UPPSALA
UNIVERSITET

IT15016
Examensarbete 30 hp
Oktober 2015

Putting Hard Real-time Applications together with Dark Silicon

Björn Forsberg

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Putting Hard Real-time Applications together with Dark Silicon

Björn Forsberg

As the number of transistors per unit square on silicon chips continues to increase, the heat production of these chips also increases, making thermal problems more common. As the heat production is dependent on the clock frequency, chips today have a defined thermal safe speed at which the cooling system is sufficient to compensate for the produced heat. A DVFS enabled processor executing at speeds over the thermal safe speed for prolonged periods of time may be forced to power down a set of cores to prevent damage due to overheating. As any deadline violations are considered as a system failure, executing hard real-time workloads on such a platform is not trivial.
This thesis presents an online scheme which enables the execution of workloads on such a system. The scheme uses a history aware bounding function for determining when the processor speed must be increased above the thermal safe speed, and as such has a constant and small memory and computation footprint.
The scheme is analysed using predefined traces within the gem5 simulator. For this purpose a new simulated hardware device is implemented which allows the scheduler to read predefined trace files.
With workloads that follow the model closely the online scheme performs on-par with the optimal offline implementation. For workloads whose arrival curve follow the model less closely, the presented scheme is still able to assign timing correct processor speeds.

Handledare: Kai Lampka
Ämnesgranskare: Stefanos Kaxiras
Examinator: Lars-Åke Nordén
IT15016
Tryckt av: Reprocentralen ITC

Populärvetenskaplig sammanfattning

Dagens och framtidens processorer kan exekveras i allt högre hastigheter. De ökade hastigheterna innebär dock betydligt ökad energiåtgång och därmed värmeproduktion. Med anledning av detta innehåller processorerna undersystem som i händelse av energiåtgång eller temperaturer över förutbestämda tröskelvärden stänger av transistorer på silikonchippet. Dessa undersystem kallas brett för Dynamic Thermal and Power Management Systems, och förkortas DTPM. Hur stora enheter som stängs av varierar mellan olika implementationer, men kan till exempel utgöras av processorn som helhet eller enstaka processorkärnor.

Eftersom att systemet drar mycket energi, respektive producerar mycket värme när de körs vid höga klockfrekvenser, vill man undvika detta utan att förlora prestanda. Lyckligtvis är det så att moderna processorer är så pass snabba att de sällan används till 100%. Eftersom att processorn mellan varven måste vänta, till exempel på data från minne eller hårddisk, ökar och minskar ofta arbetsbelastningen i intervaller. Dessa intervaller kan klassificeras som antingen processor- eller minnesbundna intervaller, där det förstnämnda är de intervaller i vilka arbetsbelastningen är hög. Istället för att låta processorn arbeta i högsta hastighet och snabbt avsluta ett processorbundet intervall och därefter utföra så kallade icke-operationer under de minnesbundna intervallen, kan man med hjälp av så kallad Dynamic Voltage/Frequency Scaling, eller DVFS, anpassa processorns klockfrekvens för att matcha arbetsbelastningen under det aktuella intervallet.

Denna teknik används aktivt i dagens persondatorer, men har ännu inte implementerats i realtidssystem. Med realtidssystem menar man ett system där systemets korrekta funktion inte bara definieras av *vad* systemet ger för utsignaler eller utdata, utan även av *när* det ger dessa. Realtidssystem utgörs ofta av inbyggda datorsystem, det vill säga system som ingår som en del i ett större system. Ett vanligt exempel på detta är t.ex. de datorsystem som finns inbyggda i bilar och styr allt större delar av dessa. Här är tidskomponentens betydelse tydlig när man resonerar om t.ex. det system som styr inbromsningen av bilen: Det är inte bara viktigt att datorn aktiverar bromsarna när man trycker på bromspedalen, utan även att den gör det *nu*, eftersom att en olycka annars kan inträffa. Att ändra processorns klockfrekvens med hjälp av DVFS-system under dessa förhållanden introducerar åtskilliga problem som behöver lösas för att inte de tidskritiska komponenternas funktion ska äventyras.

Detta examensarbete presenterar ett schema som genom att förutse den framtida arbetsbelastningen kan avgöra när processorns klockfrekvens måste

höjas för att utesluta att processer i systemet inte möter sina tidskrav. Enkelt uttryckt fungerar detta genom att systemet vet hur ofta nya processer kan starta, och genom att räkna hur många processer som redan *har* startats och jämföra detta med hur många processer som *kan* startas, kan systemet räkna ut hur många processer som kan komma att behöva schemaläggas samtidigt på processorn, och avgöra hur hög processorhastigheten behöver vara för att producera resultat i tid.

Tester gjorda på hårdvarusimulatorn *gem5* visar att det presenterade schemat klarar av att exekvera realtidslaster med garantin att alla tidkrav efterlevs. Utöver detta klarar systemet, i de fall som processer startar i god överensstämmelse med modellen, att exekvera med så liten energiåtgång att det kan mäta sig med det optimala värdet. I de fall som processerna startar under mindre ordnade former klarar systemet fortfarande av att uppfylla alla tidskrav, dock med en högre energiåtgång.

Contents

Contents **7**

List of Figures **9**

List of Algorithms **11**

1 Introduction **13**
1.1 Background and Setting 13
1.2 Purpose and Evaluation 14
1.3 Related Work 14
1.4 Statement of Originality 15
1.5 Organisation 15

2 System modelling **17**
2.1 Processor model 17
2.2 Thermal model 19
2.3 Task model 20
2.3.1 Task activation patterns and bounds 21
2.3.2 History aware bound of future task arrivals 23

3 Worst-Case Ready Queue **27**
3.1 The Worst-Case Ready Queue 27
3.2 Speed assignments 29

4 Evaluation **35**
4.1 System setup 35
4.2 Execution 37
4.3 Results 38

5 Discussion **43**

6 Conclusions **45**

Bibliography **47**

List of Figures

2.1 Visualisation of the DTM system powering down cores 18
2.2 The thermal model represented as an automata 20
2.3 Visualisation of the relationship between $\bar{\alpha}$ and $R(\Delta)$ 22
2.4 Visualisation of alpha curves 23

3.1 Simplified visual representation of the WCRQ 30

4.1 Overview of how the trace is presented to the scheduler 36
4.2 Overview of the interaction between components in the evaluation environment 38
4.3 Comparison of secondary core uptime, in percent of total uptime, between online and offline execution of each trace. The online run of the system uses the implemented scheme. In contrast, the offline run is executed with complete knowledge about all future events in the system, and is optimal with respect to speed changes, which in turn decides when the secondary cores are paused and resumed. 39
4.4 Demand, and online and offline supply for the *max* traces. 40
4.5 Demand, and online and offline supply for the *var* traces. 41

List of Algorithms

2.1 UpdateDNC - Update the dynamic counter 24
3.1 ComputeWCRQ - Add virtual entries to the WCRQ 29
3.2 Scheduler - Update WCRQ and make scheduling decision 31
3.3 SpeedAssignments - Assign speeds to jobs in WCRQ 32

Chapter 1

Introduction

As Moore's law continues to apply, the number of transistors that can be put on a single chip continues to increase. However, as the transistor density increases, it has given raise to thermal problems, as the heat production per area increases. Likewise, another factor that is driving up the heat production of silicon chips is the increase in clock frequency, and at high clock frequencies, it becomes increasingly difficult for the system to sufficiently cool the transistors.

1.1 Background and Setting

To counter the hazards introduced by increased heat production, modern processors are equipped with a Dynamic Thermal Management systems (DTM). The DTM system allows the hardware to monitor the state (temperature) of the system, and take measures to ensure that the temperature specifications of the system are not violated. In addition to this, modern processors allow the dynamic changing of the clock frequency, using techniques such as Dynamic Voltage/Frequency Scaling (DVFS). This technique enables the clock frequency to be altered from software during system execution.

In the case of multi-core variable-speed processors, this has lead to the definition of a thermal safe power (TSP), and the corresponding thermal safe speed (TSS), a speed at which the dynamic heat production of the processor can be sufficiently dissipated by the cooling system. At speeds higher than the TSS, the heat production is higher than the capacity of the cooling system, and the system can no longer run all cores for an indefinite period of time without risking undefined behaviour or physical damage to the processor [11].

The powering down sets of transistors on a chip is referred to as *dark silicon* and can occur due to thermal, power and other reasons [5]. The set of transistors to power down is chosen on a functional level, and in the case of multi-core processors, this may include the transistors that make up a processor core. In a DVFS enabled system which allows frequency settings above the TSS, the system may run into situations in which the DTM system is forced to power down a set of cores.

These systems are becoming more and more commonplace, and it is not unlikely that they will be used in hard real-time settings. This presents a problem, as deadline violations within a hard real-time system is not acceptable. While

it is possible to execute hard real-time workloads on such a system by running the processor at speeds lower than the TSS, this may lead to poor use of the hardware.

1.2 Purpose and Evaluation

Instead, this thesis presents an online scheme which uses DVFS to control the clock frequency of the processor in such a way that hard real-time workloads which require higher speeds than the TSS can be executed on hardware that suffers from thermal induced dark silicon without introducing any deadline violations. This is done by using a history-aware bound for future task activations to predict the future workload, and using this to increase the clock frequency of the processor at peak workloads. Due to the nature of the scheme, the only design time parameter required is an upper bound for the task activations, which can be derived for both simple, deterministic workloads, and workloads with a high degree of non-determinism. As such, the scheme can be used to assign speeds to a large set of workloads with non-deterministic arrival patterns, common in real-time applications. The presented scheme relies only on a small (and constant) memory and computation footprint, and is thus suited for online use.

While work has been done in the individual areas of online speed assignments, DVFS speed assignments and DTM induced dark silicon, this work is, to the best knowledge of the author, the first work which combines all these areas into a single framework for executing hard real-time workloads on such systems.

Evaluation of the presented scheme using predefined task activation traces within the gem5 simulator shows that for workloads adhearing close to the model used, the scheme performs on-par with an offline implementation with full information about the task activation times. As the trace diverges further from the upper bounds of the model, the scheme is still able to assign speeds to the jobs which ensure timing correctness, although with increased pessimism.

