# Improving multiprocessor voltage and frequency scaling for dynamic, throughput-constrained applications

Chen, L.

*Award date:*
2013

# Improving multiprocessor voltage and frequency scaling for dynamic, throughput-constrained applications

**Author :** **Le Chen**
**Supervisor:** **Dr. Ir. Sander Stuijk**
**Morteza Damavandpeyma**

**Date: October 8,2013**

## Abstract

Streaming applications with dynamic behaviors mapped onto multiprocessor platform can be effectively modeled using the scenario-aware dataflow (SADF) Model-of-Computation. Many streaming applications must perform their own tasks within a time deadline, or reach a certain value of throughput in order to assure their quality-of-service. Furthermore, the energy consumption of such applications on devices with limited battery power should be as low as possible. Dynamic voltage and frequency scaling (DVFS) is a software-controlled technique which can be used to lower the energy consumption by lowering the frequency and voltage of a processor.

As the starting point of this project, one existing technique proposed in [1] is taken, which selects a suitable multiprocessor DVFS point for each scenario of a dynamic application described by an SADF. Compared to the technique presented in [2], it solves the problem faster, and reduces energy consumption. However, the result is still sub-optimal, so in this project, further optimizations are made to improve it. After a design-time optimization, run-time optimizations are also implemented by utilizing the available slacks. The experimental results show that the new technique can reduce the energy consumption further while still providing timing guarantees.

# Table of Contents

# 1 Introduction

## 1.1 Background

Nowadays multiprocessor systems-on-chip (MPSoCs) are widely used in order to meet the growing computational demands and strict timing constraints in embedded systems. Besides these two requirements, energy consumption has also emerged as an important design consideration. To address this last design challenge, several low-energy techniques can be used in an embedded system ranging from hardware to software techniques. Compared with hardware low-energy techniques, such as clock/power gating and multiple voltage islands, software-based techniques have advantages on flexibility and cost reduction, since software is relatively easy and cheap to be modified. Dynamic voltage and frequency scaling (DVFS) is one of the most prominent software-based low-energy techniques which is used to control the energy consumption by changing the frequency and voltage of a processor dynamically.

Many signal processing applications in the embedded domain are applied iteratively on infinite input streams. They are often designed as hard real-time streaming applications that run on a multiprocessor platform. Moreover, these multiprocessor platforms are battery oriented devices in most cases. A design objective is to minimize the energy consumption in order to achieve a longer battery service time. So in this thesis, two types of requirements will be focused: timing requirements and low energy consumption.

The dataflow Model-of-Computation (MoC) is a suitable MoC to capture the iterative process performed in dynamic streaming applications. Synchronous dataflow (SDF) is a dataflow MoC that can capture mapping decisions and resource requirements and that can be used to analyze the timing behavior of applications mapped onto multiprocessor platforms. SDF is a static model and cannot capture the dynamic behavior found in many streaming applications. However, many streaming applications contain dynamic behaviors depending on their current state or input data. For example, the execution time of each function unit of an H263 decoder can be different depending on the block types in different input frames. In order to describe the dynamic behaviors, the scenario-aware data flow (SADF) MoC can be used. An SADF graph consists of a set of scenarios, each described with their own SDF graph, and a finite state machine that is used to specify the possible scenario sequences. If we can determine a suitable voltage and frequency setting for each scenario while satisfying the timing constraint, the energy consumption can be reduced.

There are many papers about determining proper frequency and voltage settings for each application scenario. In [1], an algorithm to select, at design-time, a suitable multiprocessor DVFS point for each scenario of a dynamic application is introduced. The algorithm provides strict timing guarantees to the application while trying to minimize the energy consumption of the platform. The DVFS controller proposed in [1] computes, at design-time, for each scenario the frequency settings of all processors in the platform. At run-time, the active scenario is

detected and the processors are set to the corresponding frequencies. In this algorithm, a parametric SADF model is used to accommodate the processor clock cycle periods. Instead of using concrete values, linear expressions provide the actor execution times in terms of some parameters. By identifying and resolving the longest cycle that limits the throughput, which is called the critical timing cycle, a throughput-constrained solution can be found. One novel part in [1], compared to related work [2], is that the switching cost of DVFS is considered. This makes the design-time exploration performed in [1] much more realistic.

There is another closely related approach to solve a similar problem for SADF graphs, this approach is presented in [2], by traversing the state-space of a so called power-aware SADF graph, a power management configuration that minimizes the energy consumption of a throughput-constrained application can be derived. This approach is based on a state-space exploration technique. Such techniques are known to scale poorly to larger problem sizes. Compared with the technique presented in [1], the processor frequency setting found using the technique from [2] are typically a little worse in terms of their energy consumption. The intuition behind this difference is that algorithm [1] considers all iterations involved in a critical timing cycle, i.e., the slack of one or multiple iterations can be utilized and the workload will be well balanced across multiple iterations. As a result, the processor frequencies might be lower than those found using the technique from [2].

## 1.2 Problem Statement

This thesis focuses on designing a DVFS controller. It is based on the algorithm proposed in [1], which we will refer to as the RTAS'13 algorithm.

The RTAS'13 algorithm determines a suitable multiprocessor DVFS point for each scenario of a dynamic application. It tries to lower the frequencies of all processors as much as possible in order to save energy while ensuring the throughput constraint.

Lowering the frequency of a processor too much may lead to deadline misses. In addition, lowering the frequency on one processor may save energy on this processor but the total energy consumption may increase since other processors might be forced to run at a high frequency in order to meet the throughput requirements of the applications. These tradeoffs make it a challenging task to find the optimal frequency assignment for all processors in the platform.

Even though the algorithm in [1] has many advantages, experiments show that this technique provides sub-optimal solutions. Therefore, some techniques can be designed to derive better frequency settings for an application that result in a lower energy consumption while still providing timing guarantees. In this thesis, several optimizations will be made to get a better DVFS controller.

## 1.3 Report Outline

The reminder of this thesis is structured as follows. In Section 2, some preliminary knowledge is introduced to help understand data flow graphs, while Section 3 presents the principle of dynamic voltage and frequency scaling, and gives an introduction to some related work. In Section 4, the analysis and discussion about the RTAS'13 algorithm is introduced. Section 5 gives a detailed description about the design-time extensions, while Section 6 states the run-time refinement. In section 7 we explore the behavior and performance of the approach with the use of design-time extensions and run-time refinements, discusses the result of the integration of these improvements and draws a conclusion.

# 2 Data flow Preliminaries

In this project, data flow is taken as the model of computation. Data flow can describe applications for concurrent implementation on multiprocessors. In this section, the graph definitions will be introduced and several related data flow model used in this project will also be discussed.

Data flow graphs are directed graphs. A directed graph can be represented as $G= (V, E)$ where $V$ is a set of nodes and $E$ is a set of edges. Every edge is an ordered pair $(u, v)$ where $u$ and $v$ are two nodes. For an edge $(u, v)$, it starts from source node $u$ and ends at the destination node $v$. Edges represent First-In-First-Out (FIFO) queues while nodes represent computation blocks called actors. In data flow graph, a number of concurrent actors communicate through unidirectional FIFO channels. Data is transported in discrete chunks called tokens. An actor in a data flow graph can fire when it is activated by data availability.

## 2.1 Synchronous Dataflow

Synchronous dataflow (SDF) is one kind of static data flow graphs that allows design-time analysis for multiprocessor applications; it can capture mapping decisions and resource requirements.

SDF graphs are constructed by nodes and edges, which are called actors and channels. In a SDF graph, actors are corresponding to functionality while channels are used to show data dependencies (data edges) or execution orders (sequence edges). Every channel can carry an infinite number of tokens which are present on the edges at start time. Figure 2.1(a) shows an example SDF with 3 actors, 4 channels and 2 tokens. The data flow model has a well-defined firing rule, which is an actor can only fire when there are sufficient tokens on each of its input channels. When an actor is allowed to fire, it consumes a constant number of tokens from its inputs. These amounts are called rates. After finishing computation, it produces tokens to all its outputs. The rate determines how often actors have to fire with respect to each other such that the distribution of tokens over all channels is in balance. This property is captured in the **repetition vector** [28] of an SDF. A SDF graph that has a non-zero repetition vector is called consistent. The fixed rates allow SDFs to fire in a periodic form, which is called **iteration**. An SDF that is not consistent requires unbounded memory to execute or suffers from deadlock. In this project, only consistent and deadlock-free SDFGs are considered.

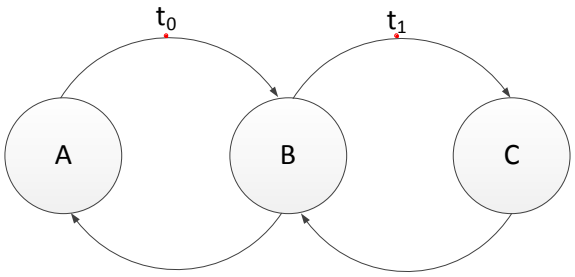
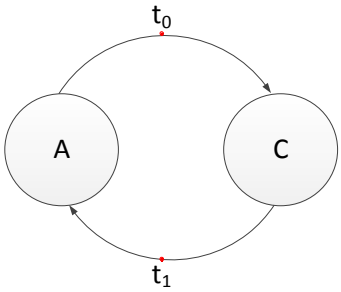



Figure 2. 1 (a)




F igure 2. 1 (b)

An actor can be associated with a certain value to represent its execution time and the time cost to read/write its input/output data. After finishing one iteration, the tokens restore to the initial token distribution. A token timestamp vector $\gamma_k(k \in \mathbb{N})$is used to specify the production time of tokens in the $k^{th}$ iteration of the graph. For each SDF, a characteristic Max-Plus matrix $G|_{n*n}$ exists that can be used to calculate $\gamma_k$. [13] In case of the example SDF graph shown in Figure 2.1(a), assume that the execution times of actor A, B, C are 30, 40, 50 time-units respectively. Using the technique presented in [14], the characteristic matrix for the SDF shown in Figure 2.1(a) is:

$$G = \begin{matrix} \\ t_0 \\ t_1 \end{matrix} \begin{matrix} \begin{matrix} t_0 & t_1 \end{matrix} \\ \begin{pmatrix} 70 & 120 \\ 40 & 90 \end{pmatrix} \end{matrix}$$

Matrix G specifies the minimum time distance from one token in this iteration to itself and other token(s) in the next iteration. For example, the shortest distance from $t_0$ in the $k^{th}$ iteration to $t_0$ in the $(k+1)^{th}$ iteration is 70 time-units.

When computing the matrix G and the initial token timestamp vector $r_0$, with the help of Max-Plus matrix multiplication $r_{k+1} = G \cdot r_k$, all the following token timestamp vectors $r_k$ can be determined. For example, assume $r_0 = \binom{0}{0}$, then $r_1$ can be calculated as follow:

$$r_1 = \begin{pmatrix} 70 & 120 \\ 40 & 90 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \max\{70+0, 120+0\} \\ \max\{40+0, 90+0\} \end{pmatrix} = \begin{pmatrix} 120 \\ 90 \end{pmatrix}$$

The characteristic Max-Plus matrix can be used to determine the throughput of an SDF graph [14]. In order to calculate the throughput, a Max-Plus automata graph (MPAG) should be built based on the characteristic Max-Plus matrix. In an MPAG, a node is created for each initial token in the SDF, and an edge with some weight, which depends on the characteristic Max-Plus matrix, is added to the MPAG. When the value of an element $G[i,j]$ is $-\infty$, meaning that there is no dependency from the $j^{th}$ token to the $i^{th}$token, then the edge can be deleted. A cycle which is limiting the throughput is called a critical cycle of the SDF which can be discovered by applying a maximum cycle mean (MCM) analysis on the MPAG. The corresponding MPAG for the SDF in Figure 2.1(a) is shown in Figure 2.2. The red edge with a value 90 is the critical cycle.

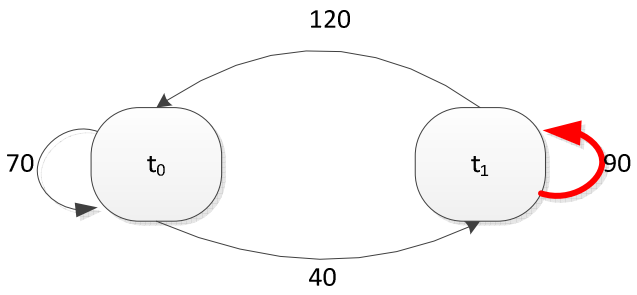



**Figure 2.2**

SDF enables simple analysis of the performance of an application. However, it is not suitable to model streaming applications with a dynamic behavior. In case of designing a system with predictable timing behavior, worst-case execution times are assigned to actors, in the situation that an application contains a lot of dynamism, the use of worst-case values could lead to over-dimensioning of the system for many run-time situations [5]. In order to capture the dynamic behavior, the scenario-aware data flow (SADF) MoC is used.

## 2.2 Scenario-aware dataflow

### 2.2.1 System Scenario

Scenario based design has been used for a long time in different areas, like human-computer interaction [17] or object oriented software engineering [18]. In case of the "4+1" view model of software architecture [21], a scenario is a set of interactions with the system integrating the models in the views and proving behavioral models inside the views. In embedded systems area, use-case scenarios are used in both hardware [19] and software design [20], focusing on the application functional and timing behaviors and on its interaction with the users and the environment, not on the resources required by an application to meet its constraints.

