

Diss. ETH No. 21755

# Hard Real-Time Guarantees in Cyber-Physical Systems

A thesis submitted to attain the degree of  
Doctor of Sciences of ETH Zurich  
(Dr. sc. ETH Zurich)

presented by  
PRATYUSH KUMAR  
Master of Technology,  
Indian Institute of Technology Bombay  
born May 4, 1987  
citizen of India

accepted on the recommendation of  
Prof. Dr. Lothar Thiele, examiner  
Prof. Dr. Sanjoy Baruah, co-examiner

2014





Institut für Technische Informatik und Kommunikationsnetze  
Computer Engineering and Networks Laboratory

---

TIK-SCHRIFTENREIHE NR. 143

Pratyush Kumar

# Hard Real-Time Guarantees in Cyber-Physical Systems



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

A dissertation submitted to  
ETH Zurich  
for the degree of Doctor of Sciences

Diss. ETH No. 21755

Prof. Dr. Lothar Thiele, examiner  
Prof. Dr. Sanjoy Baruah, co-examiner

Examination date: January 20, 2014

ISBN 978-3-906031-47-7

DOI [10.3929/ethz-a-010068802](https://doi.org/10.3929/ethz-a-010068802)

*To Maa,*

गर्भज्जन्मं स्तनाद्गुधं जङ्घात्स्वप्नं गलाद्रसं ।  
सर्वापि प्राप्य मातुः मे दक्षिणाय दरिद्रता ॥



# Abstract

By integrating components for sensing, communicating, computing and actuating, Cyber-Physical Systems (CPSs) enable software applications to monitor and control events in the physical world. It is widely anticipated that CPSs will become pervasive in personal and industrial applications.

As deployed CPSs will impact safety of humans and infrastructure, certifying their correctness is imperative. For an important class of systems, correctness requires guaranteed timing properties. For instance, in an automatic stability program of an automobile, the worst-case end-to-end delay between sensing and actuating could be upper-bounded.

Analysis of such hard real-time guarantees in CPSs is inherently challenging, because the timing models exhibit variability due to multiple reasons. Firstly, as CPSs are distributed and heterogeneous, events do not arrive periodically. Secondly, on modern processors, resource availability can be non-uniform due to physical effects such as overheating or low energy supply. Thirdly, timing models can be uncertain either due to incorrect calibration or simultaneous analysis of multiple designs. Finally, due to complex components in such CPSs, such as caches, rare and transient phenomena can result in deviation from nominal timing models.

In three parts of the thesis, we present three templates of solutions to compute hard real-time guarantees in the presence of the said variability.

- Variability in arrival patterns of events can be absorbed by a run-time manager which monitors and adapts to incoming events. We illustrate this by compositionally building demand bound servers and cool-shapers from efficient constituent units.
- Variability in timing models can be bounded with analysis of sound abstractions which compactly represent the timing-critical traces of the system. We illustrate this with the analysis of temperature-controlled speed-scaling and the analysis of multiple designs within an Satisfiability Modulo Theory (SMT) solver.
- Variability due to rare and transient phenomena can be exported through richer guarantees to verify cross-layer objectives such as stability of a plant in a networked control system. We illustrate this by proposing and computing settling-time and overshoot metrics.





# Zusammenfassung

Durch die Integration von Sensoren, Aktoren, Kommunikationsmodulen und Berechnungseinheiten, ermöglichen Cyber-physische Systeme Softwareanwendungen für die Überwachung und Steuerung von Ereignissen der physischen Welt. Es wird allgemein erwartet, dass in Zukunft Cyber-physische Systeme in personenbezogenen und industriellen Anwendungen allgegenwärtig sein werden.

Da Cyber-physische Systeme die Sicherheit von Mensch und Infrastruktur beeinflussen werden, ist die Bescheinigung ihrer Korrektheit zwingend notwendig. Für eine wichtige Klasse von Systemen erfordert Korrektheit garantierte Zeiteigenschaften. Beispielsweise könnte man so für das automatisierte Stabilitätsprogramm eines Autos, eine obere Grenze für die Worstcase Verzögerung zwischen Sensor und Aktor angeben.

Die Analyse solcher harten Echtzeitgarantien in Cyber-physischen Systemen ist von Natur aus eine Herausforderung, da Timing-Modelle wegen unterschiedlichen Gründen Variabilität aufweisen.

Erstens, da Cyber-physische Systeme verteilt und heterogen sind, treten Ereignisse unregelmässig auf. Zweitens, auf modernen Prozessoren kann die Verfügbarkeit von Ressourcen, aufgrund von physikalischen Effekten wie Überhitzung oder niedrige Energieversorgung, ungleichförmig sein. Drittens, Timing-Modelle können Unsicherheiten aufweisen, welche entweder durch eine falsche Kalibrierung oder die gleichzeitige Analyse von mehreren Entwürfen entstehen. Schliesslich können seltene und vorübergehende Phänomene, wegen den komplexen Bauteilen in Cyber-physischen Systemen, wie zum Beispiel Zwischenspeichern, zu Abweichungen der Timing-Modelle führen.

In den drei Teilen dieser Arbeit präsentieren wir drei Lösungsvorlagen für die Berechnung harter Echtzeitgarantien im Beisein der oben genannten Variabilität.