1.3 Related Work

There has been a large amount of work in the setting of thermal aware scheduling in the last 10 to 15 years.

Work has been done in executing real-time workloads in energy efficient ways. Work in this area can be coursly divided into three categories, the reduction of power usage through Dynamic Voltage/Frequency Scaling (DVFS), Dynamic Power Management (DPM) and other methods.

Methods for reducing the power usage through DVFS has been presented by [12, 9], in which somewhat different approaches have been taken to the problem. [12] present a scheme which combines using DVFS to assign the lowest possible processor speed based on the current workload, but switches to the maximum speed of the system when the workload requires a higher speed than a threshold value. This is done to prevent the system from requiring speeds higher than the maximum speed of the processor if the previous workload was executed too slowly. [9] present a scheme in which a combination of offline and online bounding of the workload are used to make decisions on which speed to run

the processors at. The offline, or static, component is calculated at design time of a system and provides a bound of the behaviour of the workload. The online component adjusts the bounds defined by the static component based on the (sliding window) history of the workload, and as such allows the system to provide tighter bounds than the static bound based on the history of the system, however, this method is computation intensive.

The use of workload history for bounding the future workload has since become a common way to produce tighter bounds than what can be made in an offline scheme. [10] present a method in which minimum distance functions are used to implement a lightweight workload monitoring scheme in which, for certain l-repetitive functions, only the l most recent events must be tested. This scheme can be implemented with a design time limit on the computation and memory overhead acceptable to the designer. [7] present another scheme for bounding the future workload behaviour using dynamic counters. The dynamic counters use an overapproximation of the bounding curves inherent to Real-Time Calculus (RTC) [15], and implement counters that trace these curves. In addition to this, the counter is decreased each time a new task activation occurs, and as such, the counters are dynamically updated to contain the difference between the task activation bound and the actual history, which is the potential future workload. Because of this design, the memory and computation overheads are small and constant. This scheme is used as a basis for the scheme presented in this master thesis. In addition to this scheme for bounding the future workload, [7] present a dynamic power management (DPM) scheme based on these dynamic counters.

A survey in dynamic speed assignment algorithms is presented in [2]. In addition to this, several papers on minimising the temperature in real-time systems without DVFS have been presented, among others [14, 6, 1].

1.4 Statement of Originality

This master thesis presents work that has been done by the author in collaboration with Kai Lampka at the Embedded Systems group of the Department for Information Technology at Uppsala University. Several of the key theoretical pieces have been provided by Kai Lampka, while the main contributions from the author has been in assisting with the formulation of algorithms and evaluating the scheme. However, all of the theory is presented for completeness.

1.5 Organisation

The rest of this thesis consists of five chapters. In Chapter 2 the models upon which the implemented scheme relies are presented, including any assumptions made upon these. The chapter is further divided into three sections, each covering one of the three main models; the Processor model (Section 2.1), the Thermal model (Section 2.2) and the Task model (Section 2.3). Chapter 3 presents the scheme itself, which is then evaluated in Chapter 4. The thesis is concluded with a discussion and conclusions drawn, in Chapters 5 and 6 respectively.

Chapter 2

System modelling and prediction of the future workload

The presented scheme relies upon three models, the processor model, the thermal model, and the task model. Each of these models describe the assumptions made upon each of the parts of the system, and precisely defines what is meant with each term. Lastly, the scheme used for deriving a history aware bounding of future task arrivals is presented.

2.1 Processor model

The processor is assumed to be a K-core DVFS capable processor, executing a hard real-time workload. It is assumed that each task within the system is mapped to a fixed core, and as such there is no task migration between cores at runtime. This is done to simplify the reasoning.

The model assumes that it is always the same set of cores that is powered down by the DTM system. This is a reasonable assumption, as cores placed next to each other will increase the heat concentration, and powering of the cores in such a pattern that the heat concentration is reduced allows the remaining cores to operate at higher frequencies. A visual representation of this is presented in Figure 2.1. The cores that are always powered on are referred to as *main* cores, while the cores that may be powered off by the DTM system are referred to as *secondary* cores. To further simplify the reasoning it is assumed that only one of the main cores is executing the decisive workload, that is, the workload which requires the processor to increase the clock frequency above the TSS. While this may seem limiting, it is easy to generalize the reasoning to include several decisive cores, by introducing a *global max* routine.

As the clock frequency f of the processor varies, so does the execution time required to finish a job. However, the execution time of a job can not be directly calculated from the number of cycles produced by the processor. While a higher clock frequency increases the number of instructions that can be processed per time unit, the higher number of cycles per time unit also increases the number of

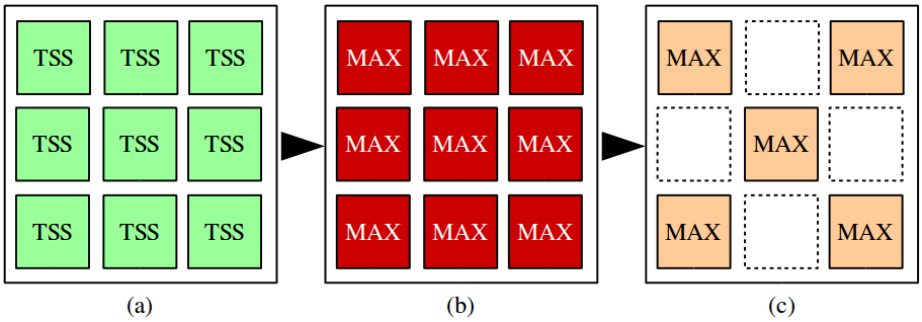


Figure 2.1: Visualisation of the DTM system powering down cores. In the leftmost image (a) all cores are running at the TSS, at which point the cooling system is able to sufficiently cool all cores for an indefinite amount of time. In the middle image (b) all cores are executing at the maximum speed supported by the processor, at which point the cooling system is insufficient, thereby causing the CPU temperature to increase. In the last image (c) the DTM system has powered down a set of cores to prevent overheating of the processor. The remaining cores can operate at the maximum speed for as long as needed, as the cooling system can once again sufficiently cool all executing cores.

cycles the processor has to wait for other subsystems. This includes the reading and writing of data from caches and main memory as these subsystems operate at a constant clock frequency. This means that the execution time increases sub-linearily as the clock frequency decreases.

As a (simplified) example, consider that the fetching of data from memory takes 100 cycles at clock frequency f, the fetching will take 50 cycles at clock frequency $0.5 \times f$, as the memory operates at a constant speed, and each clock cycle now takes twice as long. Should the program require 100 cycles of execution to finish, in addition to the fetching of data, it would take $100 + 100 = 200$ cycles at speed f, and $100 + 50 = 150$ cycles at speed $0.5 \times f$. The time required to execute this can be expressed relatively to the higher speed as x in $\frac{200}{f} = x \times \frac{150}{0.5 \times f}$, which gives $x = \frac{2}{3}$, which means that, in this simplified example, halfing the clock frequency only decreases the execution time to $\frac{2}{3}$. While the execution time relative to the time it takes to fetch data from memory may be high in this example, it serves to illustrate the sub-linear relationship between clock frequency and execution time.

Under this assumption, a better model for expressing the execution time scaling in respect to clock frequency is to normalise the execution times to the maximum clock frequency f_{max}. This means that every execution time c can be expressed on the form $c \times \frac{f}{f_{max}}$, where f is the current clock frequency. Generally, this fraction is referred to as the execution speed s, where $s_{current} = \frac{f_{current}}{f_{max}}$. This simplifies the execution time expression to $c \times s$, where c is the worst case execution speed of the task, and s is the current speed of the processor. By norming all execution times to the maximum clock frequency, the actual execution times will be lower than the assumed ones, as all memory access times are assumed to conform to the access times for the maximum speed, although as shown above, the delay relative to the clock frequency will decrease as the clock frequency decreases. This allows for the conservative scaling of execution

times with regards to the execution speed. Using this concept of normed speeds instead of clock frequencies, the TSS can be expressed as s^{th}, and the maximum speed is defined as $s^{max} = 1$.

While the processor may be able to execute at several speeds in $[s^{th}, s^{max}]$, it has been shown [8] that modern hardware will not benefit from fine-grained voltage/frequency scaling. For this reason, the only cases used by the presented scheme is when the system is executing at either s^{th} or s^{max}. This also has the added effect of limiting the overhead introduced by switching times, as the processor will execute at s^{th} for as long as possible, and then switch directly to s^{max} when the peak workload appears. In addition to this, this scheme does not consider the use of dynamic power management (DPM) techniques for completely turning the processor cores on or off. While the static power consumption, and thereby heat production, of the processor cores drops when the cores are turned off, the state switching introduces an overhead which is not negligable [3]. As the scheme is to be used in hard real-time settings in which tasks produce jobs at short intervals, it is not likely that the system will idle for periods long enough to motivate, or even permit, switching the cores into offline mode.

2.2 Thermal model

The implemented scheme relies upon a simple thermal model to reflect a simplified version of the heatup and cooldown phases of the processor. This is represented by an automata with four states. Depending on the current clock frequency and history the thermal model places the system in one of the four states, see Figure 2.2.

The states of the thermal model, representing different modes of execution, are *cooldown*, *normal*, *heatup* and *overheat*. In the first two states, *cooldown* and *normal*, the system is running at the TSS, and as such, the cooling system is able to remove any excess heat from the processor. In the first case however, the system has recently switched from a higher clock frequency, and as such, the processor is still too hot for all cores to execute normally until the temperature has dropped further. Once the temperature T has dropped below a threshold value T^{darken}, the thermal state is changed to *normal*, in which the system is able to run and cool all processors within the system.