Another different type of scenario, so-called **system scenario**, is first explicitly identified and exploited in [16]. A system scenario is used to group system behaviors that are similar from a multi-dimensional cost perspective, such as resource requirements, delay, and energy consumption, in such a way that the system can be configured to exploit this cost similarity. At design-time, these scenarios are individually optimized. In the remainder of this report, we use the term scenarios exclusively to refer to system scenarios.

A H.263 decoder supports two different types of input frames, i.e., I and P frames. When a frame of type I is found, a total of 99 macro blocks must always be processed in the motion compensation part. While for the P frame, variable number of macro blocks need to be processed. The execution time is shown in Table 3.1. According Table 3.1, the dynamic behavior of H.263 decoder can be concluded in 9 different scenarios (combinations of frame type and number of macro blocks to decode) [5].

Different data flow models can be used to capture scenarios within a streaming application. In [15], the scenario-aware data flow MoC is proposed as a design time analyzable stochastic generalization of the SDF MoC which can capture the dynamism of streaming applications expressed by scenarios.

| Execution Time | | |
|---|---|---|
| VLD | $P_0$ | 0 |
| | others | 40 |
| IDCT | $P_0$ | 0 |
| | others | 17 |
| MC | $I, P_0$ | 0 |
| | $P_{30}$ | 90 |
| | $P_{40}$ | 145 |
| | $P_{50}$ | 190 |
| | $P_{60}$ | 235 |
| | $P_{70}$ | 265 |
| | $P_{80}$ | 310 |
| | $P_{99}$ | 390 |
| RC | $I$ | 350 |
| | $P_0$ | 0 |
| | $P_{30}, P_{40}, P_{50}$ | 250 |
| | $P_{60}$ | 200 |
| | $P_{70}, P_{80}, P_{90}$ | 320 |
| FD | All | 0 |

Table 3.1 H.263 decoder Execution time

### 2.2.3 Scenario-Aware Dataflow

SADF is a dataflow model of computation for design-time analysis of dynamic and signal processing applications. An SADF graph consists of a set of scenarios, each described with their own SDF graph. The scenario concept enables SADF to capture the variable behaviors of different processes in a streaming application. SADF utilizes a finite state machine to specify the possible scenario sequences


**Figure 3.1 Finite State Machine**

Figure 3.1 shows an example SADF with two scenarios $s_0$ and $s_1$. In this example, $s_0$ uses the scenario graph of Figure 2.1(a), $s_1$ uses the scenario graph of Figure 2.1 (b), and their execution times is shown in Table 3.2. When the state transfers to another state in the FSM, a scenario associated to this state is executed for one iteration. The initial tokens in the scenario graph capture the dependencies between subsequent iterations.

| | A | B | C |
|---|---|---|---|
| Scenario $s_0$ | 30 | 40 | 50 |
| Scenario $s_1$ | 40 | - | 40 |

**Table 3.2 Actor Execution Time**

As for the individual SDF in this example, the characteristic Max-Plus matrixes can be determined for each scenario. The corresponding matrices for each scenario $s_0$ and $s_1$ are:

$$G_{s_0} = \begin{pmatrix} 70 & 120 \\ 40 & 90 \end{pmatrix}, G_{s_1} = \begin{pmatrix} 80 & 40 \\ 40 & 80 \end{pmatrix}$$

Reference [4] provides a technique to combine the MPAGs of all the scenarios into a single MPAG. If there is a state transition from a state in the FSM associated with scenario N to a state associated with scenario M, an edge with weight $G_M[y,x]$ is added from node $N/t_x$ to node $M/t_y$ in the MPAG. If the weight is equal to $-\infty$, the edge can be removed. Using the MCM analysis on this MPAG, the critical cycle can be found. Figure 2.4 shows the MPAG for the example SADF in Figure 2.3, and the red edge is the critical cycle ($MCM = 90$). The critical cycle determines the throughput of this example SADF, which is equal to 1/90 iterations/time-unit.

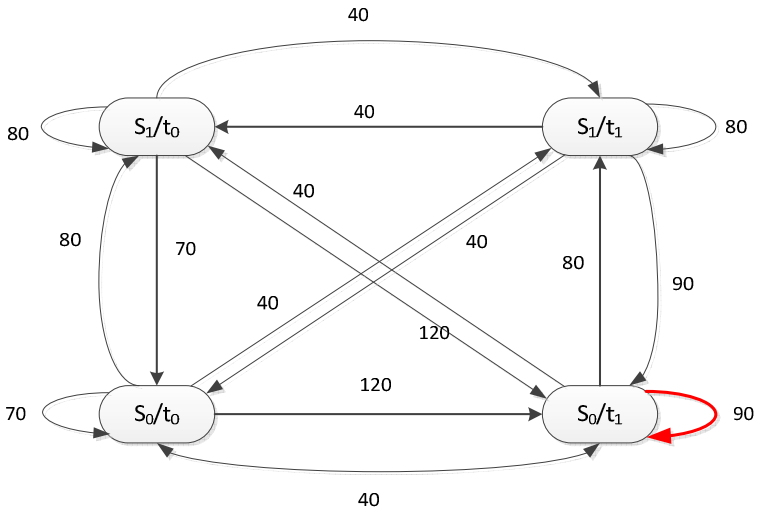



**Figure 2.4 Max-Plus Automata Graph**

## 2.3 Parametric SADF and throughput analysis

### 2.3.1 Parametric SADF

Throughput is an important criterion to determine the system performance. For example, the number of frames per second output by a video decoder should always be kept above a threshold to maintain the video quality. And the throughput threshold can be calculated according to the timing requirement. There are some design space exploration (DSE) techniques [7]-[9] related to throughput calculation to determine the performance of multiple solutions in the design space. However, the throughput analysis with fixed execution time can only provide limited information to improve the design decisions.

SADFs can capture the dynamism of applications efficiently by using different actor execution times in different scenarios. The actor execution time has a relationship to the properties of the platform onto which it is mapped, (i.e., frequency and voltage). Changing these properties, e.g. frequency level of the processors, can alter their execution times. As a result, the throughput will be changed.

A parametric SADF is identical to a common SADF except that it uses parameterized actor execution times. Instead they are a function of a set of parameters with some discrete values within an interval. The execution time of an actor depends linearly on the clock cycle period of the underlying processing element. Hence the variable execution time can be expressed by a linear expression of some parameters. These parameters are used to capture the frequency of the processing elements in different scenarios, e.g., $p_{i,j}$ expresses clock cycle period (i.e., inverse of frequency) of $j^{th}$ the processing element in the $i^{th}$ scenario.

For example, consider the SADF of Figure 2.4 in which actors A and B are mapped to processing element 1 and actor C is mapped to processing element 2. The execution times of all actors expressed in the parameters of the processing elements are shown in Table 2.3.

### 2.3.2 Symbolic throughput analysis

In Section 2.2, how to calculate the throughput based on a MPAG has already been explained when all the execution times are fixed. A parametric throughput analysis for the SADF model-of-computation based on MPAG analysis is proposed in [11]. The MPAG is extended to a symbolic MPAG in order to compute symbolic throughput expressions. First of all, a symbolic characteristic Max-Plus matrix for each scenario graph must be computed, then these matrices are combined to construct a symbolic MPAG using a similar method in Section 2.2.

For example, the symbolic characteristic Max-Plus matrices for scenario $s_0$ and $s_1$ are equal to: $G_{s_0} = \begin{pmatrix} 7p_{11} & 7p_{11}+5p_{12} \\ 4p_{11} & 4p_{11}+5p_{12} \end{pmatrix}$ $G_{s_1} = \begin{pmatrix} 4p_{21}+4p_{22} & 4p_{21} \\ 4p_{22} & 4p_{21}+4p_{22} \end{pmatrix}$

With the information of $G_{s_0}$ and $G_{s_1}$, the symbolic MPAG can be constructed, as shown in Figure 2.5. Then the symbolic MPAG can be evaluated for a set of concrete parameter values, which

results in a concrete MPAG. Using a relation between the edges in the concrete MPAG and symbolic MPAG, the critical cycle in the concrete MPAG can be translated into a symbolic cycle in the symbolic MPAG. This symbolic critical cycle is the inverse of the symbolic throughput.

| | A | B | C | Parametric range |
|---|---|---|---|---|
| Scenario $s_0$ | $3p_{1,1}$ | $4p_{1,1}$ | $5p_{1,2}$ | $SFS_1 = \{1,2,3,4,5\}$ |
| Scenario $s_1$ | $4p_{2,1}$ | - | $4p_{2,2}$ | $SFS_2 = \{1,2,3,4,5\}$ |

**Table 2.3 Actor Execution Time**

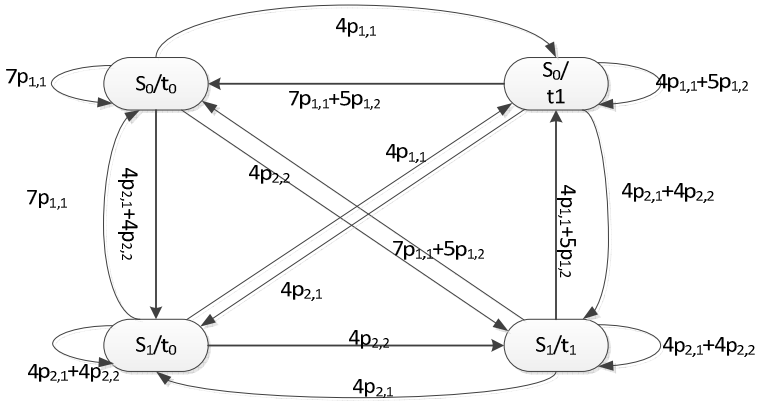



**Figure 2.5 Symbolic Max-Plus Automata Graph**

For example, when $p_{1,1} = 3, p_{1,2} = 4, p_{2,1} = 4, p_{2,2} = 2$, the cycle with the expression $4p_{1,1} + 5p_{1,2}$ , is the critical cycle. It lasts from the production time of token $t_1$in scenario $s_0$ in the previous iteration to its consumption time in the current iteration. Then the symbolic throughput expression is: $\frac{1}{4p_{1,1}+5p_{1,2}}$. With the expression of critical cycle and throughput, it is easy to satisfy the timing requirements by adjusting the values of these parameters.

# 3 State-of-the-art DVFS algorithms

Energy and power consumptions are important design concerns for contemporary microprocessors and embedded devices. Lowering the energy consumption can increase the battery life of portable devices.

In order to reduce the energy consumption while meeting the throughput requirements at the same time, many energy-aware techniques can be used. Dynamic voltage and frequency scaling (DVFS) is one of the most effective software-controlled energy reduction methods.

## 3.1 Dynamic voltage and frequency scaling

DVFS is being used in commercial processors across the entire computing range: from the embedded and mobile market up to the server market. DVFS offers opportunities to reduce energy/power consumption by adjusting both voltage and frequency levels of a system according to the changing characteristics of its workload. It is an effective way to reduce the energy consumption by providing "just-enough" performance by taking advantage of the available slack.

$$P = \alpha \cdot C_{eff} \cdot V^2 \cdot f$$

$$E = P \cdot T \propto V^2$$

$$V \propto f$$

The energy consumption depends quadratically on the supply voltage ($E \propto V_{DD}^2$), whereas the frequency also depends linearly on its supply voltage ($f \propto V_{DD}$). By reducing the processor frequency, the energy consumption can be reduced. For example, a task with workload W has to be finished before deadline D, half the value of voltage and frequency, as shown in Figure 3.1, can lead to 75% energy saving.

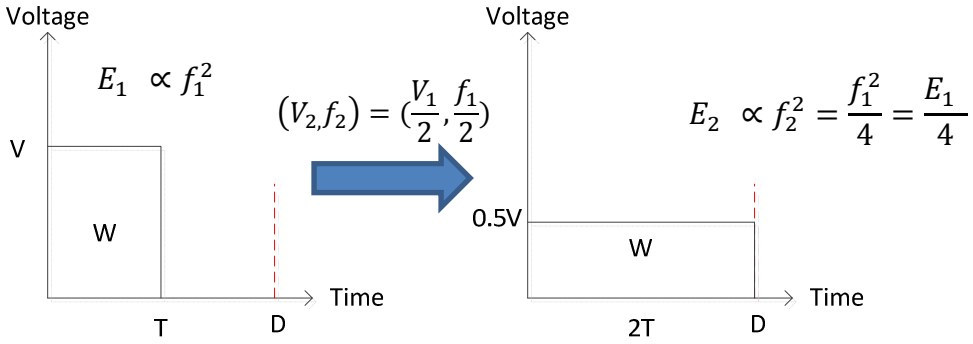



**Figure 3.1 Energy saving with DVFS**

When using a DVFS strategy, multiple Voltage Frequency Islands (VFI) are necessary for the MPSoCs [22], since different parts of multiprocessor platform need to operate at different clock frequency and supply voltage levels. Each island in a VFI system is locally clocked with an

independent voltage supply, while inter-island communication is orchestrated via a mixed-clock, mixed-voltage FIFOs [30]. Then the voltage of each island can be independently tuned to minimize the system power dissipation under performance requirements.

Considering the workload variation, the underlying processors scale their corresponding supply voltage and frequency at run-time to use the slacks to reduce the energy consumption [23]. From different viewing points, the DVFS policy can be justified from three angles [1]: (1) design policy; (2) application model; (3) solution granularity.

