Title	A Study of Reducing Jitter and Energy Consumption in Hard Real-Time Systems using Intra-task DVFS Techniques
Author(s)	Tseng, Bo-Yu
Citation	
Issue Date	2018-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/15464
Rights	
Description	Supervisor:田中　清史，先端科学技術研究科，修士（情報科学）

A study of reducing jitter and energy consumption in hard real-time systems using Intra-task DVFS techniques

BO-YU TSENG

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
September, 2018

Master's Thesis

A study of reducing jitter and energy consumption in hard real-time systems using Intra-task DVFS techniques

1610105 BO-YU TSENG

Supervisor : Associate Professor Kiyofumi Tanaka
Main Examiner : Associate Professor Kiyofumi Tanaka
Examiners : Professor Mineo Kaneko
Professor Yasushi Inoguchi
Associate Professor Yuto Lim

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]

August, 2018

Abstract

In some real-time control applications, the predictability of task's timing behaviour is as important as energy consumption. That predictability includes the response time and short finish time jitter. This thesis presents a jitter-aware Intra-task DVFS scheme for mitigating finish time jitter in hard real-time systems; meanwhile, the system still can consume energy efficiently. This work exploits Dynamic Voltage and Frequency Scaling (DVFS) technique to proactively manipulate actual execution/response times of tasks. The strategy proposed in this paper mainly applies control and data flow analysis of task program to insert additional frequency scaling codes (instructions to change processors voltage and frequency). Moreover, it determines the appropriate frequency scaling factor. Through evaluation by multitasking simulation, it is shown that jitter can be reduced by up to 16.2%-19.4%, and energy saving by up to 13.6%-18.39% as side effect.

Acknowledgments

First of all, I would like to extend my sincere gratitude to my supervisor, Associate Professor Kiyofumi Tanaka, who has been helping me a lot of things during my master programme in addition to previous research student duration. He is really a responsible and professional advisor. During this two years and a half, he was always willing to be patient with me, especially whenever I got stuck in designing the algorithm, literature review, or even had problem on my presentation and writing skill (e.g., making a presentation within laboratory seminars, writing the papers and thesis). Furthermore, I really need to appreciate him that he helped me to write the recommendation for scholarship application. Thanks to his support, I could get scholarship. Although I did not study well and accomplish any successful research at all, I will still keep going in the following three-year's doctoral programme until I become a good researcher and let him be proud of me.

Next, I must thank for the NTT DOCOMO, INC., which granted me two-year scholarship. Thanks to their financial aid, I did not need to worry about the tuition fee and daily expenses. In addition, because of this scholarship, I feel I have much more responsibility in the future for doing well in my research and making contribution to our industrial field.

Besides, I would like to thank to my laboratory senior, Amr Mostafa M. Ashmawy and Doan Duy. They also kindly assisted me whenever I had difficulties or got stress in my research, e.g., research plan, programming (when I was confused about how to design the simulator efficiently), and even correcting my English all the time. Moreover, a special thank to Dr.Diego Pinheiro from Institute of Computing IComp, Federal University of Amazonas, who provided me the analysis toolset for analysing the benchmark programs, and spent few time to teach me how to use the tool, although we have never met each other.

Last but not least, I really have to express my gratitude to my lovely family (including my family's future member) who are always with me since I decided to go for master degree. It is definitely not easy to study aboard, there are a lot of pressure from financial and psychological aspects. Thanks to their unconditional support, encouragement and love, and without which I would not have come this far.

BO-YU TSENG

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the hard real time systems, every periodic task is required to perform deterministic timing behaviour, i.e., explicit timing constraint (specific deadline, low jitter). Among them, schedulability of multitasking is the highest criticality in hard real-time system which is not allowed to be violated. To maintain the schedulability, feasible task scheduling algorithms have been studied for decades. However, predictability of response times of periodic tasks is also a critical concern in some real-time applications, e.g., control system or data acquisition. In the context of task scheduling, due to the task interaction (preemption, precedence constraint or synchronisation protocol), memory access latency, shared-resource contention or I/O device accesses, systems may face great runtime variation in response time, called finish time jitter. Especially, in the real-time control applications, finish time jitter can be tolerated whilst it does not lead a degradation of system performance. However, large jitter can impact predictability in terms of timing domain. This unpredictability can degrade stability (some fluctuation behaviour) or processing accuracy of system [11, 22]. Thereby jitter ought to be kept as low as possible.

The objective of this paper is to reduce finishing time jitter under Rate-Monotonic scheduling (RM) whilst keeping the system consuming energy efficiently. A **jitter-aware Intra-task DVFS scheme** is proposed to make task scheduling adapt to runtime variations due to both interference and execution time. The Intra-task DVFS approach [25, 31, 33, 35] promises finer granularity of frequency scaling within one instance of task's execution. Thus, it relatively outperforms the Inter-task DVFS approach in terms of energy reduction. Apart from the effect of energy efficiency, it is expected that the Intra-task DVFS approach manipulates finishing time jitter. The proposed algorithms target at reducing variation in both execution time and interference time.

This work is the first to control the finishing time jitter using Intra-task DVFS, to the best of the authors knowledge.

Chapter 2

Related Work

As stated in Chapter 1, this research is to mitigate the finish time jitter whilst keeping the energy efficiency using the Intra-task DVFS techniques. Therefore, it is necessary to introduce the concept of DVFS, how the task scheduling can cooperate with DVFS mechanism, and the existing solutions of jitter reduction.

2.1 Dynamic Voltage and Frequency Scaling

There are enormous techniques which have been proposed for reducing energy consumption or thermal issue in CMOS devices (e.g., processor). One of the widely used approaches is *Dynamic Voltage and Frequency Scaling (DVFS)*. Nowadays, most of processors (e.g., Intel XScale PXA270 [20], Texas Instruments Sitra AM335x [16], AMD Zacate [27]) support the DVFS mechanism for energy efficiency of embedded system platforms, laptop or cloud datacenters [37] when they are in active states. With the DVFS mechanism, processors can perform multiple levels of supply voltage and operating frequency in various system requirement. For instance, whilst the application's processes do not need to be executed at the high computational performance, processor may scale down the operating frequency and its corresponding supply voltage to avoid energy dissipation.

Specifically, DVFS technique is to deal with energy consumption caused by *dynamic power* ($P_{dynamic}$)[1]. When the processor is operating at clock frequency f with supply voltage V_{dd}. The $P_{dynamic}$ can be written as follows:

$$P_{dynamic} = \alpha \times C_{eff} \times V_{dd}^2 \times f \tag{2.1}$$

where α is the switching activity, C_{eff} is the load capacitance.

According to the equation 2.1, the dynamic power is proportional to the operating frequency and quadratically proportional to the supply voltage. Therefore, the processor reduces operating frequency and supply voltage together, to achieve the energy saving.

2.2 Energy/Power-Aware Scheduling

In the energy-aware task scheduling, the DVFS techniques are classified into Inter-task and Intra-task according to the scaling granularity. Inter-task DVFS [2, 7, 12, 19] determines the supply voltage and operating frequency for each execution duration on task-by-task perspective. That is, system assesses adequate operating frequency(and its corresponding supply voltage) for every instance of tasks. On the other hand, Intra-task

[1]Energy consumption of a CMOS device includes two sources [36]: dynamic power and static power.

DVFS [3,25,28,29,31–33,35,39] adjusts the supply voltage and operating frequency within each individual-task boundary. Therefore, the major difference between them is that the former deals with the existing slack time during any two consecutive instances of tasks, whereas the latter manages to predict the upcoming slack time generated by the running task itself. The Figure 2.1 to 2.3 illustrate the difference between Inter-task and Intra-task DVFS scheduling. The Figure 2.1 is the original task scheduling without DVFS mechanism. Because both task 1 and task 2 finish their execution times earlier than *worst-case execution times*, system leaves some slack times in which the processor is in idle state. Consequently, in Figure 2.2, if Inter-task DVFS mechanism is enabled, every instance of any task will combine the slack time (left by previous running task) with its available execution and decide a lower operating frequency to run through. On the other hand, in Figure 2.3, if Intra-task DVFS mechanism is enabled, every instance of any task will be run under multiple operating frequency by predicting the upcoming slack time arising from the current instance's execution itself.

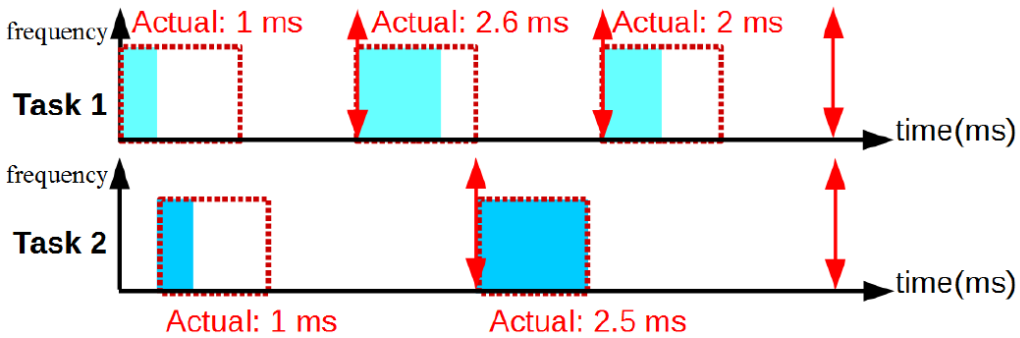


Figure 2.1: Original task scheduling

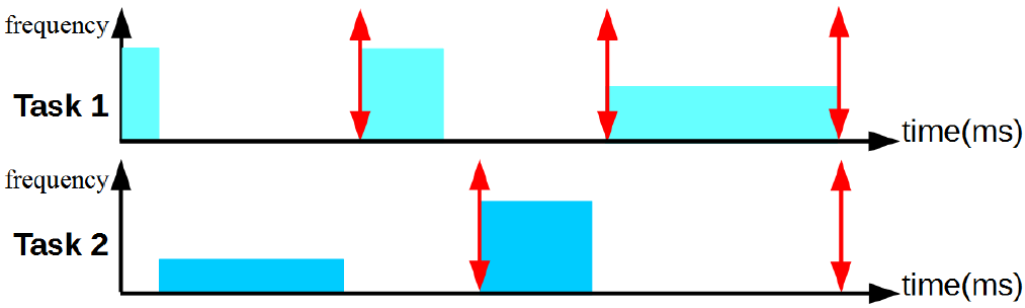


Figure 2.2: Inter-task DVFS

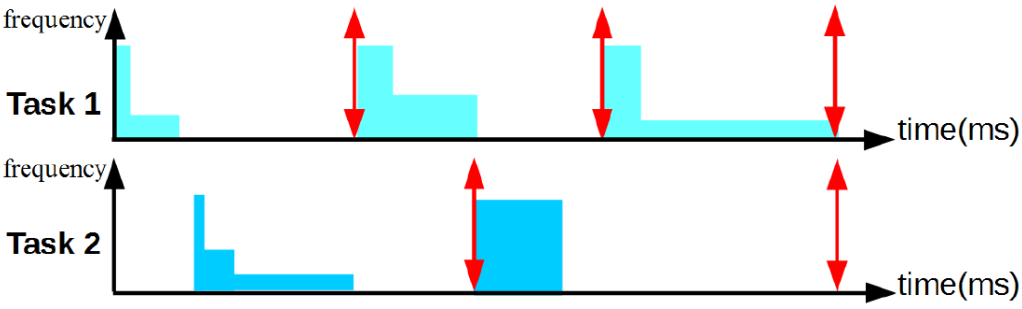


Figure 2.3: Intra-task DVFS

2.3 Intra-task DVFS

As for the main scheme of Intra-task DVFS, the objective is to decide in which execution points the processor should adjust its processing speed within the execution of one task's program; moreover the specific *speed-updated ratio* is assessed in order to reduce the energy consumption whilst preventing deadline miss. There are two different approaches to achieve Intra-task DVFS, i.e., stochastic-based Intra-task [38] and path-based Intra-task [25, 29, 31–33]. The former divides every task's program into several code sections and furthermore analyses the stochastic information of each code section (e.g., average execution time, probabilistic distribution, etc.). On the other hand, the procedures of path-base Intra-task can be generally organised into timing analysis (control and data flow information), placement of program checkpoint and calculation of speed-updated ratio. First, timing analysis can give tight WCET bounds of every program by the *control flow graph* (CFG) and data flow information (e.g., loop dependencies). It provides the hints about the locations where the runtime variation(i.e., the causes of slack times in the future) may occur. The example is the appearance of branch instruction or loop (a multiple iteration of branch instruction). Second, the program checkpoints are inserted at some program regions in order to trace the actual control flow during runtime. Accordingly, the locations of checkpoints can be at the exits of branch instructions and loops (corresponding to the *B-type voltage-scaling point* and *L-type voltage-scaling point* respectively [25, 31, 32]). Thus, the actual execution flow of one task within any instance becomes clear. Last, every program checkpoint decided by previous step is regarded as a candidate for speed adjustment (voltage/frequency scaling). If the successive execution path (of the current control flow) where it is directing is not the longest path (the execution path taking the largest execution cycles among its all branches), processor slows down the processing speed but still makes task complete before the given WCET/deadline and overall energy consumption can be further reduced.

The existing approaches of Intra-task DVFS scheme are summarised in the illustration of Figure 2.4.

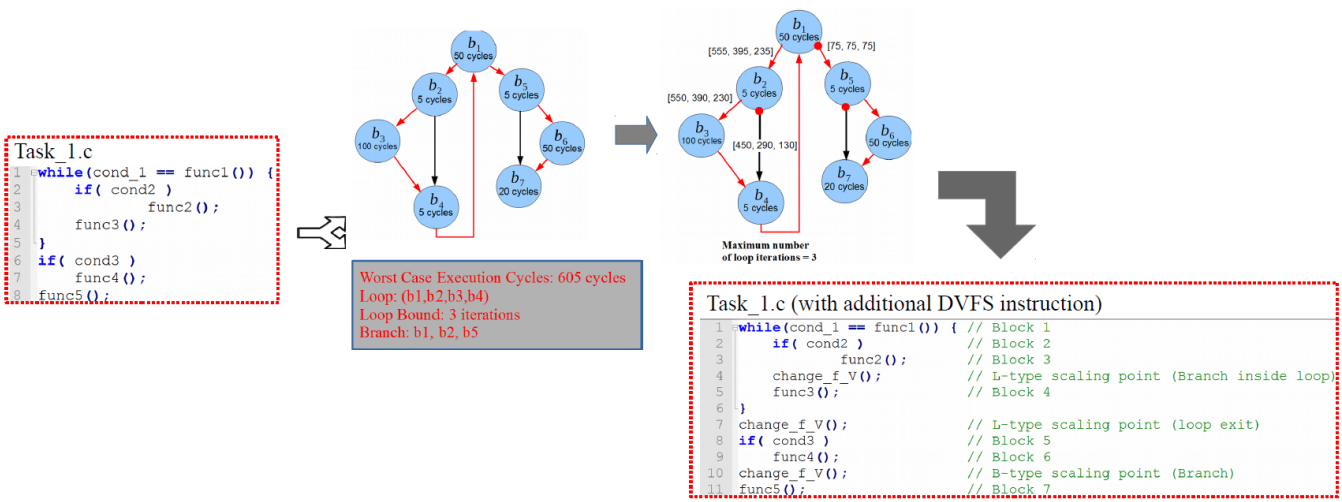


Figure 2.4: The workflow of existing Intra-task DVFS scheme

2.4 Jitter Reduction

To reduce jitter, a *deadline assignment algorithm* by linear programming was proposed [17] under the EDF scheduling environments [21]. Deadline assignment attempts to shorten

relative deadlines of some periodic tasks whilst keeping the schedulability, by promoting priorities of certain tasks to reduce the number of preemptions. Variation in preemption duration makes contribution to jitter. Accordingly the less the preemption, the less the jitter. In addition, some similar researches propose the period adjustment algorithms [6, 22] for fixed-priority preemptive scheduling environment. Especially, they apply the notion of *jitter margin* [11] (it will be introduced in Chapter 3) with feedback loop control technique to reduce the impact of task scheduling on the presence of jitter. Although those ideas reduce the finish time jitter by shortening the start time jitter, the variance of actual execution time is not directly handled. In particular, if the periods of tasks are much greater than finish time jitters then the effectiveness obtained from such approach is relatively small; moreover, the nature of their approaches manages to change the priorities of some tasks from original task schedules, whilst the literature [15] shows that it may violate the schedulability of the system in some cases.

Other works exploit DVFS to handle jitter [1, 23, 24]. They proves that DVFS mechanism enables the system to control the actual execution/response times of periodic tasks, thus it is applicable to reducing finish time jitter. Mochocki, et al. exploit only the suitable portion of slack time to scale down the operating frequency for some lower-priority tasks instances instead of aggressively using all slack time for energy reduction [23]. However, their work is based on Inter-task perspective, hence frequency scaling can be performed only at the start time of every instance of tasks.

On the other hand, Phatrapornnant and Pont point out the issue of *DVFS-induced variance* in task scheduling. This issue is discussed that, the usage of DVFS-based task scheduling schemes mainly aim at minimising systems energy consumption under the constraints of schedulability. But it also incurs a great deal of uncertainty about actual response times (especially the duration of execution time).

Hence they proposed a similar jitter-aware DVFS algorithm called TTC-jDVS algorithm. This algorithm manages to suppress the impact of DVFS on response times's predictability. Firstly, It assumes that some tasks require low jitter (defined as reduced-jitter tasks) and others are unnecessary. Then system would intentionally add certain length of delays after the completions of particular tasks which are not reduced-jitter tasks. Thus, the following reduced-jitter tasks only can get fewer slack time to perform the DVFS operation. Consequently the aggressive frequency scaling can be avoided. However, despite of their effort on TTC-jDVS algorithm, the variation in actual execution time is still not dealt with directly (the finish time jitter is still overlooked).

Chapter 3

Static Timing Analysis toward Jitter

3.1 Characteristic of Real-Time Task

In the real-time systems, the features of every periodic task τ_i in timing domain are depicted in Figure 3.1, where

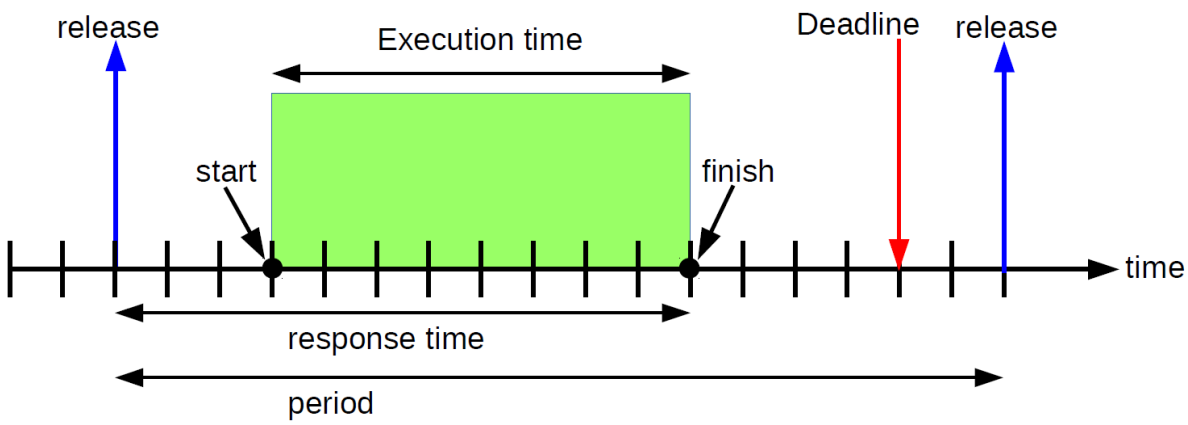


Figure 3.1: The timeline of one periodic task's feature

1. **Release.** The time at which task requests for execution; it also means the time at which task becomes ready for execution.
2. **Execution time.** The total amount of processing time necessary for execution without interruption.
3. **Deadline.** The maximum time within which task must complete its execution.
4. **Start.** The time at which system starts to run the task.
5. **Finish.** The time at which system completes the execution of task.
6. **Response time.** The total amount of time for system to respond to request of task, i.e., the interval between release and finis time.
7. **Period.** The interval between task's consecutive release times, i.e., the request rate of execution.

For ease of timing analysis by mathematics, those features can be denoted by the notations below.

- $\mathbf{r_{i,j}}$ denotes the *release time* of jth instance of task τ_i.
- $\mathbf{s_{i,j}}$ denotes the *start time* of jth instance of task τ_i.
- $\mathbf{T_i}$ denotes the *period* of task τ_i.
- $\mathbf{d_{i,j}}$ denotes the *absolute deadline* of jth instance of task τ_i.
- $\mathbf{D_i}$ denotes the *relative deadline* of task τ_i.
- $\mathbf{R_{i,j}}$ denotes the *response time* of jth instance of task τ_i.
- $\mathbf{f_{i,j}}$ denotes the *finish/completion time* of jth instance of task τ_i.
- $\mathbf{C_i}$ denotes the *execution/computation time* of task τ_i.
- $\mathbf{prt_i}$ denotes the *priority* which is assigned to task τ_i in task scheduling.

3.2 Rate Monotonic Scheduling Algorithm

This research is based on the *Rate-Monotonic* algorithm as the task scheduling policy. Hence this section is to introduce the idea of it.

Rate-Monotonic (RM) is a fixed-priority preemptive algorithm for periodic tasks [9]. Its rule is to assign priority to every periodic task according to the length of its period. That means the periodic task with shorter period is assigned higher priority. Note that, since the period of every periodic task is constant, every task's priority is decided before execution and does not change during runtime.