- Die Variabilität der Ankunftszeiten von Ereignissen kann von einem Laufzeit-Manager absorbiert werden, der eingehende Ereignisse überwacht und sich entsprechend anpasst. Wir veranschaulichen dies durch den Aufbau eines Demand-Bound-Servers und Cool-

Shapers, welche aus einzelnen effizienten Bauteilen zusammengesetzt sind.

- Die Variabilität der Timing-Modelle kann durch die Analyse von Abstraktionen eingegrenzt werden, welche kompakt die zeitkritischen Abläufe des Systems darstellen. Wir veranschaulichen dies mit der Analyse von einer temperaturgesteuerten Geschwindigkeitsskalierung und der Analyse von mehreren Entwürfen mit einem Satisfiability Modulo Theory (SMT) Löser.
- Variabilität, die durch seltene und vorübergehende Phänomene entsteht, kann durch zusätzliche Garantien ausgelagert werden um so Cross-Layer-Ziele zu verifizieren, wie zum Beispiel die Stabilität einer Anlage in einem vernetzten Kontrollsystem. Wir veranschaulichen dies durch die Empfehlung und Berechnung von Einschwingzeiten und Übersteuerungsmetriken.

# Acknowledgements

I express my deep gratitude to *mata*, *pita*, *guru*, and *brahman*: *Mata* my mother for everything that there is, *pita* my father for being a source of inspiration, Prof. Lothar Thiele for being a *guru*, a guide, in the truest sense of the word, Prof. Sanjoy Baruah for reviewing the dissertation, and finally, the ultimate reality in *brahman*. I am also grateful to my brilliant colleagues, kind support-staff, and compassionate friends.

This work was financially supported by the EU FP7 project PRO3D under grant number 247846.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Real-Time Guarantees in Embedded Systems . . . . .	2
1.2	Additional Challenges in Cyber-Physical Systems . . . . .	2
1.3	Aim of the Thesis . . . . .	4
1.4	Thesis Outline and Contributions . . . . .	5
<b>I</b>	<b>Absorbing Variability with Run-Time Managers</b>	<b>9</b>
<b>2</b>	<b>Isolation of Tasks with Demand Bound Servers</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Demand Bound Server . . . . .	14
2.3	Shifted-Periodic Demand Bound Server . . . . .	17
2.4	Composition Operators for DBSs . . . . .	25
2.5	Summary . . . . .	28
<b>3</b>	<b>Shaping Real-Time Tasks to Minimize Peak-Temperature</b>	<b>33</b>
3.1	Introduction . . . . .	33
3.2	System Model . . . . .	36
3.3	Optimal Leaky-Bucket Shaper . . . . .	38
3.4	Extensions of Convex-Hull Shaper . . . . .	42
3.5	Experimental Results . . . . .	45
3.6	Summary . . . . .	49
<b>II</b>	<b>Bounding Variability in Formal Analysis</b>	<b>51</b>
<b>4</b>	<b>Analysis of Temperature-Based Feedback Control of Speed</b>	<b>53</b>
4.1	Introduction . . . . .	53
4.2	System Model . . . . .	56
4.3	Illustrating Example . . . . .	58
4.4	Analysis for Maximum Initial Temperature . . . . .	60
4.5	Analysis for Minimum Initial Temperature . . . . .	61
4.6	Analysis for Any Initial Temperature . . . . .	65
4.7	Faulty Temperature Sensors . . . . .	69
4.8	Experimental Results . . . . .	70

4.9	Summary . . . . .	73
<b>5</b>	<b>Satisfiability Modulo Real-Time Calculus</b>	<b>79</b>
5.1	Introduction . . . . .	79
5.2	System Model and Problem Definition . . . . .	83
5.3	Abstract Arrival and Service Curves . . . . .	85
5.4	Basics of a SMT Solver . . . . .	90
5.5	SMT Solver for Speed Assignment . . . . .	92
5.6	Experimental Results . . . . .	97
5.7	Summary . . . . .	101
<b>III</b>	<b>Exporting Variability through Richer Guarantees</b>	<b>105</b>
<b>6</b>	<b>The Settling-Time Metric</b>	<b>107</b>
6.1	Introduction . . . . .	107
6.2	Rare-Events with Settling-Time (REST) . . . . .	110
6.3	Analysis of the REST model for a Single Task . . . . .	114
6.4	Analysis of the REST model for Multiple Tasks . . . . .	119
6.5	Summary . . . . .	125
<b>7</b>	<b>Conclusions</b>	<b>131</b>
7.1	Major Findings . . . . .	131
7.2	Outlook . . . . .	133
<b>A</b>	<b>Real-Time Calculus</b>	<b>135</b>
A.1	Min-Plus/Max-Plus Algebra . . . . .	135
A.2	Arrival and Service Functions and Curves . . . . .	136
A.3	Workload Conserving Resource . . . . .	136
A.4	Greedy Processing Component . . . . .	137
	<b>Bibliography</b>	<b>139</b>
	<b>List of Publications</b>	<b>147</b>

# 1

## Introduction

Cyber-physical systems (CPSs) enable software applications to monitor and control events in the physical world. For instance, the embedded systems in a modern car can sense and affect physical parameters such as the driver's interaction with the car, the car's movement on the roads, and the health conditions within the car. Networking these systems enables integration at even larger scales, for instance several cars on a street may interact [YLVZ04]. Many anticipate that such systems will become pervasive in both personal and industrial applications [RLSS10, Lee08]. However, many scientific challenges remain.