The design policy determines when to make the DVFS choices: at run time [26] [27] or at design time [1] [2]. Run-time frequency and voltage scaling is used to predict the workload and adjust the supply voltage and frequency of the underlying processing elements at run time. This type has become very attractive because it allows handling variable workloads in complex and dynamic utilization scenarios. But it also suffers from extra timing and energy overheads when reconfiguring the related processing elements (i.e., changing frequency and voltage). For design-time selection, the workload of an application is assumed to be known beforehand which can result in pessimistic resource allocations if the behavior of the application is not captured accurately. In this thesis, we make the DVFS choices at design-time.

Different models-of-computation can be used to model a system when devising a DVFS controller. When using a static model, such as SDF [24] or task graphs [25], a worst-case execution time is assigned to the tasks (actors) in the model. When using a dynamic model (i.e., SADF) [1] [2], execution time variation can be captured. Static models are simple but not able to capture the dynamisms found in dynamic applications.

When considering the solution granularity, it is important to distinguish how often the voltage and frequency switch happens. In a fine-grain solution [24] [25], frequency and voltage can be altered before the execution of each actor, while for a coarse-grained solution [1] [2], DVFS switches may only happen at the start of an iteration. The DVFS overhead limits its granularity. For instance, fine-grained solutions are beneficial only when the DVFS overhead is negligible.

## 3.2 Related work

Extensive research has been carried out on DVFS optimizations. In the following sub-section two related approaches will be introduced.

### 3.2.1 DVFS based on scenario-aware dataflow graph

Reference [1], which we refer to as the RTAS'13 algorithm, presents a multiprocessor frequency assignment technique for dynamic applications modeled by SADF graph. It analyzes SADF graphs, at design time, to devise a coarse-grain DVFS controller based on the throughput-constraint of an application. A DVFS switch may happen before executing an iteration (e.g., processing a video frame in a H.263 decoder) at run-time.

The SADF model is extended to parametric SADF in order to capture the change of the processor frequencies of the platform onto which the application is mapped. The critical cycles are identified with the help of a symbolic version of a Max-Plus automation graph analysis. At first, all the parameters are set to the lowest possible frequency which leads to the lowest energy mode. Then the critical cycles that are violating the throughput constraints are resolved repetitively by refining the parameters (i.e., increasing some processor frequencies). At the end, a set of frequency choices is made at design-time and these can be used at run-time to enable DVFS on iteration boundaries. (Details on this approach will be introduced in Chapter 4).

### 3.2.2 DVFS based on state-space exploration

Reference [2] addresses the same problem as reference [1], to devise a DVFS strategy for some application at design-time, but using a state-space based technique. A power mode is selected for each state of the application. Timestamps are used to distinguish between states and to capture the miss-aligned completion of the iterations on a platform with multiple processing elements. An initial state is selected as a starting point and leads to a new state or a recurrent state. For each possible scenario transition from that state a low-power mode that satisfies the time constraint for the upcoming iteration is created as desired power mode for that specific scenario switch. The exploration is stopped if all the states are recurrent, then all possible states within the given power modes are traversed.

Since the basic idea is to take advantage of the idle time of the processors, trying to reduce or redistribute the slack time is important. In this algorithm, only one iteration of the graph is considered in power mode selection which may lead to a greedy slack distribution within just that iteration. In contrast, the algorithm from [1] can do power mode selection among all iterations involved in all critical cycles, as a result, it can balance the workload and utilize slack across multiple iterations which can provide a better solution in terms of energy consumption.

The state-space exploration can result in a large state-space and it may suffer from the state-space explosion problem which makes the analysis very time-consuming. Another problem caused by the state-space explosion is that it requires a big storage area. Compared with the technique from [2], the algorithm proposed in [1] gets rid of these problems. The technique proposed in [1] finds solutions for all the proposed benchmark applications in seconds to minutes, and saves considerable memory space to store the controller, depending on the number scenarios and frequency points.

# 4 Overview of the extensions to the RTAS'13 algorithm

## 4.1 Analysis of RTAS'13 Algorithm

The RTAS'13 algorithm, proposed in [1], is used to select a suitable multiprocessor DVFS point for each scenario of a dynamic application. The streaming applications with dynamic behaviors are modeled by SADF graphs with strict timing constraints.

This algorithm contains a repetitive part which consists of two steps: **parameter reduction step** and **re-initialization step**. In every **repetition**, at most two parameter reduction steps will be executed if the length of the critical cycle is longer than an application-specific threshold. The threshold, which will be called ***period***, is calculated according to the timing requirements of the application, its value is equal to the longest time for an iteration defined by the throughput constraint. After executing the parameter reduction steps twice, a re-initialization step happens.

One novel part of this algorithm is that it takes the reconfiguration time into consideration. When applying DVFS at run time, delays are inevitable when switching scenarios since it takes time to reconfigure the processor, e.g. changing frequencies and voltages setting. In this algorithm, reconfiguration scenarios for each processor are added at design time. The scenario graph for the reconfiguration scenario for each processor is modeled as an actor with a self-edge and a labeled token. As shown in Figure 4.1, the token pe represents the dependency between consecutive iterations and the execution time of $a_{pe}$is set to the DVFS delay.

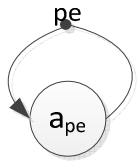



**Figure 4.1 Reconfiguration Scenario Graph**

In the FSM of the SADF, intermediate states for reconfiguration scenarios are placed between original states. In this way, the overhead of DVFS is taken into consideration when selecting a DVFS point at design time.

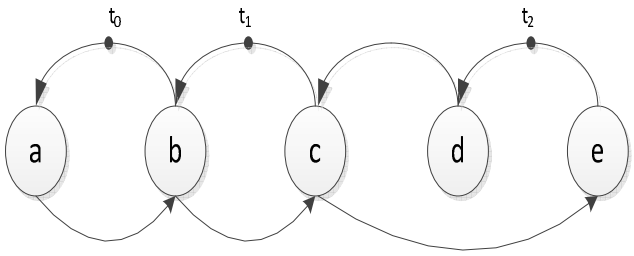



**Figure 4.2 Scenario Graph**

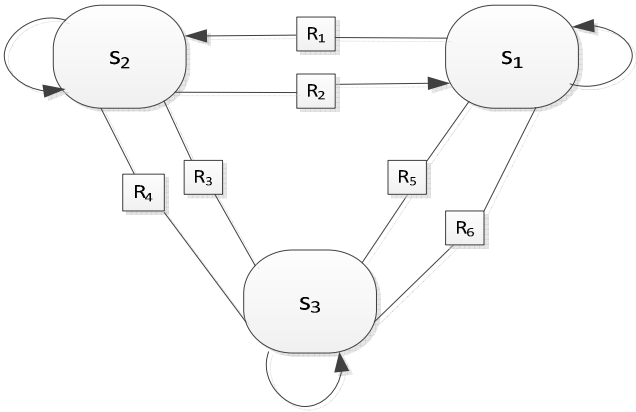



**Figure 4.3 Scenario FSM**

In order to make the algorithm easy to understand, a detailed example based on the example SADF shown in Figure 4.2 and Figure 4.3 is explained here. Figure 4.5 depicts the detailed process to implement the algorithm [1], the period is equal to 40 time-units.

In practice, a processor can only operate in specific discrete clock cycle levels. Hence, we assume that these discrete clock cycle points are known for each processor. In this example, the application with three scenarios is mapped to a three-processor platform. There are three parameters related to the frequency settings of their own processors in each scenario. For example, $p_{i,j}$ represents the parameter with which the clock cycle period of the $j^{th}$processor is scaled in scenario $s_i$. When reducing the value of $p_{i,j}$, the related frequency needs to be increased.

At the beginning, all parameters are initialized with their maximum value, (e.g., 5), in order to guarantee that the frequency settings can satisfy the timing constraint of the application. Then one critical cycle with an expression of $5p_{31} + 13p_{33}$ will be extracted and the information of its expression can be used to change the related clock cycle. As long as its length is bigger than the required period, 40 time units, the values of $p_{31}$and $p_{33}$ must be reduced in order to satisfy the following equation:

$$5p_{31} + 13p_{33} \leq 40$$

There are many possible combinations of $p_{31}$and $p_{33}$ , as shown in Figure4.4. The points marked in black are valid options. Among all the possible parameter points, $p_{31} = 3$ and $p_{33} = 2$ are selected since this combination is the local optimal that guarantees the lowest energy consumption at this stage.

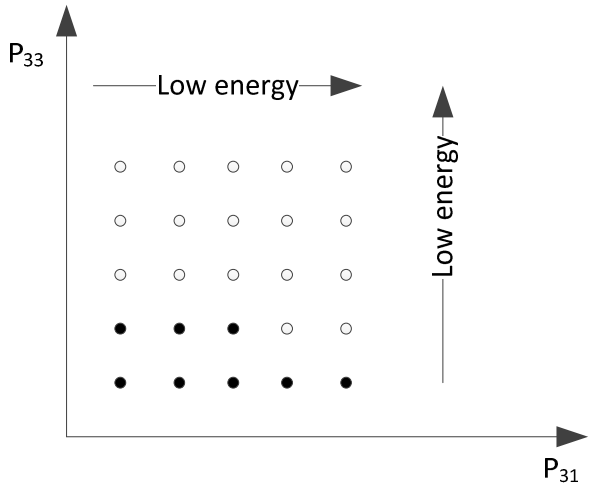



**Figure 4.4**

The resulting parameter point ($p_{31} = 3$ and $p_{33} = 2$) is fed to step 2. Another critical cycle ( $5p_{11} + 10p_{13}$) is selected and resolved in the same way as step 1, the values of $p_{11}$ and $p_{13}$ are reduced to 4 and 2 in order to make this critical cycle meet the time requirement.

Resolving a critical cycle in one step cannot enlarge other critical cycles found in prior steps, because all the derived parameter points are restricted to be smaller than or equal to the input parameter points in each step. After step 1 and step 2, if both steps decrease the frequencies of some processors, a new repetition will be executed after a re-initialization step. A re-initialization step will fix the values of those parameters which are reduced in step 2 for the subsequent repetitions. Parameters which were only changed in step 1 of the current repetition keep their original values, since they may have already been affected in the second step. This ensures that some parameters get smaller after each repetition, but also provides an opportunity to avoid unnecessary frequency increases for some processing elements involved in a critical timing cycle. The repetitions will continue until the length of all critical cycles is smaller than the required period.

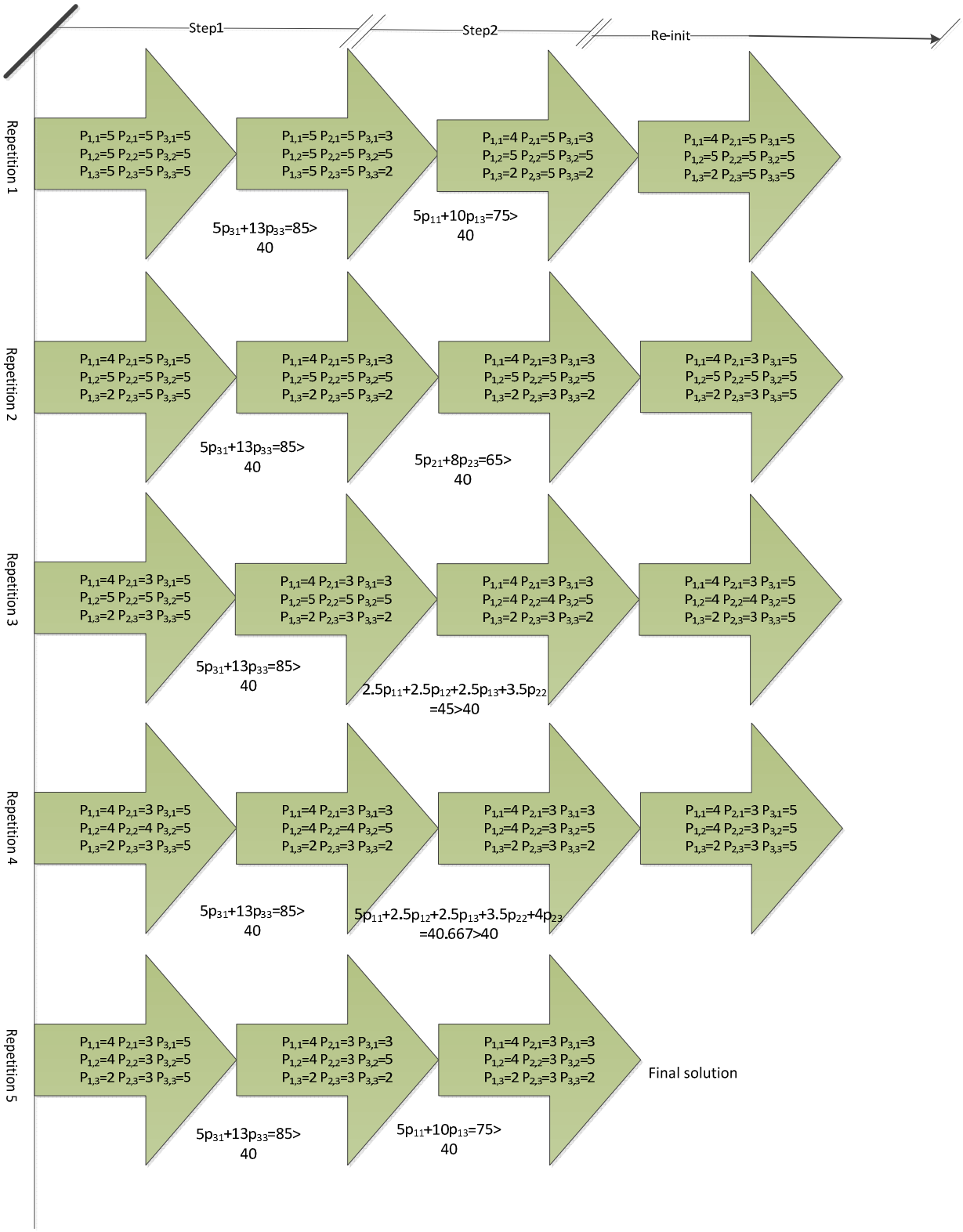



**Figure 4.5 Applying Algorithm [1] to the example SADF**

## 4.2 Non-optimality of RTAS'13 algorithm

The RTAS'13 algorithm can select a suitable multiprocessor DVFS point for each scenario of a dynamic application described by an SADF. According to the timing and energy requirement, this problem can be translated to an optimization problem of finding one best solution from all feasible solutions.