Figure 3.2 is an example of Rate-Monotonic algorithm for scheduling three periodic tasks. Task 1 τ_1 has execution time 1 ms and period 6 ms (=relative deadline). Task 2 τ_2 has execution time 2 ms and period 7 ms. Task 3 τ_3 has execution time 3ms and period 10 ms. According to the RM algorithm, τ_1 has the highest priority, τ_2 has middle priority and τ_3 has the lowest priority. In the execution point 12 ms of Figure 3.2's timeline, the third instance of τ_1 is released; meanwhile, the second instance of τ_3 is still executing. In this case, because τ_1 has higher priority than priority of τ_3, the $\tau_{3,2}$ is suspended from its execution until $\tau_{1,3}$ finishes, which is called *preemption*.

3.3 The sources and definition of Jitter

In the real-time task scheduling, different instances of a task may vary their computational behaviour in timing domain. Such variation in timing domain is called *jitter*. According to the characterisation of a periodic task τ_i defined by [9], there are four types of jitter described below.

- **Relative Start Time Jitter.** The maximum variation between relative start times of any two consecutive instances, expressed by equation (3.1):

$$RRJ_i = \max_k |(s_{i,k} - r_{i,k}) - (s_{i,k-1} - r_{i,k-1}))| \tag{3.1}$$

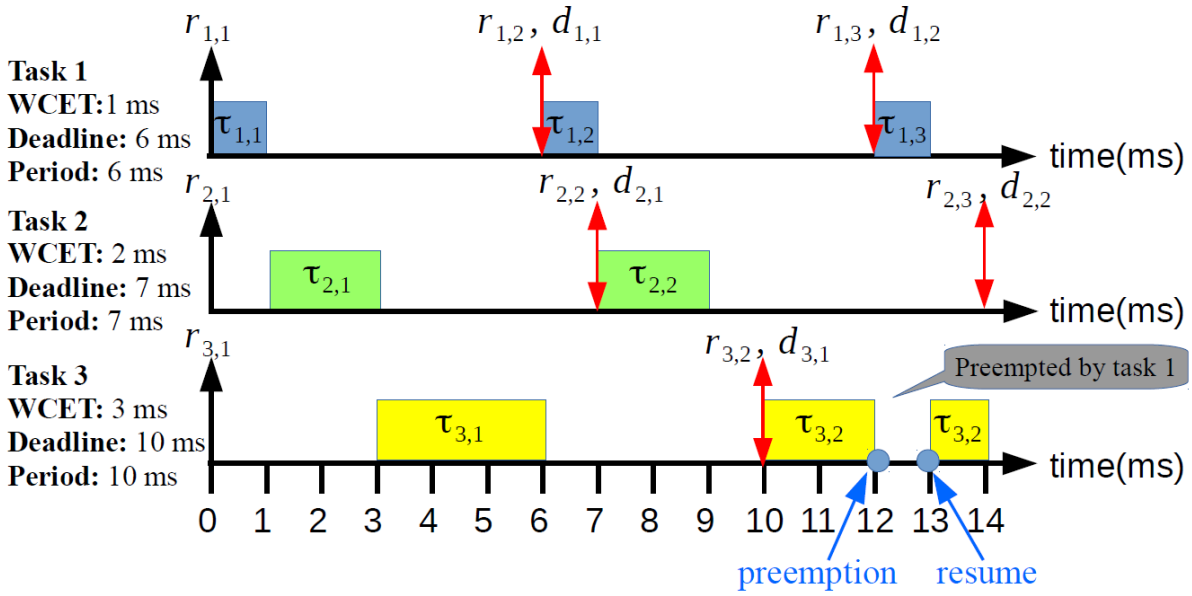


Figure 3.2: Example of Rate-Monotonic scheduling

- **Absolute Start Time Jitter.** The maximum variation of relative start times among all instances, expressed by equation (3.2):

$$ARJ_i = \max_k(s_{i,k} - r_{i,k}) - \min_k(s_{i,k} - r_{i,k}) \tag{3.2}$$

- **Relative Finish Time Jitter.** The maximum variation between response times of any two consecutive instances, expressed by equation (3.3):

$$RFJ_i = \max_k|(f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1}))| \tag{3.3}$$

- **Absolute Finish Time Jitter.** The maximum variation of response times among all instances, expressed by equation (3.4):

$$AFJ_i = \max_k(f_{i,k} - r_{i,k}) - \min_k(f_{i,k} - r_{i,k}) \tag{3.4}$$

According to the equation above, it is clear that start time jitter is one part of finish time jitter, hence finish time time jitter is the aim of jitter reduction metrics in this research. Furthermore, finish time jitter of one task is regarded as composition of **execution time variance** and **interference time variance**.

As for execution time variance, its causes can be traced by structure of its source code (described in Chapter 4.3) and target processor's architecture (such as pipline and cache hierarchy) [14, 26]. The former will be the main focus in this study. On the other hand,

interference time variance denotes runtime variation of the total duration in which the task is suspended in *Ready Queue* at any instance, i.e., the preemption duration. As a result, the interference time variance of a task is dominated by the timing behaviour of higher-priority tasks.

The Figure 3.3 shows an example of presence of finish time jitter with three periodic tasks under Rate-Monotonic scheduling[1]. The first periodic task τ_0 has period $T_0 = 40$ ms and worst-case execution time $WCET_0 = 10$ ms, the second task τ_1 has $T_1 = 80$ ms and $WCET_1 = 20$ ms. Last, the task τ_2 has $T_2 = 120$ ms and $WCET_2 = 30$ ms. And thereby those tasks show their priorities in a descend order. Because task τ_0 gets the highest priority, the variability of its response times among all instances only arises from runtime variation of actual execution time between different instances. On the other hand, the sources of finish time jitters happening to both task τ_1 and τ_2 do not only depend on their execution time variance, but also interference time variances which are inherently affected by their higher-priority tasks' jitters.

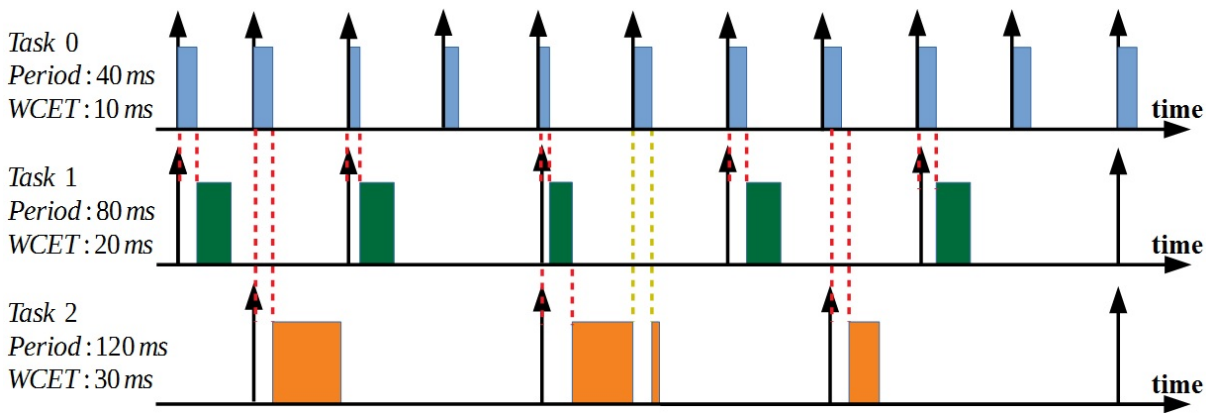


Figure 3.3: Example of Rate-Monotonic task scheduling

Accordingly, if a task τ_i is assigned a lower priority, then the more impacts from higher-priority tasks that the response time of task τ_i will be greatly uncertain. Hence the downside of low-priority task in terms of jitter reduction's complexity will be taken into account in proposed methodology of this research.

3.4 Modelling Timing Attributes of Jitter

To further quantify the possible finish time jitter of every periodic task regarding its WCET, period and the chosen task scheduling algorithm, Cervin et al. [10,11] introduced the notion of *jitter margin*. It defined the upper and lower bound of input-output jitter[2] of a periodic task in real-time control system, which can help designers to guarantee the system stability of their task scheduling algorithm. In their definition, the input-output delay means the response time of a control task is divided into a **constant delay,** $L \geq 0$ and **time-varying delay (the jitter),** $J_m \geq 0$ shown in Figure 3.4. The minimum possible delay is equal to L, and the maximum possible delay is given by $L+J_m$. Therefore, *jitter margin* is defined as the largest variance of input-output delay, $J_m(L)$.

Accordingly, this thesis redefines the viewpoint of original *jitter margin*. That is, every periodic task's execution must take system a constant interval of response time denoted as *constant response* $R_{constant}$. Moreover, system would spend an uncertain interval of

[1]In Rate-Monotonic scheduling, relative deadline D_i of one task τ_i is equal to its period T_i

[2]In the general real-time system's point of view, **input** and **output time** of a control task are equivalent to release and finish time respectively.

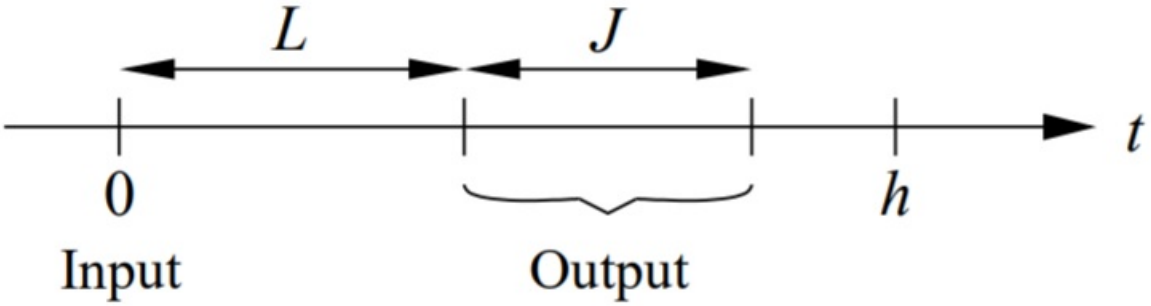


Figure 3.4: The input-output delay in a control task. The symbol h is period of the control task. [11]

response time denoted as *variance response* $R_{variance}$. The $R_{constant}$ and $R_{variance}$ of each periodic task can be computed by **response-time analysis** (for fixed priority preemptive scheduling environments) [8, 18].

- **Calculation of *best-case response time* ($\mathbf{BCRT_i}$):** it is assumed that task τ_i will spend its *best-case execution time* ($BCET_i$) at one instance whilst all higher-priority tasks which are released during τ_i's one period, also perform their BCETs. In this case, the *best-case response time* of τ_i ($BCRT_i$) is derived. The calculation of $BCRT_i$ can be done by the recursive manner in the equation

$$BCRT_i = BCET_i + \sum_{j \in hp(i)} \left\lceil \frac{BCRT_i}{T_j} - 1 \right\rceil \cdot BCET_j \tag{3.5}$$

  where $hp(i)$ indicates the set of tasks assigned higher priorities than task τ_i. Accordingly, the corresponding *best-case interference time* ($i^{best}(i)$) is $\sum_{j \in hp(i)} \left\lceil \frac{BCRT_i}{T_j} - 1 \right\rceil \cdot BCET_j$.

- **Calculation of *worst-case response time* ($\mathbf{WCRT_i}$):** first, defining the upper bound of **variance response** by the notion of *worst-case response time*. That is, assuming one instance of task τ_i takes its *worst-case execution time* ($WCET_i$); meanwhile, all higher-priority tasks released within the period of τ_i's instance will perform their WCETs as well. The $WCRT_i$ is given by the equation

$$WCRT_i = WCET_i + \sum_{j \in hp(i)} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \cdot WCET_j \tag{3.6}$$

  Accordingly, the corresponding *worst-case interference time* ($i^{worst}(i)$) is $\sum_{j \in hp(i)} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \cdot WCET_j$.

- **The bounds of $\mathbf{R_{constant}}$ and $\mathbf{R_{variance}}$:** based on the response-time analysis above, the $BCRT_i$ can be regarded as *constant response* $R_{constant}$.

$$R_{constant} = BCRT_i \tag{3.7}$$

On the other hand, because the maximum possible response time of τ_i is $WCRT_i$ thereby the range of *variance response* is within the interval of $[0, \mathbf{WCRT_i} - \mathbf{R_{constant}}]$, which is so-called *jitter margin* in this research's definition. It can be represented by a parametric way below

$$R_i^{variance} = \alpha_i \cdot (WCRT_i - BCRT_i) \tag{3.8}$$

which the α_i value ranges from 0.0 to 1.0 (0% to 100%).

Finally, the response time of a task τ_i can be illustrated by Figure 3.5.

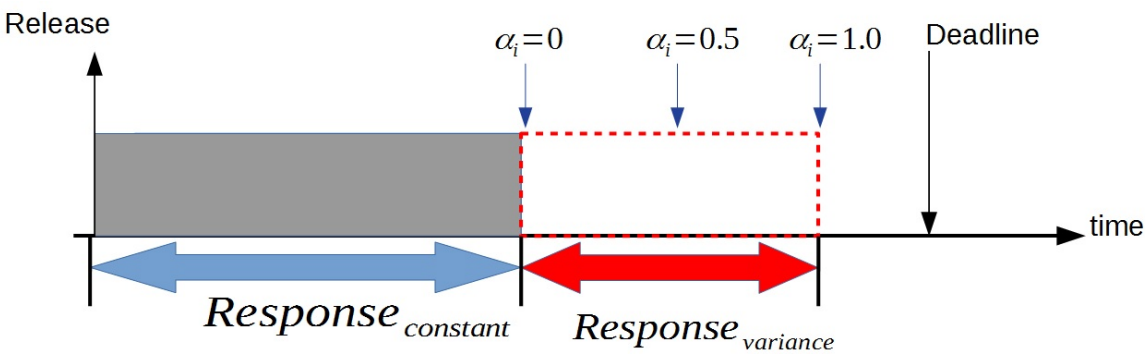


Figure 3.5: The target response time from the perspective of *Jitter Margin*

The detail of utilising this *jitter margin* for DVFS operation will be addressed in Section 4.6.

Chapter 4

Jitter-Aware Intra-Task DVFS Scheme

The timing attributes of finish time jitter of periodic tasks and its causes have been analysed in Chapter 3, and then bring out the clue how DVFS mechanism of target processor can handle the addressed problem. In order to reduce finish time jitter of periodic tasks, a DVFS-centric design called **Jitter-aware Intra-task DVFS scheme** is proposed in this chapter. Simply speaking, this scheme manages to provide a general systematic approach for DVFS mechanism of target processor to operate under hard real-time systems without modifying task scheduling policy of system kernel.

The chapter is organised into six sections: first shows a framework of proposed approach; second is a preliminary including some extension work about *runtime profiling* for system kernel; the following three sections are off-line works for giving DVFS operation guideline; and sixth is an runtime (on-line) work for performing DVFS operation.

4.1 System Framework

In this section, the framework of proposed **Jitter-aware Intra-task DVFS scheme** is presented, in which the implementation is based on the idea of existing Intra-task DVFS [31, 33] and makes an extension.

The main strategy of this scheme is to control the actual response time of periodic tasks (within any instance) by changing the processing speed of the system, according to the runtime variation in both interference and execution times. The overall framework of the proposed approach is shown in Figure 4.1. It consists of four phases, that is

1. **Control and data flow analysis.** Analysing the diversity of every periodic task's execution behaviours at source code level.

2. **Execution cycle estimation.** Evaluating the processing cost of each task's execution behaviours.

3. **Frequency-scaling point placement.** Determining the invocation points of DVFS operation within each task's runtime.

4. **Frequency-updated ratio calculation.** Deciding the new operating frequency for meeting given jitter constraints.

As shown in Figure 4.1, in the off-line stages, source code (C codes) of given target tasks are analysed in order to obtain their control flow graphs (CFGs) and data flow information. Then each basic block of CFGs is examined by **execution trace mining** [33]

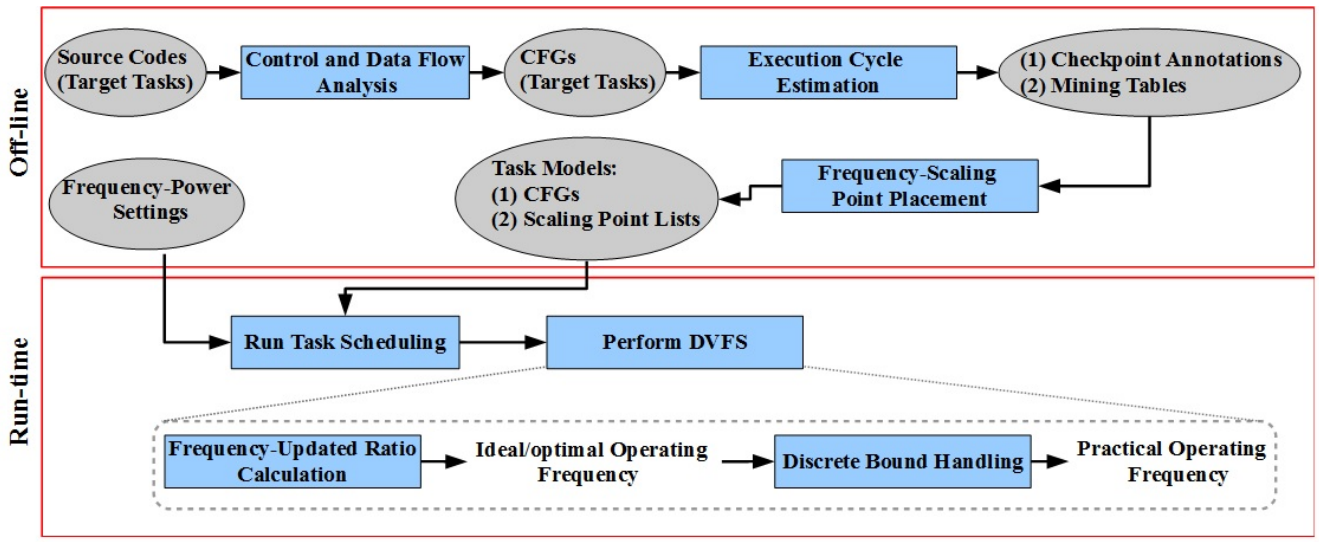


Figure 4.1: The framework of Jitter-aware Intra-task DVFS scheme

to record the worst-case remaining execution cycles (processing cost). Finally, locations of frequency-scaling points are determined. The details are described in Section 4.3 to Section 4.5.

In the run-time stage, the system mainly performs DVFS operation as a part of the task scheduling. The new operating frequency is decided by referring to the given frequency(and power) settings and scaling point lists. The details are described in Section 4.6.

4.2 Runtime Profiling

In this section, a simple *runtime profiling mechanism* is proposed to make system kernel clarify the specific runtime situation arising from finish time jitter at every instance of all periodic tasks. Those profiling informations will be the references for DVFS operation to decide an adequate *frequency-updated ratio*, detailed in Section 4.6, and also be evaluation metric of jitter reduction.

In any practical real-time operating system, there is one main data structure, called *Task Control Block* (TCB_i), for storing the feature of every task [9]. In particular, it characterises the set of tasks in a relation of $\Gamma\{\tau_i(Prt_i, C_i, T_i, D_i), i = 1, ..., n\}$ by designers, and provides the kernel for managing the task scheduling. The illustration of TCB_i is shown in Figure 4.2.

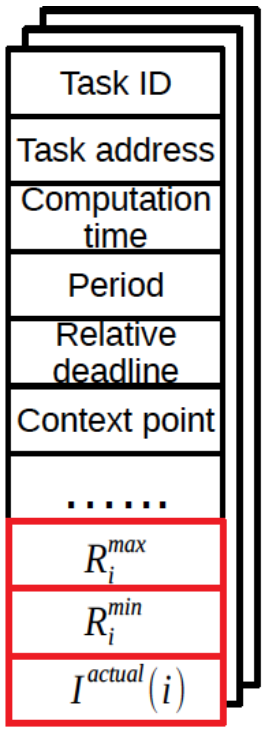
Based on this context, there are three additional control parameters added into TCB_i to get required profiling information: (i) *recorded maximum response time* $\mathbf{R_i^{max}}$, (ii) *recorded minimum response time* $\mathbf{R_i^{min}}$, and (iii) *actual interference time* $\mathbf{I^{actual}(i)}$). In addition, there is one global control parameter for the whole task set: *global slack time* $\mathbf{Slack_{global}}$.

4.2.1 Recorded Maximum/Minimum Response Time

R_i^{max} and R_i^{min} are used to record the maximal and minimal response time among all past instances of task τ_i. Assuming that $\tau_{i,j}$ is the currently running task, at start time of $\tau_{i,j}$, R_i^{max} is initialised to zero and R_i^{min} is initialised to ∞. Once τ_i finishes its execution, the system kernel will check if the actual response time of $\tau_{i,j}$, denoted $R_{i,j}^{actual}$, is bigger than R_i^{max} or not. Then updating R_i^{max} by assigning the value of $R_{i,j}^{actual}$ otherwise keeping the

```
typedef struct TCB {
    char TskID;
    proc (*addr)();
    int type;
    int state;
    long dline;
    int period;
    int prt;
    int wcet;
    float util;
    int *context;
    proc next;
    proc prev;
} tcb_t;
```

(a) An example of implementing TCB_i



(b) Structure of TCB_i

Figure 4.2: The task control blocks featuring tasks for system kernel to manage

current value of R_i^{max}. Similarly, updating R_i^{min} by assigning the value of $R_{i,j}^{actual}$ if $R_{i,j}^{actual}$ is smaller than R_i^{min}.