As CPSs impact safety of humans and infrastructure, certifying their correctness is imperative [SLMR05]. To this end, governmental agencies have been set up for the avionics (e.g., European Aviation Safety Agency, U.S. Federal Aviation Authority) and the automotive (e.g., European Transport Safety Council, National Highway Traffic Safety Administration) domains. Among other interventions, these authorities publish standards, such as the ARINC-600 series [C+97] and the DO178B [Joh98], which specify the design guidelines and safety-levels for different classes of applications.

For an important class of systems, correctness requires guaranteed timeliness properties. An example is the Electronic Stability Control Program (ESP) deployed in modern cars [LMSN04]. The ESP constantly senses the car's parameters and actuates stabilizing corrections "to mitigate rollover accidents". Indeed, the time-delay between sensing and actuating affects the correctness of such a program. Other examples from the automotive domain include the automatic deployment of airbags and drive-by-wire steering control. Thus, computing certifiable timing

guarantees is an integral part of the design process of CPSs.

## 1.1 Real-Time Guarantees in Embedded Systems

In designing real-time embedded systems, timeliness properties have been studied since the 80's. An embedded system often refers to a resource-constrained computing device that offers a specific functionality, and is embedded into a larger system. In the following, we highlight some of the important strands of research in real-time embedded systems.

- *Static analysis of software:* The primary challenge in any real-time system is to identify the worst-case execution time (WCET) of each code-block. Tools, such as aiT [Gmb08], perform such static analysis using the principles of abstract interpretation [CC77].
- *Managing concurrency:* For systems with multiple concurrent tasks, the operating system must manage concurrency while meeting timing guarantees. This includes scheduling the tasks [ABD<sup>+</sup>95], and providing locking mechanisms to share resources [Bra11].
- *Modeling architectural effects:* As new computer architectures are used, the design and analysis of real-time systems must be adapted. Two areas which received attention are multiprocessor scheduling [SB09] and analysis of caches [LMW96].
- *Focus on analytical methods:* Conclusively, existing research recommends the use of analytical methods over simulation-based methods. Apart from soundness, which is essential in certification, analytical methods benefit from modularity [Wan06] and sensitivity analysis [HHJ<sup>+</sup>05].

## 1.2 Additional Challenges in Cyber-Physical Systems

The above research results derived for embedded systems are also applicable for CPSs. Indeed, to compute timing guarantees for a CPS, software must be statically analyzed, concurrency of multiple tasks must be managed, cache and multiprocessor behavior must be considered, and analytical methods must be adopted. However, the journey from



embedded systems to CPSs involves several additional challenges. In the following, we identify some such challenges.

- *Massively distributed systems:* As noted, CPSs are envisaged to be networked with each other, and thereby constitute massively distributed systems. A commercially developed example is the LS-460 car from Lexus which has about 100 Electronic Control Units (ECUs) [Tak12]. Furthermore, the ECUs are interconnected with a variety of communication platforms, such as CAN, FlexRay, and IDB-1394 [NSSLW05]. In such systems, the commonly adopted abstraction of periodic arrivals of jobs do not apply. Indeed, identifying bursts in traffic patterns and avoiding congestion are major focuses in large heterogeneous networks such as the Internet [BCC+98].
- *New trends in processor architectures:* As VLSI feature sizes have continued to shrink and functional integration has increased, the power density in microprocessors has multiplied. This has two major consequences. One, chips are now much hotter, and require active dynamic thermal management, such as speed-scaling, active throttling, and task migration [BM01]. This is further complicated by the spatial locality at several scales: neighboring transistors, cores on the same processor, different levels on a 3D chip, and across racks in a server farm. Two, an extrapolation of the current trends reveals that future processors will be unable to power all available transistors. This gloomy situation, referred to as “dark silicon”, is driving micro-architectural decisions [EBA+11]. In both these cases, the availability of the processor and its offered performance are not constants, but instead depend on physical parameters like temperature and energy.
- *Irregular timing requirements:* The focus on CPSs gives primacy to cross-layer considerations. A good example is a networked control system (NCS), wherein sensing, actuation, and communication are integrated. A cross-layer analysis of an NCS considers the impact of the delay inserted by the network, including interference and protocol effects, on the stability of the controlled plant [ZBP01]. Such analysis reveals that timing requirements for a stable plant do not imply that a fixed deadline must be met by each control signal [WA07]. The guarantee of stability can be satisfied even if a certain number of control signals do not arrive within a certain deadline. In other words, the timing requirements of CPS applications need not be regular, as is often assumed in schedulability analysis.

- *Uncertainties in design-time models:* As CPSs interact with users, the ambient, and computing units, identifying accurate and tight design-time models is a considerable challenge. A remedy, in part, is to identify several models with varying degrees of fidelity. A case in point is the current approach in mixed-criticality scheduling [BLS10]: Different levels of pessimism in the design-time models are translated to different levels of criticality, and each application is certified at a certain level of criticality. In other words, an *if-then* analysis, predicated by different model assumptions, specifies the timing guarantees. With such multiplicity of design-time models, there is an additional burden of monitoring and adapting to changes at run-time.
- *Difficult design problems:* Configuring large CPSs requires making several design choices, while satisfying multiple, often conflicting, hard constraints. Often these design problems do not admit standard solutions as they do not satisfy properties such as linearity or convexity. A computational approach to optimally solve such hard problems is the use of Satisfiability Modulo Theory (SMT) solvers [DMB11]. An SMT solver can be efficiently applied only if we can simultaneously analyze a large number of designs. Different designs may have different parameters. This variability needs to be effectively abstracted to represent and analyze multiple designs.