With different frequency settings, the total energy consumptions are different. When trying to get the lowest energy consumption , the problem can be translated to an optimization problem, to find a suitable multiprocessor DVFS point setting which leads to the lowest energy consumption while meeting temporal behavior constraint. It can be described as follows:

$$\begin{aligned} &\textbf{minimize}_{p_{i,j}} && E(p_{i,j}) \\ &\textbf{subject to} && 0 \le p_{i,j} \le n \\ & && T(p_{i,j}) \le Period \end{aligned}$$

where $E(p_{i,j})$ is the **target function** of energy consumption. $T(p_{i,j}) \le$ Period are called inequality constraints. $T(p_{i,j})$ are the expressions for all the critical cycles, they can derived by RTAS'13. The value of Period is given by the timing requirement. Since real platforms support limited number of frequency levels, so the values of each $p_{i,j}$ are set as integers and can only be changed between the range from 0 to n, while the initial values of all parameters are equal to their maximum value, n.

For example, an example SADF shown in Figure 4.2 and Figure 4.3, its execution times for different scenarios are listed in Table 4.1.

| Scenario | a | b | c | d | e | Scale factor sets |
|---|---|---|---|---|---|---|
| $s_1$ | $5p_{1,1}$ | $5p_{1,2}$ | $5p_{1,3}$ | $5p_{1,1}$ | $5p_{1,3}$ | $SFS_1 = \{1,2,3,4,5\}$ |
| $s_2$ | $3p_{2,1}$ | $7p_{2,2}$ | $3p_{2,3}$ | $5p_{2,1}$ | $5p_{2,3}$ | $SFS_2 = \{1,2,3,4,5\}$ |
| $s_3$ | $6p_{3,1}$ | $2p_{3,2}$ | $7p_{3,3}$ | $5p_{3,1}$ | $5p_{3,3}$ | $SFS_3 = \{1,2,3,4,5\}$ |

**Table 4.1 Actor execution times of the example parametric SADF**

Let $p_{i,j}$ express the scale factor with which the clock cycle period of the processing element $pe_i$ in scenario $s_j$. So the final solution derived by RTAS'13 algorithm is equal to:

$$S_1 \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} = \begin{bmatrix} 4 & 3 & 3 \\ 4 & 3 & 5 \\ 2 & 3 & 2 \end{bmatrix}.$$

In our project, it is assumed that homogenous processing elements consume equal amounts of energy when they are operating at the same frequency, and all the scenarios happen with the same probability, so the energy consumption is only related to the frequency. In other words, the energy consumption is inversely proportional to the sum of reciprocals of the squares of the

value of these parameters. So the target function $E(p_{i,j})$

$$E(p_{i,j}) = \sum\left(\frac{1}{p_{i,j}^2}\right)$$

With the information we get from the process of applying the RTAS'13 algorithm, all the unsatisfied critical cycles can be found. Then all the constraint inequalities are listed below:

$$5p_{11} + 10p_{13} \leq 40$$
$$5p_{21} + 8p_{23} \leq 40$$
$$2.5p_{11} + 2.5p_{12} + 2.5p_{13} + 3.5p_{22} \leq 40$$
$$5p_{11} + 2.5p_{12} + 2.5p_{13} + 3.5p_{22} + 4p_{23} \leq 40$$
$$for\ all\ 1 \leq p_{ij} \leq 5$$

When considering all the cycles and throughput requirements, one optimal solution can be calculated by LINGO. LINGO is a tool that can be used to find a global optimum solution. An optimal solution for this example is given by:

$$S_2 \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} = \begin{bmatrix} 4 & 3 & 3 \\ 4 & 4 & 5 \\ 2 & 3 & 2 \end{bmatrix}.$$

Compare $S_1$ and $S_2$, the value of $p_{2,2}$ are enlarged from 3 to 4, which means the frequency of the second processing element in scenario 2 can be lowered further while the throughput requirement is still met. The reduction of frequency can lead to a lower energy consumption. From this simple example, it is shown that the result solution of RTAS'13 algorithm is not optimal, so some techniques can be used to improve the algorithm.

## 4.3 Overall Approach

### 4.3.1 Dynamic number of parameter reduction steps

Even though the usage of the re-initialization step is to avoid an unnecessary frequency increase of some processors, some parameters may still suffer from over-reduction in the RTAS'13 algorithm (i.e., some processor frequencies might become higher than strictly necessary). In each repetition, at most two critical cycles can be taken into consideration for the point selection. It is easy to induce some biases for further reduction of other parameters. So this cannot guarantee an optimal frequency set.

If we increase the number of parameter reduction steps, more critical cycles can be considered in one repetition of the point selection step. Subsequently, the following re-initialization step should also be changed. In the re-initialization step, the parameters which are only modified at last will be kept. In the above example, when we increase the number of parameter reduction steps to 5, a

better set of parameters $S_3$ can be derived in terms of energy consumption, which is shown below:

$$S_3 \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} = \begin{bmatrix} 4 & 3 & 3 \\ 4 & 4 & 5 \\ 2 & 3 & 2 \end{bmatrix}.$$

Compare $S_3$ with $S_1$, the value of $p_{2,2}$ are enlarged from 3 to 4, which means the frequency of the second processing element in scenario 2 can be lower when the throughput requirements are met. When lowering the frequency, the energy consumption can be reduced.

Increasing the number of parameter reduction steps allows considering more critical cycles in the frequency point selection, as a result the point selection can be more accurate. However, it still cannot guarantee an optimal solution. And for different applications and targeted platforms, the number of necessary steps varies, so it is hard to determine a general number of steps which is suitable to all applications and platforms.

One idea is to use a dynamic number of parameter reduction steps instead of setting a fixed number of steps in every repetition. In each repetition, as long as there is a critical cycle whose bounds are out of the required period, the parameter reduction step will not stop. After all the parameter reduction steps completed, a re-initialize step follows, it only keeps the parameters modified at last, and then a new repetition begins. The repetitions will not finish until all the values of parameters are not reduced any longer.

### 4.3.2 Multiple critical cycles

In the RTAS'13 algorithm, when extracting critical cycles in the parameter reduction step, one critical cycle will be found randomly. However, there may be multiple critical cycles which violate the throughput requirements, and the order of selecting critical cycles with the same length may improve the final result.

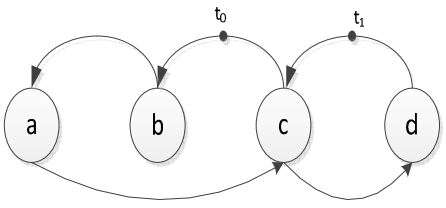



**Figure 4.6(a) Scenario graph**

| scenario | a | b | c | d |
|---|---|---|---|---|
| S1 | x | y | z | 2z |

**Figure 4.6(b) Actor execution time of the example parametric SADF**

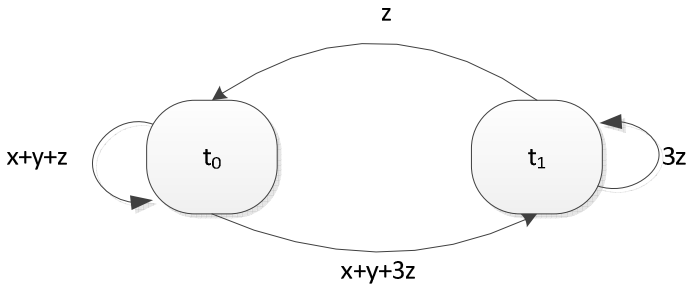



**Figure 4.7 Symbolic MPAG of the example SADF**

In the above example SADF (as shown in Figure 4.6) with only one scenario, when assigning the slowest frequency to each processing element, there are three critical cycles with the same length of 15 time-units.

As shown in Figure 4.7, the first found critical cycle expression is $x + y + z$ (from $t_0$ in scenario 0 to $t_0$ in scenario 1). There are three possible sets of answers when resolving the critical cycle and each of them can be selected with the same probability. The three sets are shown as follows:

$$[x\ y\ z] = [3\ 3\ 4] \text{ or } [4\ 3\ 3] \text{ or } [3\ 4\ 3]$$

In Figure 4.8, [3 3 4] is selected at random, and the final solution is [3 3 3].

As shown in Figure 4.9, the first found critical cycle expression is $3z$ (from $t_0$ in scenario 0 to $t_0$ in scenario 2 ), but the final solution is [4 3 3]. Obviously, this solution is better than the final solution found in Figure 4.8 in terms of energy consumption.

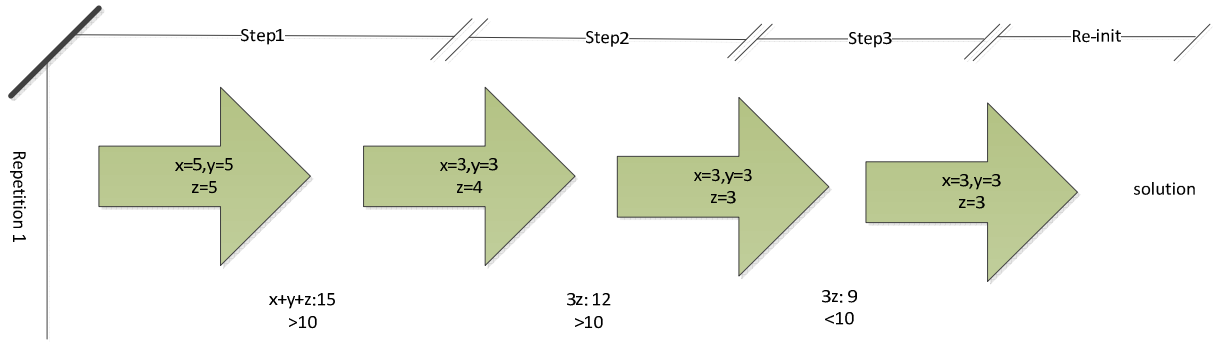



**Figure 4.8 Applying Algorithm proposed in [1] to example SADF**

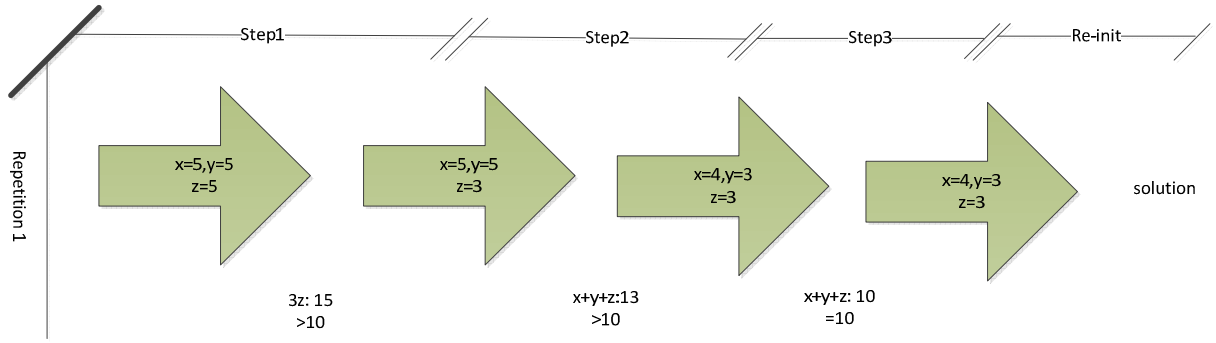



**Figure 4.9 Applying Algorithm proposed in [1] to example SADF**

Since different critical cycles may share some parameters with other cycles which have the potential to limit the throughput, changing one parameter here may make a difference to other related parameters in the future. When resolving one critical cycle with fewer parameters, taking the extreme situation in which this cycle is only related to one parameter into consideration, the parameter is reduced to its limit in one step immediately, and then it will leave a bigger room for other related parameters in the following steps. So an assumption is that the fewer parameters one critical cycle possesses, the higher priority it has to be resolved first when there are several critical cycles with the same length. Changing the sequence when selecting critical cycles with the same length does not influence the final solution, but it makes a difference to its runtime, which is also an important criterion to evaluate a strategy. So adding a mechanism to detect and choose the proper critical cycle can be beneficial to reduce its runtime.