4.2.2 Updating the Recorded Slack Time

In regard to the execution time variance discussed in Chapter 3.3, once a running task finishes its current instance's execution, it may show an earliness on actual execution time. And then leaving a certain length of idle time during which processor serves none of tasks, such idle time is called *slack time* or *laxity*. This slack time will be one of basic factors that enables the usage of DVFS [2].

The $Slack_{global}$ is used to record such slack time arising from a running task at its completion time. However, when the ready queue is empty, that means that even if a running task finishes early no other task can make use it to perform DVFS. Thus, $Slack_{global}$ will be updated to currently remaining slack time otherwise reset to zero.

4.2.3 Updating Actual Interference Time

According to the interference time variance discussed in Chapter 3.3, the total interference time of a running task is dependent to higher-priority tasks' finish time jitter (how long the running task have to wait for completion of higher-priority tasks' preemptions). The *actual interference time* $I^{actual}(i)$ is proposed to record the maximum bound of interference time within every task's instance in order to give DVFS operation a tighter response time[1] against overestimation of $WCRT$.

[1] A more accurate interference time information inside the lower bound (best-case interference time) and upper bound (worst-case interference time)

At release time of the task, the $I^{actual}(i)$ of a running task is initialised to its $I^{worst}(i)$. Then $I^{actual}(i)$ will be updated by $I^{actual}(i) = I^{actual}(i) - Slack_{global}$ at start and resume time of the running task.

4.3 Control and Data Flow Analysis

From perspective of task's possible execution flows, the execution time variance always occurs whenever (i) there is a branch in the control flow [4] of task's source code and each partial execution path (right after the branch) involves different *execution cycles* (processing cost); or (ii) there is a loop and the number of actual loop iterations varies between instances' execution. Hence this research attempts to extract *control flow graph* (CFG) of every task' source code, and identify the *execution-varying factors* from those CFGs. In addition, to support prediction of the number of actual loop iterations during runtime, data flow analysis is made use of to trace the loop dependency in CFG [30, 35] (or called *data flow tracing*).

4.3.1 Extracting Control Flow graph

To extract CFG from every task's source code, an open source toolbox [5] for WCET analysis is utilised. The procedures are shown in Figure 4.3 and the result from the toolbox is visualised in Figure 4.4. In this example, the chosen task's source code is compiled to get binary code and then the generated binary code is imported to OTAWA WCET analyser. This toolbox can break the code into several *basic blocks* connected with arrows.

One basic block represents a sequence of instructions from corresponding binary code. Moreover, one basic block with multiple outgoing arrows represents that the last instruction inside the basic block is a *branch instruction*[2], such as the partial execution paths of

- **BB2→BB3→BB5→BB7→BB8→BB9→BB2→BB10→end**
- **BB2→BB10→end**

in Figure 4.4 show a presence of branch instruction in basic block B2. On the other hand, the partial execution path marked by grey colour in Figure 4.4 represents a *loop* region.

Accordingly, the presences of **branch** and **loop** inside CFGs are regarded as execution-varying factors in this research.

4.3.2 Data Flow Tracing against Loop Variability

Regarding the loop structure in a program, there are two data-flow boundaries defined in this research and a **for-loop** example is introduced in Figure 4.5.

- **Static loop bound ($\mathbf{Bound_{static}}$).** Defining the maximum iteration regardless of any variable or parameter assignment during runtime. The line 4 in Figure 4.5 shows ten as maximum possible iterations.
- **Runtime loop bound ($\mathbf{Bound_{runtime}}$).** Defining the actual iteration setting at runtime. The line 3 in Figure 4.5 decides the actual iteration, for instance, if x is assigned to 5 then the actual iteration of following for-loop will be five iterations.

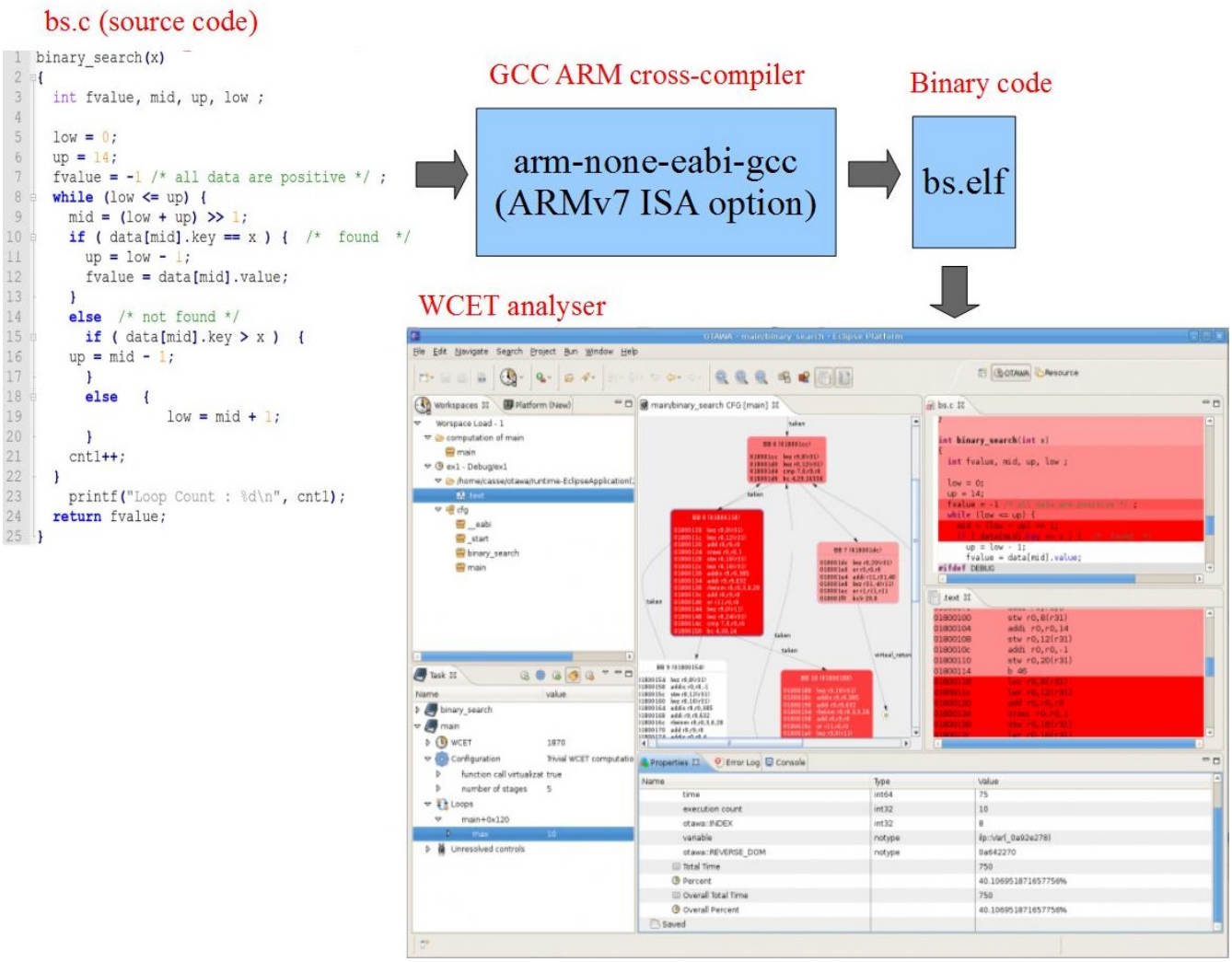
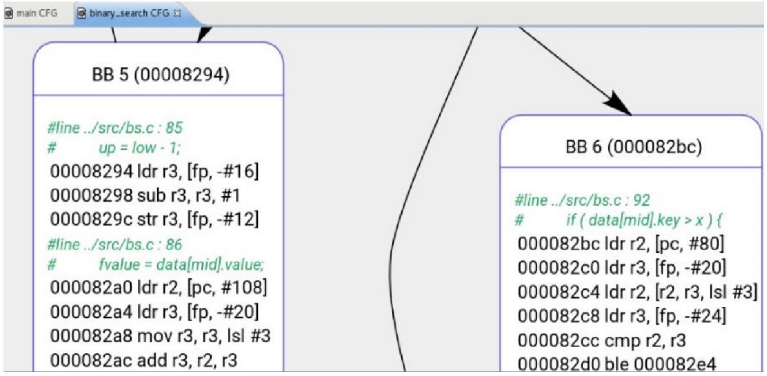


Figure 4.3: Extracting CFG using OTAWA WCET analyser

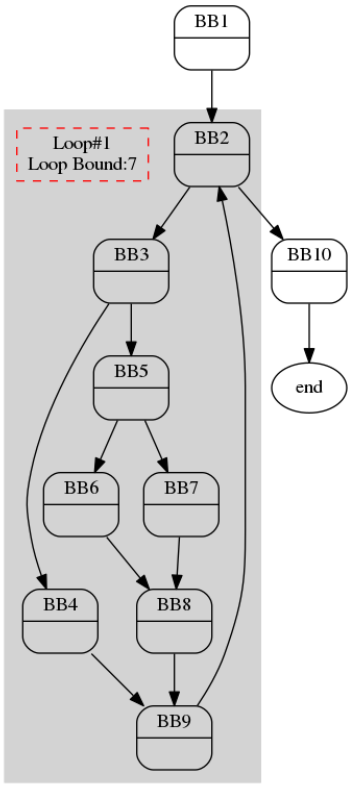
In [30,35]'s idea, they managed to move the frequency-scaling points to earlier execution point with respect to the task's actual control flow in any instance. They proved that if system can predict the execution time variance and perform DVFS operation earlier, then much more workload will be involved in new operating frequency hence energy consumption will be further reduced. Accordingly, this thesis makes use of their concept to let more basic blocks under the control flow deal with reduction of execution time variance (for ease of explanation and comprehension, another source code is given with corresponding CFG for illustration as shown in Figure 4.6).

Specifically, when a source code contains either **while-** or **for-loop**, the *runtime loop bound* is determined by particular variables. In the example of Figure 4.6a, the *runtime loop bound* of while-loop depends on the variable **x**. Thus it is regarded as the induction variable of the while-loop. Since the value of induction variable can differ from one instance to another, it varies task's execution time. Thereby the line 1 of Figure 4.6a is the point making system possible to predict the execution time early. Furthermore, the line 1's instructions are mapped to the basic block BB1 in Figure 4.6b, thus basic block BB1 is said to be **loop dependency** of the following loop region.

[2]In the C/C++ language, a branch instruction can be a **if-statement**.



(a) One basic block represents a sequence of intructions



(b) CFG of chosen task's program

Figure 4.4: Control flow of target task's source code

```
1:  void output(void)  {
2:    int i;
3:    x = func1(); /*x = 0 ... 9*/
4:    for(i = x; i < 10; i++)  {
5:      func2();
6:    }
7:  }
```

Figure 4.5: Example program with **for-loop**

4.4 Execution Cycle Estimation

4.4.1 Estimation of Processing Cost

The control flow graphs of target tasks are extracted, then in order to know the total processing costs of CFG's execution paths, a CFG-based computation tool called **cfg-wcec**[3] [25], is used to obtain the *execution cycles*.

The calculation of processing cost using cfg-wcec tool in this research, is done by the following procedures:

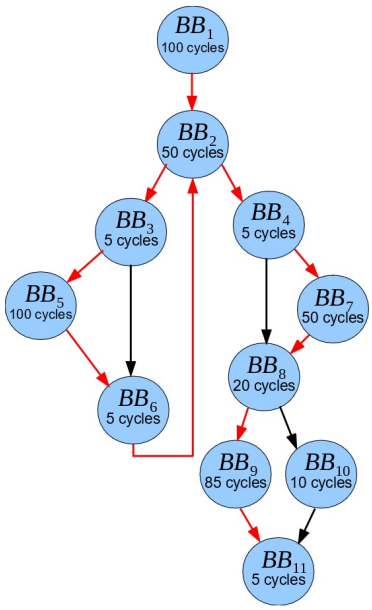
1. Converting CFG of every task (obtained from previous section) into **graph xml format (.graphml)** file [34], and inputting the .graphml file and task's assembly code to the tool.

2. Computing the number of execution cycles required for processing each basic block's

[3]The open source tool can be found on https://github.com/diegoValhalla/cfg-wcec

```
1:  x = func1();
2:  func2();
3:  while(x != 0) {
4:    y = func3();
5:    if(y != 0)
6:      func4();
7:    z = func5();
8:    x-=1;
9:  }
10: if(z != 0)
11:   func6();
12: a = func7();
13: if(a == 1)
14:   func8();
15: else
16:   func9();
17: func10()
```

(a) Source code



(b) CFG

Figure 4.6: An example of loop dependency

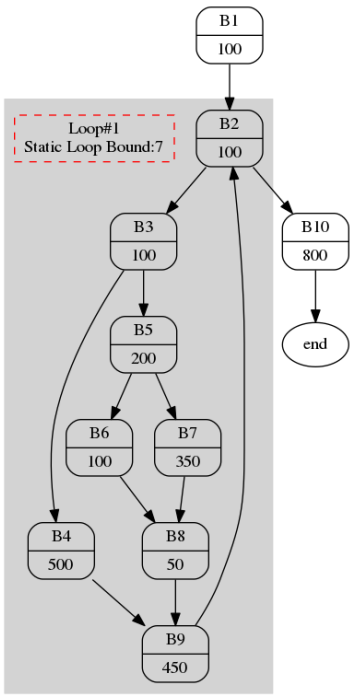
corresponding sequence of instructions. The example of Figure 4.4 is presented in Figure 4.7a. Additionally, the required execution cycles for running each instruction is based on the specification of target processor. The detail can be referred to Appendix A.

3. Finding the *best-case execution path (BCEP)* and *worst-case execution path (WCEP)*. For instance, the BCEP costing 1000 execution cycles, and WCEP costing 9750 exeuction cycles are shown in Figure 4.7b and Figure 4.7c respectively. In addition, the processing costs through BCEP and WCEP are denoted as *BCEC* and *WCEC* respectively.

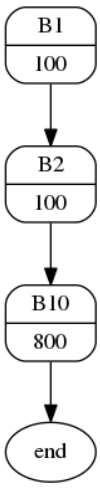
4.4.2 Checkpoint and Mining Table Placement

After analysing the CFGs and processing cost of each basic block, next step is to mark the location of execution-varying points and quantify the execution time variance from each of them. The approach here is to calculate the total processing cost of each partial execution path until the end of CFG's traversal (i.e., the basic block with the identifier: *end*). Specifically, there are two steps to make the quantification of the execution time variance happening at execution-varying point: (i) checkpoint insertion and (ii) *mining table* establishment [33].

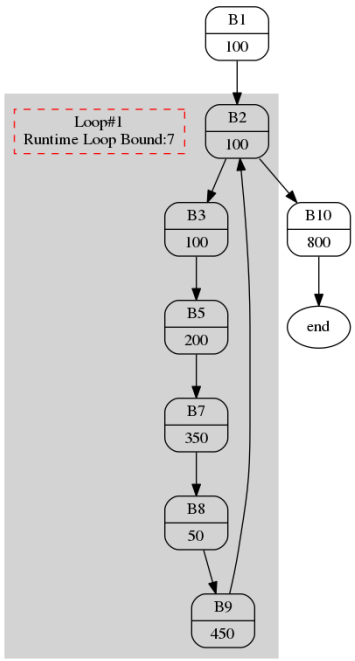
In the first step, it is expected that basic blocks corresponding to the three factors (branch, loop entry and loop dependency) should be selected as the location of execution-varying points, also known as *program checkpoints* in this thesis. A *B-type checkpoint* is inserted right after every branch's corresponding basic block in [33], in order to trace the execution time variance from each . Also, a *L-type checkpoint* is inserted at every loop exit to trace the variety of runtime loop bound (referring to the Section 4.3.2) in [25, 31]. However, the research manages to analyse more fine granularity of execution time variance in every CFG. Consequently, it not only makes use of B-type checkpoint from previous



(a) CFG with annotation of processing cost



(b) The bset-case execution path with processing cost of 1000 cycles



(c) The worst-case execution path with processing cost of 9750 cycles

Figure 4.7: The control flow graph after processing cost estimation

work but also redefines the L-type checkpoint and extends a new type of checkpoint against loop dependency, called *P-type checkpoint*.

In the second step, the purpose is to calculate *remaining worst-case execution cycles (RWCECs)* from each checkpoint to the end of the task's execution. The remaining worst-case execution cycles here mean the maximum processing cost through every checkpoint's successor execution paths. According to the **Execution Trace Mining [33]**, RWCECs of execution paths from each B-type checkpoint as well as their corresponding instruction addresses can be recorded in a *mining table*. Therefore, the approach here makes an extension of mining table, constructing the B-type, L-type and P-type mining tables which correspond to their types of checkpoints. The details of checkpoints and mining tables are described below.

- **B-type checkpoint and mining table.** The example shown in the Figure 4.8 presents a simple CFG containing a branch after basic block $Block_1$. There are two successor execution paths: (i) $\mathbf{Successor_1}$: $Block_1 \rightarrow Block_2 \rightarrow$end, and (ii) $\mathbf{Successor_2}$: $Block_1 \rightarrow Block_3 \rightarrow Block_2 \rightarrow$end. Hence a checkpoint is inserted right after $Block_1$.

  Referring to the Table 4.1, there are two successor execution paths from B-type checkpoint. And because none of them contains another branch or loop, hence the RWCECs under successor execution paths are exactly equal to their total processing cost, i.e., $RWCEC_{successor_1} = 150$ cycles and $RWCEC_{successor_2} = 350$ cycles. In addition, the first column, *Address*, indicates the location of its corresponding checkpoint.

  In summary, the RWCEC of B-type checkpoint can be formulated according to the

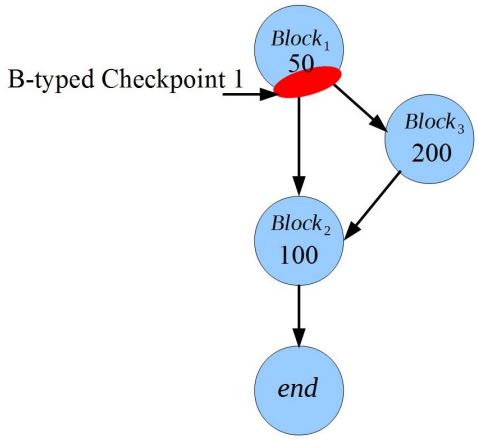


Figure 4.8: The location of B-type checkpoint

B-type Mining Table				
Address	$Successor_1$	$RWCEC_{successor_1}$	$Successor_2$	$RWCEC_{successor_2}$
$\#1(Block_1)$	$Block_2$	$100(cycles)$	$Block_3$	$300(cycles)$

Table 4.1: B-type mining table

mining table

$$RWCEC = \begin{cases} RWCEC_{successor_1} & ,(successor = Successor_1) \\ RWCEC_{successor_2} & ,(successor = Successor_2) \end{cases} \tag{4.1}$$

- **L-type checkpoint and mining table.** As for the insertion of L-type checkpoint in [25,31], they put the checkpoint at the end of every loop exit, only. Their strategy is to focus on the difference between the *static loop bound* and *runtime loop bound* which can be known after the process of the loop; meanwhile, if there are multiple partial execution paths in the loop (due to the presence of branch inside), task's execution will always branch to the longest path in terms of processing cost. For example, in the loop region in the Figure 4.9, five partial execution paths might occur during runtime, each of which is called *loop-inner execution path* in this thesis:

  1. $Block_1 \rightarrow Block_3$
  2. $Block_1 \rightarrow Block_2 \rightarrow Block_4 \rightarrow Block_6 \rightarrow Block_1$
  3. $Block_1 \rightarrow Block_2 \rightarrow Block_5 \rightarrow Block_4 \rightarrow Block_6 \rightarrow Block1$
  4. $Block_1 \rightarrow Block_2 \rightarrow Block_4 \rightarrow Block_7 \rightarrow Block_6 \rightarrow Block_1$
  5. $Block_1 \rightarrow Block_2 \rightarrow Block_5 \rightarrow Block_4 \rightarrow Block_7 \rightarrow Block_6 \rightarrow Block_1$

  It is clear that the fifth partial execution path will cost system the most execution cycles and thereby the original usage of L-type checkpoint always considers the only possibility of fifth partial execution path during every loop iteration. However, no matter how much the target reduces the energy consumption or finish time jitter, the execution time variance caused by such diverse partial execution paths in one loop can affect the effectiveness of DVFS operation significantly. According to the drawback of previous work, instead of inserting this type of checkpoint at loop exit, this research relocates the checkpoint to any *branch inside the loop* (called *loop-inner*

branch in this research). For instance, the $Block_1$, $Block_2$ and $Block_4$ are defined as checkpoints' locations.