A common imperative in all the above challenges is the need to tolerate *variability*. Whether it is because jobs arrive in bursts, or processing speed varies to manage temperature, or control applications have irregular timing requirements, or validity of models changes at run-time, or multiple abstracted designs have different parameters, the timing properties of CPSs should be expected to vary. We argue that a higher degree of variability is the defining difference in computing hard real-time guarantees for CPSs, as opposed to embedded systems.

### 1.3 Aim of the Thesis

With this work, we aim to defend the following thesis:

*The essential complexity in providing hard real-time guarantees for cyber-physical systems is the presence of variability in timing models. It is possible to effectively manage this variability in several ways: by hiding it with a run-time manager, by bounding it in formal analysis, and by exporting it through richer guarantees.*

## 1.4 Thesis Outline and Contributions

### Part I: Absorbing Variability with Run-Time Managers

Prima facie, any variability in a deployed CPS can be absorbed by a sufficiently intricate run-time manager which (a) monitors variable timing models, and (b) responds such that timing guarantees are satisfied. However, this must be tempered by the, often overriding, need for frugal implementations of run-time managers, both in terms of timing and memory requirements. In this part of this thesis, we argue that for the specific case of variable job arrival patterns, modularly designing a run-time manager can efficiently and effectively absorb variability.

### Chapter 2: Isolation of Tasks with Demand Bound Servers

When a task with uncertain execution time executes on a processor, all other tasks on that processor, may (depending on the scheduling algorithm and schedulability) miss their deadlines. A principled approach to mitigate such uncertainty propagation was first proposed in [SB94], and has hence been referred to as serving tasks through *servers*. A server is a run-time manager which ensures that the task(s) served by it, do not execute for more than a certain *reservation*. In all the known servers, this reservation is specified as a certain utilization (or fraction) of the processor. For tasks with irregular job arrival patterns, which we expect in a distributed CPS, we show that the utilization-based reservation introduces a *schedulability-gap*. We propose a Demand Bound Server (DBS) wherein the reservation of the server is the sum of the demand bound functions of all tasks served by the server. We show that DBSs form the optimal class of servers, with no schedulability-gap. We propose Shifted-Periodic DBS (SP-DBS) as a specific DBS with an efficient implementation. Crucially, we show that SP-DBSs can be modularly composed to generate DBSs with no schedulability gap for a large class of tasks with irregular arrival patterns. In conclusion, by modularly composing SP-DBSs, the uncertainty in execution time and the variability in the arrivals patterns of a task can be effectively and efficiently abstracted from the analysis of other tasks.

### Chapter 3: Shaping Real-Time Tasks To Minimize Peak Temperature

Given the high power density in current generation of processors, limiting the peak on-chip temperature is a first-class design consideration. Due to the slow diffusive process of heat dissipation, executing a burst of jobs can lead to high temperatures. Consequently, spacing out the execution of jobs, by delaying individual jobs can reduce the peak temperature.

However, such delaying must not cause real-time jobs to miss their deadlines. We propose the run-time manager, referred to as *cool-shaper*, to balance the temperature and timing objectives. A cool-shaper delays incoming jobs to conform to a shaping-curve which is the convex-hull of the demand bound function of the stream of jobs. More specifically, a cool-shaper monitors the arrival patterns of jobs and dynamically adjusts the spacing of the jobs according to the shaping-curve. A cool-shaper is implemented by modularly composing leaky-bucket shapers, which were first proposed for ATM networks [Rat91] and are efficient in their operation. We show that a cool-shaper optimally minimizes the peak temperature while meeting the deadlines of all jobs. In conclusion, by modularly composing leaky-bucket shapers, the dependence of thermal objectives on the irregular job arrival patterns can be abstracted out: we are guaranteed to minimize the peak temperature while meeting all deadlines.

## Part II: Bounding Variability in Formal Analysis

In some CPSs, a run-time manager cannot be designed to absorb variability in timing models. This could be either because the run-time manager is designed to satisfy objectives other than timing guarantees, or because the run-time manager is required to be especially efficient. In such settings, the variability in the timing models is exposed to the analysis, whose role it is to bound the timing properties in the presence of variability. Furthermore, in the design optimization of large CPSs, multiple designs must be simultaneously analyzed. In this part of the thesis, we argue that formal analysis procedures can be derived to compute bounds in the presence of such variability. In particular, we highlight the effectiveness of deriving *abstractions* which represent and bound the variability.