### 4.3.3 Run-time energy reduction

In [1], the algorithm assumes workloads based on the worst case execution time (WCET) when designing a strategy which meets the throughput requirement. However, application workload varies and most of the workload is less than the WCET. Thus, algorithm [1] leads to pessimistic voltage and frequency settings since the assumed workload may be much higher than the actual workload. Moreover, the real hardware only supports a few numbers of frequency and voltage levels, it leads to limited usage of slacks.

When using the derived DVFS controller at run-time, there are available slacks which leave space to lower the energy consumption by further run-time refinement.

After finishing one task, the actual execution time ($t_{actual}$) may be less than worst case execution time, then there is a time slack ($WECT - t_{actual}$) before the execution of the subsequent actors. The following actors can utilize the available time slack so that they can work in a lower voltage and frequency level while still meeting the throughput requirement.

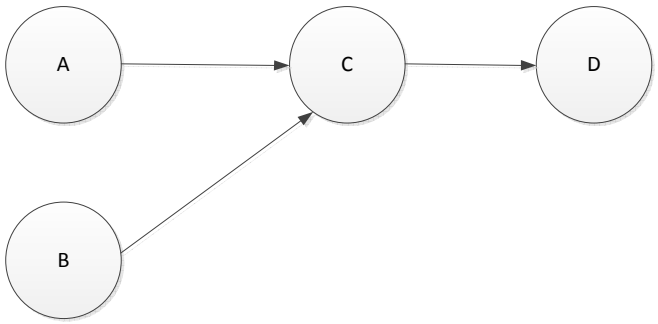



**Figure 4.10 an example SDF**

For example, in Figure 4.10, there are four actors (A, B, C, D) mapped to two processing elements ($pe_1, pe_2$). Actors A, C, D are mapped to $pe_1$ and actor B is mapped to $pe_2$. Actor C can only start to work after actor B finishes, the deadline of actor A is 60 time-unit. After applying a DVFS controller to the system, the actual timeline of the system is shown in Figure 4.11, there is a time slack of 60 – 40 = 20 time units before the execution of C. It is a waste of time and energy for actor C to wait until these 60 time-units have passed. If the available time slack is used, as shown in Figure 4.12, the value of frequency and voltage of $pe_1$can be reduced further.

In the above example, we do not consider the DVFS overhead and actor C monopolizes the time slack. In fact, before the system is going to utilize the extra slack, a reconfiguration of the voltage and frequency has to be done. After the slack is used, a reconfiguration is also needed in order to switch to the original or another DVFS point. Hence, at least two reconfigurations must be taken into consideration when adjusting the DVFS operating point of a processor at run-time. Before using the time slack, assuming that this time slack is more than twice as large as the reconfiguration cost, the run-time switch may take place, or there is no need to lower the frequency and voltage to utilize the time slack.

How to utilize these slacks efficiently is another difficulty. The detected slack at run-time cannot be utilized by the finished actors, but it can be utilized by one or more subsequent actors. For example, the time slack can be used by C alone (see Figure 4.12), or C and D together (see Figure 4.13). After the found time slack is used, the scheduler continues to detect the time slack that can be used by the following actors.

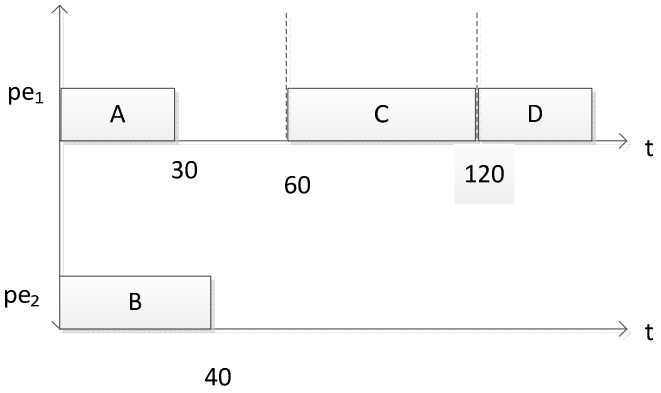



**Figure 4.11**

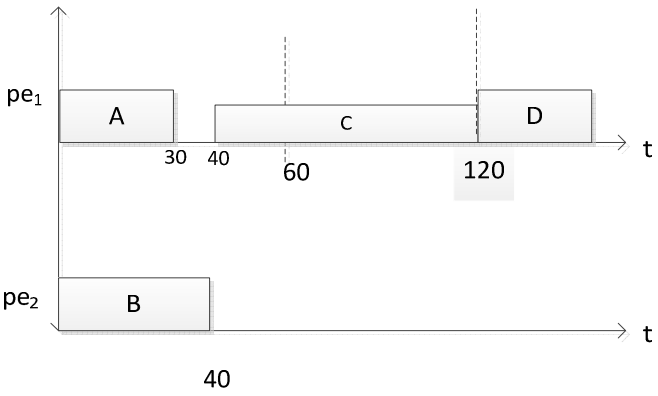



**Figure 4.12**

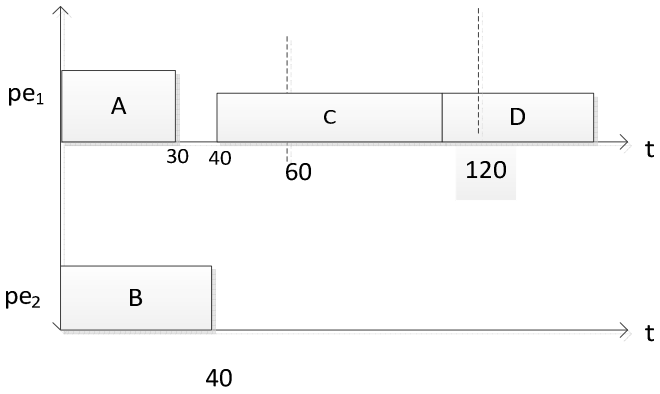



**Figure 4.13**

# 5 Design-time extension

## 5.1 Dynamic Number of parameter reduction steps

When finding the suitable frequency settings for each scenario by the RTAS'13 algorithm, there are two parameter reduction steps and one re-initialization step. The first extension of the RTAS'13 algorithm is to use a dynamic number of parameter reduction steps.

Instead of executing parameter reduction steps maximal twice in every repetition, the new algorithm will continue to find critical cycles until all the critical cycles satisfy the timing requirement. For different SADF graphs, the number of parameter reduction steps is different. Moreover, for the same application SADF, the number of parameter reduction steps in different repetitions is also different. Consequently, the original re-initialization, which fixes the values of those parameters reduced in steps of two, the number of steps should be changed.

The RTAS'13 DVFS is implemented with three variables (*paramResult, paramResult2 and paramResult3*). The initial parameter setting is stored in *paramResult*, while the other two keep the intermediate results for the use of the re-initialization step. The *budget* is determined by the throughput requirement, if the length of the critical cycle is bigger than *budget*, some parameters needs to be reduced.

```
DVFS_RTAS'13
1     while (true) do
2     // step 1
3       if budget < MCM then
4         CheckAllPowerModes (paramResult,ParamResult2)
5         //(delete) UpdateResult(ParamResult2)
6       else
7         return paramResult
8     //step 2
9       if budget < MCM then
10        CheckAllPowerModes (paramResult2,ParamResult3)
11        Re-initialization (ParamResult)
12      else
13        paramResult =paramResult 2
14        return paramResult
```

Figure 5.1

In the re-initialization step, after comparing the result of step 1and step 2, only the parameters reduced in the second step will be kept, otherwise, the parameters keep the same.

```
Re-initialization
1     for each item a in paramResult3
2       if paramResult3[a] <paramResult2[a]
3         paramResult[a]=paramResult3[a]
```

Figure 5.2

When trying to apply a dynamic number of parameter reduction steps to the RTAS'13 algorithm, there are three issues to be solved:

1. When should the parameter reduction steps stop in each repetition
2. How the new re-initialization step needs to be performed
3. When the whole process finishes

In order to increase the number of parameter reduction steps to unlimited, we can add another **while**-loop inside the original **while**-loop. If the length of the found critical cycle is longer than the *budget*, the parameter reduction step continues, otherwise, the parameter reduction steps stops and breaks the inner **while**-loop, then a re-initialization step follows.

In the new re-initialization step, only the parameters that are modified at last will be kept for the following repetition. All the other parameters keep the same as the input value of this repetition. In order to find the last changed parameters, another variable need to be added.

The point selection process does not finish until all the parameters are not reduced any more. So it is useful to add a flag signal in the real code to identify the end of the process. When running the modified DVFS algorithm, the number of parameter reduction steps in different repetitions is descending. When the length of all cycles is shorter than the *budget*, the process comes to an end, and there is only one step in the last repetition.

| **Modified DVFS** | |
|---|---|
| **1** | **while** (true) |
| **2** | **while**(true) |
| **3** | *ParamResult1=paramResult2* |
| **4** | *ParamResult2=paramResult3* |
| **5** | **if** budget < MCM **then** |
| **6** | CheckAllPowerModes (*paramResult2, ParamResult3*) |
| **7** | **else** |
| **8** | **return** *paramResult2* |
| **9** | //re-initialization |
| **10** | flag=0 |
| **11** | **for** each item a in *paramResult2* |
| **12** | **if** *paramResult2*[a] < *paramResult1*[a] **then** |
| **13** | flag++ |
| **14** | *paramResult3*[a]= *paramResult2*[a] |
| **15** | *paramResult*[a]= *paramResult2*[a] |
| **16** | **else** |
| **17** | *paramResult3*[a]=*paramResult*[a] |
| **18** | **if** flag=1 **then** |
| **19** | **break** |

Figure 5.3 The modified DVFS

There is another optimization about the number of parameter reduction steps. If the extracted critical cycle is only related to one parameter, the parameter reduction step can stop immediately,

then the repetition will enter a re-initialization step in order to keep the changed parameter. It does not influence the final frequency setting, but it reduces the analysis time.

## 5.2 Finding more critical cycles

The length of the critical cycle(s) determines the performance of a system. If the system works slower than expected, the critical cycle will be longer than the *budget*, then timing constraint violation happen. In order to meet the timing constraint, the frequency of some processors should be increased, then the length of related critical cycle will be reduced. All the critical cycles with the same length are responsible for the timing violation. Changing one parameter here may make a difference to the following steps. The problem here is that we do not know which critical cycle should be changed first. Before solving this problem, all the critical cycles need to be found.

However, in the original RTAS'13 algorithm, it only selects one critical cycle randomly. In every repetition, once one critical cycle is found, no critical cycle is detected any more. As a result, very limited information can be used for further parameter reduction. Currently, our solution is to provide as many critical cycles as possible. Before making decisions about how to change the frequency setting, all the cycles with the same longest length should be found and kept.

Consider a finite directed cyclic graph $G$ with $n$ nodes $v$ and $m$ arcs $a$. The simplest and easiest way to find all the critical cycles is to detect all the cycles, and calculate their length. Then the longest cycles can be found. However, the number of cycles is exponential in the number of nodes. For example, in a complete graph, every single permutation of every possible size from 2 to n results in a cycle. When the number of nodes increases, the scalability of this method is not good since the execution time will be increased exponentially. So it is not feasible and efficient to find all the critical cycles in the DVFS algorithms.

There are other feasible methods to find all the critical cycles. Finding all the critical cycles can be applied with the help of a maximum cycle mean/ maximum cycle ratio (MCM/MCR) analysis on the concrete MPAG of the corresponding SADF. There are many MCM/MCR algorithms. Before explaining the second extension for RTAS'13 algorithm, some details about MCM algorithms, especially the algorithm proposed by Young, Tarjan, and Orlin[29], will be introduced.

### 5.2.1 Maximum Cycle Ratio algorithm

Consider a finite directed cyclic graph $G$ with $n$ nodes $v$ and $m$ arcs $a$. Suppose that every arc $a$ is associated with two weights: its cost and its transit time. The cycle ratio is defined by the following equation:

$$Cycle\ ratio\ = \frac{\sum_a transit\ time}{\sum_a cost}$$

The problem to find the maximum cycle ratio or mean is called MCM/MCR problem. MCM and MCR problem can be transformed easily if the sum of transit time is equal to the cycle length. If the cycle mean of some cycles are the maximum, then these cycles are critical cycles.

Let $G = (V, A, \omega, \tau)$ be the input finite directed graph for each MCR algorithm. In the input graph, there are n nodes ($|V| = n$) and m arcs ($|A| = m$), two weight functions $\omega$ $and$ $\tau$ map arcs to numbers. Each arc $a$ from node $u$ to node $v$ is mapped to an arbitrary number $\omega(a)$ and a nonnegative integer $\tau(a)$, which equal to its cost and transit time, separately. The maximum cycle mean $\lambda^*$ can be defined in the following linear program:

$$\textbf{minimize} \quad \lambda$$

$$\textbf{subject to } \mathrm{d(v)} \geq \mathrm{d(u)} + \omega(\mathrm{u,v}) - \lambda, \forall(\mathrm{u,v}) \in \mathrm{E}$$

Here $d(u)$ is the weight of the maximum weighted path from s to u in G when $\lambda^*$ is subtracted from every arc weight. This subtraction eliminates all the positive weighted cycles from G so that it becomes possible to find the longest paths in G. If the longest path is a cycle, then it can be defined as critical cycle. After calculating the $\lambda$, and keeping the node information or arc information about the critical cycle, one critical cycle can be kept.