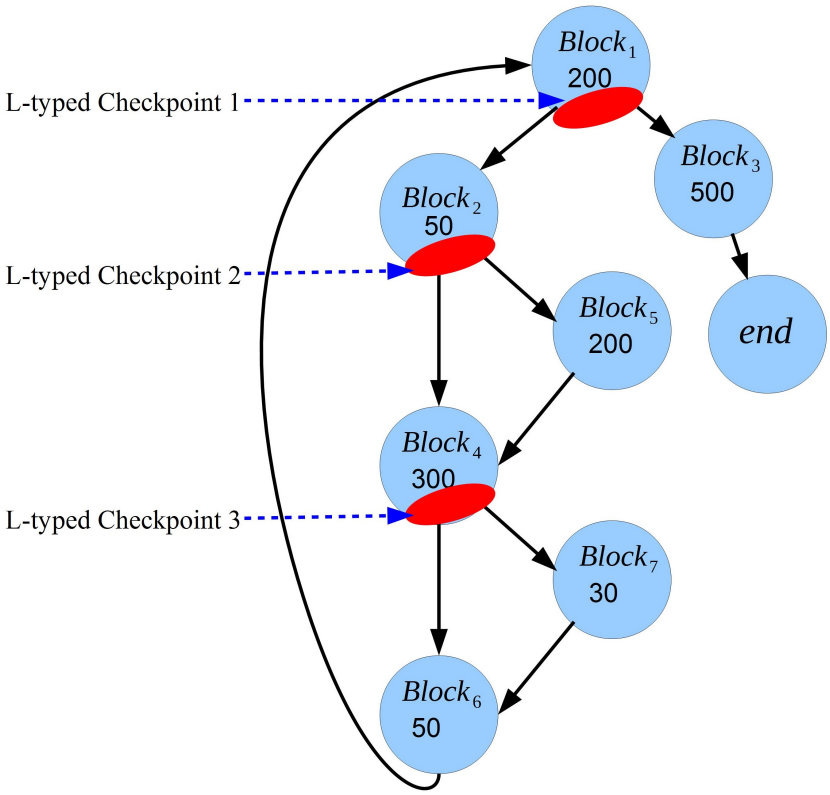


Figure 4.9: The location of L-type checkpoint

Similar to the B-type mining table, the P-type mining table records the RWCECs of successor execution paths of every *loop-inner branch*. In addition, the last column of the mining table, $RWCEC_{after\_loop}$, indicates the remaining worst-case execution cycles after the process of loop, i.e., the processing cost of path $Block_3 \rightarrow end$.

L-type Mining Table					
$\mathbf{Bound_{static}}$: 10 (iterations) $\mathbf{WCEC_{iter}}$: 830 (cycles) $\mathbf{Loop_{entry}}$: $Block_1$, $\mathbf{Loop_{exit}}$: $Block_1$					
Address	$Successor_1$	$RWCEC_{successor_1}$	$Successor_2$	$RWCEC_{successor_2}$	$RWCEC_{after\_loop}$
$\#1(Block_1)$	$Block_2$	$830(cycles)$	$Block_3$	$500(cycles)$	$500(cycles)$
$\#2(Block_2)$	$Block_4$	$580(cycles)$	$Block_5$	$780(cycles)$	$500(cycles)$
$\#3(Block_4)$	$Block_6$	$250(cycles)$	$Block_7$	$280(cycles)$	$500(cycles)$

Table 4.2: L-type mining table

There are three different possible RWCECs whilst task's execution reaches a L-type checkpoint. First case is that task branches to the $Successor_1$ referring to the $condition1$ in equation 4.2. Second case is that task branches to the $Successor_2$ referring to the $condition2$. Last case is that when task's execution passes through the loop exit ($Block_1$) and branches to a successor which does not belong to loop (such as $Block_1$ branches to $Block_3$), it also means the end of loop process. The

RWCECs of those three cases can be calculated by following equation.

$$RWCEC = \begin{cases} RWCEC_{successor_1} + WCEC_{iter} \cdot (Bound_{runtime} - Iter_{executed}) + RWCEC_{after\_loop} \ , (condition1) \\ RWCEC_{successor_2} + WCEC_{iter} \cdot (Bound_{runtime} - Iter_{executed}) + RWCEC_{after\_loop} \ , (condition2) \\ RWCEC_{after\_loop} \ , (otherwise) \end{cases} \tag{4.2}$$

where

- **condition1** : $successor = Successor_1 \wedge Address \neq Loop_{exit}$
- **condition2** : $successor = Successor_2 \wedge Address \neq Loop_{exit}$

Consequently, more accurate quantification of execution time variance in a loop is expected.

- **P-type checkpoint and mining table.** This type of checkpoint is designed to exploit the data flow tracing of $Bound_{runtime}$ predicting its value in advance (knowing the value of $Bound_{runtime}$ before the task's execution reaches upcoming loop process). The concept of P-type checkpoint is similar to the idea of original L-type checkpoint [25, 31] which takes the loop-inner execution path of maximum processing cost into account within every loop iteration. However, it is located at loop dependency's corresponding basic block instead of loop exit.

  Assuming that the $Block_1$ of Figure 4.10 contains the instruction deciding the $Bound_{runtime}$ and thereby $Block_1$ is considered as a loop dependency (according to the principle defined in Section 4.3.2). In this case, a P-type checkpoint can be inserted right after $Block_1$. Once task's execution finishes the process of $Block_1$ the P-type checkpoint here can predict that the workload required for upcoming loop process is equal to

$$\max\{\forall C(\textit{loop-inner execution paths})\} \cdot Bound_{runtime} \tag{4.3}$$

  where the $\mathbf{C}(\mathbf{x})$ indicates the processing cost of execution path $\mathbf{x}$.

  Subsequently, the P-type mining table is to give the identification that each loop region belongs to which specific loop dependency, by pointing the location of checkpoint to the loop entry of target loop region. Such as the Table 4.3, the location of one loop dependency is at $Block_1$ and the loop entry of its target loop region is $Block_2$. The fourth ($WCEC_{iter}$) column records the processing cost of $\max\{\forall C(\textit{oop-inner execution paths})\}$, and the fifth column ($RWCEC_{outside\_loop}$) records the required workload from checkpoint to the the end of CFG excluding the workload of the loop region. In this example, the processing cost of partial execution path $Blcok_4 \rightarrow end$ is the only required workload outside the loop region, thus the $\mathbf{RWCEC_{outside\_loop}}$: 500 cycles.

The RWCEC at the checkpoint is calculated by following equation.

$$\begin{aligned} RWCEC = C(LoopEntry) + Bound_{runtime} \cdot WCEC_{iter} + \\ RWCEC_{outside\_loop} \end{aligned} \tag{4.4}$$

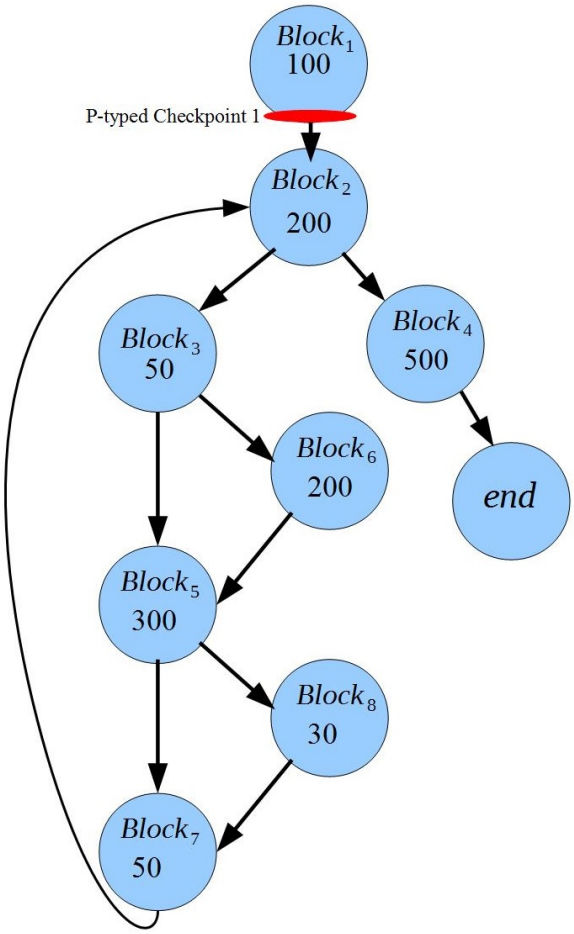


Figure 4.10: The location of P-type checkpoint

P-type Mining Table				
Address	*Loop Entry*	$Bound_{static}$	$WCEC_{iter}$	$RWCEC_{outside\_loop}$
$\#1(Block_1)$	$Block_2$	$10(iterations)$	$830(cycles)$	$500(cycles)$

Table 4.3: P-type mining table

In conclusion, the research manages to make P-type checkpoint cooperate with L-type checkpoint for dealing with the execution time variance arising from loop region. The procedure of their cooperation is described below.

1. First, P-type checkpoint is utilised to obtain the information of the number of actual loop iteration against upcoming loop, then operating frequency is adjusted against loop-iteration variance.
2. Second, the variance of *loop-inner execution paths* is further identified to handle the exact execution time of the loop region.

4.5 Frequency-Scaling Point Placement

In order to shorten the finish time jitter, variance of interference and execution time is handled. The main purpose of this phase is to determine when the system invokes DVFS operation to reduce those variances. There are four execution points where performing frequency-scaling is possible. The frequency-scaling point at each of them targets specific

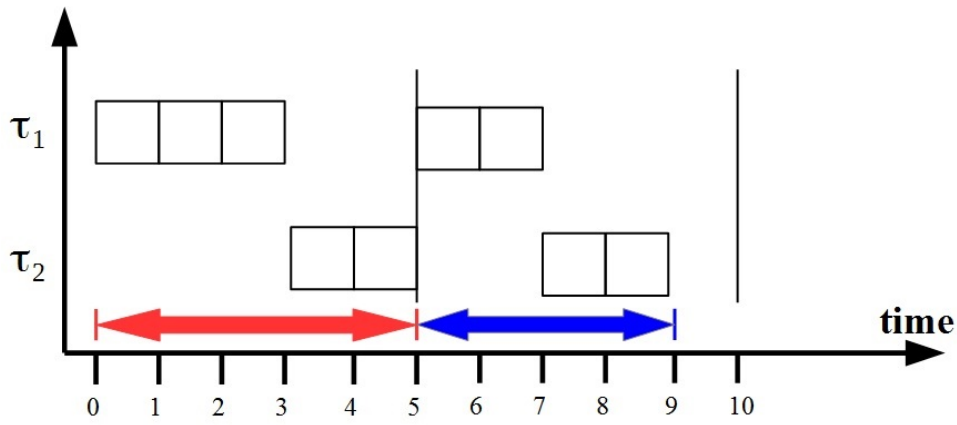


Figure 4.11: The finish time jitter caused by the variance of interference time

factor of jitter. The following bullet points describe the location of frequency-scaling point in addition to the summary/review of jitter factors addressed in Section 3 and 4.2.

- **Start-time frequency-scaling point.** Referring to the *actual interference time* in Section 4.2.3, this type of frequency-scaling point is placed at start time $s_{i,j}$. It aims to reconfigure a default operating frequency due to updated actual interference time. Namely, it focuses on interference time variance affected by higher-priority tasks.

  For example, in Figure 4.11, τ_1 has WCET of 3 and period(=Deadline) of 5, and τ_2 has WCET of 2 and period(=Deadline) of 5, with τ_1 having higher priority than τ_2. This example shows a finish time jitter of one tick for τ_2 where response times of $\tau_{1,1}$ and $\tau_{1,2}$ are not constant. This leads to different interference times on the τ_2's instances. It is obvious that the actual start time of lower-priority task is affected by higher-priority tasks. Here, frequency-scaling points are inserted at the start time of a lower-priority task. As a result, shorter response time for $\tau_{2,1}$ or longer response time for the $\tau_{2,2}$ is obtained, which can reduce the difference from all τ_2's instances.

- **B-type frequency-scaling point.** It is placed at every B-type checkpoint's corresponding execution point. It focuses on variant processing cost among successor execution paths of branches.

- **P-type frequency scaling point.** It is placed at every P-type checkpoint's corresponding execution point. It focuses on variant processing cost due to different $Bound_{runtime}$ on loops.

- **L-type frequency scaling point.** It is placed at every L-type checkpoint's corresponding execution point. It focuses on variant processing cost among loop-inner execution paths of every loop's iteration.

On the other hand, as the aforementioned variety of execution paths in one task's CFG, it incurs variation in execution time. Therefore, frequency-scaling points are placed at respective checkpoint's corresponding location. Such approach can equalise the response times of the running task even if it follows different execution paths. Table 4.4 shows an example of task's source code with its corresponding frequency-scaling points' location.

Finally, although performing DVFS operation at start time can handle the interference time variance, the exact resume times of the running task which is preempted by higher-priority tasks is another factor. A strategy for placing frequency-scaling points either (i)

(a) Source code	(b) CFG	(c) DVFS code

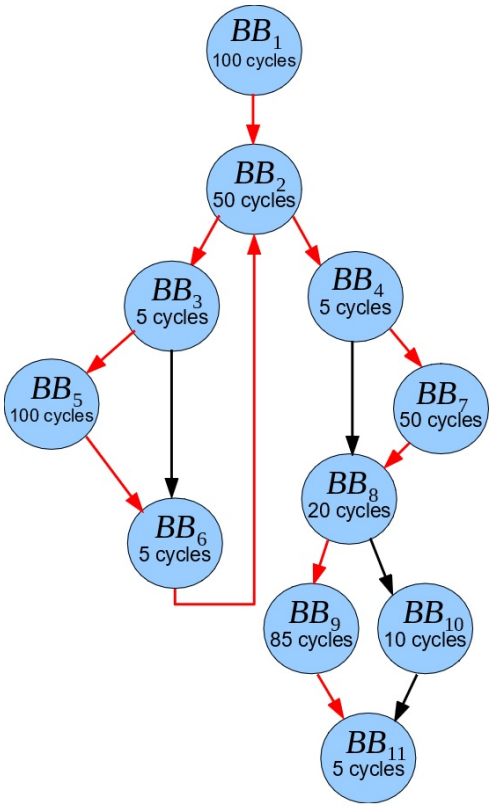
1: x = func1();
2: func2();
3: while(x != 0)
{
4: y = func3();
5: if(y != 0)
6: func4();
7: z = func5();
8: x-=1;
9: }
10: if(z != 0)
11: func6();
12: a = func7();
13: if(a == 1)
14: func8();
15: else
16: func9();
17: func10()
		1: x = func1();
		2: freq_scaling(P_type_1);
		3: func2();
		4: while(x != 0) {
		5: y = func3();
		6: if(y != 0) {
		7: freq_scaling(L_type_1);
		8: func4();
		9: }
		7: freq_scaling(L_type_1);
		10: z = func5();
		11: x-=1;
		12:}
		13: if(z != 0) {
		14: freq_scaling(B_type_1);
		15: func6();
		16:}
		17: a = func7();
		18: if(a == 1) {
		19: freq_scaling(B_type_2);
		20: func8();
		21:}
		22: else {
		23: freq_scaling(B_type_2);
		24: func9();
		25:}
		26: func10()

Table 4.4: The DVFS-aware code instrumentation in task's source code

right after resume time, (ii) or somewhere after resume time but before completion time, can further control response time. As a result, this enhancement is left for the future work.

4.6 Frequency-Updated Ratio Calculation

To identify and mitigate the execution and interference time variance, previous sections have already determined the specific execution points in which DVFS operation can be performed such as Table 4.4(c). Consequently, the next step is to design the functionality of DVFS operation. First, the response time constraint is imposed to every periodic task. Second, the frequency-updated ratio (frequency-scaling factor) is determined for respective frequency-scaling point. There are two different contexts of DVFS operatoin proposed in this research, (i) *static-based DVFS* and (ii) *profile-based DVFS*.

Regarding to the response time constraint, more specifically, giving every jitter-sensitive

task[4] τ_i^{jitter} an guideline called *target response time* $\mathbf{R_i^{target}}$. Ideally, once the DVFS operation is invoked by any frequency-scaling point of τ_i^{jitter}, the system starts calculating the frequency-updated ratio in the aim of making actual response time get closer/equal to given R_i^{target}. In this manner, the range of execution time variance and interference time variance can be mitigated even if each instance takes different amount of execution cycles.

4.6.1 Static-Based DVFS

Assignment of Response Time Constraint

According to the timing attributes of jitter described in Section 3.4, the lower bound and upper bound of finish time jitter of a periodic task, *jitter margin*, is assessed regardless of frequency scaling. This section further utilises the *jitter margin* to determine the **tolerable variance of response time** of every τ_i^{jitter}. Similarly, $\forall \tau_i^{jitter}$ are given one specified target response time ratio $\boldsymbol{\alpha_i}$ ranging from 0 to 1 (or 0% - 100%) by user in advance. Hence all instances of every τ_i^{jitter} are requested to respond with target response time:

$$R_i^{target} = BCRT_i + \alpha_i \cdot (WCRT_i - BCRT_i) \tag{4.5}$$

An example of specifying R_i^{target} is depicted in Figure 4.12. If the value of α_i is assigned to 0.0, it means that all j^{th} instances of τ_i^{jitter} are demanded to finish its execution at $r_{i,j} + BCRT_i$ as shown in Figure 4.12a. On the other hand, $\alpha_i = 0.6$ and $\alpha_i = 1.0$ mean all j^{th} instances of τ_i^{jitter} are demanded to perform the actual response time of $r_{i,j} + R_i^{variance} \cdot 0.6$ and $r_{i,j} + WCRT_i$ respectively, as shown in Figure 4.12b and 4.12c.

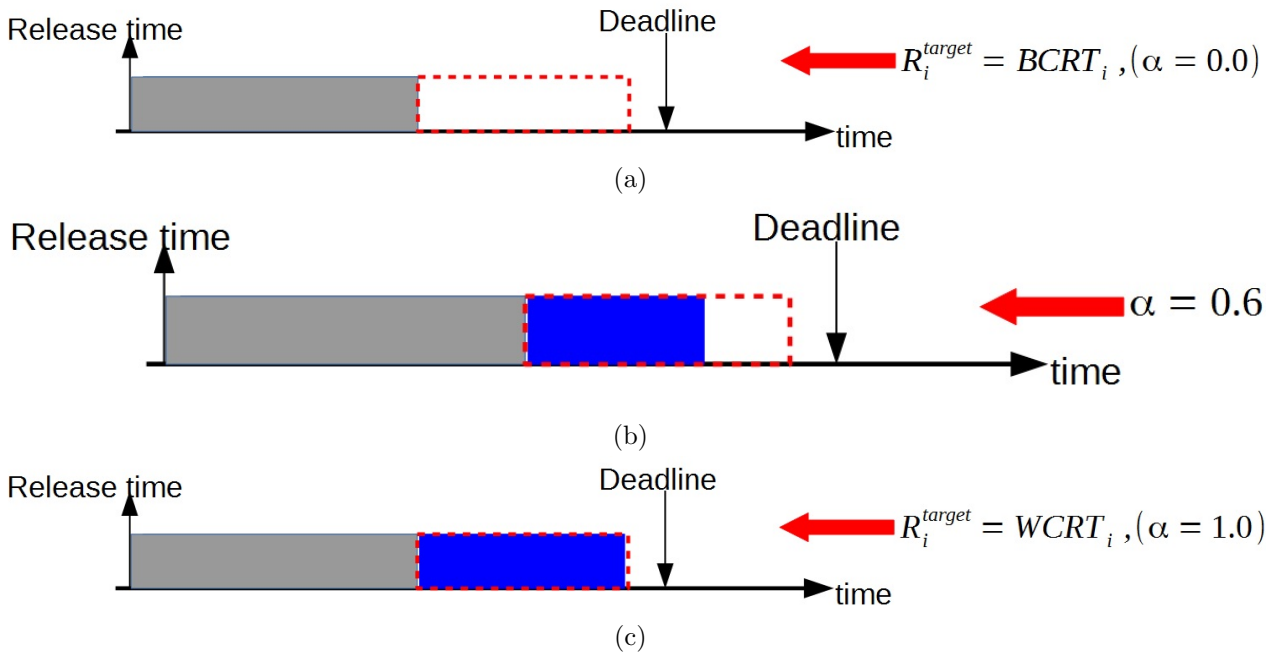


Figure 4.12: An example of user-specified target response time

[4]The specific task which is highly sensitive to jitter, i.e., low jitter tolerance.

Ideal Operating Frequency

In order to get an ideal operating frequency at a frequency-scaling point, the system has to know the available time before R_i^{target} expires and the remaining worst-case execution cycles ($RWCEC_i$) which τ_i^{jitter} is supposed to spend from the current time. The ideal operating frequency is calculated by the following equation.

$$f_{ideal} = \frac{RWCEC_i}{R_i^{target} - time_i^{executed} - I^{actual}(i)} \tag{4.6}$$

In this equation, $\mathbf{R_i^{target} - time_i^{executed} - I^{actual}(i)}$ represents the **available time** for task τ_i at the considered frequency-scaling point. The available time is substantially subject to the length of interference time $I^{actual}(i)$ from higher-priority tasks.

To further distinguish the DVFS opertion between (i) *start-time frequency-scaling point* and (ii) *B-/L-/P-type frequency-scaling point*, this thesis shows each of their algorithms by pseudo-codes below.

- **Start-time frequency-scaling point.** The line 1 of the Algorithm 1 is to calculate the so-called *expected actual response time* based on the assumption that, current instance will spend its worst-case execution cycles ($WCEC_i$) under the default operating frequency ($f_{default}$). In addition, the R_i^{actual} here is aware of actual interference time ($I^{actual}(i)$ depicted in Section 4.2.3). The $time_{elapsed}$ is the duration from *release time* to *start time*, denoted as $r_{i,j} - s_{i,j}$. Next, from line 2 to line 4 is to deal with the interference time variance that, if higher-priority tasks finish earlier and leave some global slack ($Slack_{global}$) then current instance can scale down the operating frequency to avoid the R_i^{min} being updated in the future. The line 5 manages to handle the *discrete bound of operating frequencies range* which will be introduced in Section 4.6.3. Finally, the line 9 is really to configure the new operating frequency to the target processor.