### Chapter 4: Analysis of Temperature-Based Feedback Control of Speed

Speed-scaling is an effective technique to manage the on-chip temperature. However, exact thermal models can be difficult to estimate at design-time, as they depend on power consumption when executing different tasks, parameters of heat dissipation, and the ambient temperature. This uncertainty motivates a run-time manager based on reactive feedback-control [WB08]: The temperature of the processor is measured with on-board sensors, and accordingly the speed of the processor is regulated. This is an example of a CPS where the run-time manager is designed for an objective different from that of meeting timing guarantees. Apparently, the computation of timing guarantees for such a CPS is particularly

challenging: The processing power available to execute a job depends on the temperature of the processor, which in turn depends on when and how many jobs were executed earlier. In particular, when the job arrival patterns are variable, there can be a large number of traces of jobs, each generating different temperature traces. We show that for reactive speed-scaling and variable job arrival patterns, there is a specific *critical trace* for which the response-time of a specific job is maximized. Crucially, this critical trace is independent of the thermal parameters of the processor and the parameters of the run-time manager. Then, the worst-case response time is obtained by simulating the execution of the critical trace and observing the response time of a particular job. In conclusion, identifying the critical trace enables analysis of timing properties even in the presence of a run-time manager that dynamically changes the speed of the processor.

## Chapter 5: Satisfiability Modulo Real-Time Calculus

Design of CPSs involves configuring several design knobs to satisfy multiple, often conflicting, design constraints. As an example, speeds of processors in a distributed real-time system could be chosen to satisfy delay, buffer-space, and energy constraints. We illustrate with examples that such a design problem does not have intuitive solutions. For finding optimal designs for such hard problems, Satisfiability Modulo Theory (SMT) solvers [BSST09] provide a template for a solution strategy, as demonstrated for a variety of domains [DMB11]. For the efficient use of SMT solvers, we need to abstractly represent and analyze multiple designs. In the speed assignment problem, this means analyzing the delay, buffer-space and energy constraints for processors whose speeds may not be known precisely. To this end, we propose *abstract arrival and service curves* which extend the arrival and service curves defined in Real-Time Calculus [TCN00]. We show that operations of RTC can be extended to abstract curves, based on certain monotonicity properties of the abstract curves and the operations. We built an SMT solver with the OpenSMT framework [BPST10]. Experimentally, we showed that very large speed assignment problems (with  $\approx 3 \times 10^{17}$  designs) can be optimally solved in under 20 minutes on a typical desktop computer. In conclusion, we confirm the computational success of SMT solvers for the speed assignment problem, and thereby provide a template for other design problems in CPSs.

## Part III: Exporting Variability through Richer Guarantees

Variability in timing models of CPSs can affect the worst-case timing guarantees. For instance, variability in execution demand can increase the worst-case response-time. In some CPSs, there is a need for specifying more than the worst-case guarantees. This is motivated by two different reasons: one, if the variability is seldom observed, and two, if the application can tolerate a certain number of violations in the worst-case timing guarantee. In this part of the thesis, we propose richer timing guarantees which expose such variability.

### Chapter 6: The Settling-Time Metric

In certain CPSs, the timing models exhibit a dual nature: They conform to a *nominal* model at most times, which is violated under some *exceptional* conditions. An example is the dependence of execution time on the cache contents. The execution time of a job is exceptionally higher with a “cold cache”, which is encountered during the first instance of the task. For computing timing guarantees, both the nominal model and the exceptional events must be considered together. However, we make an alternate case of computing *two* timing guarantees: one, when the nominal model is always satisfied, and two, when an exceptional event occurs. In particular, for the latter guarantee we propose the *settling-time* metric which specifies the longest interval of time for which jobs can miss their deadlines after the occurrence of an exceptional event. We show that settling-time can be analytically computed for different scheduling algorithms and task models. We show that the Earliest Deadline First (EDF) scheduling algorithm, which is optimal in terms of schedulability, optimally minimizes the settling-time. In conclusion, if design-time models can resolve the difference between nominal models and exceptional events, we can compute richer timing guarantees.

## **Part I**

# **Absorbing Variability with Run-Time Managers**





# 2

## Isolation of Tasks with Demand Bound Servers

### 2.1 Introduction

When a task with an uncertain execution time executes on a processor, all other tasks on that processor may (depending on the scheduling algorithm and schedulability) miss their deadlines. This argument also extends to tasks with uncertain arrival patterns, for instance an unexpected jitter in a periodic task. A principled approach to mitigate such uncertainty propagation was first proposed in [SB94], and has hence been referred to as serving tasks through *servers*. A server is a run-time manager which ensures that the task(s) served by it, do not execute for more than a certain *reservation*.

Over the years, several servers have been proposed. They are classified into dynamic- and static-priority servers, depending on the underlying scheduler. We study dynamic-priority servers, which expect an EDF scheduler. Some of the proposed dynamic-priority servers are the two-level hierarchical scheduler [DL97], the Dynamic Priority Exchange Server (DPE) [SB94], the Dynamic Sporadic Server (DSS) [SB96, GB95], the Total Bandwidth Server (TBS) [SB94], the Constant Bandwidth Server (CBS) [AB98], the Bandwidth Sharing Server (BSS) [LB00], and the PShED algorithm [LCB00]. In each of these servers, the reservation of the server is a certain utilization (or bandwidth) of the processor, i.e., the server reserves a certain fraction of the processor for the task(s) it serves. Even with hierarchical composition of such servers, the unit of reservation remains a fraction of the processor. We refer to such a reservation as

