yang 2012]. All experiments have been performed on an Intel Core i7-4790K 4.00GHz machine with 8GB main memory. Internal parameters of the GA framework are set as shown in Table II. μ and λ denote the number of parents and offspring, respectively.

Figure 11 shows task graphs which are used for experiments. Task graphs of MP3 decoder and printer pipeline applications in Yang [2012] are omitted due to lack of space, but the task graph and the profiling information of each application are same with Yang [2012]. For all graphs in Figure 11, $\text{Rate}(\rho, \text{mode})$ for each port $\rho \in P_t$ is one if it is not specified. For the task graph of vocoder application in Zhai [2015], we reduce

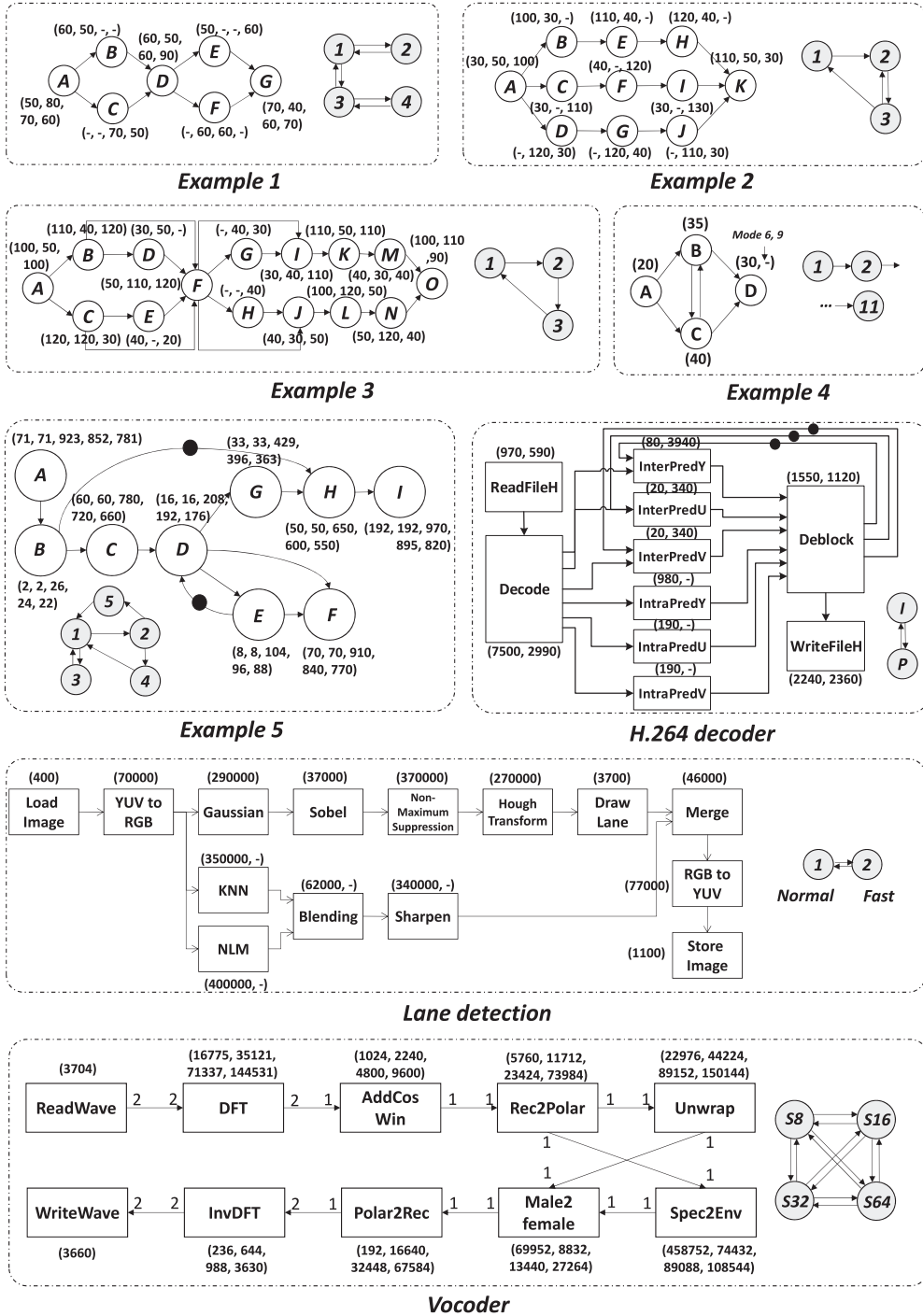


Fig. 11. MMDF graph examples used in experiments.