In the two second states, *heatup* and *overheat*, the system is executing at a speed above the TSS, namely the maximum speed supported by the processor. In the *heatup* state, all processor cores are executing normally, as the temperature T is still below the threshold value T^{darken}, but as the processor is executing at a speed above the TSS, T is constantly increasing. Once $T \geq T^{darken}$, the system has to switch of the secondary cores to prevent overheating problems, and the thermal state changes to the *overheat* state, in which only the primary cores are running, but the temperature increase is prevented. In the *overheat* state, the value of T is continously incremented up to T^{max}. This means that the processor is executing normally in the temperature interval $[0, T^{darken}]$, and in reduced mode, in the interval $[T^{darken}, T^{max}]$. The effect of this is that the difference $T^{max} - T^{darken}$ is the maximum cooldown time of the system, while $T^{darken} - 0$ is the maximum heatup time. The state of the system when $T = T^{darken}$ can be either *normal* or *overheat*, depending on the current tem-

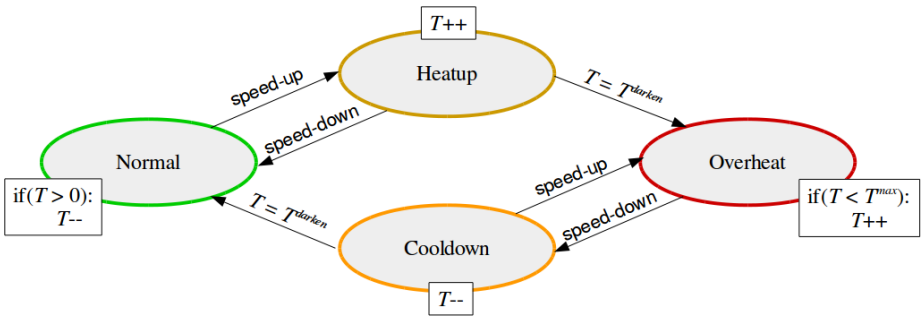


Figure 2.2: The thermal model represented as an automata. In the *normal* and *cooldown* states the cores are executing at the TSS, while the processor is executing at the maximum speed in *heatup* and *overheat*. The processor is cool enough to operate all processor cores at once in the *normal* and *heatup* states, while only the main cores are executing in states *overheat* and *cooldown*.

perature trend, i.e. if the previous state is *cooldown* or *heatup*.

The main idea is that the changing of clock frequencies alters the thermal model in the following way. Initially the system is in the *normal* state, in which the processor executes at the TSS, and has been doing so for a long enough period of time for all cores of the processor to execute normally. Should the decisive core encounter a peak workload and increase the clock frequency above the TSS, a transition to the *heatup* state is triggered. At this point, the processor is still cool enough for all cores to operate, but as the cooling is insufficient, the temperature is steadily rising in this state. Once the temperature has increased above a threshold T^{darken}, the transition to the *overheat* state is triggered, in which the DTM is activated and powers down the secondary cores. While in this state, the temperature does no longer increase, as the power down of the secondary cores means that the cooling system regains sufficiency.

Once the peak workload has been processed by the decisive core, the system will return to the TSS, which in turn triggers the transition to the *cooldown* state. At this point, the processor is still too warm to power up the secondary cores, but as the processor is now running at the TSS, the cooling system is sufficient, and the temperature is decreasing. Once the temperature decreases below the threshold temperature T^{darken}, the DTM system is once again activated and triggers the transition to the *normal* state, in which all cores are once again running at the TSS, and the cooling is sufficient to continue their execution indefinitely.

There are also some complimentary transitions possible within the thermal model. Should the peak workload be finished before the processor overheats, the model allows the transition directly from the *heatup* back to *normal*, and equivalently from *cooldown* to *overheat* if another peak workload appears before the processor has cooled down since the last one.

2.3 Task model

A hard real-time task τ_i is defined by a worst case execution time C_i, relative deadline D_i and an upper bound on task activations $\bar{\alpha}_i$. A task activation is an

invocation of the task, and each invocation is known as a *job*, which is the entity executed by the job scheduler. A job $J_{i,j}$, where j represents the j^{th} activation of task τ_i, has an absolute deadline $d_{i,j}$, which is defined as the relative deadline D_i of the task τ_i added to the arrival time of the job $J_{i,j}$. The residual execution time $c_{i,j}$ of the task is initialised at job arrival to the worst case execution time C_i of the corresponding task. The execution times are, as previously presented, normed to the maximum clock frequency of the processor.

As previously presented, the processor has a thermal stable speed (TSS, s^{th}) at which the cooling is sufficient to keep all processor cores powered up. At any speed above s^{th}, the DTM system will eventually power off the secondary cores to be able to safely execute the remaining cores at the requested higher speed. For a hard real-time workload, a feasible speed s^{fs} is defined, which is the minimum speed required for all task activations to finish executing before their respective deadlines under the assumed scheduling principle. It is, of course, possible to choose the workload such that $s^{fs} \leq s^{th}$, in which case the dark silicon problem will never arise, but this might lead to low utilisation of the hardware. Instead, the implemented scheme ensures that a hard real-time system meets its deadlines even if $s^{fs} > s^{th}$.

The scheme assumes that the system initially executes at the thermal stable speed, but has to switch to the maximum speed s^{max} at peak workloads to prevent deadline misses.

As the model assumes that all tasks are mapped to a single core, the feasability of a core can be calculated on a per core basis. In addition to this, it is clear that the feasible speed s^{fs} is the same as the utilization u of the core in question, as the speeds are normalized to the maximum speed supported by the processor, and as such represent the usage of the core. It is assumed that the workload has been proven feasible beforehand, that is, the feasible speed of the workload on which the scheme is executed is within the interval supported by the processor, thereby giving $0 \leq s^{min} \leq s^{fs} = u \leq s^{max} = 1$. As the secondary cores may be powered off due to the main cores (to which the decisive core belongs) operate at speeds above the TSS, they are not suitable for hosting hard real-time tasks, but may be used for executing best-effort workloads.

2.3.1 Task activation patterns and bounds

The task activation pattern of a task is a representation of how and when task activations occur for a task within a system. While a simple activation pattern, like strictly periodic arrivals, present an easy way to reason about the workload of a system, it conforms badly to the practical scenarios in which these workloads are expected to execute. The task arrival patterns of actual systems are often non-deterministic, and depend on several factors within the system itself, but, as is even more difficult to model, the activation of tasks may depend on external events. For the system to safely schedule the workload, the system must be able to not only schedule the current jobs, but any future jobs as well. This requires a more sophisticated model for the task activation pattern. For this purpose, the task activations can be described by a discrete event curve as defined by Real-Time Calculus (RTC) [15], and can be bounded on an interval $[t, t+\Delta]$ by a staircase curve $\bar{\alpha}_i$. The bounding curves are sub-additive curves in which the following holds true

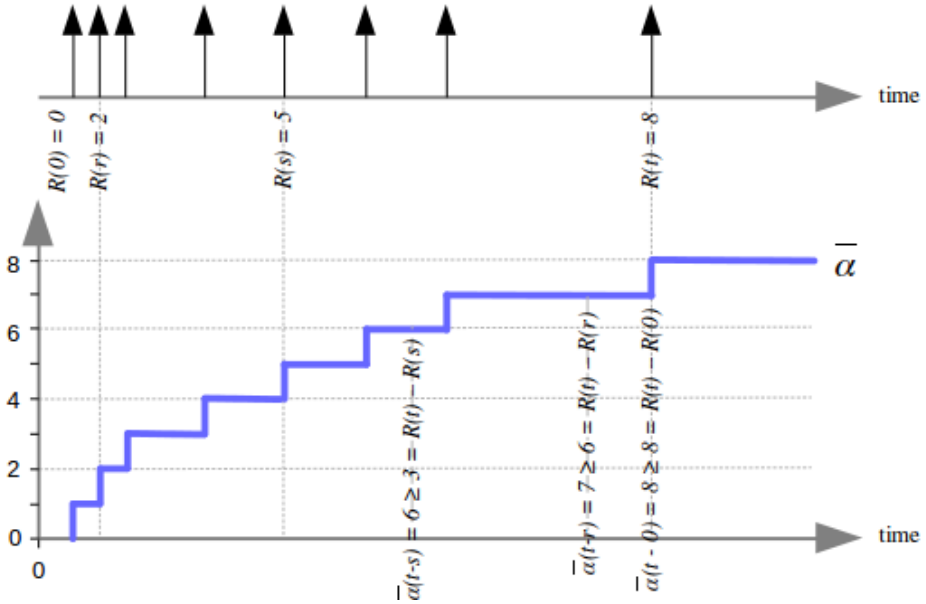


Figure 2.3: Visualisation of the relationship between the sub-additive $\bar{\alpha}$ and $R(\Delta)$ curves. The top graph shows when task activations occur, and the new values of the cummulative task activation counting function $R(\Delta)$. The bottom graph displays the $\bar{\alpha}$ function, and it's relation to the $R(\Delta)$ values at some key points. Initially the relation holds, as $\bar{\alpha}(0) = 0 \geq 0 = R(0)$.

$$\bar{\alpha}(t-s) \geq R(t) - R(s) \quad \forall s \leq t \tag{2.1}$$

in which $R(\Delta)$ is the cumulative number of task activations up to time Δ. A visualisation of this relation is presented in Figure 2.3.

For the workload in a system, each task τ_i has an individual task activation bounding curve $\bar{\alpha_i}$. The $\bar{\alpha_i}$ curve can be over-estimated by constructing a new curve α_i, such that $\alpha_i(\Delta) \leq \bar{\alpha_i}(\Delta) \quad \forall \Delta \geq 0$. This curve α_i is created by combining a set K_i of linear staircase functions $a_{i,j}$ of which the *minimum* is taken in each point, see Equation 2.2. A visualisation of this is presented in Figure 2.4.

$$\alpha_i(t) = min_{j \in K}(a_{i,j}(t)) \tag{2.2}$$

The linear staircase functions a_i are defined by two variables, the step-width δ and the intersection with the Y-axis N. These variables in turn represent the inter-arrival time of events (δ), and the number of events that can appear at the current instant in time, also called the burst capacity (N) [7].