Algorithm 1: Slack Reclamation at Start Time

1 $R_i^{actual} \leftarrow time_{elapsed} + \frac{WCEC_i}{f_{default}} + I^{actual}(i)$;
2 **if** $Slack_{global} > 0$ **then**
3 **if** $R_i^{actual} - Slack_{global} < R_i^{min}$ **then**
4   $f_{ideal} \leftarrow \frac{WCEC_i}{R_i^{min} - I^{actual}(i) + Slack_{global}}$;
5   $f_{new} \leftarrow Discrete\_freq\_handle(f_{ideal})$;
6   $freq\_config(f_{new})$;
7 **else**
8   $f_{new} \leftarrow f_{default}$;
9 **end**
10 **end**

- **B-/L-/P-type frequency-scaling point.** When the task τ_i^{jitter}'s execution reaches one certain frequency-scaling point (no matter B-type, L-type or P-type), first step in line 1 of Algorithm 2 is going to look the $RWCEC_i$ up in each point's corresponding mining table. Then the available execution time ($time_{available}$) from current time to specified R_i^{target} is calculated in line 2. From line 3 to line 5 is to calculate the new operating frequency with discrete bound handling according to the $RWCEC_i$ and $time_{available}$.

Algorithm 2: User-specified DVFS Operation

Input: i, $block_{current}$

1 $RWCEC_i \leftarrow mining\_table(block_{current})$;

2 $time_{available} \leftarrow R_i^{target} - I^{actual}(i) - time_{executed}$;

3 $f_{ideal} \leftarrow RWCEC_i / time_{available}$;

4 **if** $f_{ideal} \neq f_{current}$ **then**

5 $\quad f_{new} \leftarrow Discrete\_freq\_handle(f_{ideal})$;

6 $\quad freq\_config(f_{new})$;

7 **end**

4.6.2 Profile-Based DVFS

Assignment of Response Time Constraint

In the *profile-base DVFS* settings, the target response time (R_i^{target}) of task τ_i^{jitter} will be kept regulating by system during runtime, instead of being specified by user/designer statically.

The system performs one procedure called dynamic assignment of target response time during runtime. It decides a target response time by referring to the *profiling informations* (R_i^{max} and R_i^{min} defined in Section 4.2) as well as estimating the *currently expected response time* given by the following equation.

$$R_i^{expect} = time_i^{executed} + \frac{RWCEC_i}{f_{current}} + I^{actual}(i) \tag{4.7}$$

The R_i^{expect} indicates the actual response time if current operating frequency ($f_{current}$) is kept to run the task τ_i^{jitter} until the end of its execution. The $time_i^{executed}$ is the duration from start time to current time which indicates the total amount of time system spent for executing task τ_i^{jitter}.

In the second step, the obtained R_i^{expect} is compared with R_i^{min} and R_i^{max} at any type of frequency-scaling point. Then target response time R_i^{target} is assigned in the manner of the following equation.

$$R_i^{target} = \begin{cases} R_i^{min} & (R_i^{expect} < R_i^{min}) \\ R_i^{max} & (R_i^{expect} > R_i^{max}) \end{cases} \tag{4.8}$$

However, there is one exception that DVFS operation is not performed. That is, when $\mathbf{R_i^{min}} \leq \mathbf{R_i^{expect}} \leq \mathbf{R_i^{max}}$. In this case, the response time of the current instance will not increase the finish time jitter even if the system keeps the current operating frequency $f_{current}$. Hence R_i^{target} does not need to be considered.

Ideal Operating Frequency

- **Start-time frequency-scaling point.** The strategy of start-time frequency-scaling point in *profile-based DVFS* settings is same as the algorithm addressed in *static-based DVFS*.

- **B-/L-/P-type frequency-scaling point.** As for the procedure of deciding the new operating frequency in the Algorithm 3, from line 2 to line 4 is to check if current operating frequency will make R_i^{expect} end up with new (shorter) R_i^{min}, and calculate a lower operating frequency for f_{ideal}. Similarly, From line 6 to line 9 is to check whether R_i^{expect} will lead to longer R_i^{max} or not, and calculate a higher operating frequency for f_{ideal}. Otherwise, the DVFS operation does not change the current operating frequency ($f_{current}$).

Algorithm 3: Profile-based DVFS Operation

1 $R_i^{expect} \leftarrow time_{executed} + \frac{RWCEC_i}{f_{current}} + I(i)^{actual}$;
2 **if** $R_i^{expect} < R_i^{min}$ **then**
3 $\quad f_{ideal} \leftarrow \frac{RWCEC_i}{R_i^{min} - time_i^{executed} - I(i)^{actual}}$;
4 $\quad f_{new} \leftarrow Discrete\_frequ\_handle(f_{ideal})$;
5 $\quad freq\_config(f_{new})$;
6 **else if** $R_i^{expect} > R_i^{max}$ **then**
7 $\quad f_{ideal} \leftarrow \frac{RWCEC_i}{R_i^{max} - time_i^{executed} - I(i)^{average}}$;
8 $\quad f_{new} \leftarrow Discrete\_freq\_handle(f_{ideal})$;
9 $\quad freq\_config(f_{new})$;
10 **else**
11 $\quad f_{new} \leftarrow f_{current}$;
12 **end**

4.6.3 Discrete Bound Handling

The **ideal operating frequency** assumes that the system can use continuous frequencies from one to infinity. However it is impossible in practical processors which can operate only with a limited number of discrete operating frequencies(from maximal frequency f_{max} to minimal frequency f_{min}). For instance, in the Texas Instruments Sitara AM335x

Processor (ARM Cortex-A8) [16], there are only five operating points (OPPs) which microprocessor unit subsystem (MPU) can perform as shown in Table 4.5.

MPUOPP	*MPU Frequency* (MHz)	*Supply Voltage* (v)	*VDD_MPU Power* (mW)
*OPP*50	300	0.95	114.38
*OPP*100	600	1.1	303.15
*OPP*120	720	1.2	437.49
Tubo	800	1.26	542.73
Nitro	1000	1.325	736.08

Table 4.5: The operating points of Sitara AM335x family processor

Therefore, the obtained ideal operating frequency needs to be converted to one of those practical frequencies in the target processor model. We assume the set of practical frequencies $\mathbf{F^{discrete}} = \{\mathbf{f_1}, \mathbf{f_2}, ..., \mathbf{f_n}\}$ where f_1 and f_n are f_{min} and f_{max}, respectively. The frequency conversion corresponding to the function $Discrete\_freq\_handle(f_{ideal})$ in previous section's algorithm is described as follows.

$$f_{new} = \begin{cases} f_{min} & (f_{ideal} \leq f_0) \\ f_{a+1} & (f_a < f_{ideal} \leq f_{a+1}) \\ f_{max} & (f_{ideal} \geq f_n) \end{cases} \tag{4.9}$$

If f_{ideal} is between f_a and f_{a+1}, f_{a+1} is chosen as the updated frequency f_{new} in order to avoid deadline misses. Because this research targets to the *hard real-time systems*, thus the criticality of deadline miss is higher than low jitter or energy demand. The selection of practical frequency is illustrated in Figure 4.13.

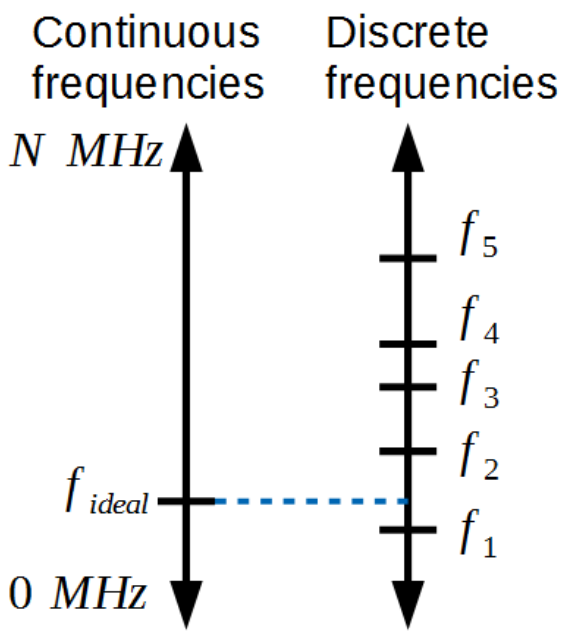


Figure 4.13: The discrete bound of practical processor

Chapter 5

Evaluation

5.1 Experimental Setup

To evaluate the proposed jitter reduction approach, a multitasking simulator, benchmark programs representing periodic tasks, and task sets as well as timing constraint parameters are essential. And the evaluation way is basically based on the comparison of *absolute finish time jitter* and *energy saving* between (i) the environment of Rate-Monotonic scheduling with single operating frequency and (ii) the environment of Rate-Monotonic scheduling with DVFS mechanism.

This section is organised into four subsections: Section 5.1.1 introduces the chosen benchmark programs, Section 5.1.2 will show the algorithm how to generate the test patterns, and Section 5.1.3 will give each chosen periodic task's timing constraints. Section 5.1.4 is about the framework of simulator.

5.1.1 Benchmark Programs

In this experiment, five benchmark programs are prepared for periodic task set. Those contain four programs (written in C) which are provided by WCET project [13], and one simple case study's CFG is created. The description of benchmarks is shown below.

- **bs.c**: Binary search for the array of fifteen integer elements, including one loop with four loop-inner execution paths.
- **compress.c**: Data compression program which contains three loops.
- **matmul.c**: Matrix multiplication program containing four nest loop structures[1] and one branch.
- **ludcmp.c**: LU decomposition algorithm containing six nest loop structures and four branches.
- **cfg_1**: Simple CFG containing one loop with two loop-inner execution paths and one branch.

The former four programs were input into the off-line timing analysis introduced in the proposed system framework of Section 4.1, e.g., (i) control and data flow analysis, (ii) execution cycle estimation and (iii) frequency-scaling point placement. In addition, the case study CFG, cfg_1 was input from the second phase, execution cycle estimation. The results of those off-line timing analysis are shown in Appendix B.

[1]The program structure that one loop is inside the other loop.

Finally, the informations of these benchmarks which will be used in simulation, are summarised as task models in Table 5.1. The second column shows the number of basic blocks of each benchmark CFG, the third column indicates the number of frequency-scaling points placed inside each CFG. The fourth and fifth columns present the worst-case execution cycles (WCECs) and best-case execution cycles (BCECs) of each CFG.

Task	# Basic Block	# Scaling Point	WCEC (cycle)	BCEC (cycle)
bs	10	2	9750	1000
compress	11	4	11 950	52
cfg_1	9	3	1810	260
matmul	23	4	1 890 395	95
ludcmp	46	9	27 546	338

Table 5.1: The information of chosen benchmarks

5.1.2 Task-Set and Test Pattern Generation Algorithm

After obtaining the CFGs of target benchmarks, next steps are to (ii) make the feasible task sets for simulating the task scheduling, and (ii) generate the various execution paths for each CFG to reflect runtime variation in execution of the target tasks among instances.

Generating task sets

The worst-case execution time (WCET) and best-case execution time (BCET) of every task are essential for generating the task set. In this experiment, the occurrence of WCET (ns) is considered that task takes its WCEC (cycles) under the default operating frequency, i.e, $WCET = \frac{WCEC}{f_{default}}$. On the other hand, the BCET (ns) occurs in the case that the task takes its BCEC (cycles) under the default operating frequency, i.e., $BCET = \frac{BCEC}{f_{default}}$. The frequency settings is referred to the Texas Instruments Sitrara AM335x processor shown in Table 4.5 in which the running clock frequency is set to 300, 600, 720, 800, or 1000 MHz. The default operating frequency is always set to its maximal frequency, $f_{default} = f_{max} = 1000MHz$.

The content of one task set here contains the five target tasks' features in terms of timing domain. Every task's feature includes priority[2], computation time, period, relative deadline, $\tau_i(Prt_i, C_i, T_i, D_i),\ i = 1, ...,\ 5$. The generation procedures are described below.

1. **Calculation of utilisation bound.** According to the Rate-Monotonic scheduling policy, the utilisation bound is $N \times (2^{1/N} - 1)$ where N is the number of tasks. Therefore, the utilisation bound of scheduling five tasks is 0.74349 (74.349%)[3].

2. **Random assignment of utilisation of every task.** Next step is to randomly assign an utilisation to every task τ_i in the range of [0.001,0.4] by exponential distribution, denoted as $Util_i$. In addition, the accumulation of those five tasks' assigned utilisation values ($\sum_{n=1}^{5} Util_i$) is limited by the bound 0.74349. If $\sum_{n=1}^{5} Util_i$ exceeds the bound, this step will randomly select one task and keep decreasing its utilisation value until $\sum_{n=1}^{5} Util_i \leq 0.7439$.

[2]The priority is based on descend order, i.e., the smaller the value, the higher the priority.
[3]In the uniprocessor system, the full utilisation is up to 1.0 (100%)

3. **Calculation of period of each task.** The utilisation of each task τ_i can be simply considered as the following equation, $Util = \frac{C_i}{T_i}$. Therefore, the period of every task can be set to $T_i = \frac{WCET_i}{Util_i}$. Moreover, the relative deadline of every task is set to be equal to its period.

In this experiment, there are two task sets which were generated as shown in Table 5.2.

$Task$	WCET (ns)	Period (ns)	Deadline (ns)	Priority
bs	9750	75582	75582	0
compress	11950	173189	173189	3
cfg_1	1810	164546	164546	2
matmul	1890395	9110699	9110699	4
ludcmp	27546	84239	84239	1

(a) The task-set 1 whose total utilisation is **0.74349**

$Task$	WCET (ns)	Period (ns)	Deadline (ns)	Priority
bs	9750	162500	162500	3
compress	11950	35949	35949	0
cfg_1	1810	35951	35951	1
matmul	1890395	37807900	37807900	4
ludcmp	27546	121349	121349	2

(b) The task-set 2 whose total utilisation is **0.71976**

Table 5.2: The generated task sets

In addition, in order to perform the static-based DVFS settings, the response-time analysis (described in Section 3.4) ought to be done. Table 5.3 shows the $WCRT_i$, $BCRT_i$ and $R_i^{variance}$ of every task τ_i. The analysis result of task-set 1 is shown in Table 5.3a, and the result of task-set 2 is shown in Table 5.3b

$Task$	WCRT	BCRT	$R_i^{variance}$
bs	9750	1000	8750
compress	11950	1650	10300
cfg_1	39106	1598	37508
matmul	4108449	1745	4106704
ludcmp	37296	1338	35958

(a) Task-set 1

$Task$	WCRT	BCRT	$R_i^{variance}$
bs	64816	1650	63166
compress	11950	52	11898
cfg_1	13760	312	13448
matmul	5778963	1745	5777218
ludcmp	55066	650	54416

(b) Task-set 2

Table 5.3: The result of response-time analysis

Generating various execution paths

In Section 3.4, the concept of *variance response/jitter margin* has been addressed. The possible *variance response* of a task τ_i is in the range of $[0, WCRT_i - R_{constant}]$. In order to make simulation reflect such bounded variance during every instance of the running task a **text pattern generator** is built. The generator randomly generates fifty execution paths for every target task that is based on their CFG structure.

The strategy of generating those execution path patterns is described in Algorithm 4. Initially, through the **Input** port, number of execution path patterns which is required for generating, e.g., 50 is given in this experiment. And the CFG of target task and static loop bound of every loop (introduced in Section 4.3.2) are input. On the other hand, the output port is to return a set of execution path patterns, each of which is constructed as a linking list. Hence fifty linking lists will be obtained from output port.

From line 2 to line 4, generator starts to traverse the CFG from its first basic block. Once the generator passes through a basic block, it will put it into current test pattern's linking list; moreover, line 4 randomly determines the runtime loop bound ($Bound_{runtime}$) of each loop region in the range of [0,$Bound_{static}$][4]. From line 5 to line 23 are to make different combination of basic blocks by following three manners:

- **line 6 - line 7:** When generator reaches a branch's corresponding basic block, it will randomly take one of successor execution paths and continue traversing.
- **line 8 - line 17:** These lines deal with the pattern of loop-inner execution path. The given $Bound_{runtime}$ by *line 4* is set as a counter, and it will keep generating $Bound_{runtime}$ number of execution paths from its loop entry to loop exit. Each of those execution path is to simulate the inner-execution path within every loop iteration. When generator reaches a loop exit's corresponding basic block, if the counter is subtracted to 0 then the generator will leave the loop region.
- **line 19 - line 20:** When generator reaches a basic block which is neither a loop entry/exit nor branch, it will traverse to its only successor basic block.

The illustration of generating the execution path patterns for benchmark **compress**'s CFG is presented in Figure 5.1.

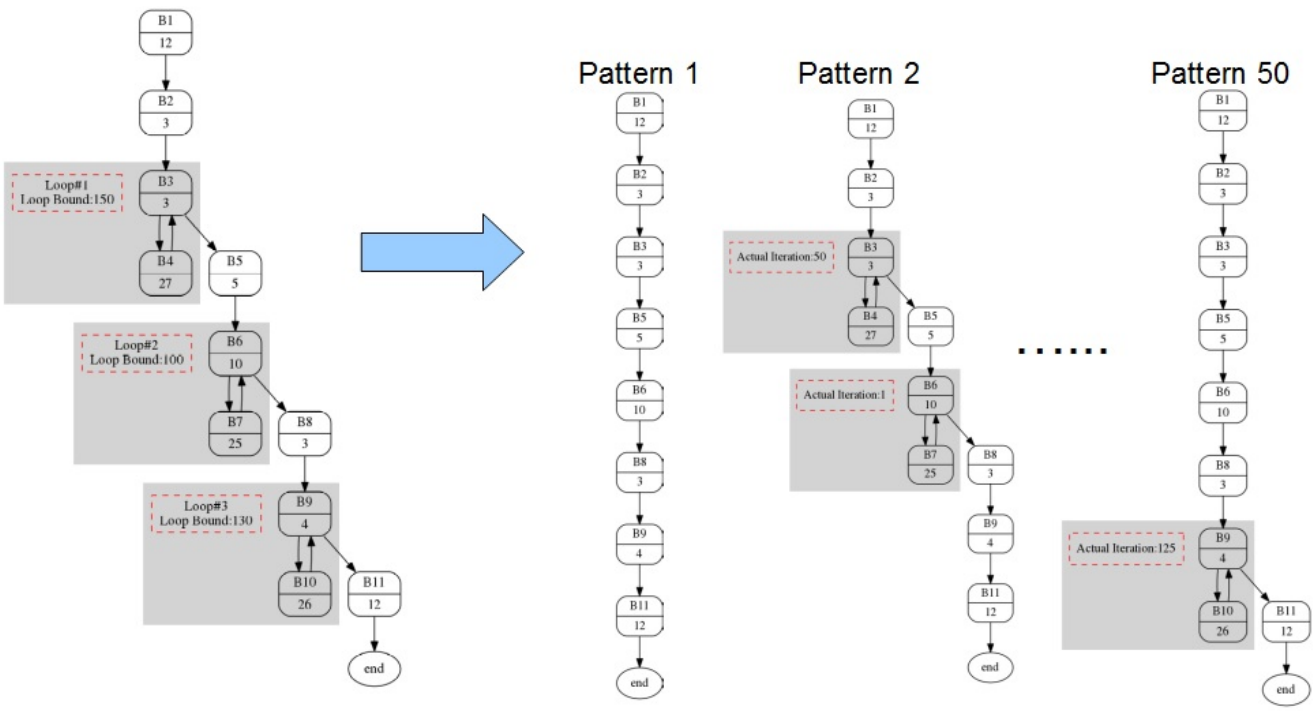


Figure 5.1: Generating fifty execution path patterns for **compress**'s CFG

[4]The number of loops inside CFG can be known by counting the number of loop entries

Algorithm 4: Randomly Generating Execution Path(s)

```
Input: number of required patterns, target Control Flow Graph,
       Bound_static of every loop
Output: A set of test patterns
1  while pattern_num ≠ 0 do
2      cur_NodeID ← CFG_ path.startBlock;
3      insert Block(cur_ NodeID) into current test pattern;
4      Bound_runtime[i] ← random number from [0,Bound_static[i]],
        ∀i ∈ CFG_paths' LoopIndex;
5      repeat
6          if Block(cur_ NodeID) is a branch then
7              succ_ NodeID ← random number ϵ Block(cur_
                NodeID).successors ;
8          else if Block(cur_ NodeID) is a loop exit then
9              cur_ NodeID ← Block(Loop[loop_ ID]'s entry);
10             loop_ID ← search_ LoopEntryID(cur_ NodeID);
11             insert Block(cur_ NodeID) into current test pattern;
12             if Bound_runtime[LoopID] = 1 then
13                 succ_ NodeID ← random number ϵ
                    Block(cur_NodeID).successors ∧ outside loop;
14             else
15                 Bound_runtime[LoopID] ← Bound_runtime[LoopID] - 1;
16                 succ_ NodeID ← random number ϵ
                    Block(cur_NodeID).successors ∧ inside loop;
17             end
18         else
19             succ_ NodeID ← Block(cur_ NodeID).succssor[0];
20         end
21         cur_ NodeID ← succ_ NodeID;
22         insert Block(cur_ NodeID) into current test pattern;
23     until cur_ NodeID = CFG_ path.sinkBlock;
24     insert current test pattern into test pattern set;
25     pattern_ num ← pattern_ num - 1;
26 end
```

5.1.3 Jitter Constraint Settings

For each task set, it randomly classifies certain tasks into the *jitter-sensitivity task* τ_i^{jitter} aforementioned in Section 4.6. Every task set is configured to have five different combination of jitter-sensitivity tasks as shown in Table 5.4. For instance, the set of jitter-sensitivity tasks in the first row of Table 5.7, $(bs, compress)$, indicates that this experiment setting only focuses on reducing the finish time jitter of task **bs** and task **compress**. More specifically, when system starts running, the DVFS operation will only be invoked during the executions of task **bs** and task **compress**; meanwhile, the rest of tasks will be executed under the default operating frequency without frequency scaling.