Table III. Four Approaches Used in Experiments

Approach	Consideration of other modes	Task migration	Mode-overlapped schedule
<i>Base</i>	Do not consider	Allow	Allow
<i>Fixed</i>	Consider	Do not allow	Allow
<i>Blocked</i>	Consider	Allow	Do not allow
<i>Proposed</i>	Consider	Allow	Allow

the number of invocations for specific tasks (from *AddCosWin* to *Polar2Rec*) from 128 to 2 by clustering, so the given WCETs of those tasks in Zhai [2015] are multiplied by 64. Also, we allow that each instance of the same node can be mapped onto different processors for data parallelism. The numbers above or under the tasks in Figure 11 indicate the $WCET(t, m, p)$ in each mode. In the case where the $WCET(t, m, p)$ of a task is constant in all modes, a single number is denoted. For synthetic examples, the WCET of each task is set to an arbitrary value, and the WCET of each task in the H.264 decoder and lane detection applications is set to profiled data in *us* unit. Also, for all examples, we assume that the source task of each task graph is the mode decision task t_{mode} that determines the mode of the current iteration.

7.1. MMDF Scheduling Technique

We compared the proposed technique with three different approaches listed in Table III. The first approach, *Base*, schedules SDF graphs independently and performs the processor renaming heuristic. It is an iterative algorithm. For each mode, it constructs a set of pareto-optimal solutions that are optimized with throughput and the number of processors, using a genetic algorithm. Then it selects an initial schedule that satisfies the throughput constraint with the minimum number of processors for each mode. Based on the mapping/scheduling results, it performs the processor renaming heuristic and adjusts the throughput requirement as discussed in the previous section, considering the mode transition delay incurred by the initial schedules. If a schedule does not satisfy the calculated throughput requirement, then it is replaced with another schedule that uses one more processor. Unless all scheduling results satisfy the newly adjusted throughput requirement in all modes, it repeats the mapping/scheduling with the new adjusted throughput requirement until the mapping/scheduling results satisfy the adjusted throughput requirement. Comparison with *Base* approach will show the reason why all modes should be scheduled simultaneously in the proposed approach.

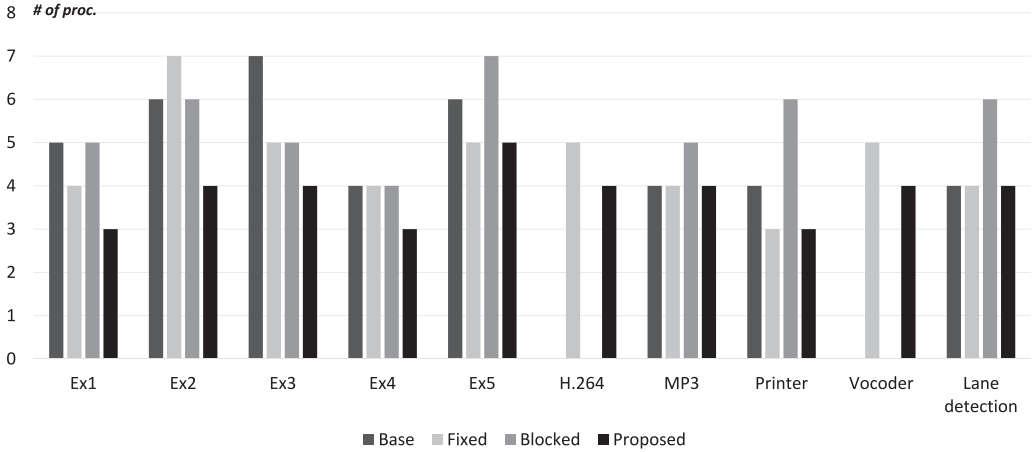
The second approach fixes task mapping in all modes disallowing task migration as the existing approaches usually assume. This technique is denoted as *Fixed*. The *Fixed* technique is implemented in the same GA framework as the proposed framework with disallowing task migration. Comparison with *Fixed* approach will show how task migration helps reduction of the resource requirement.