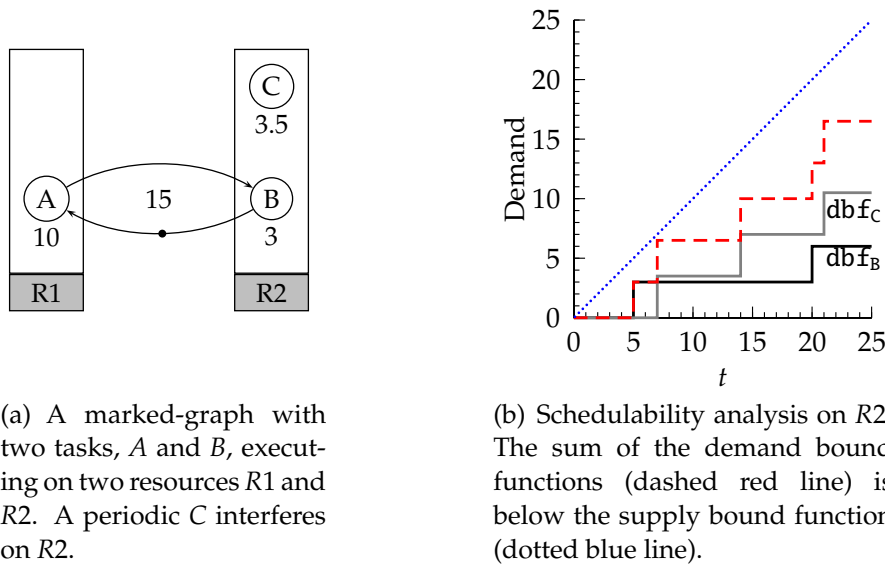


Fig. 2.1 Schedulability-gap: The disadvantage of utilization-based reservation.

utilization-based, and a server which enforces such a reservation as a utilization-based server.

A utilization-based reservation has some clear advantages. Firstly, proving schedulability is straight-forward: If the sum of the utilizations of the servers does not exceed the available processor utilization, then the servers can be scheduled together. This simple interface leads to the second advantage, namely, hierarchical decomposition: A reserved fraction of the processor can be hierarchically divided into smaller fractions which can be appropriately utilized.

However, utilization-based reservation has a major disadvantage which we refer to as the *schedulability-gap*. We illustrate this with an example.

**Example 2.1:** Consider two identical processors R1 and R2 scheduling three tasks A, B, and C, as illustrated in Figure 2.1(a). Task A is mapped on R1, while tasks B and C are mapped on R2. Tasks A and B belong to a marked-graph that executes periodically with period 15. The Worst-Case Execution Times (WCETs) of tasks A and B are 10 and 3, respectively. An iteration of the marked-graph must complete before the next iteration begins. Task C is an independent periodic task with WCET 3.5, and period and relative deadline equal to 7.

We now compute the utilization-based reservations on R2 to isolate the two tasks B and C. These reservations may be enforced with any of the server algorithms mentioned earlier. To serve task B, we need a reservation  $U_B = 3/5 = 60\%$ . To serve task C, we need a reservation  $U_C = 3.5/7 = 50\%$ . As  $U_B + U_C = 110\% > 100\%$ , the two reservations are not schedulable on R2.

Instead of using utilization-based reservations, let tasks  $B$  and  $C$  execute on  $R2$  directly through an EDF scheduler with relative deadlines 5 and 7, respectively. Then the two tasks are schedulable as shown in Figure 2.1(b): The sum of the demand bound functions of the two tasks is below the supply function of the full processor, i.e., a line through the origin with slope 1.

To conclude, tasks  $B$  and  $C$  can execute on  $R2$  without servers with guaranteed schedulability, but no utilization-based server can enforce reservations for the tasks. We refer to this as the *schedulability-gap*. In this example, the schedulability-gap arose because the relative deadline of task  $B$  was smaller than its period. Indeed, this is to be expected for distributed CPSs. Similarly, schedulability-gaps can be identified for tasks where jobs arrive in bursts, or experience jitter. Thus, we argue that the schedulability-gap is an outstanding problem in providing timing isolation amongst tasks sharing resources in CPSs.

We note two points which relate schedulability-gap with existing works, but highlight key differences.

- *Resource reclamation:* Consider again Example 2.1 with different reservations for the tasks on  $R2$ : Let us reserve 60% for task  $B$  and the remaining utilization of 40% for task  $C$ . With this modification the two reservations (not necessarily the tasks) are schedulable. If the server serving the task  $C$  allows for resource reclamation, as proposed in [LB00], [CBS00] or [MLBC04], then task  $C$  would reclaim unused utilization from the reservation of task  $B$ , and indeed meet all its deadlines. However, this approach detracts from the motivation of using servers, namely, to isolate tasks such that timeliness properties of a task can be analyzed *independent of the behavior of other tasks*.
- *Interfaces for resource demand and supply.* Interfaces exist that greatly generalize the utilization (or bandwidth) representation. Well known examples include the periodic supply function [SL04], explicit deadline periodic supply function [EAL07], and service curve used in Network Calculus [LBT01]. From the perspective of the schedulability-gap, the key question is whether there are server algorithms which *enforce* such interfaces of resource demand and supply, and thereby implement task isolation. To the best of our knowledge, all existing server algorithms enforce utilization-based resource interfaces, while the more generic interfaces are employed to *analyze* (distributed) real-time systems.