It can be shown that for well chosen over-approximations of $\bar{\alpha_i}$ and large values of Δ, the long term task activation *rate* is $\rho_i = min_{j \in K_i}(\lfloor \frac{\Delta}{\delta_{i,j}} \rfloor)$ [16], where K_i is the set of staircase functions for over-estimating the arrival curve of task τ_i. This will be used in Section 3.1 as a basis for discussing which future jobs that are relevant for processor speed assignments.

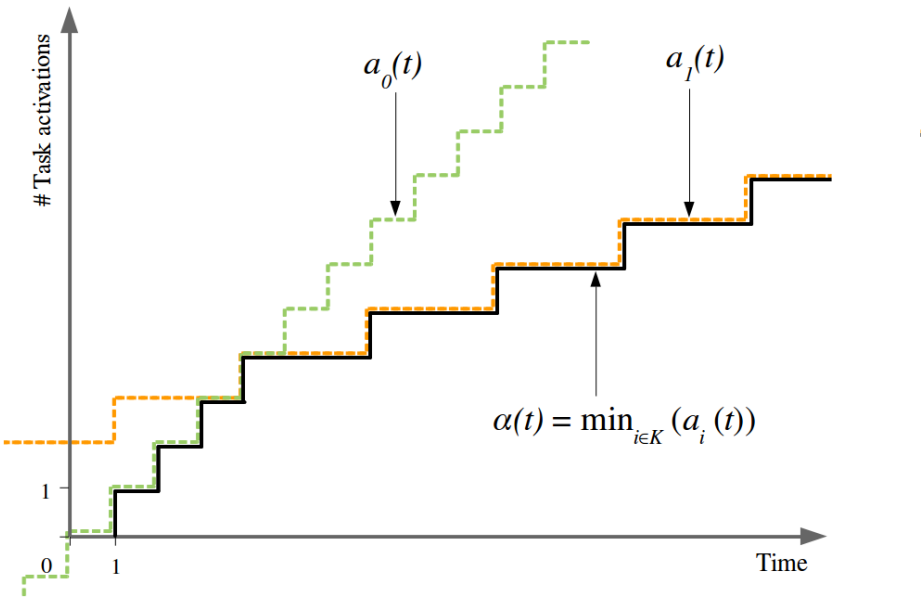


Figure 2.4: Visualisation of how the two linear staircase functions $a_0(t)$ and $a_1(t)$ are used to create the linear task activation over-approximation curve $\alpha(t)$ of $\bar{\alpha}$ in the interval $[0, \infty]$.

2.3.2 History aware bound of future task arrivals

While the task activation bound is useful for reasoning about the behaviour of a workload, it is not useful to an online scheme unless it can be used to derive the current, and predict future, states of the system. It is possible to implement this behaviour by keeping a history of previous events, and use this history-awareness to predict the future behaviour of the system as a whole (see Section 1.3, Related Work). This, however, introduces both memory and processor overheads, as the history must be stored for as long as possible for the future predictions to be accurate. A less memory intensive way of creating a history aware task activation bound is by the use of dynamic counters [7].

The dynamic counters are operating on one of the linear staircase functions $a_{i,j}$ each, meaning that there is one dynamic counter $DNC_{i,j}$for each staircase function $a_{i,j}$ in K_i that make up α_i. Each of these dynamic counters use the two defining characteristics of the staircase curve they are coupled with, the step-width $\delta_{i,j}$, and the burst capacity $N_{i,j}$. Using these characteristics the dynamic counter $DNC_{i,j}$, tracking curve $a_{i,j}$, is able to give a bound on the number of task activations that may occur at the current instant in time. In short, this is done by letting a timer elapse at the end of every step in the staircase curve, at which point the counter is increased. Likewise, each time a task is activated the counter is decreased. As the staircase curve represents (one of the components of) the task activation bound, increasing the counter at each step causes the counter to reflect the value of the bounding curve at the current point in time. In addition to this, by also decreasing the counter for each job arrival, the counter becomes history-aware, as the difference between the bounding curve and the number of actual task activations represent the

potential jobs that have yet to arrive. This means that the value of the counter will not bound *all* task activations up to the current point in time, but only the *potential* activations that have yet to occur. The full algorithm for keeping the dynamic counters updated is presented in Algorithm 2.1.

Algorithm 2.1 The UPDATEDNC algorithm, which updates a dynamic counter $DNC_{i,j}$ based on either event arrivals or elapsed timer, given by the s signal.

```
1: function UPDATEDNC(index, i, j, signal s)

2:     if s == event_arrival then
3:         if DNC[i,j] == N_{i,j} then
4:             set(TIMER[i,j], δ[i,j])
5:             set(T[i,j], 0)
6:             A[i,j] = 0
7:         end if
8:         DNC[i,j] --
9:     end if

10:    if s == timer_elapsed then
11:        DNC[i,j] = min(DNC[i,j] + 1, N_{i,j})
12:        A[i,j] ++
13:        set(TIMER[i,j], δ[i,j])
14:    end if

15: end function
```

As shown by [7] there are points in time at which the previous task activation history becomes irrelevant for the future behaviour. These points are referred to as *renewal points*, and occur when a job arrives when the dynamic counter is equal to the burst capacity. Once this occurs for dynamic counter DNC_i, the corresponding timer T_i, which gives the time since the last renewal point, and A_i which gives the number of activations since the last renewal point, are reset to zero. These renewal point aware values will be used for predicting the future workload.

As the dynamic counter in itself only bounds the number of jobs that may arrive at the current point in time, or the burst capacity, the counters in themselves do not provide sufficient information about the future workload to guarantee that both the current and future workload will meet their deadlines. Therefore, in addition to the burst capacity, we need to take into account the future task activations. To create a demand bound function for the future task activations in the interval $[t, t + \Delta]$, the formula presented in Equation 2.3 can be used. In this equation $T_{i,j}$ is the time since the latest renewal point, and $A_{i,j}$ is the number of staircase steps that occur within this time, both of them are kept up to date by Algorithm 2.1. Furthermore, $\delta_{i,j}$ and $N_{i,j}$ are the parameters that define the staircase function handled by the dynamic counter. In all cases, the subscript i, j referrs to the jth staircase function in the set K_i that make up the curves of α_i, where τ_i is the ith task executing on the decisive core.

$$Q_{i,j}(\Delta) = \begin{cases} \lfloor \frac{\Delta + T_{i,j} - A_{i,j} \times \delta_{i,j}}{d_{i,j}} \rfloor & \text{if } DNC_{i,j} < N_{i,j} \\ \lfloor \frac{\Delta}{\delta_{i,j}} \rfloor & \text{if } DNC_{i,j} = N_{i,j} \end{cases} \tag{2.3}$$

$Q_{i,j}(\Delta)$ is the number of *new* task activations that may occur in the interval $[t, t + \Delta]$. This function calculates the number of steps that $a_{i,j}$ will take within

the given interval, and, if neccesary, adds the number of task activations that have yet to occur from previous intervals.

As the dynamic counter $DNC_{i,j}$ bounds the number of task activations that may occur *now*, and $Q_{i,j}(\Delta)$ bounds the number of *new* task activations that may occur in the interval $[now, now + \Delta]$, the total number of task activations within the interval is the sum of these two terms $U_{i,j}$, as shown in Equation 2.4.

$$U_{i,j}(\Delta) = DNC_{i,j} + Q_{i,j}(\Delta) \tag{2.4}$$

By combining the history aware $U_{i,j}$ values for each staircase function $a_{i,j}$, a history aware task activation bound can be created for each task τ_i by, as with the arrival curves of Equation 2.2, combining them using the minimum operation, as shown in Equation 2.5. The history aware demand bound function for the future workload $FDBF$ is then created by multiplying the task activation bounds U_i with the worst case execution time C_i for each task in τ_i executing on the decisive core, as shown in Equation 2.6.

$$U_i(\Delta) = min_{j \in K_i}(U_{i,j}(\Delta)) \tag{2.5}$$

$$FDBF(\Delta) = \sum_{i \in \tau} U_i(\Delta) \times C_i \tag{2.6}$$

The demand bound function for the current workload $CDBF(\Delta)$ is the remaining worst case execution time for all jobs currently present in the system, up until point Δ. These jobs are stored in what is commonly referred Ready Queue RQ of the scheduler, and with $RQ(\Delta)$ define all jobs up until Δ. This is presented in Equation 2.7

$$CDBF(\Delta) = \sum_{\tau_i \in RQ(\Delta)} c_i \tag{2.7}$$

Lastly, the total demand bound function for the workload on the decisive core $DBF(\Delta)$, is the sum of the current demand bound $CDBF(\Delta)$ and the future demand bound $FDBF(\Delta)$, as presented in Equation 2.8.

$$DBF(\Delta) = CDBF(\Delta) + FDBF(\Delta) \tag{2.8}$$

As the core of interest is the decisive core, whose workload defines at which speed the processor will execute, the demand bound functions for the remaining cores is of no interest for the implementation of the presented scheme, and all references to the DBF will referr to the demand bound function for the decisive core.

Noteworthy is that all variables and function values to create the history aware demand bound functions can be calculated at runtime as long as the parameters for the α curves, δ and N, are given at design time.

Chapter 3

The Worst-Case Ready Queue for EDF scheduling and speed assignments

In combination with the algorithms to operate on it, the data structure that enables the safe assignment of speeds to current and future jobs is the Worst-Case Ready Queue (WCRQ). In this data structure, all information about current and future jobs is stored, and these are then scheduled by the system using the common Earliest Deadline First (EDF) dynamic priority scheduling policy.

3.1 The Worst-Case Ready Queue

Traditionally, the scheduling of jobs have been performed via the concept of a Ready Queue (RQ), in which all jobs that are currently ready to be executed are represented. In the case of EDF scheduling, this queue is sorted by the deadline of the tasks. The entries in the RQ are represented by a tuple (id, c, d), where id identifies the task to which the job belongs, c is the residual execution time at s^{max} and d is the absolute deadline of the job.

Using the history aware prediction of future job arrivals, it is possible to create a similar queue for all jobs that have not arrived yet, the so called *potential* jobs. This queue can then be thought of as the potential ready queue (PRQ). To contrast the jobs in this queue from the jobs that have already arrived, the jobs in the RQ are referred to as *real* jobs.