The third approach assumes that mode transition and task migration is performed in a *blocking* fashion. This technique is denoted as *Blocked*. The *Blocked* technique is also implemented in the same GA framework as the proposed framework but uses a different formulation for the start offset of the next mode. Instead of Definition 5.4, $Lat(prev_mode) + MigCost(prev_mode, next_mode)$ is used for χ . Comparison with the *Blocked* approach will show how much benefit is expected by allowing mode-overlapped schedules for the resource requirement.

For all configurations in Table IV, we compared four techniques: *Base*, *Fixed*, *Blocked*, and *Proposed*. We assume that the minimum repetition count (*MRC*) for all modes in each example is set to the given value in Table IV except the H.264 decoder application, since the mode transition pattern of the H.264 decoder is known and fixed (e.g.,

Table IV. Configurations for Experiments

	$MRC(m)$	$MC(t)$	$ThrConst$
<i>Example 1</i>	$MRC(m) = 10$	$MC(t) = 10$	1/130 iteration/time-unit
<i>Example 2</i>	$MRC(m) = 10$	$MC(t) = 10$	1/170 iteration/time-unit
<i>Example 3</i>	$MRC(m) = 5$	$MC(t) = 10$	1/300 iteration/time-unit
<i>Example 4</i>	$MRC(m) = 5$	$MC(t) = 50$	1/80 iteration/time-unit
<i>Example 5</i>	$MRC(m) = 3$	$MC(t) = 50$	1/2000 iteration/time-unit
<i>H.264 decoder</i>	$MRC(I) = 1$ $MRC(P) = 4$	$MC(t) = 10$	1/5600 iteration/us
<i>Lane detection</i>	$MRC(m) = 5$	$MC(t) = 10$	1/700000 iteration/us
<i>Vocoder</i>	$MRC(m) = 5$	$MC(t) = 10000$	1/480000 iteration/cycle
<i>MP3 decoder</i>	$MRC(m) = 3$	$MC(t) = 10000$	1/5000000 iteration/time-unit
<i>Printer pipeline</i>	$MRC(m) = 3$	$MC(t) = 50$	1/1600 iteration/time-unit

Fig. 12. Comparison results in terms of the number of processors: *Base*, *Fixed*, *Blocked*, and *Proposed*.

I-P-P-P-P-I-P-P-P-...). Throughput constraints are set arbitrarily with considering the WCET of tasks. In the synthetic examples, vocoder, MP3 decoder, and printer receiver applications, the task migration cost is fixed to $MC(t)$ for all tasks. In H.264 decoder and lane detection applications, however, $MC(t)$ is scaled based on the actual task code size for all $t \in T$: The task migration cost of a task is computed as the product of $MC(t)$ values in Table IV.

Figure 12 shows the experimental results for all applications. The y-axis indicates the number of required processors. The results show that the *Proposed* approach requires no more processors than the other approaches in all applications. It is observed that the *Blocked* approach requires more processors than the other approaches that allow overlapped schedules during mode transition. In the cases of H.264 decoder and vocoder applications, it could not find feasible solutions. This means that the *blocking* scheme degrades the overall throughput performance of an MMDF graph due to high mode transition delay.