### 2.1.1 Contributions

Utilization-based reservation suffers from schedulability-gap, i.e., task-sets exist which are schedulable with an EDF scheduler, but cannot be isolated with utilization-based servers. Our goal in this chapter is to define a new class of servers to bridge the schedulability-gap. To this end, we propose the Demand Bound Server (DBS), which generalizes the reservation to a demand bound function. This reservation allows a DBS to tightly reserve the execution demanded by a task. From this tightness, we derive an optimality result: For any task-set that is schedulable, a configuration of DBSs exists to isolate the tasks.

As noted, any run-time manager in a real-time CPS, such as the DBS, must have an efficient implementation. To this end, we take two steps. Firstly, we propose a Shifted-Periodic Demand Bound Server (SP-DBS), which is a specific DBS that reserves a shifted-periodic demand bound function. We show that SP-DBS can be efficiently implemented. Secondly, we propose composition operations which can hierarchically combine several SP-DBSs to behave as a single DBS. This enables us to extend the class of SP-DBSs to a much wider class of DBSs. We argue that this wider class of DBSs can bridge the schedulability-gap for most practically arising task-sets in a CPS.

The rest of the chapter is organized as follows. In Section 2.2, we introduce a DBS and formulate its defining properties and guarantees. In Section 2.3, we present the implementation of a SP-DBS. In Section 2.4, we propose compositional operations which operate on SP-DBS and generate a richer class of DBSs. We summarize in Section 2.5, and include proofs of the results in an appendix.

## 2.2 Demand Bound Server

In this section, we present the definition of a Demand Bound Server (DBS). We then identify two defining properties of a DBS, and consequently two guarantees of a DBS.

### 2.2.1 Task-Model

A real-time task is a stream of jobs, where a job is also called an *instance* of the task. Each job  $J_i$  is characterized by an arrival time  $a_i$ , an absolute deadline  $d_i$ , and a worst-case execution time (WCET)  $C_i$ . We assume that the relative deadline (difference between absolute deadline and the

arrival time) of each job of a task is the same.<sup>1</sup> For any specific trace of jobs, the arrival function  $R$  characterizes the resource demand, as defined in Appendix A.2. In the presence of variability in arrival times and execution demands of jobs, the set of all possible arrival functions are abstracted by an arrival curve  $\alpha$ , as defined in Appendix A.2.

The resource demand of a trace of jobs can instead be represented by the demand bound function (DBF), which was first proposed by Baruah et al. in [BMR90] and is defined thus.

**Definition 2.1: (Demand Bound Function)** *A task has a demand bound function,  $\text{dbf}$ , if in the interval of time  $[t, t + \Delta]$  for any  $t, \Delta > 0$ , the sum of the execution times of all jobs that arrive not earlier than  $t$  and have deadline not later than  $(t + \Delta)$  does not exceed  $\text{dbf}(\Delta)$ .*

A demand bound function is non-negative and non-decreasing. On a dedicated processor, a task-set is EDF-schedulable if the sum of the demand bound functions of all tasks is bounded as below

$$\sum_i \text{dbf}_i(\Delta) \leq \Delta, \quad \forall \Delta \geq 0. \quad (2.1)$$

### 2.2.2 Defining the DBS

In this work, we define the Demand Bound Server (DBS) as a dynamic-priority server with a reservation which is given by a demand bound function denoted as  $\text{dbf}_s$ .<sup>2</sup> In this section, we will present the interpretation of such a reservation.

We first define some notation that characterizes how a DBS behaves. To serve a task, the DBS must make a sequence of requests for execution. Since the underlying scheduling discipline is EDF, each such request is characterized by a 3-tuple  $(x, y, z)$  interpreted as follows: At time  $x$  the server requests  $z$  time-units to be provided within an absolute deadline  $y$ . Let  $\rho$  denote the set of all such 3-tuples of requests made by a DBS to the EDF scheduler. In terms of these 3-tuples, we define the following property.

**Definition 2.2: (DBS Property I)** *A DBS characterized by the demand bound function  $\text{dbf}_s$  must satisfy the following property:*

$$\sum_{\{(x,y,z) \in \rho \mid x \geq t \wedge y \leq t + \Delta\}} z \leq \text{dbf}_s(\Delta), \quad \forall t, \Delta \geq 0. \quad (2.2)$$

<sup>1</sup>With minor modifications to our approach, jobs with different relative deadlines can be considered. However, we require that the absolute deadline of a later arriving job cannot be earlier.

<sup>2</sup>We will use the sub-script  $s$  to represent parameters of servers.

The above property says that for the EDF scheduler, a DBS is indistinguishable from a task with a demand bound function  $\text{dbf}_s$ .

The  $\text{dbf}_s$  of a DBS also specifies when jobs served by it will complete. We formally define this specification, by relating  $\text{dbf}_s$ , the arrival function  $R$ , and the output arrival function  $R'$  which is defined in Appendix A.3.