By selecting a *horizon* Δ for which to populate the PRQ, the PRQ will contain all jobs that will arrive in the interval $[now, now + \Delta]$ by creating a job entry for each job returned by Equation 2.5 for each task. These job entries represent future arrivals of real jobs, and as such will be on the same form as the entries of the RQ.

The model used allows for non-determinism regarding task activations, that is, the jobs are not guaranteed to arrive in the order assumed in the PRQ. As the jobs are scheduled using EDF, a job j_a that is delayed might be forced to execute later than assigned within the PRQ. Since the absolute deadlines of the jobs are assigned in accordance with their arrival times, this means that another

job j_b may now have an earlier deadline. Because j_a would originally have been executed before j_b, any speed assignments that would have been sufficient when j_a executes before j_b are not necessarily correct if the jobs arrive in opposite order. To take this into account, the entries of the PRQ are expanded to include one last parameter, max, which is the maximum number of pre-emptions that may cause a job to be delayed. This gives the entries of the PRQ the form (id, c, d, max).

As the relative deadline D_i of the tasks is known, all task activations that occur within $now + D_i$ have the worst case release time, and thus the worst case deadline, at the current point in time. Likewise, any potential task activation that occurs later than $now + D_i$ has $now + D_i$ as worst case release time, as it can not appear before that time. This means that the worst case release time of any potential job j of task τ_i is $now + n \times D_i$, where n is the smallest number so that $n \times D_i \leq \Delta$. For future reference, this is referred to as the deadline period of the virtual jobs.

In addition, since these jobs are purely potential and have had no chance to execute, it is clear that the residual execution time c is the worst case execution time C_τ. This means that every potential job within the PRQ is given with the worst case scenario values.

As both the real and potential jobs are on similar form, they can be moved from their respective data structures to a new, shared structure. This structure is the Worst Case Ready Queue (WCRQ) and it removes the need for both the RQ and PRQ, as they are now subsets of the new structure. The reason it is Worst Case is that, as has already been presented, the PRQ contains the worst case parameters for every future job, and the RQ itself uses the worst case execution time as base for the residual execution time. To differentiate between the real jobs from the RQ and the potential jobs from the PRQ, a *type* parameter must be added to the queue entries for book keeping purposes. The type is either *real* for real jobs, or *virtual* for potential jobs.

Finally, to allow speed assignments to each job, a s parameter is included in the WCRQ, which represents the speed, either s^{th} or s^{max}, at which the job should be executed. This means that the final form of the WCRQ entries is $(id, c, d, s, type, max)$.

At runtime, the WCRQ is updated before the scheduler is executed, as to give the scheduler the most up to date information about the state of the system. At this point, the arrivals of new real jobs will replace the virtual job to which it corresponds. The virtual jobs are added to the WCRQ by executing the steps presented in Algorithm 3.1. A visual representation of the workings of the WCRQ is presented in Figure 3.1.

To be able to safely calculate the speeds, it is neccesary for the WCRQ to be large enough to hold all relevant jobs. As the long term task activation rates for each task τ_i is defined by ρ_i, the global task activations rate ρ is defined by the highest individual rate, that is, $\rho = max_{\forall \tau_i \in \tau}(\rho_i)$. It can be assumed that $s^{th} \geq \rho$, that is, that the long-term rate of task activations is lower than the thermal stable speed s^{th}. This is reasonable, due to the concave shape of the activation bound, as the slope becomes lower as Δ increases, see again, Figure 2.4.

Under this assumption, there exists a maximum number of jobs which are relevant to assign safe speeds to all jobs, given by $\sup_{\Delta \geq 0} \left\{ \alpha_i(\Delta) - s^{th}\Delta \right\}$ [15]. This expression gives the maximum number of jobs that may still be pending at

Algorithm 3.1 The ComputeWCRQ algorithm which adds new virtual jobs to the WCRQ up until *horizon*.

Require: Time elapsed since renewal point T
1: **function** ComputeWCRQ($WCRQ$, $horizon$)
2: ▷ Loop over all task sets
3: **for** $i = 1$ **to** $i = N$ **do**
4: $x = 0$
5: $sum = 0$
6: $s = \infty$
7: **repeat**
8: ▷ Number of newly expiring jobs at $k \times D_i$
9: $new = U_i(x \times D_i) - sum$
10: $d = (x + 1) \times D_i$
11: **for** $j = |WCRQ|$ **to** $new + |WCRQ|$ **do**
12: $WCRQ \cup= (i, C_i, d, s, v, sum\ ++)$
13: **end for**
14: $x\ ++$
15: **until** $x \times D_i \geq horizon$
16: **end for**
17: **end function**

any point in time if the system is executed at the TSS. By making the WCRQ large enough to store these jobs, the scheme is able to predict this worst case behaviour, and thereby assign safe speeds at all times. The Δ which satisfies the above expression is therefore the *horizon* used within the presented algorithms.

3.2 Speed assignments

The WCRQ contains all the information needed to safely assign speeds to each task within the queue. Assuming that the horizon has been chosen large enough, as outlined in the previous section, the scheme only needs to consider the deadlines of the jobs within the WCRQ. The WCRQ is populated by Algorithm 3.1 which is executed before each execution of the speed assignment algorithm. Furthermore, the speed assignment algorithm is only executed if the next job in the WCRQ does not yet have a speed assignment, to prevent uneccesary overhead. This is safe to do, as the speed assignments are done in such a way that the worst case future workload will meet its deadlines. Once the first job in the queue doesn't have a speed assignment, safe assignments will be done for the whole queue, including the virtual jobs, which are the worst case representations of the future job arrivals. Once the speed assignments have been made, the next job, from an EDF perspective, is chosen to be executed. The full operation of the scheduler is presented in Algorithm 3.2.

The algorithm for assigning the speeds is presented in Algorithm 3.3, and in a simplified form contains the following steps:

1. Iterate over the WCRQ, and for every job e:
   (a) Set the speed of the job to s^{max}.
   (b) Check if the job will meet its deadline at s^{th}. If it will, lower the speed of the job to s^{th}, update the $CTime$ (required computation time), and **continue** to the next job entry in the queue. This is done in lines 6-11.

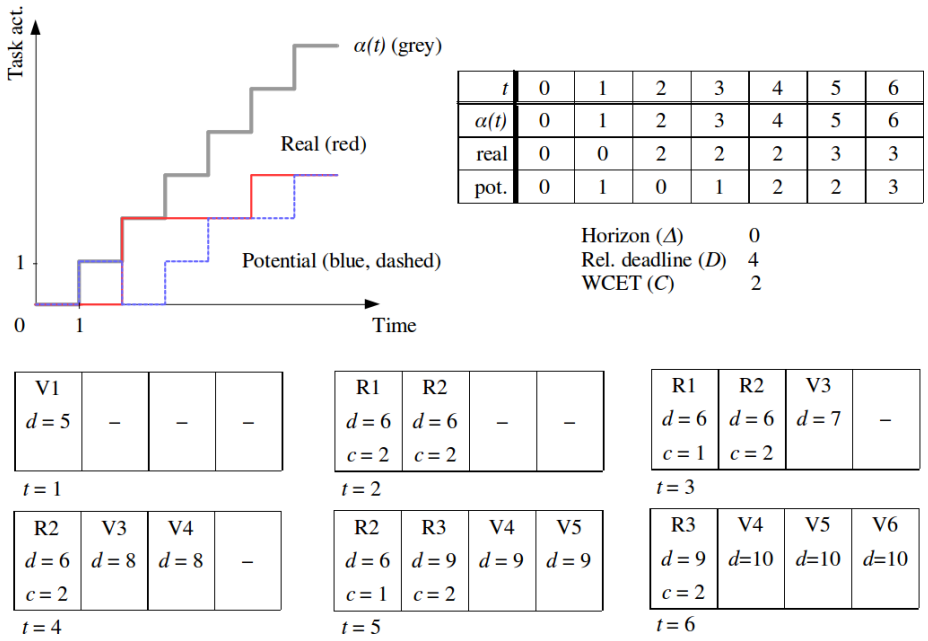


Figure 3.1: A simplified visual representation of the operation of the WCRQ in which there is only a single task and the horizon is set to zero. The plot shows the alpha curve (upper bound on task activations, cummulative), the actual (real) task activations (cummulative) and the potential task activations U (non-cummulative). The table shows the values of each of the curves at each point in time. Because the horizon (Δ) is zero, the potential jobs given by U is only dependent on the dynamic counter DNC, which gives the number of potential jobs that might appear *now*. In the bottom half, the contents of the WCRQ are given at each point in time. It may be interesting to note the replacement of virtual jobs V with real jobs R, as it shows how the deadlines of the virtual jobs are more pessimistic than those of the real jobs once they occur.

(c) Check if the job will meet its deadline at s^{max}. If it will, update the $CTime$, and **continue** to the next entry in the queue. This is done in lines 12-16.

(d) If the algorithm continues this far, the job e will not meet its deadline at s^{max}. This means that the earlier jobs must execute at higher speeds to create enough slack for this job to finish on time. At this point we know that the current job e will have to execute at the maximum speed, so update the $CTime$ accordingly. This is done in lines 17-18.