Similarly to the *Blocked* approach, the *Base* approach requires more processors than the *Proposed* approaches. Because it constructs a static schedule of each mode without

Table V. Experimental Results of *Proposed* Approach

	$\max_{m \in Mode} MaxTransDelay(m)$	$\min_{m \in Mode} ThrRequire(m)$	Output buffer size	$MigCost_{total}$
<i>Example 1</i>	0 time unit	1/130	2	20 time unit
<i>Example 2</i>	20 time unit	1/168	2	110 time unit
<i>Example 3</i>	80 time unit	1/284	2	40 time unit
<i>Example 4</i>	10 time unit	1/78	2	50 time unit
<i>Example 5</i>	2,622 time unit	1/1126	2	1,200 time unit
<i>H.264 decoder</i>	660 μs	1/5435	2	280 μs
<i>Lane detection</i>	183,000 μs	1/663400	1	4500 μs
<i>Vocoder</i>	88,175 cycles	1/462364.8	2	340,000 cycles
<i>MP3 decoder</i>	1,412,947 time unit	1/4529017.6	2	340,000 time unit
<i>Printer pipeline</i>	804 time unit	1/1332	2	1,000 time unit

considering other modes, there is less of a chance of overlapping between the schedules during the mode transition. Therefore, the mode transition delay is likely to be higher than the case of *Proposed* approach. It also could not find feasible solutions in cases of H.264 decoder and vocoder applications. Even though the *Fixed* approach allows a mode-overlapped schedule and constructs schedules of all modes simultaneously, it requires more processors than the *Proposed* approach in many cases. This is because it does not allow task migration among modes.

In the lane detection application, there exists a dominant mode in which all tasks in an MMDF graph are executed. Since the dominant mode creates the critical path in all modes, if the mapping and scheduling result satisfies the throughput constraint in the dominant mode, then results in the other modes automatically satisfy the throughput constraint. Hence, *Fixed*, *Base*, and *Proposed* approaches produce the same results for the application.

Table V presents the detailed experimental results from the *Proposed* approach in Figure 12. The table shows that the throughput which an application should satisfy becomes tighter than the given throughput constraint in Table IV due to the mode transition delay. The table also presents the total task migration cost and the required output buffer sizes for benchmark applications.

7.2. Scalability of the Proposed Framework

Because the proposed framework is based on the genetic algorithm, its convergence speed depends on the size of solution space. As shown in Figure 9, the size of the solution space depends on the number of nodes and modes. So we perform experiments for different configurations of these factors. Figure 13 shows the experimental results on the scalability of the proposed framework for synthetic examples. The results show that the number of nodes more contributes to the convergence speed than the number of modes.

8. CONCLUSION

In this article, we address the multiprocessor scheduling problem of an MMDF graph allowing task migration with non-negligible mode transition delay. An MMDF graph has a finite set of modes, and each mode is specified by an SDF graph. We observe that the mode transition delay should be considered in many streaming applications in which the mode transition occurs frequently, in order to satisfy the throughput constraint. Thus we propose a mapping/scheduling framework based on a genetic algorithm that

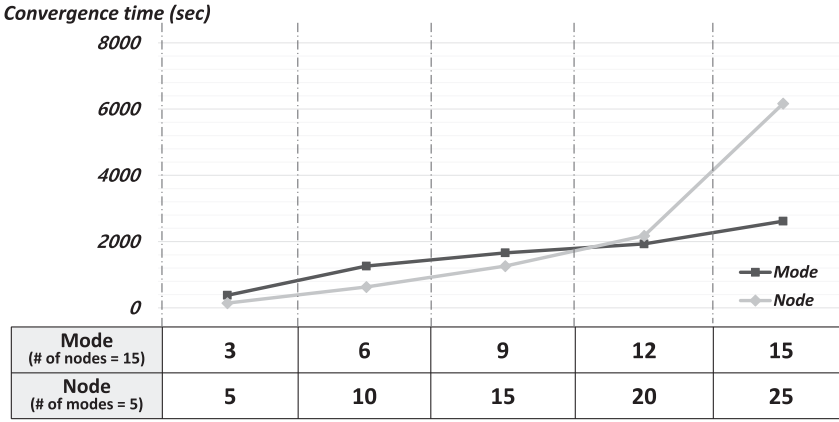


Fig. 13. Experimental result for the scalability property.

schedules all SDF graphs simultaneously to minimize the number of processors while keeping the throughput constraint. Also, we propose the formulations to compute the required buffer size and the required throughput performance of the MMDF graph to satisfy the given throughput constraint of the system by estimating the mode transition delay conservatively. To minimize the resource requirement, the proposed framework finds the maximally overlapped schedule that minimizes the mode transition delay. To show the viability of the proposed technique, we compare the proposed technique with three other approaches with some synthetic examples and five real applications. Experimental results confirm the superiority of the proposed technique over the other approaches.