Set	Targeted Jitter-sensitive Tasks
1	(bs, compress)
2	(bs,compress,cfg_1)
3	(bs,compress,cfg_1,ludcmp)
4	(bs,cfg_1,ludcmp)
5	(compress,cfg_1)

(a) Task-set 1

Set	Targeted Jitter-sensitive Tasks
1	(compress,cfg_1)
2	(bs,compress,cfg_1)
3	(compress,cfg_1)
4	(bs,cfg_1)
5	(bs,compress,cfg_1,ludcmp)

(b) Task-set 2

Table 5.4: The sets of jitter-sensitivity tasks

5.1.4 Implementation of Simulator

In this subsection, the simulation environment is introduced. A CFG-based multitasking simulator is built for evaluating jitter reduction and energy saving by the proposed approach in C++11 and Bash script. The simulator mainly is composed of (i) *CFG traversing model* and (ii) *task scheduling model*. The former is built for simulating a processor's behaviour, and the latter is to simulate the real-time kernel's task scheduler.

The platform used for running the simulator is Intel Core i5-5200U 2.20GHz with 4GB main memory, in which in on the top of openSUSE Leap 42.2 Linux kernel 4.4.27.

CFG traversing model

Basically, this model is based on the *Intra-task perspective*, and simulate every single task's execution paths individually. First, before running the CFG traversing model, there are several configuration files which must be input. And there are dedicated parsers for reading each kind of configuration file. The files are listed below.

After CFGs of target tasks are input with aforementioned configuration files, the CFG traversing model is started running. It is performed on a tasks' CFGs basis, where execution cycles of traversed basic blocks are counted.

- **Input**
  - **DVFS settings.** The voltage-frequency sets and corresponding power consumption constructed by the C++ head file, shown in Figure 5.2.
  - **Frequency-scaling point list.** To tell the simulator in which the DVFS operation should be invoked. The informations are arranged by a file called *scaling_list.txt* in Figure C.1 of Appendix C. In addition, the content of this file is used for building the mining tables.

```
#ifdef  PROCESSOR_AM335x
    #define FREQ_CNT      5    // Five discrete set(MPU OPP)
    #define MAX_speed     1000    // 1000 MHz
    #define MIN_speed     300 // 300 MHz
    static int OverheadCycle_B = 50; // 50 execution cycle of performing B-type checkpoint
    static int OverheadCycle_L = 60; // 60 execution cycle of performing L-type checkpoint
    static int OverheadCycle_P = 60; // 60 execution cycle of performing P-type checkpoint
    static double  OverheadTime = 1; // 1 ns of delay during frequency/voltage scaling
    static double  OverheadEnergy = 50.0; // 50.0 uJ of additional energy consumption during
                                          // frequency scaling
    enum { // The id of MPU OPP
        OPP50  = 0,
        OPP100 = 1,
        OPP120 = 2,
        TURBO  = 3,
        NITRO  = 4
    };
    static double freq_vol[][2] = {
        {300 , 950},  // 300 MHz ,  950 mV
        {600 , 1100}, // 600 MHz , 1100 mV
        {720 , 1200}, // 720 MHz , 1200 mV
        {800 , 1260}, // 800 MHz , 1260 mV
        {1000, 1325} // 1000 MHz, 1325 mV
    };
    // Power proportional to freq and Vol^2
    static double MPU_POWER[] = {
        270.75,//114.38, // 114.38 mW
        726.0,//303.15, // 303.15 mW
        1036.8,//437.49, // 437.49 mW
        1270.1,//542.73, // 542.73 mW
        1755.6//736.08  // 736.08 mW
    };
#endif
```

Figure 5.2: The header file as input element to configure the DVFS settings

- **Execution path patterns.** There are two configuration files related to the execution path pattern. One is the list of fifty patterns. The example of task **bs**'s corresponding file is shown in Figure 5.3a. Another one is the list of $Bound_{runtime}$ for each loop which is generated by Algorithm 4's line 4. The example of task **matmul**'s corresponding file is shown in Figure 5.3b.

Task scheduling model

This model is based on the *Inter-task perspective*. It is responsible for scheduling the target task set based on Rate-Monotonic scheduling, and giving the CFG traversing model the instruction to run the prioritising task (the task in the *Run* state) at any time. Simply speaking, the task scheduler keeps managing every task in the three states: *Ready*, *Run*, and *Idle*. The relation of those three states is explained below and an illustration is shown in Figure 5.4.

- **Idle.** The task is in the *Idle* state if it has not been released yet.
- **Ready.** Once the task is released but it is not prioritised, in this case, task will transits from *Idle* to *Ready*. That is, the task in the state *Ready* means it is waiting for the processor to execute it in the *ready queue*.
- **Run.** Once the task is assigned the highest priority, it transits from *Ready* to *Run*. Then the dispatcher will assign processor to execute the task's corresponding program.

```
PATTERN_SET_SIZE: 50
CASE: 0
SIZE: 9
1
2
3
4
6
3
7
8
9
CASE: 1
SIZE: 21
1
2
3
4
5
6
3
4
5
6
3
4
5
6
3
4
6
3
7
8
9
```

(a) List file of exeuction path patterns

```
PATTERN_SET_SIZE: 50
CASE: 0
SIZE: 6
19
17
15
17
14
13
CASE: 1
SIZE: 6
18
18
3
3
5
5
CASE: 2
SIZE: 6
4
9
3
5
7
16
```

(b) List file of $R_i^{runtime}$ of all loops

Figure 5.3: The configuration files of target task

As for the simulation of preemption, the mechanism designed in this simulator is based on an Interrupt timer and a preemption stack to manage. There is an *interconnection bus* which connects the task scheduling model, CFG traversing model and time management for communication. First the time management includes one system tick for counting the **elapsed time from starting point of simulation (the time unit is nanosecond)**; meanwhile the time management also includes an interrupt timer. Such interrupt timer is used for periodically pausing the work of both task scheduling model and CFG scheduling model, to see if it is necessary to perform preemption[5]. If preemption occurs, the execution context of the CFG which is run by the CFG traversing model will be extracted and temporarily saved in the *preemption stack*, for instance, the task identifier of the current CFG in the model, the basic block where the model currently traverses to, the execution path pattens which current model is following and the current operating frequency. The preemption stack is constructed by a data structure as shown in Figure 5.5.

Finally, the framework of whole simulator is depicted in Figure 5.6

[5]If there is any task just released with the highest priority comparing to the priorities of all tasks in the ready queue

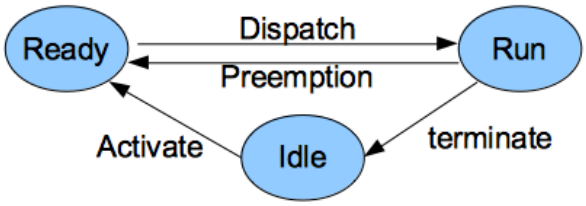


Figure 5.4: The states of every task during runtime

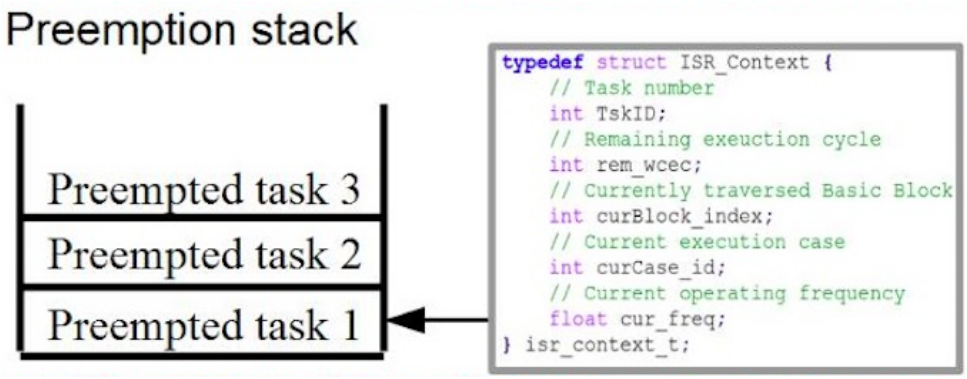


Figure 5.5: The structure of preemption stack

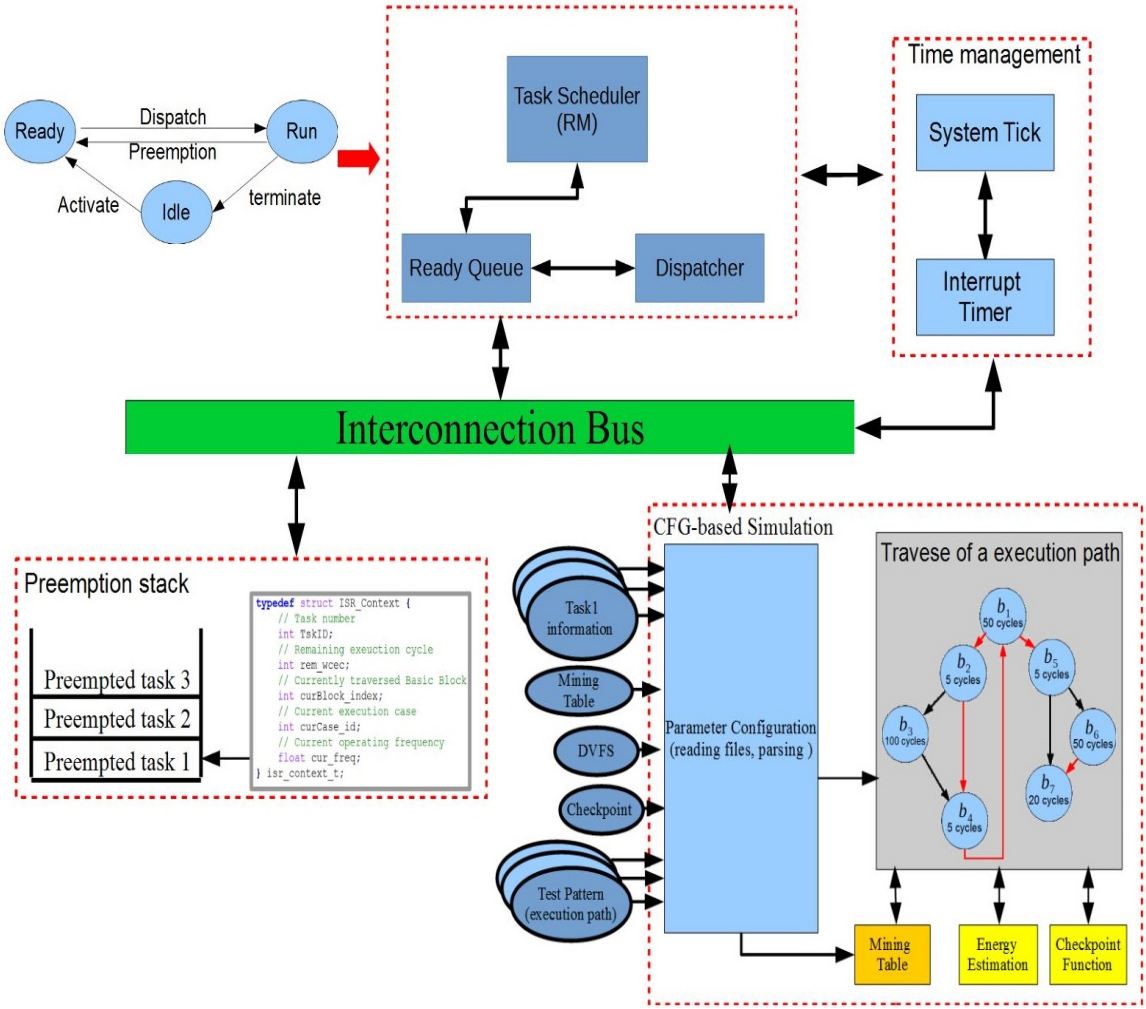


Figure 5.6: The whole system framework

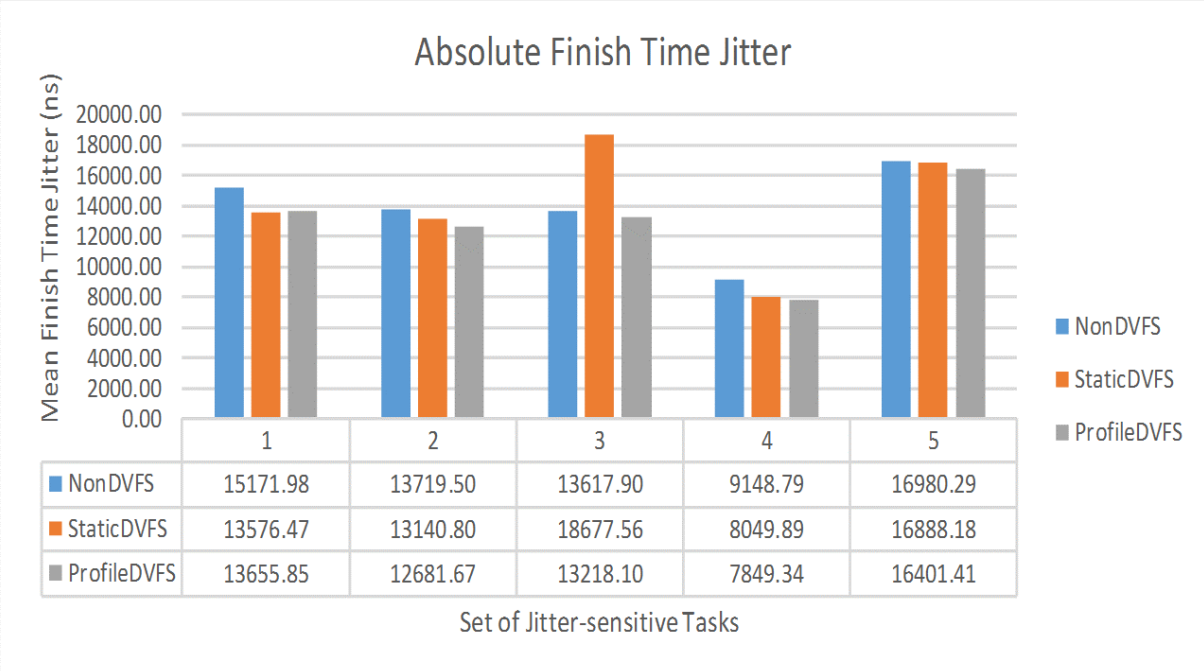
5.2 Experimental Results

In the experiments, the following three settings are compared:

- **NonDVFS.** A system without DVFS operation (with fixed operating frequency of f_{max}).
- **StaticDVFS.** A system with DVFS using the user-specified target response times for jitter-sensitive tasks referred to as the Static-based DVFS context in Section 4.6.1.
- **ProfileDVFS.** A system with DVFS using the profile-based target response times for jitter-sensitive tasks referred to as the Profile-based DVFS context in Section 4.6.2.

Each target task set is simulated five times with different execution paths generated by the test pattern generator, and every task scheduling of each task set is run until the the system tick counts of 500000 nanosecond. The average value of (i) absolute finish time jitter and (ii) energy consumption of the jitter-sensitive tasks are used in the comparison.

Figure 5.7 shows the mean value of absolute finish time jitter for each set of jitter-sensitivity tasks, in task-set 1; moreover, Figure 5.8 shows each of their corresponding jitter reduction rates.



	1	2	3	4	5
NonDVFS	15171.98	13719.50	13617.90	9148.79	16980.29
StaticDVFS	13576.47	13140.80	18677.56	8049.89	16888.18
ProfileDVFS	13655.85	12681.67	13218.10	7849.34	16401.41

Figure 5.7: Absolute finish time jitter of task-set 1

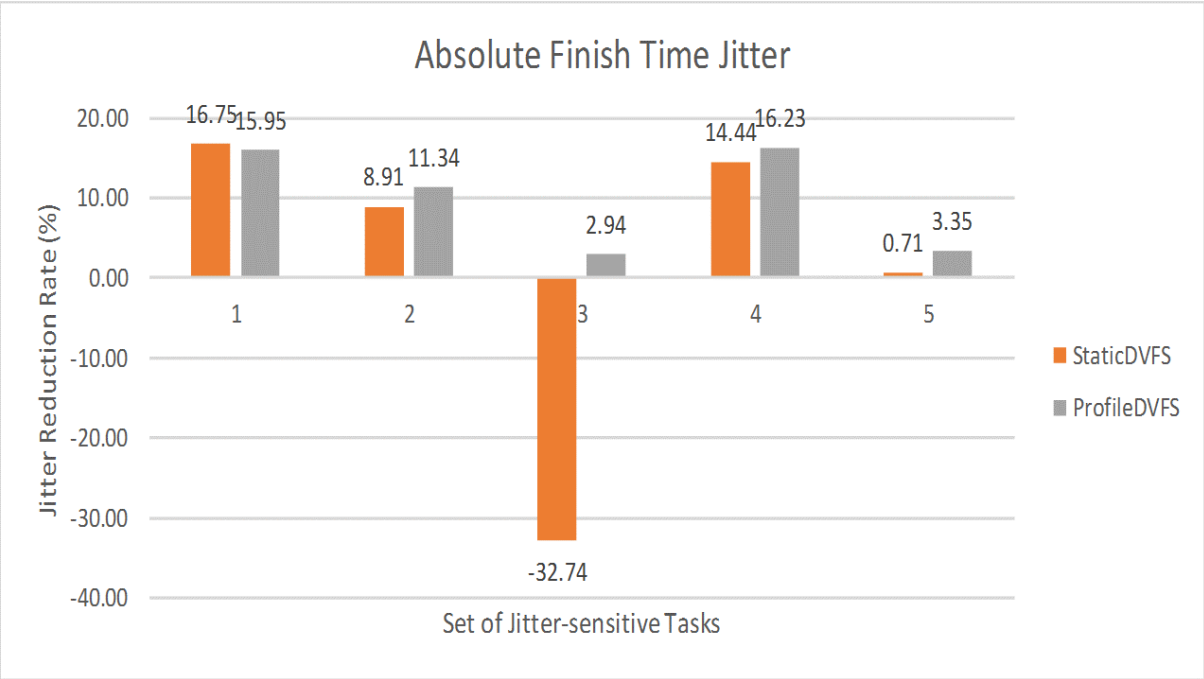
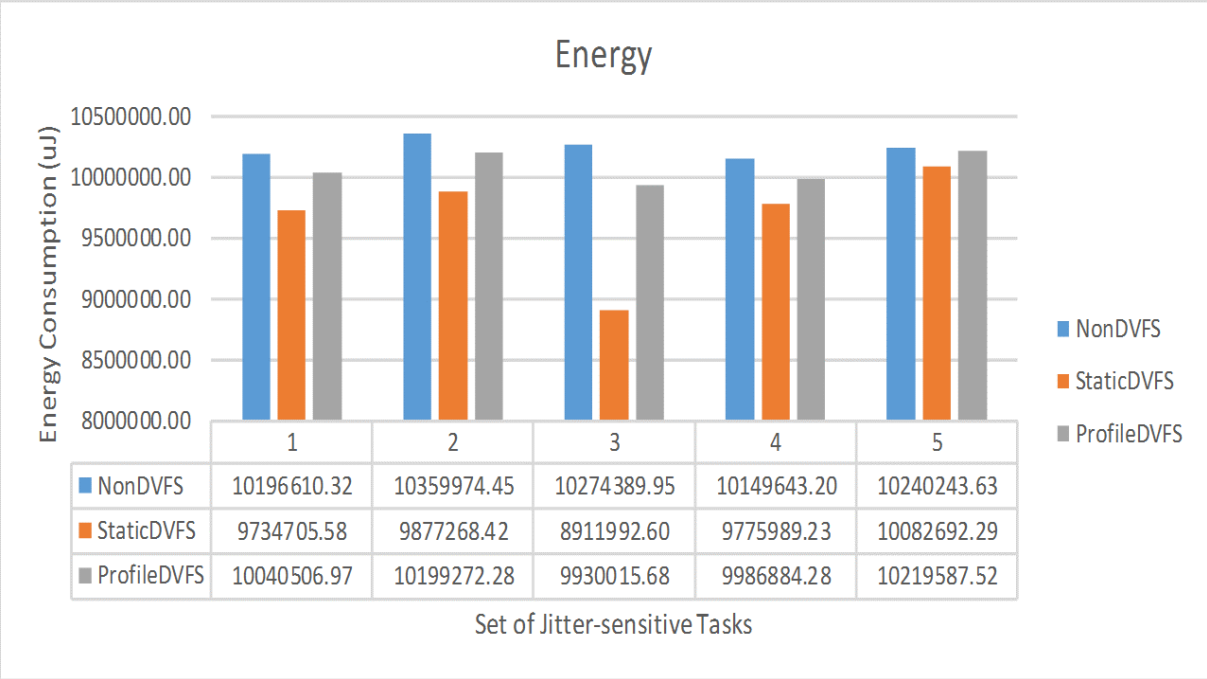


Figure 5.8: Jitter reduction rate of task-set 1

Figure 5.9 shows the energy consumption for each set of jitter-sensitivity tasks, in task-set 1; moreover, Figure 5.10 shows each of their corresponding energy-saving rates.