(e) Iterate backwards in the queue from job e, and set the speed of each previous job n.

i. If job e will still not meet its deadline, change the speed of earlier job n to s^{max} and update $CTime$ accordingly. This is done in lines 22-27.

ii. If the job e is virtual, or n is either a real job, or a virtual job

Algorithm 3.2 The SCHEDULER algorithm, which handles book keeping for pre-empted jobs, updates the WCRQ and then makes a scheduling decision.

```
 1: function SCHEDULER(WCRQ, signal s, job r, job n)
 2:     if s == job_release then
 3:         ▷ Update the residual execution time of the pre-empted job.
 4:         r.c = r.c − (now − t_start)
 5:         ▷ Set parameters for newly released job and replace virtual job in WCRQ
 6:         n.c = C_n.id
 7:         n.d = now + D_n.id
 8:         n.s = ∞
 9:         insertRJ(WCRQ, n)
10:     end if
11:     if s == job_finished then
12:         removeRJ(WCRQ, r)
13:     end if
14:     j = peekRJ(WCRQ)
15:     if j.s == ∞ then
16:         ComputeWCRQ(WCRQ, horizon)
17:         SpeedAssignments(WCRQ)
18:     end if
19:     t_start = now
20:     setProcessorSpeed(j.s)
21:     executeJob(j);
22: end function
```

that has already been reviewed (is in V), it is safe to **continue** to the next earlier job, as the job cannot be preempted in such a way that it might miss its deadline.

iii. If the algorithm continues this far, ensure that the virtual job n will meet its deadline even if it is preempted. Once this is done, add the job to the list of previosly reviewed virtual jobs V. This is done in lines 32-37.

It is clear that this will create safe speed assignments for the jobs within the WCRQ, because as pointed out in Section 3.1, all jobs, both real and virtual, are represented as their respective worst case situation. This means that the virtual jobs are represented with their deadlines as if they arrived at the earliest possible point in time, in every deadline period (as shown earlier), which in turn means that they will have the most possible overlap with the real jobs executing on the processor. This in turn means that the real jobs are assumed to remove the most possible processing time from the virtual jobs, meaning that the virtual jobs are assumed to have the least possible amount of slack time to complete their processing in. In addition to this, the WCRQ includes the maximum possible number of virtual jobs that can appear, and how many times they can be preempted, and as such, all operations on the WCRQ are done from a *worst case* perspective.

Should a virtual job v appear later than assumed, this pushes the deadline d_v of the job further into the future, in effect giving the currently executing real job r more processor time. This means that the speed assignment with virtual jobs assumed to appear *now* gives the tightest bound possible, and therefore safe speed assignments, as d_v is pushed forward with the delay of v's arrival. In the case in which v is delayed so much that its deadline d_v lands after the deadline of the real job d_r, the virtual job will not even pre-empt r, as EDF

Algorithm 3.3 The SpeedAssignments algorithm which iterates through the WCRQ, assigning speeds to all jobs in order to meet their deadlines.

```
1: function SpeedAssignments(WCRQ)
2:     for CTime = now, i = 1 to i = |WCRQ| do

3:         ▷ Look at the next job, and set its speed to max
4:         e = peek(WCRQ, i)
5:         e.s = s^max

6:         ▷ If job meets deadline at TSS, set speed to TSS and continue
7:         if 1/s^th × c_e.id ≤ d_e.id − CTime then
8:             e.s = s^th
9:             CTime += 1/e.s × c_e.id
10:            continue
11:        end if

12:        ▷ TSS not enough, if MAX enough, continue to next job
13:        if 1/s^max × c_e.id ≤ d_e.id − CTime then
14:            CTime += 1/e.s × c_e.id
15:            continue
16:        end if

17:        ▷ Job will miss deadline at MAX
18:        CTime += 1/e.s × c_e.id

19:        ▷ Iterate backwards through WCRQ
20:        for V = ∅, j = i − 1 to j = 1 do

21:            n = peek(WCRQ, j)

22:            ▷ Meet deadline by speeding up earlier job
23:            if CTime > e.d then
24:                ▷ Change the speed to MAX and update CTime accordingly
25:                CTime −= c_n.id × (1/n.s − 1)
26:                n.s = s^max
27:            end if

28:            if e.type == v ∨ n.type == r ∨ n ∈ V then
29:                continue
30:            end if

31:            ▷ Check if TSS is safe for virtual job n
32:            if n.max × c_n.id + c_e.id × 1/e.s > D_e.id then
33:                CTime −= c_n.id × (1/n.s − 1)
34:                n.s = s^max
35:            end if

36:            V ∪= n.id

37:        end for
38:    end for
39: end function
```

scheduling is applied. This increases the processing time for r, while it decreases the overlap of v with r, both leading to better cases than the assumed worst case for which the speed assignments were made.

Chapter 4

Evaluation

The evaluation of the scheme is done in regard to how much downtime the system produces for the secondary cores, as compared to an ideal execution of the scheme. Ideal referrs to the scheme having complete access to information about all future event arrivals, and as such, can determine exactly when, and for how long, it needs to execute at the maximum speed.

4.1 System setup

To evaluate the scheme, the scheme was implemented as a C program executing on the gem5 [4] simulator. The gem5 simulator is a hardware simulator capable of simulating several different platforms. It is primarily intended for architecture research. The simulator can easily be configured via its Python based object oriented configuration files, which makes it easy to add new devices.

For the evaluation to be repeatable it is necessary to be able to execute the same test again. To do this, the system was executed with predefined traces as generated in accordance with the PJD model by a MATLAB script. How to express a PJD modelled workload using α curves is explained in [13]. The trace files are then read by a custom designed hardware device attached to the simulated system. This device is the SchedHelper device.

The SchedHelper device provides a set of memory mapped registers, from which the software can read events. At system startup, before the first write to a status register of the device, the registers contain the required values for setting up the scheduler. This includes the N and δ curves defining the bounding staircase curves for task activations. In this case, the evaluation is done with only a single task defined by two staircase curves. As such, the values presented are the N_0, N_1, δ_0 and δ_1. The use of more than a single task could have been implemented, but would not provide any more information regarding the downtime of the secondary cores, which is the main point of interest.

Once a write has occured to the *status* register of the SchedHelper device, the SchedHelper starts to read from the trace file, and update the memory mapped registers with information about the arrival of tasks. The scheduler executing in software can thus read the arrival time arr and execution time ex of the job to be created.

This is where one of the differences between executing the presented scheme,

or the idealised scheme to compare with appears. The former is referred to as an *online* execution of the scheme, and the latter an *offline* execution. In the online scheme, the actual execution time ex is only used to emulate the execution of the task, and is never presented to the algorithms that make up the scheme, instead, the worst-case execution time $wcet$ is used. In contrast, when performing an offline execution, the actual execution time ex is used within the scheme.

Lastly, the SchedHelper device presents two additional memory mapped registers used exclusively by the offline execution. These two memory mapped registers are used to iterate through the trace ahead of time. Using this, the offline scheme can place the actual jobs (that will arrive in the future) in the WCRQ, instead of placing virtual placeholder jobs. This means that the offline execution will in effect not have a WCRQ, but a complete list of all jobs (current and future) to be executed during the lifetime of the system. The same speed assignment algorithm is executed on the scheme, but now with perfect information about the current and future workload.

An overview of how the trace file is presented to the scheduler is shown in Figure 4.1.

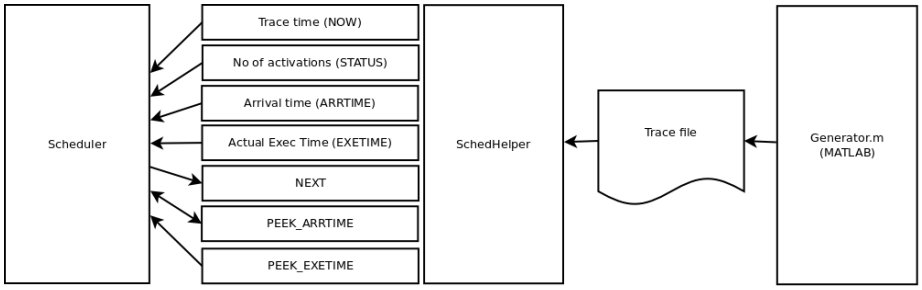


Figure 4.1: An overview of how the trace data is presented to the scheduler executing within the gem5 scheduler. The Generator.m MATLAB script generates a task activation trace which is loaded by the SchedHelper device into memory mapped registers which are then read by the scheduler.

The MATLAB script writes the generated trace into a trace file, which contains some meta data about the trace itself, followed by a listing of the arrival times and execution times of each task activation, as shown below.

$\#\ p\ =\ PPP\ j\ =\ JJJ\ d\ =\ DDD$
$SchedHelper1\ deadline\ wcet\ d\_0\ d\_1\ N\_0\ N\_1$
$activation\_time\ actual\_execution\_time$
$activation\_time\ actual\_execution\_time$
$\ldots$
$activation\_time\ actual\_execution\_time$

The first line in the trace file does not contain any actual information used by the SchedHelper device when reading the trace, instead, it just presents the PJD parameters used when creating the trace, to ease future identification of the trace. The second line contains the trace file version, to enable future expansion of the file format while easily keeping backwards compatibility. Following this

is the metadata used by the scheme, namely the relative deadline D_i and the worst case execution time C_i of the task, and the parameters of a two curve bounding function α for the task activations. Following this, starting with the third row, the actual task activations are listed in order of activation time. To remove unneeded complexity, this format only supports a single task whose α curve is comprised of two curves, as required for the evaluation of the presented scheme.

While the gem5 simulator operates in ticks with a resolution of $1ps$, the resolution of the trace file is lowered to $1ms$. This lowers the amount of bits required for the memory mapped registers to express the current time, while the resolution is still high enough to produce reasonable results. As a result of this, all values within the trace files are expressed in milliseconds. Furthermore, as all times are normalised to the maximum processor speed, as explained in Section 2.3, the trace files are also expressed normed to the maximum processor speed, meaning that the trace file itself does not need to contain this information.

In addition to the handling of the trace file the SchedHelper device monitors the gem5 DVFS/EnergyCtrl subsystem to identify at which speed the system is currently running. Depending on the current clock frequency and the amount of time it has executed at that speed, the SchedHelper device updates its internal representation of the thermal model, and shuts down the secondary cores when the thermal model demands it. All of these events are logged to the gem5 log files, which can thereafter be parsed to identify at which points in time the secondary cores were powered down. The thermal model is implemented as described in Section 2.2.