Ali Dasdan shows that YTO algorithm is the best MCR algorithm [29]. So in this thesis, we only pay attention to the YTO algorithm. YTO is a parametric shortest path algorithm [30]. It tries to find the largest value of maximum cycle mean $\lambda^*$.

YTO associates a node key *nk(n)* with every node and an arc key *ak(a)* with every arc. When finding the arc key *ak(a)* for arc *a=(x, y)*, $\Delta t$ should be calculated first according the following equation:

$$\Delta t = t(x) + \tau(a) - t(y)$$

If $\Delta t$ is smaller than zero, then the arc key is defined as infinite. Otherwise, $\Delta d$ , the difference between the distance of $x$ and the distance of $y$, should be calculated according to the following equation:

$$\Delta d = d(x) + \omega(a) - d(y)$$

Then the arc key $ak(a)$ is equal to $\frac{\Delta d}{\Delta t}$. After finding the arc key for each arc, the node key can be determined by the minimum arc key among all the input arcs of this node.

YTO algorithm operates on a finite sequence of shortest paths trees. The first step is to construct an initial tree by adding a new source node to the input graph, and arcs from the source node to all other nodes in the graph. In the next step all the arc keys and node keys will be calculated and this information will be used to create a new shortest path $T_p$. Arc $(u, v)$ is the arc with the minimum arc key among all arc keys, after replacing the predecessor of $v$ in $T_p$ by $(u, v)$, there

is a cycle, as a result, the ratio of this cycle is the MCR. Otherwise, YTO constructs the new tree by recalculating the node keys and arc keys after refreshing the path trees.

When applying the YTO algorithm to the max-plus automata graph in our project, the max-plus automata graph should be translated to the desired YTO input graph. There are two important values in the YTO input graph, the weight $d(a)$ and transit time $\tau(a)$ for each arc $a$. These values can be easily derived by checking the corresponding arc in the max-plus automata graph.

```
1   V = V ∪ {s}
2   A = A ∪ {(s, u) | w(s, u)=0}
3   T_p(s) = {(s,v) |(s,v) ∈ A}
4   for each node v ∈ V do
5           d(v)=0; t(v)=1; nk(v)=NIL
6   t(s) =0
7   for each arc a ∈ A do
8           ak(a)=FIND-ARC-KEY(a, G)
9           if nk(v)=NIL or ak(a)<ak(nk(v)) then
10                  nk(v)=a
11  PQ-INSERT(H,<∞,->)
12  for each node v ∈ V do
13  PQ-INSERT(H,<ak(nk(v)), nk(v)>)
14  lambda = -∞
15  while (true) do                                  if r = ∞ then
16      <r,(u, v)> =PQ-FIND-MIN(H)                       return r
17      if r= ∞ or u ∈ T_p(v) then                   if lambda = -∞
18          return r                                     lambda = r
19                                                       if (r < lambda)
20  // continue to construct a new tree                  return lambda
21  for each node x ∈ T_p(v) do                      if ( u ∈ T_p(v) )
22      d(x)=d(x)+(d(u)+w(u, v)-d(v))                    SAVE-A-CYCLE
23      t(x)=t(x)+(t(u)+τ(u, v)-t(v))
24  p(v)=u
25  for each node x ∈ T_p(v) do
26      ak( nk(x) ) = ∞
27      for each arc a =(x, y) entering T_p(v) do
28          ak(a)=FIND-ARC-KEY(a, G)
29          if ak(a)<= ak(nk(y)) then
30              nk(y)=a
31  PQ-UPDATE (H,<ak(nk(y)),nk(y)>)
32  for each node x ∈ T_p(v) do
33      for each arc a =(x, y) leaving T_p(v) do
34          ak(a)=FIND-ARC-KEY(a, G)
35          if ak(a)<= ak(nk(y)) then
36              nk(y)=a
37  PQ-UPDATE (H,<ak(nk(y)),nk(y)>)
```

Figure 5.4 YTO algorithm

In the YTO algorithm which is shown as the black part in Figure 5.4, only one critical cycle can be detected in each repetition. If we want to collect more critical cycles, the algorithm should be modified.

### 5.2.2 Modified YTO

In the original YTO algorithm, the process ends immediately when one critical cycle is found. However, this is not what we want. In order to make it possible to find all the critical cycles, some changes are made by replacing the original 17-18 lines in Figure 5.4.

In the modified YTO, the process continues to find another critical cycle when one critical cycle has already been found. So instead of breaking the **while** loop immediately, the program proceeds to search another shortest path with the same length. There are two situations to break the **while** loop:

1. The MCM of the new found cycle is smaller than the former MCM.
2. The maximum cycle mean is equal to infinity.

There are two types of critical cycles: self-cycle and nonself-cycle. Some extra steps should be added to avoid deadlock. For example in Figure 5.5, both self-cycle ($a_1$) and nonself-cycle ($a_2$ $and$ $a_3$) are critical cycles. If we do not differentiate the detecting process, a deadlock situation occurs. Because the parent node and child node of arc $a_1$ are the same node.

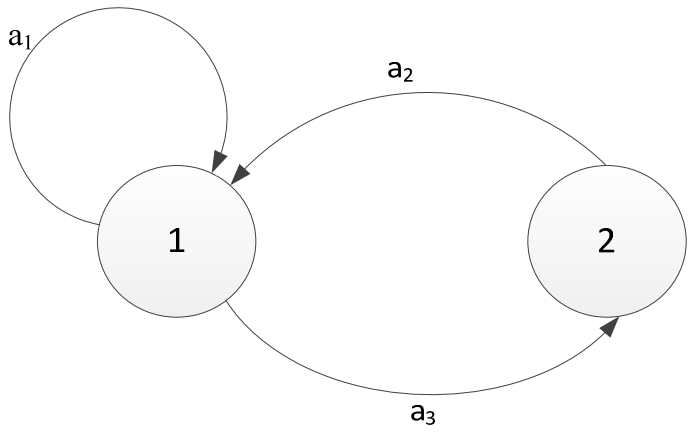



Figure 5.5 Example YTO Paths

When storing multiple critical cycles, one important problem is to find a proper data structure to store these critical cycles. The cycles are expressed by their arcs. So the variable *YTOCycles* expressed by a list of list can be used to store all the critical cycles. In the internal list, its element keeps the information of each arc. Its data structure is shown in Figure 5.6.

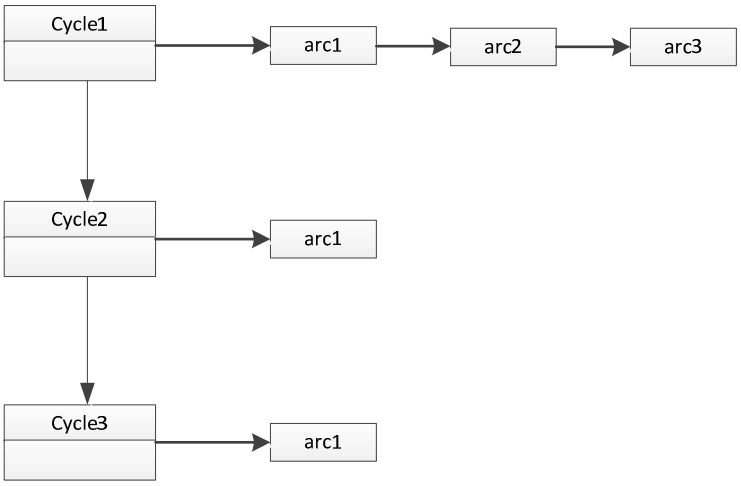



Figure 5.6 List of List Structure

There is another way to find all the critical cycles of the input graph, which is called as simple cycle detection. In the simple cycle detection, we can find all cycles in the graph, compare the length of each cycle, finally, we can select all the critical cycles. When implementing the simple cycle detection, the input max-plus automata graph should be divided into several strongly connected components. How to split the directed graph into a set of strongly connected components is a complex issue, but it is not the main point in this thesis. Since this function has already been realized in the SDF3 tool, it is called directly. So instead of describing the related implementation in detail, only the basic algorithm is shown in Figure 5.7 and Figure 5.8, if you are interested in more detail, you can read related work, such as [31].

```
1  for each component s ∈ SCC do
2      for each actor a in s do
3          color[a]←white
4          pi[a]← NIL
5      for an actor a in s do
6          color[a]←gray
7          for each actor b in Adj(a, s) do
8              pi[a]← a
9              SimpleCycleVisit(a)
```

Figure 5.7 Simple Cycle Detection Algorithm

```
1  color[a] ← gray
2  for each b in Adj(a, s) do
3      if color[a] = white then
4          pi[b] ← a
5          simpleCycleVisit(b)
6      else if color[a] = gray
7          create new cycle c
8  color[a] ← white
```

Figure 5.8 SimpleCycleVisit Algorithm

After getting all simple cycles and calculating their length, the set of critical cycles with the same longest length will be stored in a variable *SimpleCycles,* which has the same data structure as the *YTOCycles.* However, the simple cycle detection costs quite a long time to find all critical cycles, so it is not suitable to find all the critical cycles in the MPAG of its related SADF in the DVFS algorithm.

### 5.2.3 Selection of critical cycles

Since different critical cycles may share some parameters, changing one parameter here may make a difference to other related parameters in the future. In the parameter reduction step of RTAS'13, the first found critical cycle will be selected, and its parameters will be changed. This randomness may lead to a local optimal. When we get the information of multiple critical cycles, we can choose a specific critical cycle instead of using a random cycle.

When resolving one critical cycle with fewer parameters, taking the extreme situation that one critical cycle is only related to one parameter into consideration, this parameter is reduced to its limit in one step immediately, then it will leave more space for other related parameters in the following steps.

In the new DVFS controller, in order to select a specific critical cycle, the number of parameters of each critical cycle will be counted and saved. The critical cycle with the smallest number of parameters will be selected when multiple critical cycles are found and their lengths are not satisfied with the throughput constraint in the parameter reduction step. For example, the initial value of any $p_{i,j}$ in Figure 5.9 is 5, two critical cycles can be detected after executing the modified YTO algorithm, $10p_{1,1} + 6p_{1,2}$ and $16p_{1,2}$. After comparing their number of parameters, critical cycle with the expression of $16p_{1,2}$ will be selected and resolved first in the parameter reduction step.

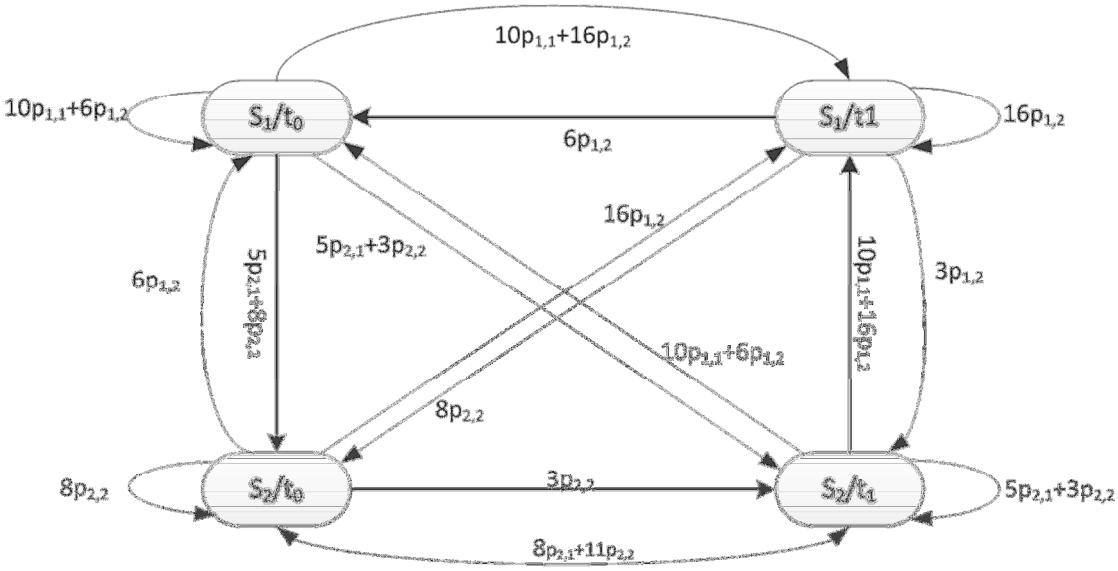



Figure 5.9 Critical Cycle Analysis

# 6. Run-time Optimization

At run-time, the system predicts and selects one scenario based on the input items. Then it scales the voltage and frequency according to the values determined at design-time, reducing the energy consumption while still meeting the timing requirement [16]. The workload prediction at design time is based on worst case execution time (WCET) of the application. However, the workload is varied at run-time, and the actual execution time can be much shorter than the worst case execution time, as a result, there are some slacks. If we utilize these slacks, the frequency and voltage level can be reduced to a lower level. Thus, the worst case situation based workload prediction leads to a pessimistic voltage and frequency scaling.