	1	2	3	4	5
NonDVFS	10196610.32	10359974.45	10274389.95	10149643.20	10240243.63
StaticDVFS	9734705.58	9877268.42	8911992.60	9775989.23	10082692.29
ProfileDVFS	10040506.97	10199272.28	9930015.68	9986884.28	10219587.52

Figure 5.9: Energy consumption of task-set 1

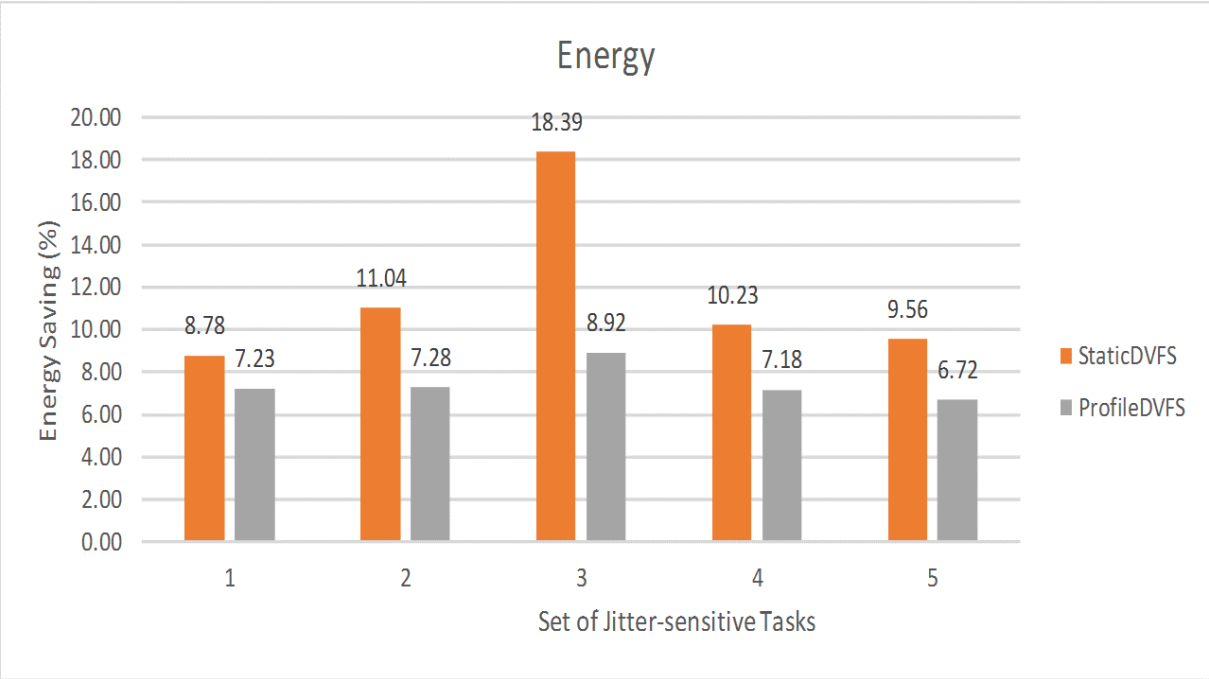
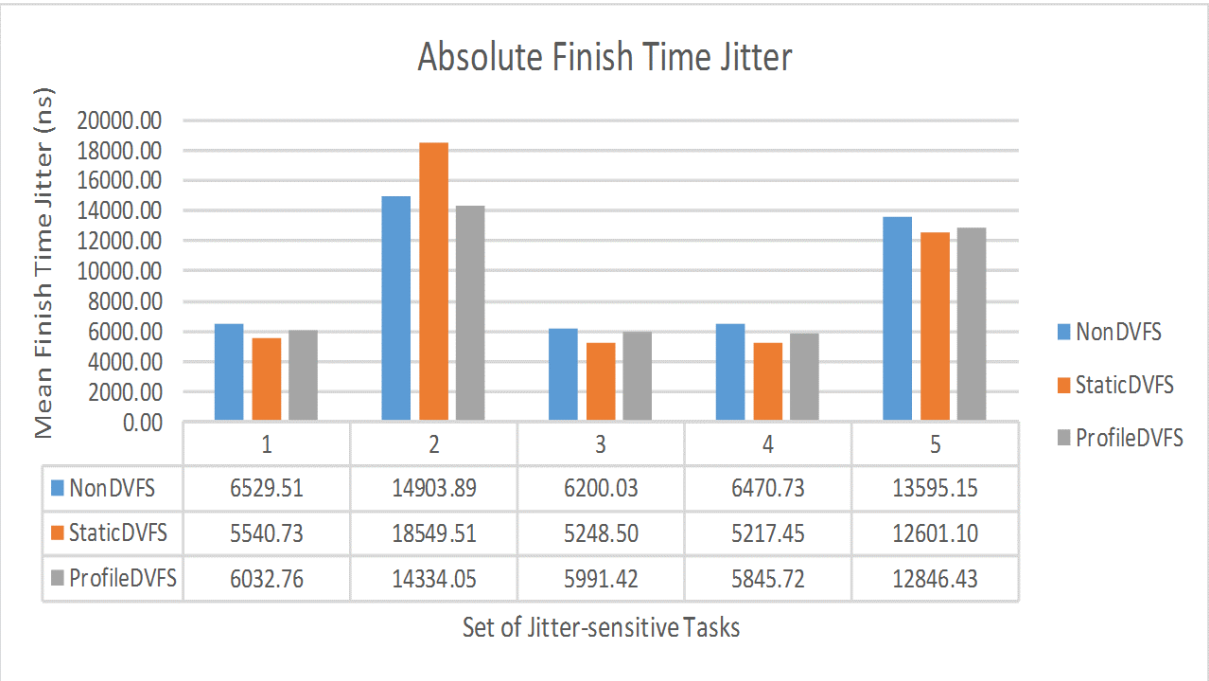


Figure 5.10: Energy-saving rate of task-set 1

Figure 5.11 shows the mean value of absolute finish time jitter for each set of jitter-sensitivity tasks, in task-set 2; moreover, Figure 5.12 shows each of their corresponding jitter reduction rates.



	1	2	3	4	5
NonDVFS	6529.51	14903.89	6200.03	6470.73	13595.15
StaticDVFS	5540.73	18549.51	5248.50	5217.45	12601.10
ProfileDVFS	6032.76	14334.05	5991.42	5845.72	12846.43

Figure 5.11: Absolute finish time jitter of task-set 2

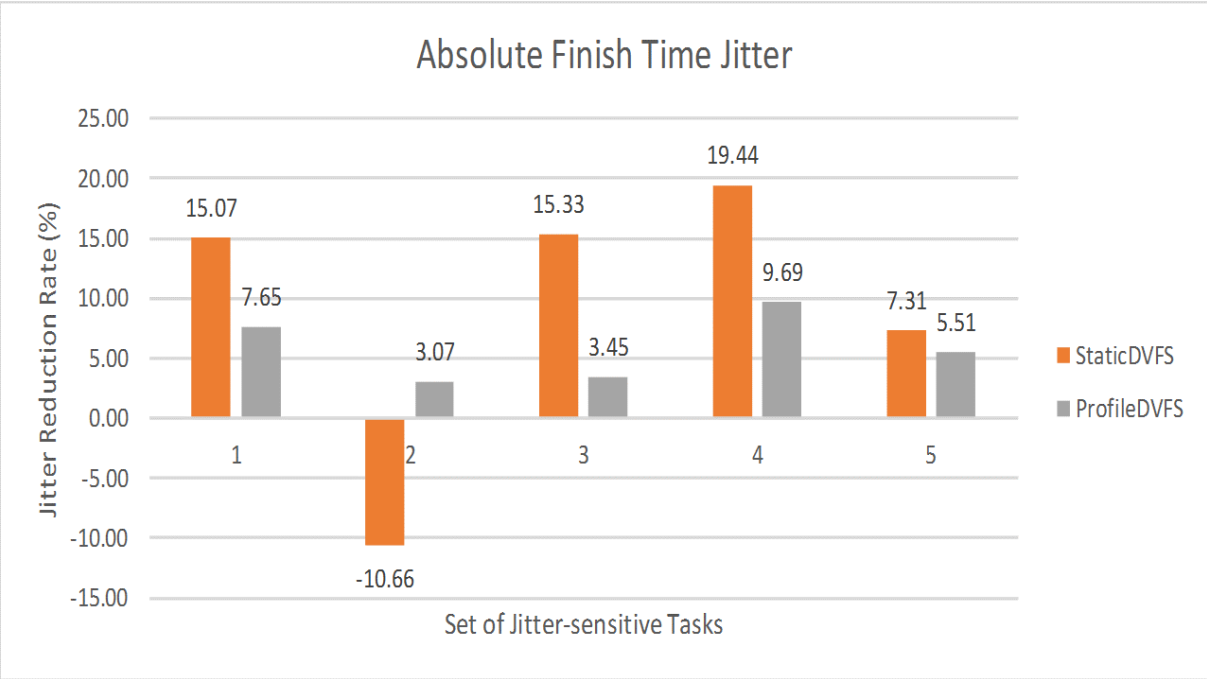
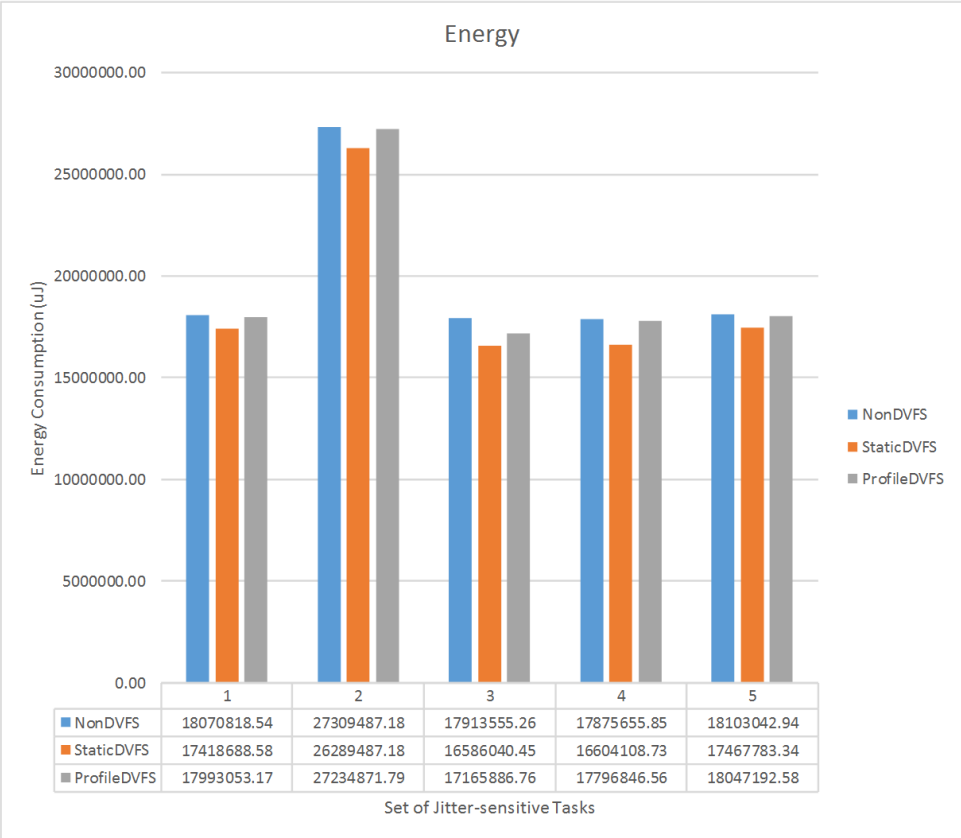


Figure 5.12: Jitter reduction rate of task-set 2

Figure 5.13 shows the energy consumption for each set of jitter-sensitivity tasks, in task-set 2; moreover, Figure 5.14 shows each of their corresponding energy-saving rates.



	1	2	3	4	5
NonDVFS	18070818.54	27309487.18	17913555.26	17875655.85	18103042.94
StaticDVFS	17418688.58	26289487.18	16586040.45	16604108.73	17467783.34
ProfileDVFS	17993053.17	27234871.79	17165886.76	17796846.56	18047192.58

Figure 5.13: Energy consumption of task-set 2

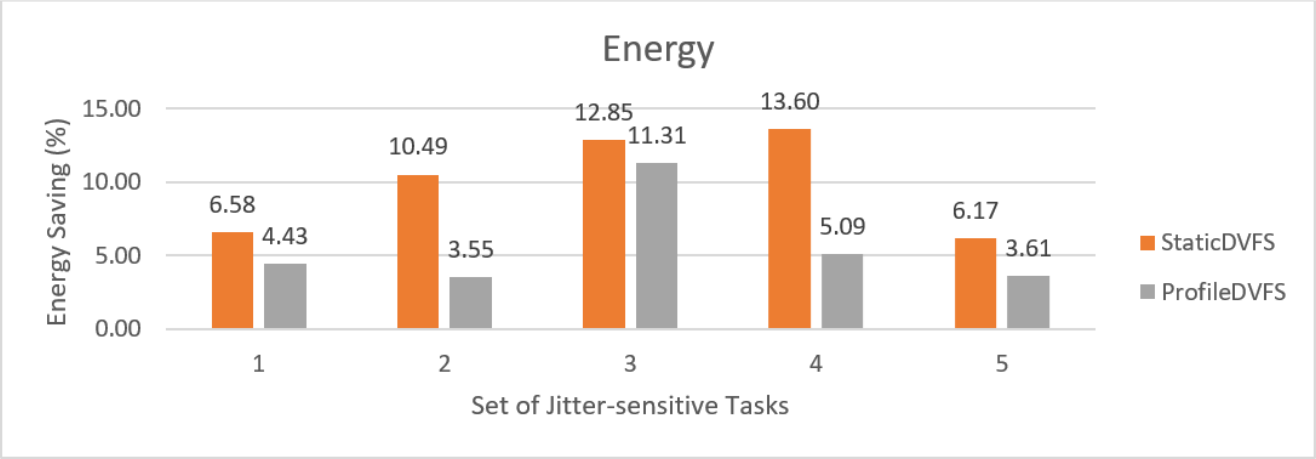


Figure 5.14: Energy-saving rate of task-set 2

Chapter 6

Conclusion

6.1 Summary

This research proposed a jitter-aware Intra-task DVFS techniques for reducing finish time jitter in hard real-time systems. It exploited DVFS technique to reduce runtime variation in both interference and execution time, with the cooperation of control and data flow analysis. To decide effective frequency-scaling factor at every DVFS operation, a jitter margin was defined to clarify the lower and upper bounds of possible finish time jitter, also four control parameters were prepared for profiling runtime situation manipulated by system. Through the simulation, it was shown that jitter can be reduced by 16.2% to 19.4%; meanwhile, the system still could get some opportunity to scale down the operating frequency. Thereby it was shown that energy was saving by 13.6% to 18.39% as the side effect.

However, the proposed approach may have some drawback which leads to larger finish time jitter . For instance, the experimental result of task-set 1 on the setting of third set of jitter-sensitivity shown in the third bar of Figure 5.8, ended up with 32.74% larger finish time jitter comparing with the result under NonDVFS settings. Similarly, the experimental result of task-set 2 on the setting of second set of jitter-sensitivity shown in the second bar of Figure 5.12, ended up with 10.66% larger finish time jitter comparing with the result under NonDVFS settings. That is because of the jitter-sensitivity task τ_i^{jitter}'s assigned priority. When the higher-priority task and lower-priority task are defined as τ_i^{jitter} at the same time, the response time control by DVFS operation for higher-priority task would inherently give the lower-priority task additional interference time variance. Therefore, even if the lower-priority task performs DVFS operation, the effectiveness of its jitter reduction is still violated.

Besides, because the practical processor only can work under limited number of operating frequencies, the *discrete bound handling* in Chapter 4.6.3 would make the actual response time of τ_i^{jitter} present a big gap to the given target response time. Thus, such gap caused by discrete frequency bound is one of the significant factors which will degrade the effectiveness of proposed approach of jitter reduction.

6.2 Future Work

According to the drawback of proposed approach mentioned in the previous section, a further strategy for limiting the DVFS operation on higher-priority task is necessary. In addition, an enhancement algorithm of the *discrete bound handling* in Chapter 4.6.3 should be taken into account.

On the other hand, currently the ongoing work is trying to find a tradeoff between jitter and energy. Different power profiles are being mapped to the frequency settings used in this research. Moreover, thorough assessment under various jitter and energy constraints is to be considered as the future extension. Currently overlooked switching overhead could possibly limit the number of frequency-scaling points.

Publications

1. Boyu Tseng, Kiyofumi Tanaka. Reducing Jitter and Energy in Hard Real-time Systems Using Intra-task DVFS Technique. In IPSJ80, volume 2018, pages 113-114, mar 2018.

2. Bo-Yu Tseng, Kiyofumi Tanaka. Jitter Reduction in Hard Real-Time Systems using Intra-task DVFS Techniques. In Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Application, in conjunction with 30th Euromicro Conference on Real-Time Systems (ECRTS), Barcelona, Spain, 2018.

Appendix A

The Required Execution Cycles for each instruction

According to the open source tool developed by [25], it arranged about the information of every ARMv7 instruction's execution cycles which processor need to spend. Note that, in their specification, the architectural-level accuracy is not taken into account, e.g., regardless of (i) cache miss/hit and (ii) dependency within any consecutive instructions (pipeline hazards).

Instruction	*Cycles*	*Instruction*	*Cycles*	*Instruction*	*Cycles*	*Instruction*	*Cycles*
ADC	3	MCR	3	LDM	6	SMULL	4
ADFD	3	MRC	4	LDR	5	STC	4
ADCS	3	MUFD	3	LDFD	5	STM	3
ADD	3	MNFD	3	LDRB	5	SUFD	3
DVFD	3	MLA	3	LDRH	5	STFD	3
ADDS	3	MLAL	4	LDRSB	5	STR	3
AND	3	MOV	3	LDRSH	5	STRB	3
ANDS	3	MOVS	3	STRH	3	NOP	3
B	3	MRS	3	SUB	3	POP	3
BL	3	MSR	3	SUBS	3	PUSH	3
BLS	3	MUL	3	SWI	3	ROR	3
BIC	4	MULL	3	SWP	4	RRX	3
BICS	4	MVN	3	SWPB	4	stmfd	4
BNE	4	MVNS	3	TEQ	3	stmia	4
BX	3	ORR	3	TST	3	ldmia	6
CDP	3	ORRS	3	UMLAL	4	bge	3
LDC	4	RSB	3	UMULL	4	BGT	3
CMN	3	RSC	3	ADR	3	BMI	3
CMF	3	RSBS	3	ADRL	3	BEQ	3
CMFE	3	RSCS	3	ASR	3	BLT	3
CMP	3	SBC	3	LSL	3	LDMFD	4
EOR	3	SBCS	3	LSR	3	ldmea	6
EORS	3	SMLAL	4	ble	3		