Included with the gem5 simulator is a configuration file for a RealView Versatile Express EMM ARM board, which is used as base for the hardware setup. The only changes done to the configuration is to include the SchedHelper device in the hardware configuration, and configure the required speed settings for the DVFS subsystem. All other implementation details are performed in the C program implemented to execute the presented scheme. As, at the point of implementation, DVFS was not supported for other platforms than the ARM, and was only supported in Full System (FS) mode, the software is executed as a bare metal approach, and thus executes directly on the hardware without an intermediate operating system. This means that the software has full access to the hardware, and does not risk being preempted or otherwise stalled by an operating system. Furthermore, at the start of the system, all required hardware is set up by the C program, including the UART, DVFS and SchedHelper subsystems. When setting up the DVFS subsystem, the program automatically calculates the values of s^{th} and s^{max}, under the assumption that the processor is configured to support two speeds only, and the lower speed is configured to provide the intended s^{th}.

An overview of the full gem5 system setup is presented in Figure 4.2.

4.2 Execution

The scheme is evaluated in both online and offline modes for two different traces. The first trace *max* is a trace in which the event arrivals matches the upper bound curve α, and the second trace *var* is a trace in which the event arrivals vary more, that is, the arrivals occur somewhere within the upper bound as

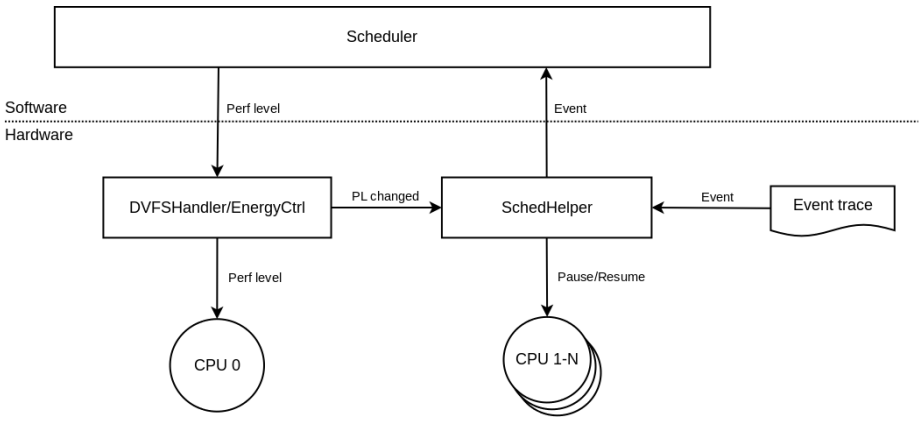


Figure 4.2: A schematic overview of the interaction between the components in the evaluation environment.

given by α, and a lower bound α^{lower}, of which the scheme is not aware. In both cases, the traces are executed once for $8s$ and once for $32s$, to capture both the initial and long-term behaviour of the traces. In both cases the trace contains only a single task, as the system is evaluated with one single task only, to decrease the complexity of the code. Note that the traces are not completely identical, so the overlapping interval between the $8s$ and $32s$ are not from the (exact) same trace.

In both cases, the PJD parameters given to the MATLAB script for generating the trace files were $p = 220ms$, $j = 388ms$ and $d = 48ms$. In addition to this, the timing requirements for the task were $D = 1250ms$ and $C = 150ms$. As explained previously, all values are normed to the maximum speed s^{max}. The thermal stable speed s^{th} of the system is configured as $0.5 \times s^{max}$, while the feasible speed s^{fs} of the workload is $0.68 \times s^{max}$, thus fulfilling the initial assumption that $s^{th} < s^{fs} < s^{max}$.

The thermal model permits the decisive core to execute at the maximum speed for $50ms$ before the secondary cores are powered down. Once the processor returns to the thermal stable speed, the secondary cores are powered up again after $100ms$, thereby assuming that the cooldown takes twice the time of the heatup. As is presented in Section 2.2, the switching of states within the thermal model is based on a counter, and as such, the cooldown period might be shorter if the processor ran at the maximum speed for a shorter period of time, thereby aborting the countup before the maximum value.

4.3 Results

The execution of each of the *max* and *var* traces is presented in Figures 4.4 and 4.5 respectively. Furthermore, a comparison between the amount of time the secondary curves are turned off between each trace is presented in Figure 4.3

From the *max*8 and *max*32 traces, in Figure 4.4a and 4.4b respectively, it is clear that the online scheme speeds up above the offline reference at the beginning of the trace. In this case, this more pessimistic speed assignment ends after about $2s$, after which it continues more parallel with the offline curve, as

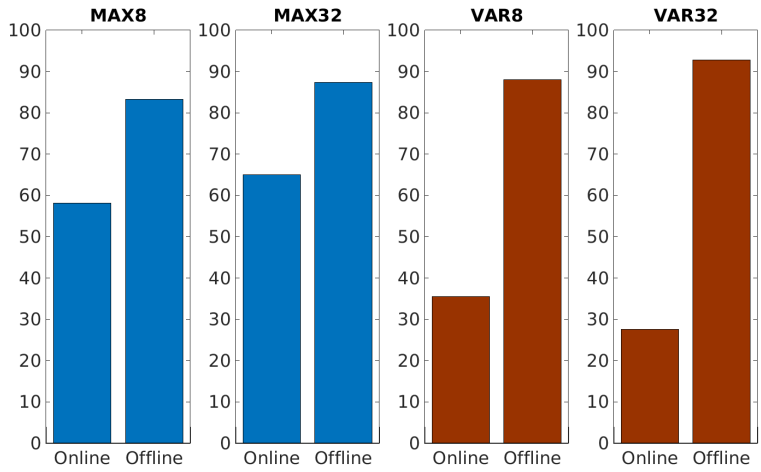
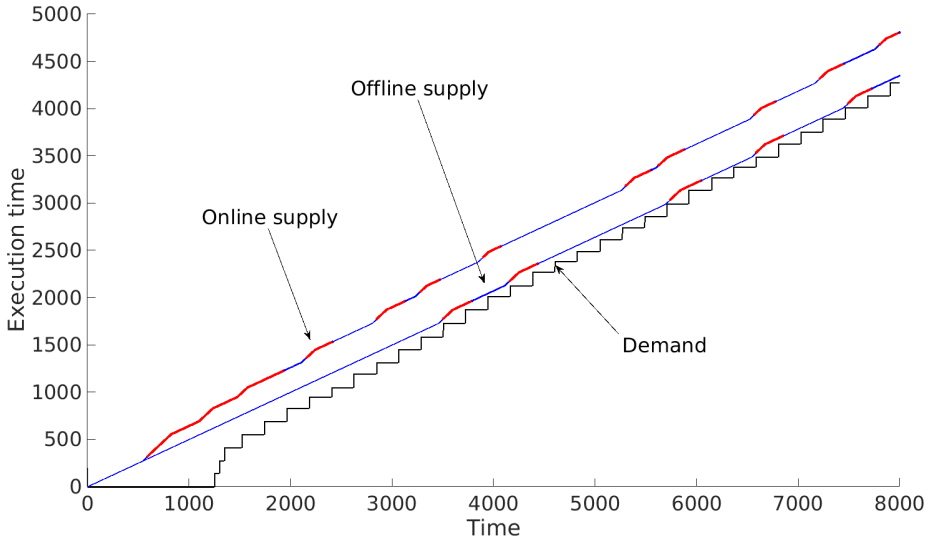


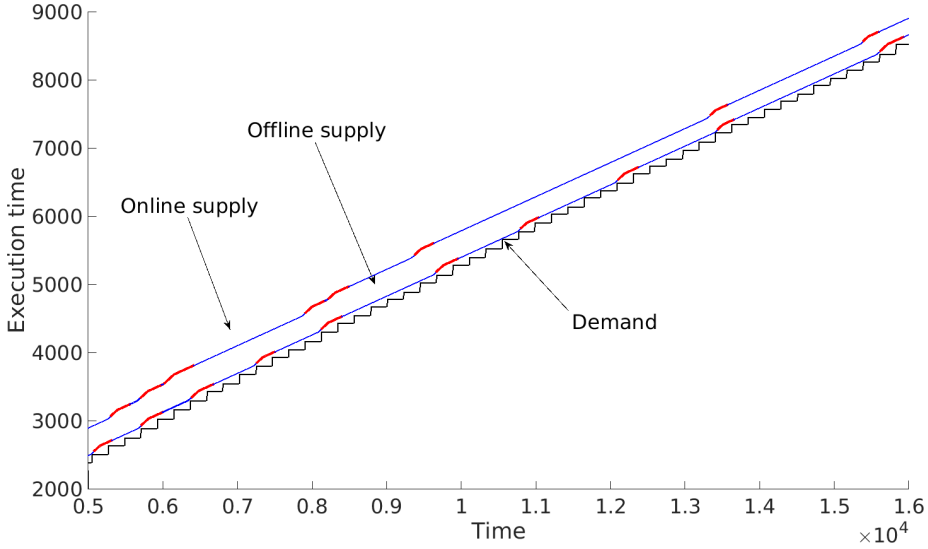
Figure 4.3: Comparison of secondary core uptime, in percent of total uptime, between online and offline execution of each trace. The online run of the system uses the implemented scheme. In contrast, the offline run is executed with complete knowledge about all future events in the system, and is optimal with respect to speed changes, which in turn decides when the secondary cores are paused and resumed.

can be seen in the $max32$ curve, which shows a similar execution on the interval $[5s, 16s]$. Once the initial burst is over, the speedups do not appear at the exact same positions in time, instead, the online curve, which has to operate on the worst case assumption, speeds up somewhat more often than the optimal offline curve. This leads to a somewhat higher percentage of time the secondary cores are turned off, as illustrated in Figure 4.3. From the same figure, it is also clear that the initial burst of the online curve becomes more and more negligible with respect to the offline time, as the secondary core uptime increases towards the offline curve once the trace gets longer.

Similarily, in the $var8$ and $var32$ traces, in Figure 4.5a and 4.5b respectively, it is shown that the initial pessimism of the online curve appears here as well. However, in this case, it is not stabilized after this, instead, it may continue to execute at higher speeds. Most notably these occurences occur more often once the number of task activations decrease further away from the task activation bound. It can be seen that the curves seem to eventually become more parallel to the offline curve, both in the $var8$ and $var32$ curves. However, as the pessimistic speed assignments when task activations are overdue continue, the downtime introduced onto the secondary cores does not become more and more negligable, as can be seen in Figure 4.3. Instead, the uptimes of the secondary cores continue to drop once the traces get longer.