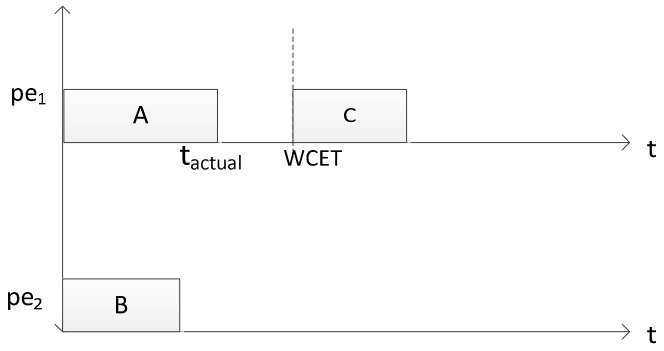



Figure 6.1 Possible run-time situation

As shown in Figure 6.1, the actual execution time of actor A is $t_{actual}$ which is less than its worst case execution time WCET, then there is a time slack ($WECT-t_{actual}$). In general, the heart of DVFS techniques is to explore and use the slacks. Even though the slack ($WECT-t_{actual}$) cannot be used by actor A anymore, the following actor C can use this slack so that it can work in a lower voltage and frequency level while still meeting the throughput requirement. These available slacks can not only be used by the closest actor, but also be saved and used by other following actors.

At run time, the processor frequency is recalculated according to how the slacks are used. After finding the available time slack, if we use different percentage of slacks, the frequency can be different. Theoretically, the more slacks we use, the more frequencies and energy consumptions can be reduced. In Figure 6.2-B, which is proposed by Morteza in his thesis, all the slacks will be used immediately by their following actor.

Utilizing all slacks is always beneficial for reducing the energy consumption when we do not consider other additional cost, such as the reconfiguration cost. However, if the processor setting is changed all the time, the energy consumption caused by reconfiguring will be increased in reality, as a result, the energy consumption may increase. So if the reconfiguration cost cannot be

negligible, some checks are necessary in order to make sure the energy efficiency when changing the current processor frequency. The time slack can be used only when the time slack is longer than twice the reconfigure time at least. It is better to find a smart way to use these slacks.

In this project, our DVFS controller uses the run-time technique that only higher frequency processors utilize slacks, then reconfiguration times should be reduced. After detecting a slack at run-time, instead of using it immediately, the system will check the frequency of current actor and next actor. The current actor keeps its working at its current(design-time selected)frequency and voltage level if next actor works at a higher frequency level. Hence, the slack is preserved temporarily and will be used by the following actor. For example, as shown in Figure 6.2-C, even though there is an available slack before actor B, actor B decides in this fictive DVFS strategy not to reduce its frequency level. As a result, the next actor C can get a longer slack, then the frequency of actor C can be reduced much lower.

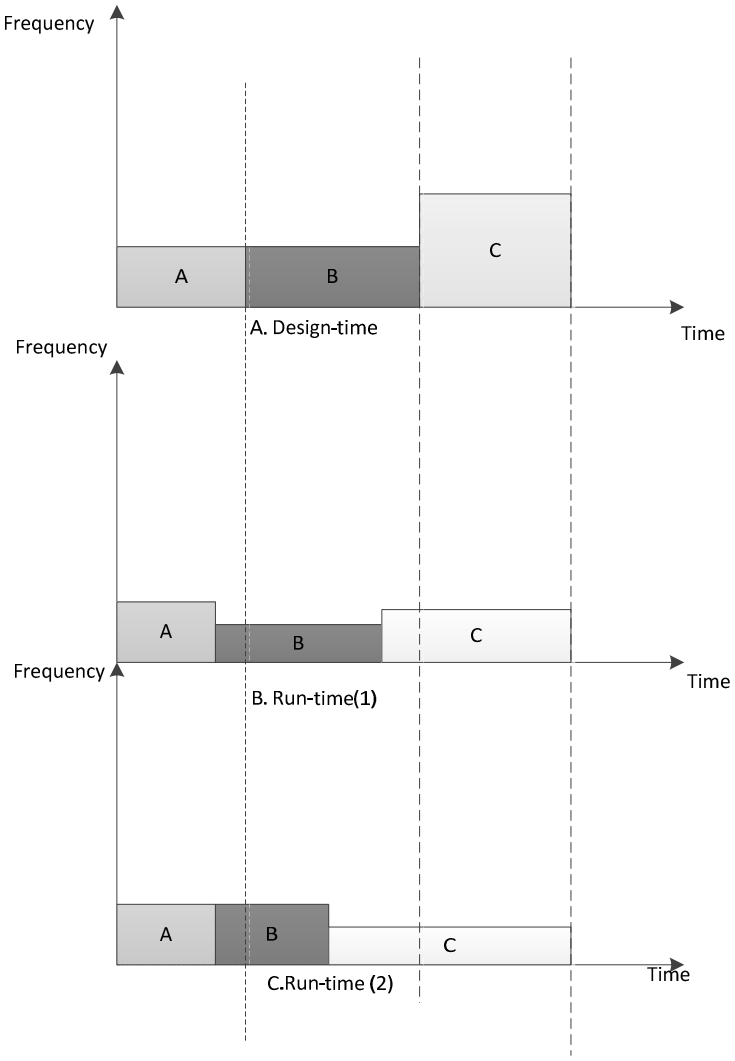



Figure 6.2 Execution Situations

# 7 Experimental result and analysis

## 7.1 Design-time Experiment

In the experimental part, firstly the influence of each modification will be tested separately by comparing their energy consumptions. After the separate comparison and analysis, we will combine the modifications and find the best DVFS algorithm.

In order to make the different versions of modified DVFS algorithm easy to be understood and referred to, we name them in a systematic way as shown in Table 7.1.

| Name | Modification |
|---|---|
| RTAS'13 | Original RTAS'13 DVFS |
| DVFS_1 | Extend RTAS'13 with dynamic number of steps |
| DVFS_2 | Select the critical cycle with minimum number of parameters |
| DVFS_3 | DVFS_1+DVFS_2 |

Table 7.1 Name Table

For all the experiments, it is assumed that the hardware platforms only support limited number of frequency levels. And for each frequency level, there is one related voltage level. Moreover, it is assumed that homogenous processing elements consume equal amounts of energy when they are operating at the same frequency, all the scenarios happen with the same probability. So the total system energy consumption is only related to the frequency. In other words, the energy consumption is inversely proportional to the sum of reciprocals of the squares of the value of these parameters.

There are three input experimental SADF graphs, SADF1, SADF2, SADF3. SADF1 is an application with three scenarios which is mapped to a three-processor platform. Its required timing constraint is 40 time units. SADF2 is a two-scenario-application and its platform has two processors. For the last SADF graph, SADF3, it is also an application with three scenarios and is mapped to a platform with three processors, but the mapping and timing requirement are different as compared to the first SADF graph.

After applying the four algorithms (RTAS'13 and DVFS_1, DVFS_2, DVFS_3) to the three examples, all the DVFS frequency points are selected, as shown in Table 7.2. If we apply the DVFS points selected at design-time to calculate the average energy consumption, the result energy consumption is shown in Table 7.3. The baseline energy consumption of RATS'13 is 1, and other energy consumptions are relative values compared to the energy consumption of RTAS'13 .

| - | SADF 1 | SADF 2 | SADF 3 |
|---|---|---|---|
| | Parameter values | Parameter values | Parameter values |
| RTAS'13 | $p_{11}=4$ $p_{21}=3$ $p_{31}=3$<br>$p_{12}=4$ $p_{22}=3$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=3$ $p_{33}=2$ | $p_{11}=4$ $p_{21}=4$<br>$p_{12}=4$ $p_{22}=4$<br>$p_{13}=2$ $p_{23}=2$ | $p_{11}=3$ $p_{21}=3$ $p_{31}=2$<br>$p_{12}=1$ $p_{22}=2$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=2$ $p_{33}=2$ |
| DVFS_1 | $p_{11}=4$ $p_{21}=3$ $p_{31}=3$<br>$p_{12}=4$ $p_{22}=4$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=3$ $p_{33}=2$ | $p_{11}=4$ $p_{21}=4$<br>$p_{12}=4$ $p_{22}=4$<br>$p_{13}=2$ $p_{23}=2$ | $p_{11}=3$ $p_{21}=3$ $p_{31}=2$<br>$p_{12}=1$ $p_{22}=2$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=2$ $p_{33}=2$ |
| DVFS_2 | $p_{11}=4$ $p_{21}=3$ $p_{31}=3$<br>$p_{12}=4$ $p_{22}=3$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=3$ $p_{33}=2$ | $p_{11}=4$ $p_{21}=4$<br>$p_{12}=4$ $p_{22}=4$<br>$p_{13}=2$ $p_{23}=2$ | $p_{11}=3$ $p_{21}=4$ $p_{31}=2$<br>$p_{12}=1$ $p_{22}=2$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=2$ $p_{33}=2$ |
| DVFS_3 | $p_{11}=4$ $p_{21}=3$ $p_{31}=3$<br>$p_{12}=4$ $p_{22}=4$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=3$ $p_{33}=2$ | $p_{11}=4$ $p_{21}=4$<br>$p_{12}=4$ $p_{22}=4$<br>$p_{13}=2$ $p_{23}=2$ | $p_{11}=3$ $p_{21}=4$ $p_{31}=2$<br>$p_{12}=1$ $p_{22}=2$ $p_{32}=5$<br>$p_{13}=2$ $p_{23}=2$ $p_{33}=2$ |

Table 7.2 Result frequency settings

| | SADF 1 | SADF 2 | SADF 3 |
|---|---|---|---|
| | Estimated Energy consumption | Estimated Energy consumption | Estimated Energy consumption |
| RTAS'13 | 1 | 1 | 1 |
| DVFS_1 | 0.96 | 1 | 1 |
| DVFS_2 | 1 | 1 | 0.98 |
| DVFS_3 | 0.96 | 1 | 0.98 |

Table 7.3 Relative energy consumption

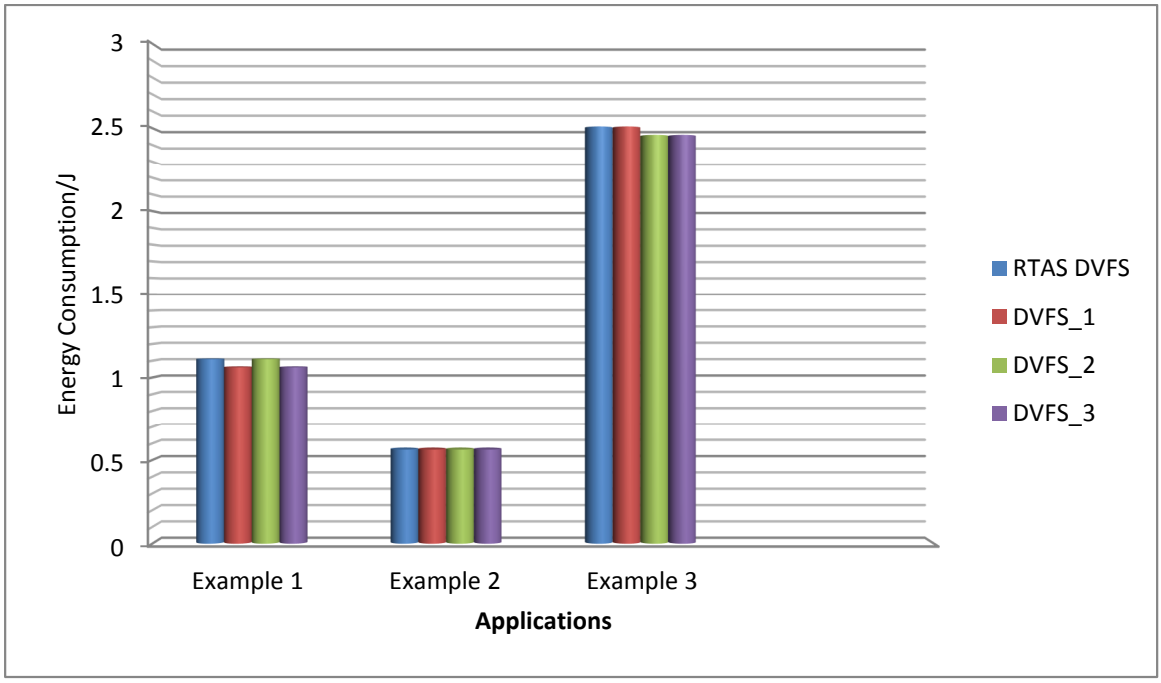



Figure 7.1 Energy Reduction

As shown in Figure 7.1, the energy consumption of Example 2 and Example 3 keeps the same, but the energy consumption of Example 1 is lower when using DVFS_1. So it is possible to improve the frequency settings to lower the energy consumption when using dynamic number of parameter reduction steps in each iteration.

When applying DVFS_2 to the example SADF graphs, the analysis time will be much longer than that of RTAS'13. But this is the design-time process, the analysis time is not an important concern. According to green and blue bars in Figure 7.1, the energy consumption of Example 1 and Example 2 keeps the same, but the energy consumption of Example 3 is lower when using DVFS_2. It shows that starting with the critical cycle with minimal number of parameters can potentially derive a better DVFS controller in terms of lower energy consumption.

When applying DVFS_3 to the example SADF graphs, the analysis time will be increased compared with than that of DVFS_2. But the estimated energy consumption of Example 1 and Example 3 is reduced. From Figure 7.1, we can conclude that both the two design-time extensions are beneficial to reduce energy consumption, and DVFS_3 can give best solution among these four algorithms.