Table A.1: The list of required execution cycles of each instruction

Appendix B

CFGs of Benchmarks

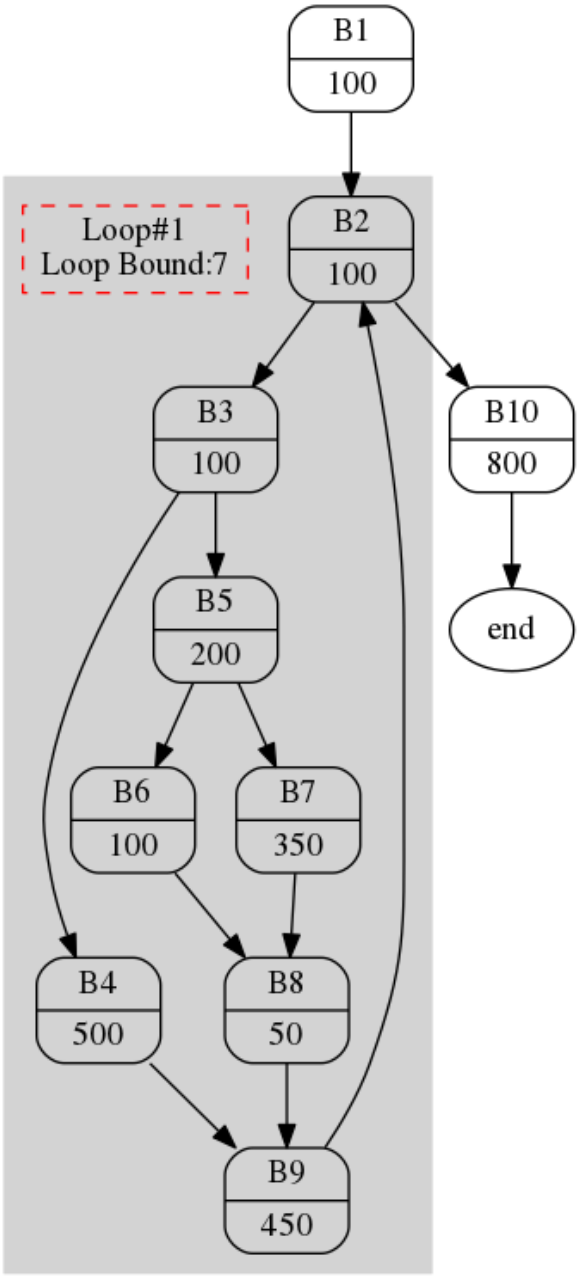


Figure B.1: The CFG of bs.c

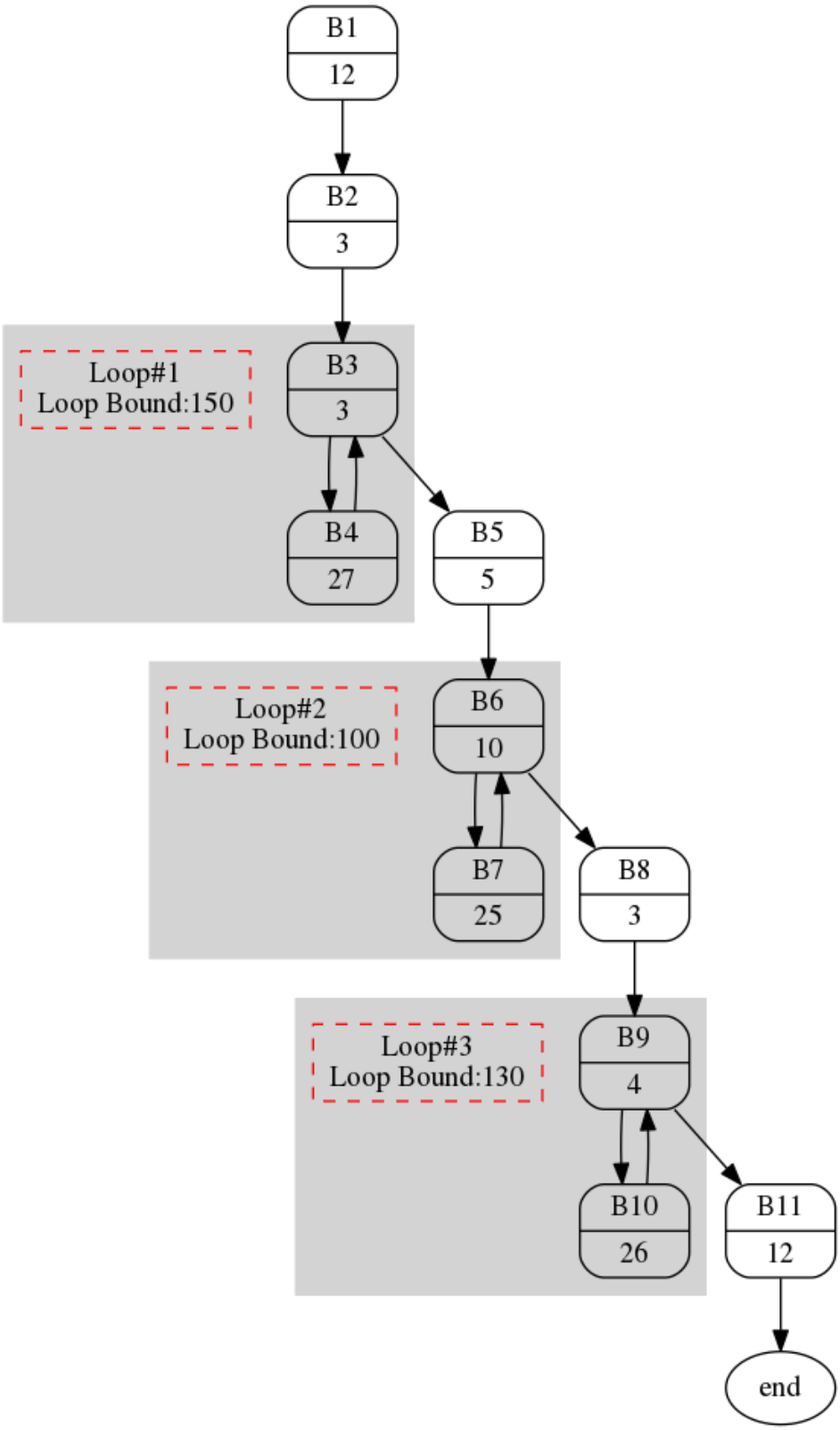


Figure B.2: The CFG of compress.c

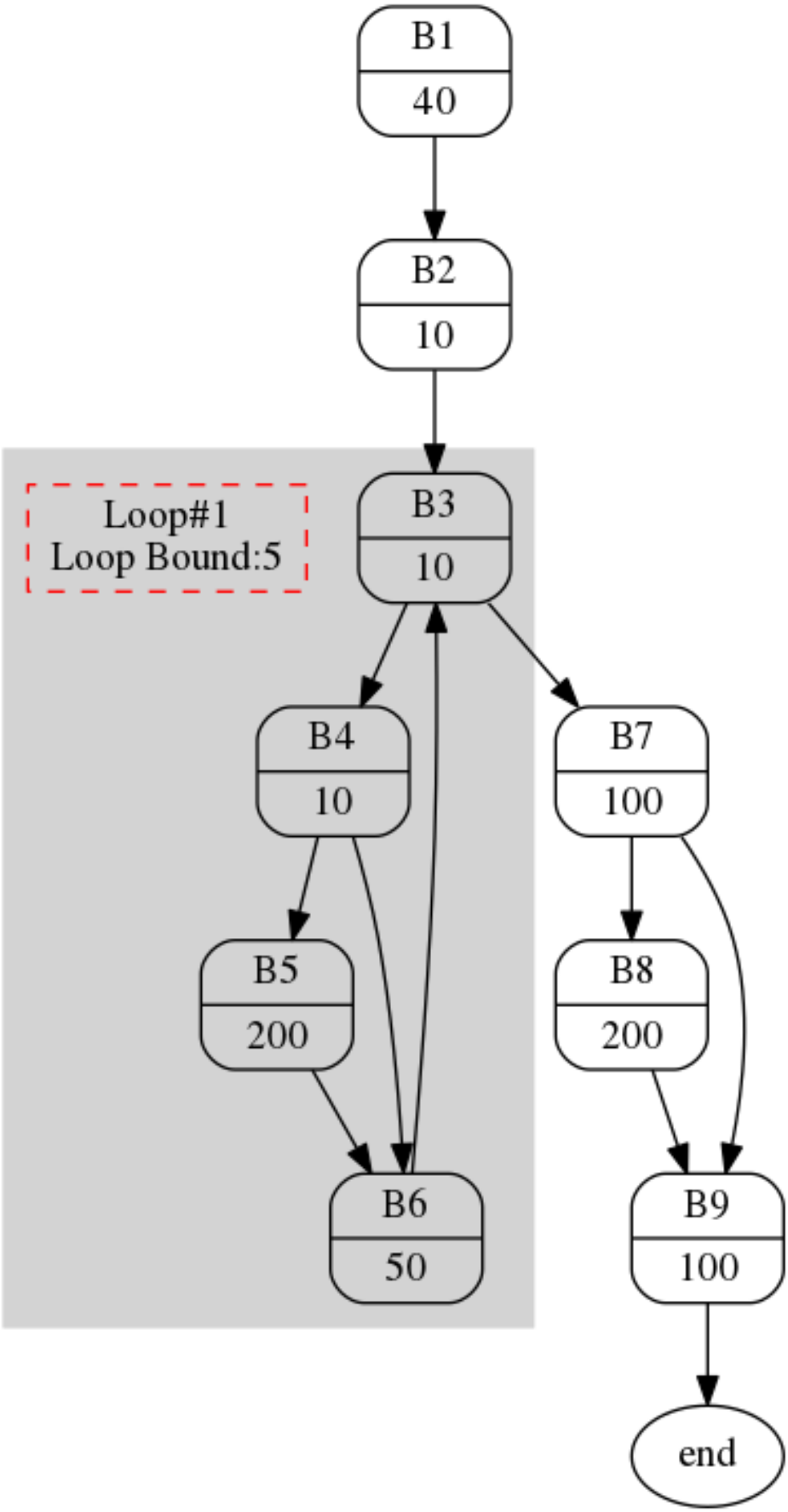


Figure B.3: The CFG of case study

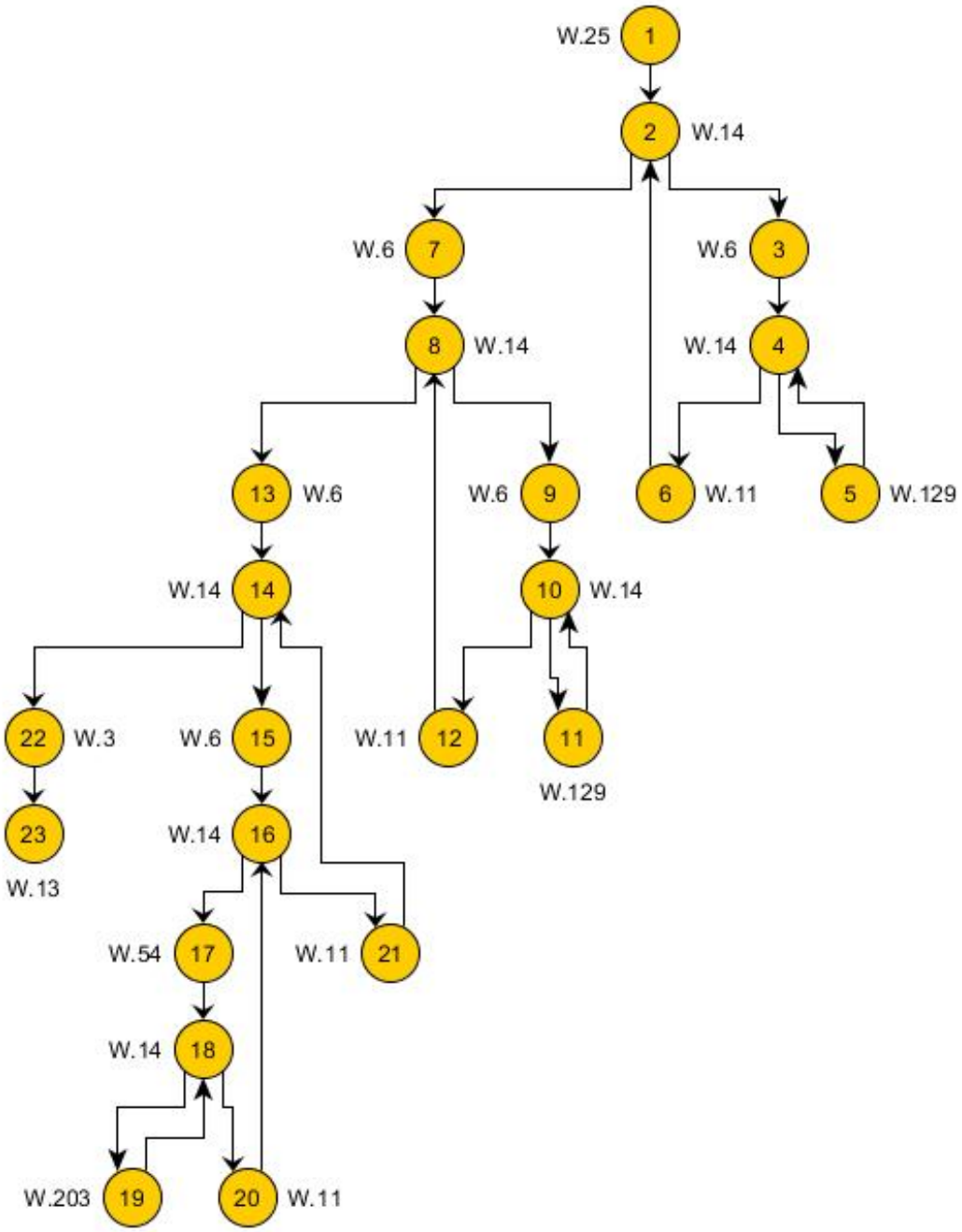


Figure B.4: The CFG of matmult.c

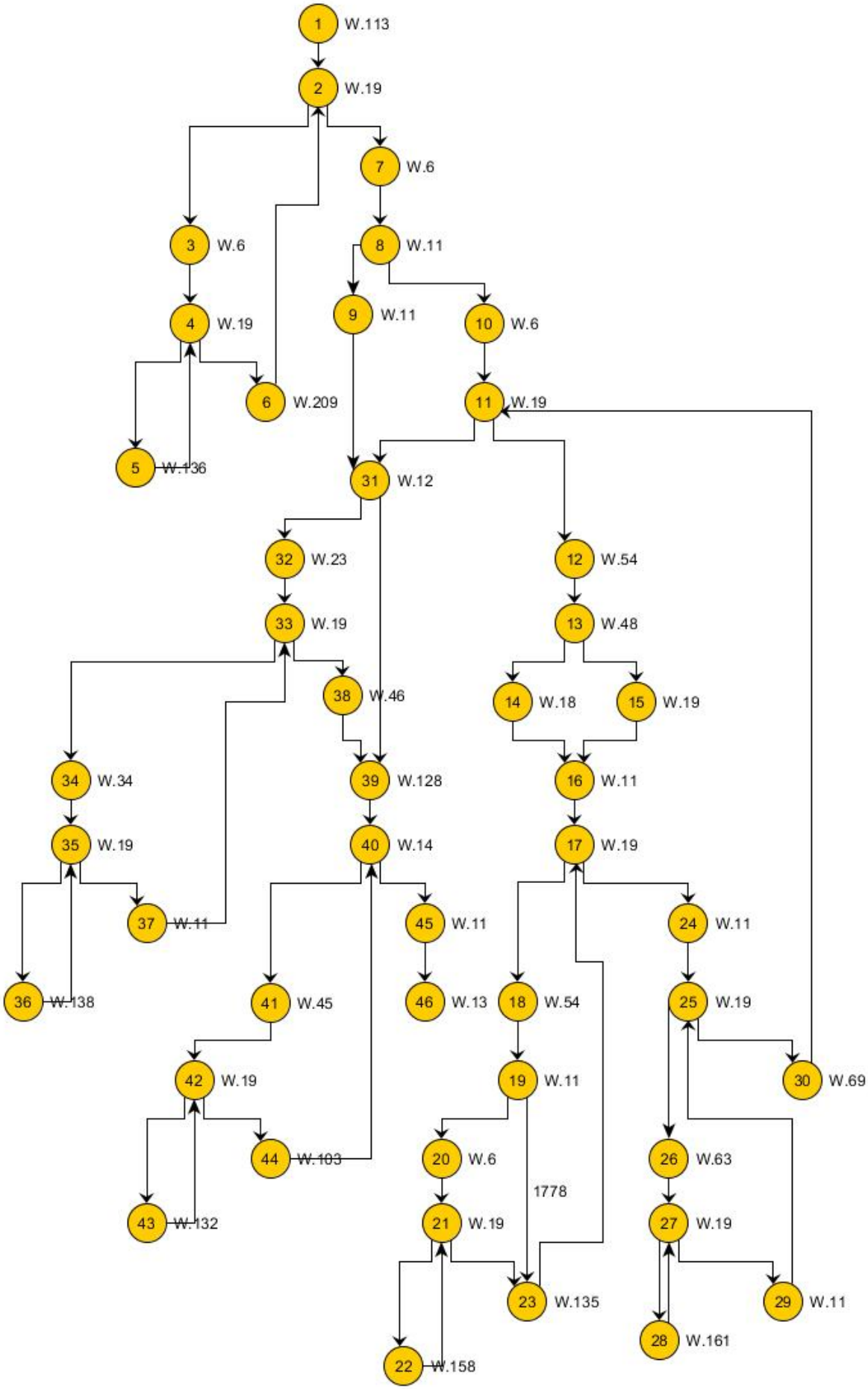


Figure B.5: The CFG of ludcmp.c

Appendix C

Annotation of Scaling Point

```
Task_Numbers: 5
Task_ID: 1
Checkpoint_Numbers: 0 0 1
P1: 1 2 7 1250 900
Task_ID: 2
Checkpoint_Numbers: 0 0 3
P1: 2 3 150 30 7437
P2: 5 6 100 35 3929
P3: 8 9 130 30 16
Task_ID: 3
Checkpoint_Numbers: 1 0 1
B1: 7 8 300 9 100
P1: 2 3 5 270 410
Task_ID: 4
Checkpoint_Numbers: 0 0 6
P1: 1 2 20 2905 1832256
P2: 7 8 20 2905 1774136
P3: 13 14 20 88705 16
P4: 17 18 20 217 0
P5: 9 10 20 143 0
P6: 3 4 20 143 0
Task_ID: 5
Checkpoint_Numbers: 3 0 10
B1: 8 9 9297 10 20266
B2: 11 12 20241 31 9286
B3: 31 32 9274 39 4846
P1: 1 2 6 1183 20283
P2: 12 17 5 1129 14523
P3: 24 25 5 1012 9355
P4: 32 33 5 868 4892
P5: 38 40 5 936 152
P6: 3 4 6 155 0
P7: 20 21 5 177 0
P8: 26 27 5 180 0
P9: 34 35 5 157 0
P10: 41 42 5 151 0
#
```

Figure C.1: The annotation file for configuring the frequency-scaling points at every CFG

Bibliography

[1] Alireza Salami Abyaneh and Mehdi Kargahi. Energy-efficient scheduling for stability-guaranteed embedded control systems. In *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, pages 1–8. IEEE, 2015.

[2] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, 2004.

[3] Ana Azevedo, Ilya Issenin, Radu Cornea, Rajesh Gupta, Nikil Dutt, Alex Veidenbaum, and Alexandru Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 168–175. IEEE, 2002.

[4] Thomas Ball and James R Larus. Using paths to measure, explain, and enhance program behavior. *Computer*, 33(7):57–65, 2000.

[5] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[6] Moris Behnam and Damir Isovic. Real-time control and scheduling co-design for efficient jitter handling. In *null*, pages 516–524. IEEE, 2007.

[7] Muhammad Khurram Bhatti, Cécile Belleudy, and Michel Auguin. An inter-task real time dvfs scheme for multiprocessor embedded systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 136–143. IEEE, 2010.

[8] Reinder J Bril, Elisabeth FM Steffens, and Wim FJ Verhaegh. Best-case response times and jitter analysis of real-time tasks. *Journal of Scheduling*, 7(2):133–147, 2004.

[9] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[10] Anton Cervin. Stability and worst-case performance analysis of sampled-data control systems with input and output jitter. In *American Control Conference (ACC), 2012*, pages 3760–3765. IEEE, 2012.

[11] Anton Cervin, Bo Lincoln, Johan Eker, Karl-Erik Arzén, and Giorgio Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 1–9. Gothenburg, Sweden, 2004.

[12] Marco ET Gerards, Johann L Hurink, and Jan Kuper. On the interplay between global dvfs and scheduling tasks with precedence constraints. *IEEE Transactions on Computers*, 64(6):1742–1754, 2015.

[13] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[14] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 8, page 12, 2017.

[15] Shengyan Hong, Xiaobo Sharon Hu, and Michael D Lemmon. Reducing delay jitter of real-time control tasks through adaptive deadline adjustments. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 229–238. IEEE, 2010.

[16] Texas Instruments. AM335x Power Consumption Summary. `http://processors.wiki.ti.com/index.php/AM335x_Power_Consumption_Summary`, 2016. [Online; accessed 19-July-2008].

[17] Taewoong Kim, Heonshik Shin, and Naehyuck Chang. Deadline assignment to reduce output jitter of real-time tasks. *IFAC Proceedings Volumes*, 33(30):51–56, 2000.

[18] Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A practitioners handbook for real-time analysis: guide to rate monotonic analysis for real-time systems*. Springer Science & Business Media, 2012.

[19] Keqin Li. Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers. *IEEE Transactions on Computers*, 61(12):1668–1681, 2012.

[20] Y Liang, P Lai, and C Chiou. An energy conservation dvfs algorithm for the android operating system. *Journal of Convergence*, 1(1), 2010.

[21] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[22] Pau Marti, Josep M Fuertes, Gerhard Fohler, and Krithi Ramamritham. Jitter compensation for real-time control systems. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 39–48. IEEE, 2001.

[23] Bren Mochocki, Razvan Racu, and Rolf Ernst. Dynamic voltage scaling for the schedulability of jitter-constrained real-time embedded systems. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 446–449. IEEE Computer Society, 2005.

[24] Teera Phatrapornnant and Michael J Pont. Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling. *IEEE Transactions on Computers*, 55(2):113–124, 2006.

[25] D. Pinheiro, R. Goncalves, E. Valentin, H. d. Oliveira, and R. Barreto. Inserting dvfs code in hard real-time system tasks. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 23–30, Nov 2017.

[26] Jan Reineke. Challenges for worst-case execution time analysis of multi-core architectures. 2014.

[27] Sonal Saha and Binoy Ravindran. An experimental evaluation of real-time dvfs scheduling algorithms. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 7. ACM, 2012.

[28] Hiroshi Sasaki, Yoshimichi Ikeda, Masaaki Kondo, and Hiroshi Nakamura. An intra-task dvfs technique based on statistical analysis of hardware events. In *Proceedings of the 4th international conference on Computing frontiers*, pages 123–130. ACM, 2007.

[29] Jaewon Seo, Taewhan Kim, and Ki-Seok Chung. Profile-based optimal intra-task voltage scheduling for hard real-time applications. In *Proceedings of the 41st annual Design Automation Conference*, pages 87–92. ACM, 2004.

[30] Dongkun Shin and Jihong Kim. Optimizing intra-task voltage scheduling using data flow analysis. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 703–708 Vol. 2, Jan 2005.

[31] Dongkun Shin and Jihong Kim. Optimizing intratask voltage scheduling using profile and data-flow information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):369–385, 2007.

[32] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2):20–30, 2001.

[33] Tomohiro Tatematsu, Hideki Takase, Gang Zeng, Hiroyuki Tomiyama, and Hiroaki Takada. Checkpoint extraction using execution traces for intra-task dvfs in embedded systems. In *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, pages 19–24. IEEE, 2011.

[34] GraphML Team. The graphml file format. *Homepage: http://graphml. graphdrawing. org*, 2015.

[35] Burt Walsh, Robert Van Engelen, Kyle Gallivan, Johnnie Birch, and Yixin Shou. Parametric intra-task dynamic voltage scheduling. In *Proceedings of the Workshop on Compilers and Operating Systems for Lower Power (COLP 2003)*, 2003.

[36] Neil HE Weste and David Harris. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[37] Chia-Ming Wu, Ruay-Shiung Chang, and Hsin-Yu Chan. A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters. *Future Generation Computer Systems*, 37:141–147, 2014.

[38] Changjiu Xian and Yung-Hsiang Lu. Dynamic voltage scaling for multitasking real-time systems with uncertain execution time. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 392–397. ACM, 2006.

[39] Thorsten Zitterell and Christoph Scholl. Improving energy-efficient real-time scheduling by exploiting code instrumentation. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 763–771. IEEE, 2008.