REFERENCES

- Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *Proceedings of the 11th ACM International Conference on Embedded Software (EMSOFT'13)*. IEEE Press, Piscataway, NJ, Article 3, 10 pages. <http://dl.acm.org/citation.cfm?id=2555754.2555757>
- Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Trans. Sign. Process.* 49, 10 (Oct. 2001), 2408–2421. DOI: <http://dx.doi.org/10.1109/78.950795>
- Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. 1995. Cyclo-static data flow. In *Proceedings of the 1995 International Conference on Acoustics, Speech, and Signal Processing, 1995 (ICASSP'95)*, Vol. 5. 3255–3258. DOI: <http://dx.doi.org/10.1109/ICASSP.1995.479579>
- Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. 2013. Throughput-constrained DVFS for scenario-aware dataflow graphs. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. 175–184. DOI: <http://dx.doi.org/10.1109/RTAS.2013.6531090>
- Marc Geilen and Sander Stuijk. 2010. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'10)*. ACM, New York, NY, 125–134. DOI: <http://dx.doi.org/10.1145/1878961.1878985>
- Alain Girault, Bilung Lee, and Edward A. Lee. 1999. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 18, 6 (Jun 1999), 742–760. DOI: <http://dx.doi.org/10.1109/43.766725>
- Hanwoong Jung, Chanhee Lee, Shin haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. 2014. Dynamic behavior specification and dynamic mapping for real-time embedded systems: HOPES approach. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 135 (Apr. 2014), 26 pages. DOI: <http://dx.doi.org/10.1145/2584658>

- Chanhee Lee, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. 2013. Failure-aware task scheduling of synchronous data flow graphs under real-time constraints. *J. Sign. Process. Syst.* 73, 2 (2013), 201–212. DOI: <http://dx.doi.org/10.1007/s11265-013-0753-3>
- Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept. 1987), 1235–1245. DOI: <http://dx.doi.org/10.1109/PROC.1987.13876>
- Orlando Moreira. 2012. *Temporal Analysis and Scheduling of Hard Real-time Radios Running on a Multi-processor*. Ph.D. thesis. Technische Universiteit Eindhoven.
- Sander Stuijk, Marc Geilen, and Twan Basten. 2010. A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD'10)*. 548–555. DOI: <http://dx.doi.org/10.1109/DSD.2010.31>
- Sander Stuijk, Marc Geilen, Bart D. Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Proceedings of the 2011 International Conference on Embedded Computer Systems (SAMOS'11)*. 404–411. DOI: <http://dx.doi.org/10.1109/SAMOS.2011.6045491>
- Sander Stuijk, Amirhossein Ghamarian, Bart D. Theelen, Marc Geilen, and Twan Basten. 2008. *FSM-based SADF*. Technical Report. Citeseer.
- Lothar Thiele, Samarjit Chakrabort, and Martin Naedele. 2000. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems, 2000. (ISCAS'00)*, Vol. 4. 101–104. DOI: <http://dx.doi.org/10.1109/ISCAS.2000.858698>
- Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. 2008. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)* 183–194. DOI: <http://dx.doi.org/10.1109/RTAS.2008.10>
- Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. 2011. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Trans. Embed. Comput. Syst.* 10, 2, Article 17 (January 2011), 59 pages. DOI: <http://dx.doi.org/10.1145/1880050.1880053>
- Yang Yang. 2012. *Exploring Resource/performance Trade-offs for Streaming Applications on Embedded Multiprocessors*. Ph.D. Dissertation. Ph.D. thesis, Eindhoven University of Technology, Eindhoven.
- Jiali T. Zhai. 2015. *Adaptive Streaming Applications: Analysis and Implementation Models*. Ph.D. Dissertation. Leiden Embedded Research Center, Faculty of Science (LERC), Leiden Institute of Advanced Computer Science (LIACS), Leiden University.

Received January 2016; revised July 2016; accepted September 2016