## 7.2 Run-time Experiment

For the run-time experiments, two dynamic streaming applications are used- MP3 decoder and MPEG4 decoder. Both of these applications are divided into five scenarios, and mapped to a

three-core platform. After determining the frequency for every processor in each scenario, the run-time simulator will simulate the actual execution. The energy consumption is calculated at that time. The time unit of applications is nanosecond(ns), and the time unit of final energy consumption is Joule. But we only use a relative number for the purpose of energy comparison. They do not represent any real energy consumptions. The energy consumption without any run time techniques is equal to 1.

Similarly, in order to make the comparison easy to distinguish, the rename table is as shown in Table 7.4. The experiment is conducted with three different methods to use the slacks. In Run-time-1, which is proposed by Morteza in his thesis, once a slack is found before the execution of some actor, the actor will use the whole slack immediately, while the difference between Run-time-1 and Run-time-2 is only using different percentage of slacks. In Run-time-3, instead of using the slack immediately, it can be preserved for later utilization if the actor is working in a lower frequency level than its following actor.

| Name | Actions |
|---|---|
| Run-time-1 | Use all slacks |
| Run-time-2 | Use half slacks |
| Run-time-3 | Only higher frequency actor uses slack |

Table 7.6 Name Table

In the run-time experiment, the reconfiguration cost is taken into consideration. The energy consumption is shown in Table 7.7.

| Energy consumption | Utilize all slacks (Run-time-1) | Utilize half slacks (Run-time-2) | Higher frequency use slacks(Run-time-3) | Without Run-time Optimization |
|---|---|---|---|---|
| MPEG4 | 0.61 | 0.68 | 0.61 | 1 |
| MP3 | 0.97 | 1.79 | 0.97 | 1 |

Table 7.7 Energy Consumption

From Table 7.7, we can find that the energy reduction of Run-time-1 and Run-time-3 is almost the same, while Run-time-2 may cause the energy consumption increase dynamically. When applying Run-time-3, higher frequency actors can use longer slacks, the energy consumption of these actors can be reduced, as a result, the total energy consumption may be reduced.

However, some factors should be considered here. Firstly, there are many slacks which are potentially not used. For example, in Figure 7.3 and Figure 7.4, actors A, B, C, D are working in the same frequency level, while actor E works in a higher frequency level. Actor A and B are working on the same processor, while actor C, D, E are mapped to another processor. The worst case execution times are the same for each actor, which are equal to 100 time units. However, the actual execution time is much shorter; there is a time slack before the execution of actor E. If we use Run-time-1, after the execution of actor C finishes, actor D can use a 50 time units slack.

However, if we use Run-time-3, actor D cannot use the detected slack, so the time slack cannot be used by any actors.

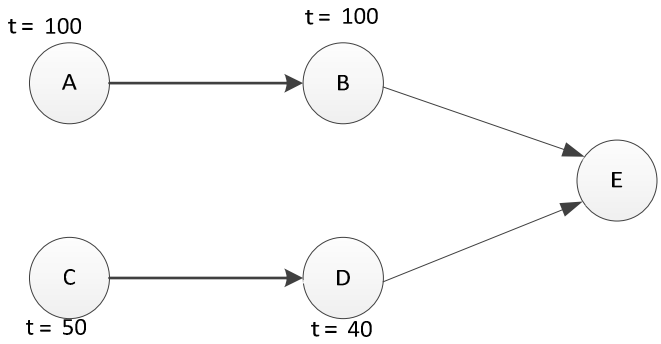



Figure 7.3 SDF

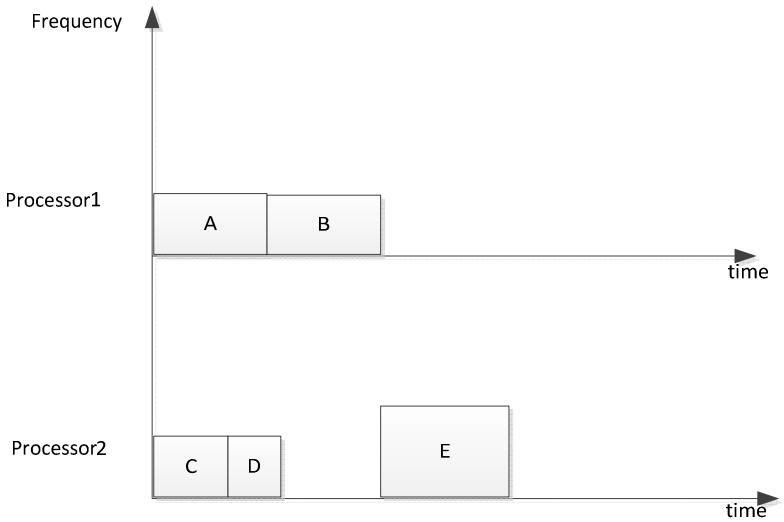



Figure 7.4 Timeline

Secondly, when executing Run-time-3, the run-time overhead is huge, it may be much longer than the length of available slacks. The processor needs to predict next scenario, compare the frequency of current and next scenario before making the decision about when to use the slack,. However, our method cannot get the accurate value of the run-time overhead. So the experimental energy consumption is not accurate because it does not include the additional run-time overhead. And since the run-time overhead can be much longer than the length of available slacks, more slacks will be wasted. Thirdly, core to core communication will lead to additional energy consumption. Compared with Morteza's approach, the run-time overhead of Run-time-3 is too high to get an energy reduction, while Run-time-1 only needs local knowledge which does not consume too much energy.

So without further verification, we can only conclude that Run-time-3 is not energy efficient, and Run-time-1 is a better choice to reduce energy consumption.

# 8 Conclusion and Future Work

In this project, we propose some techniques to minimize the energy consumption for real-time dynamic streaming applications. These streaming applications with dynamic behaviors are mapped to multiprocessor platforms. They are modeled by scenario-aware dataflow graph.

We start the project based on the RTAS'13 algorithm, which is aimed at selecting a suitable multiprocessor DVFS point for each scenario of a dynamic application. After realizing the non-optimality of RTAS'13 algorithm, two design-time extensions are proposed and implemented. The design-time extensions are increasing the parameter-reduction steps and starting to resolve critical cycle with minimum number of parameters.

When increasing the number of parameter-reduction steps in each repetition, more critical cycles can be considered in one repetition of the point selection step. Subsequently, the following re-initialization step should also be changed. In the re-initialization step, the parameters which are only modified at last will be kept. Before trying to find the critical cycle with minimum number of parameters, the algorithm should find all the critical cycles with the same length which violate the timing requirements. After these two extensions, the result is derived after considering more information. The experiment shows that the new algorithm can find better solutions in terms of energy consumption.

After making the design-time optimizations, the frequency setting will be used in run-time simulation. But the design-time DVFS is pessimistic, since the workload information used in design-time is based on the worst case execution situation while the actual execution time may be much shorter at run time. Then there are available slacks which leave space to lower the energy consumption by further run-time refinement. In the run-time energy refinement step, a simulator is used to simulate the execution of the application, the time slack is found by comparing the worst case start time and actual start time. The run-time switch can only happen when this time slack is more than twice as large as the reconfiguration cost, or there is no need to lower the frequency and voltage to utilize the time slack.

In the original run-time switch, Run-time-1, once there is an available slack, the following actor can use this slack to reduce its frequency, and consequently, it reduces the energy consumption. Ideally, when there is an available slack, the frequency of the related actor may work at a lower frequency level, and then the energy consumption can be reduced. In reality, the hardware only supports limited number of frequency levels. It is not always possible to reduce to the desired frequency for the processor. If the desired frequency is much close to the original frequency level, the frequency level will not be changed, and then the detected slack will not be used.

In this project, we propose a run-time switch method, Run-time-3, in which the slacks can be preserved and used by higher frequency actors. Actors working in a higher frequency level can utilize the accumulated slacks, then they can work in much lower frequency levels, then the

related energy consumption can be reduced, at the same time, the reconfiguration times can be reduced. However, it is possible to waste some available slacks because of data dependency. Some slacks cannot be used because of mapping. For example, the mapping determines for example which processor unit of the platform executes which actor and which communication units realize the dependencies between the actors. It influences the data/control dependency between different actors which in turn may lead to many slacks wasted.

The run-time switch Run-time-3 has another problem, it requires global knowledge to predict next scenario and decide when to use the slack. This consumes additional energy, but it is not included in the experimental evaluation. If we add them together, the method will consume more energy than the existing Run-time-1. So an accurate additional analysis is needed to investigate the quality of run-time method 3. We leave this as the future work.

# Bibliography

[1]Damavandpeyma, M., Stuijk, S., Basten, T., Geilen, M., & Corporaal, H. Throughput-constrained DVFS for scenarioaware dataflow graphs. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*(RTAS '13). IEEE Computer Society, Washington, DC, USA, 175-184.

[2] Zimmermann, J., Bringmann, O., & Rosenstiely, W. (2012, March). Analysis of multi-domain scenarios for optimized dynamic power management strategies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*(pp. 862-865). IEEE.

[3] Yang, Y., Geilen, M., Basten, T., Stuijk, S., & Corporaal, H. (2012, March). Playing games with scenario-and resource-aware SDF graphs through policy iteration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012* (pp. 194-199). IEEE.

[4] Geilen, M., & Stuijk, S. (2010, October). Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*(pp. 125-134). ACM.

[5] Stuijk, S., Ghamarian, A. H., Theelen, B. D., Geilen, M. C. W., & Basten, T. (2008). *FSM-based SADF*. MNEMEE internal report, TU Eindhoven.

[6] Gheorghita, S. V.,(2007,Dec) Dealing with dynamism in embedded system design: Application Scenarios. PhD thesis. Eindhoven University of Technology. Department of Electrical Engineering.

[7] Brekling, A., Hansen, M. R., & Madsen, J. (2008). Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, *77*(1), 1-19.

[8] Sander, I., & Jantsch, A. (2004). System modeling and transformational design refinement in forsyde [formal system design]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *23*(1), 17-32.

[9] Stuijk, S. (2007). Predictable mapping of streaming applications on multiprocessors. *Dissertation Abstracts International*, *68*(04).

[10] Ghamarian, A. H., Geilen, M. C. W., Basten, T., & Stuijk, S. (2008, March). Parametric throughput analysis of synchronous data flow graphs. In *Design, Automation and Test in Europe, 2008. DATE'08* (pp. 116-121). IEEE.

[11] Damavandpeyma, M., Stuijk, S., Geilen, M., Basten, T., & Corporaal, H. (2012, September). Parametric throughput analysis of scenario-aware dataflow graphs. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on* (pp. 219-226). IEEE.

[13] Baccelli, F., Cohen, G., Olsder, G. J., & Quadrat, J. P. (1992). *Synchronization and linearity* (Vol. 3). New York: Wiley.

[14] Geilen, M. (2010). Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems (TECS)*, *10*(2), 16.

[15] Theelen, B. D., Geilen, M. C. W., Basten, T., Voeten, J. P. M., Gheorghita, S. V., & Stuijk, S. (2006, July). A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on* (pp. 185-194). IEEE.

[16] Gheorghita, S. V., Palkovic, M., Hamers, J., Vandecappelle, A., Mamagkakis, S., Basten, T., ... & Bosschere, K. D. (2009). System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *14*(1), 3.

[17] Sears, A., & Jacko, J. A. (Eds.). (2007). *The human-computer interaction handbook: fundamentals, evolving technologies and emerging applications*. CRC Press.

[18] Design, S. B. (1995). Envisioning Work and Technology in System Development. *Ed JM Carroll*, 279-30895.

[19] Gheorghita, S. V., Palkovic, M., Hamers, J., Vandecappelle, A., Mamagkakis, S., Basten, T., ... & Bosschere, K. D. (2009). System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *14*(1), 3.

[20] Douglass, B. P. (2004). *Real Time Uml: Advances In The Uml For Real-Time Systems, 3/E*. Pearson Education India.

[21] Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *Software, IEEE*,*12*(6), 42-50.

[22] Jang, W., & Pan, D. Z. (2011). A voltage-frequency island aware energy optimization framework for networks-on-chip. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, *1*(3), 420-432.

[23] Niyogi, K., & Marculescu, D. (2005, January). Speed and voltage selection for GALS systems based on voltage/frequency islands. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference* (pp. 292-297). ACM.

[24] Nelson, A., Moreira, O., Molnos, A., Stuijk, S., Nguyen, B. T., & Goossens, K. (2011, August). Power minimisation for real-time dataflow applications. In *Digital System Design (DSD), 2011 14th Euromicro Conference on* (pp. 117-124). IEEE.

[25] Shin, D., & Kim, J. (2003, August). Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *Proceedings of the 2003 international symposium on Low power electronics and design* (pp. 408-413). ACM.

[26] Alimonda, A., Carta, S., Acquaviva, A., Pisano, A., & Benini, L. (2009). A feedback-based approach to dvfs in data-flow applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *28*(11), 1691-1704.

[27] Choudhury, P., Chakrabarti, P. P., & Kumar, R. (2007, January). Online dynamic voltage scaling using task graph mapping analysis for multiprocessors. In *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on* (pp. 89-94). IEEE.

[28] Bhattacharyya, S. S., Murthy, P. K., & Lee, E. A. (1999). Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, *21*(2), 151-166.

[29] Dasdan, A. (2004). Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *9*(4), 385-418.

[30] Garg, S., & Marculescu, D. (2008). System-level throughput analysis for process variation aware multiple voltage-frequency island designs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *13*(4), 59.