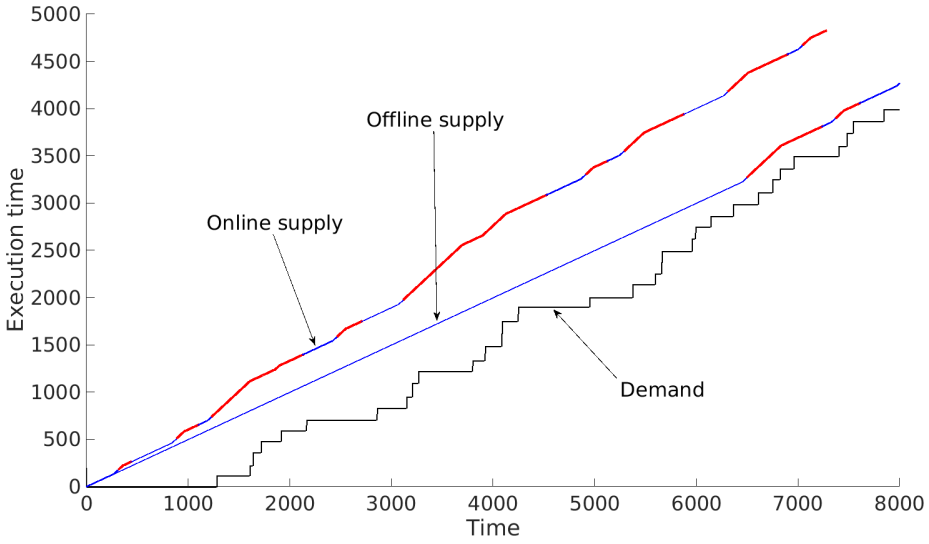


(a) max8

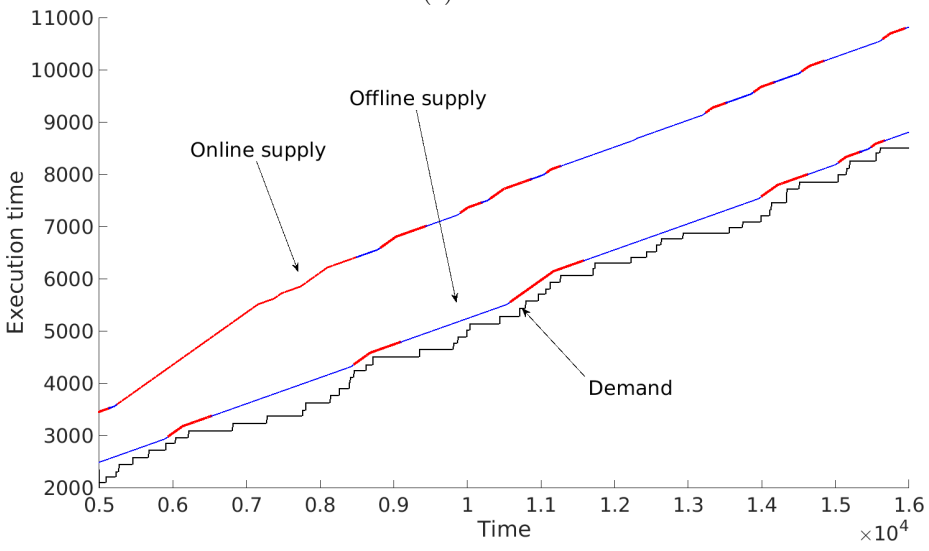


(b) max32

Figure 4.4: Demand, and online and offline supply for the *max* traces.



(a) var8



(b) var32

Figure 4.5: Demand, and online and offline supply for the *var* traces.

Chapter 5

Discussion

When executing both the *max* and the *var* traces, the initial burst gives rise to an increase in the clock frequency of the processor. The offline scheme has access to information about all future task activations, and can easily calculate that the current speed (TSS) is high enough to execute the burst in a timing correct way. The online execution, however, is not aware of when the actual task activations will occur, but does instead rely on the virtual jobs in the WCRQ to determine the correct speed.

The virtual jobs are added to the WCRQ as a result of Algorithm 3.1, which in turn depends upon the dynamic counter, which operate on the task activation bound curve. All virtual jobs in the queue that occur before $now + D$ are assumed to appear *now*, and thereby they all have a deadline at the same point in time at $now + D$, much like the case at $t = 6$ in Figure 3.1. Likewise, all virtual jobs with arrival time after $now + D$ are assumed to appear at the beginning of their respective deadline period. Because of this, the WCRQ does indeed reflect the *worst case* scenario. When there is a large number of virtual jobs in the WCRQ, the clock frequency must be increased to ensure that the timing correctness is not violated even in this worst case scenario.

Since the virtual jobs represent the worst case scenario, once the virtual jobs of the WCRQ are replaced with real jobs, the pessimism of the WCRQ decreases. As the number of virtual jobs decrease, the speed assignment algorithm can work on more real data instead of worst case assumptions, and the speed assignments become more optimistic. This can be seen after the initial execution at the maximum speed, when the speed returns to the thermally stable speed.

Similarily, in the *var* traces, the absense of task activations compared to the demand bound curve, increases the number of virtual jobs within the queue. Once again, this increases the pessimism of the speed assignments, as the worst case scenario is once again assumed. When all the delayed task activations occur, the number of virtual jobs within the WCRQ is quickly reduced, and once again the speed assignments are more optimistic.

As presented by [7] the task activation bound can not only be upper bounded, but also lower bounded using the same dynamic counter approach. This could help reduce the pessimism of delayed jobs, as the lower bound will give a theoretical last point in time when the job will arrive. This is left as future work.

Chapter 6

Conclusions

This thesis work has presented an online scheme for safely assigning speeds to hard real-time workloads on a system suffering from DVFS/DTM induced dark silicon, in which the system has to shut down a set of cores to be able to execute the remaining cores at a higher temperature.

The scheme has been implemented as a C program and executed in the gem5 simulator software. In addition to the scheme implemented as software within the simulator, a simulated hardware device has been implemented to allow predefined task activation traces to be executed on the scheme.

The scheme is, as far as known to the author, the first such scheme to be implemented and as such no comparisons to existing schemes can be made, however, in comparison to the optimal solution, the scheme performs on-par when the workload behaves closely to the worst-case predictions made from the workload model. In the cases when the actual behaviour of the workload differs significantly from the theoretical model, the scheme is still able to do predictions, but with an increased pessimism, that is, the DTM induced downtime increases. The use of dynamic counters to make the required predictions about the future behaviour of the workload means the scheme has a low overhead and can be used online applications with bounded processor and memory footprints.

In [7] the authors use dynamic counters for predicting both the upper and lower bound of the future workload. In the presented scheme, only the upper bound is used to define the behaviour of the future workload. By also implementing dynamic counters for predicting the lower bound, it might be possible to reduce the pessimism introduced to the system when the actual workload deviates from the theoretical upper bound.

Bibliography

[1] Ahmed, R., Ramanathan, P., and Saluja, K. Temperature minimization using power redistribution in embedded systems. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on* (Jan 2014), pp. 264–269.

[2] Albers, S. Algorithms for dynamic speed scaling. In *STACS* (2011), vol. 11, pp. 1–11.

[3] Benini, L., Bogliolo, A., and De Micheli, G. A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 8*, 3 (June 2000), 299–316.

[4] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[5] Esmaeilzadeh, H., Blem, E., St.Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on* (June 2011), pp. 365–376.

[6] Kumar, P., and Thiele, L. Timing analysis on a processor with temperature-controlled speed scaling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th* (April 2012), pp. 77–86.

[7] Lampka, K., Huang, K., and Chen, J.-J. Dynamic counters and the efficient and effective online power management of embedded real-time systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (New York, NY, USA, 2011), CODES+ISSS '11, ACM, pp. 267–276.

[8] Le Sueur, E., and Heiser, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems* (2010), USENIX Association, pp. 1–8.

[9] Maxiaguine, A., Chakraborty, S., and Thiele, L. Dvs for buffer-constrained architectures with predictable qos-energy tradeoffs. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (New York, NY, USA, 2005), CODES+ISSS '05, ACM, pp. 111–116.

[10] Neukirchner, M., Michaels, T., Axer, P., Quinton, S., and Ernst, R. Monitoring arbitrary activation patterns in real-time systems. In *Proceedings of the 2012 IEEE 33rd Real-Time Systems Symposium* (Washington, DC, USA, 2012), RTSS '12, IEEE Computer Society, pp. 293–302.

[11] Pagani, S., Khdr, H., Munawar, W., Chen, J.-J., Shafique, M., Li, M., and Henkel, J. Tsp: Thermal safe power: Efficient power budgeting for many-core systems in dark silicon. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis* (New York, NY, USA, 2014), CODES '14, ACM, pp. 10:1–10:10.

[12] Perathoner, S., Lampka, K., Stoimenov, N., Thiele, L., and Chen, J.-J. Combining optimistic and pessimistic dvs scheduling: An adaptive scheme and analysis. In *Proceedings of the International Conference on Computer-Aided Design* (Piscataway, NJ, USA, 2010), ICCAD '10, IEEE Press, pp. 131–138.

[13] Perathoner, S., Lampka, K., and Thiele, L. Composing heterogeneous components for system-wide performance analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011* (March 2011), pp. 1–6.

[14] Schor, L., Bacivarov, I., Yang, H., and Thiele, L. Worst-case temperature guarantees for real-time applications on multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th* (April 2012), pp. 87–96.

[15] Thiele, L., Chakraborty, S., and Naedele, M. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on* (2000), vol. 4, pp. 101–104 vol.4.

[16] Wrege, D., and Liebherr, J. Video traffic characterization for multimedia networks with a deterministic service. In *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE* (Mar 1996), vol. 2, pp. 537–544 vol.2.