**Definition 2.3: (DBS Property II)** *Let a stream of jobs with an arrival function  $R$  be served by a DBS characterized by  $\text{dbf}_s$ . Let the resultant output arrival function be given by  $R'$ . Then, for any  $t \geq 0$  there exists  $s \in [0, t]$ , such that*

$$R'(t) \geq R(s) + \text{dbf}_s(t - s). \quad (2.3)$$

The above property says that for *any* time  $t$ , there is *some* interval  $[s, t]$  such that the jobs that have arrived after  $s$  receive at least  $\text{dbf}_s(t - s)$  service no later than  $t$ .<sup>3</sup>

To conclude, a DBS has a reservation  $\text{dbf}_s$  which is interpreted by the DBS Properties I and II. Indeed, a server is said to be a DBS if and only if it satisfies these two properties for some  $\text{dbf}_s$ .

### 2.2.3 Analysis of a DBS

A server must be characterized by two guarantees, (a) a *demand guarantee* which is an upper-bound on the amount of resource that the server can consume, and (b) a *supply guarantee* which is the minimum amount of resource that the server supplies to the task it serves. We now derive these guarantees of a DBS from its two defining properties.

From DBS Property I we know that a DBS does not request for more resources than a task with a demand bound function  $\text{dbf}_s$ . Thus, we can directly extend the EDF schedulability test of (2.1) into the following demand guarantee.

**Theorem 2.1: (Demand Guarantee of a DBS)** *Let  $S = \{\text{DBS}_i\}$  denote a set of DBSs executing on a resource, such that  $\text{DBS}_i$  is characterized by a demand bound function  $(\text{dbf}_s)_i$ . Then, the set of servers is EDF schedulable if and only if*

$$\sum_{\{i \mid \text{DBS}_i \in S\}} (\text{dbf}_s)_i(\Delta) \leq \Delta, \quad \forall \Delta \geq 0. \quad (2.4)$$

<sup>3</sup> It is important to distinguish between DBS Property II and similar concepts of a lower service curve in Real-Time Calculus (RTC) [TCN00] and supply bound function defined in [SL04]. The lower service curve,  $\beta^l$ , as used in RTC, guarantees that in *every* interval of any length  $\Delta$ , the minimum amount of execution guaranteed is  $\beta^l(\Delta)$ . Supply bound function is similarly defined. DBS Property II, on the other hand, provides no such guarantees, for every interval.

We choose to specify the supply guarantee of a DBS by the maximum delay of any job served by the DBS, denoted as  $d^{\max}$ . Then, deriving from DBS Property II and known results in Network Calculus [LBT01] we have the following relation.

**Theorem 2.2: (Supply Guarantee of a DBS)** *When a stream of jobs with an arrival curve  $\alpha$  is served by a DBS characterized by  $\text{dbf}_s$ , we have*

$$d^{\max} \leq \text{Del}(\alpha, \alpha \otimes \text{dbf}_s). \quad (2.5)$$

In the above relation, the operator  $\otimes$  and the function  $\text{Del}$  are as defined in Appendix A.1 and Appendix A.3, respectively.

To conclude, a DBS provides demand and supply guarantees wholly based on its characteristic  $\text{dbf}_s$ . From these results, it follows that a task with demand bound function  $\text{dbf}$  can be served by any DBS while meeting all deadlines if and only if  $\text{dbf}_s \geq \text{dbf}$ . Combining this with the optimality of the EDF scheduler, we have the following result.

*Optimality of DBS:* For a given task-set, if no schedulable configuration of DBSs exists to serve the tasks while meeting all deadlines, then the task-set cannot be temporally isolated.

## 2.3 Shifted-Periodic Demand Bound Server

Thus far, we have defined (with two properties) and characterized (with two guarantees) a Demand Bound Server (DBS). In this section, we propose a concrete implementation of a specific class of DBSs, with an especial focus on an efficient implementation. In particular, we describe the server algorithm of a DBS where the demand bound function  $\text{dbf}_s$  is the following periodic stair-case function with an arbitrary initial offset

$$\text{dbf}_s(\Delta) = \max \left\{ 0, \left( \left\lfloor \frac{\Delta - D_s}{P_s} \right\rfloor + 1 \right) \times Q_s \right\}. \quad (2.6)$$

We refer to such a function as a shifted-periodic function, and a DBS with  $\text{dbf}_s$  equal to a shifted-periodic function as a Shifted Periodic DBS (SP-DBS). A SP-DBS is characterized by a 3-tuple which parameterizes (2.6), namely  $(Q_s, P_s, D_s)$ , where  $Q_s$  is the maximum capacity,  $P_s$  is the period, and  $D_s$  is the relative deadline.

For Example 2.1 considered earlier, demand bound functions of both tasks  $B$  and  $C$  are shifted-periodic functions (Figure 2.1(b)). Thus, SP-DBSs, if they can be designed, would solve the schedulability-gap for this specific example, and thereby extend the existing utilization-based servers. This is the goal of this section.

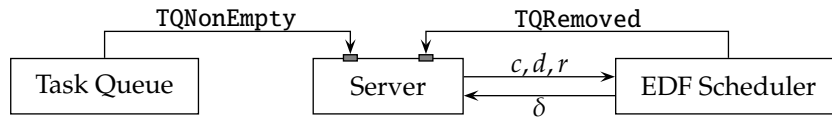


Fig. 2.2 Block diagram of a template for specifying dynamic-priority servers.

### 2.3.1 General Template of Dynamic-Priority Servers

We first formulate a general template of dynamic-priority servers. We will use this template to specify the working of a SP-DBS.

To implement task isolation with a dynamic-priority server, we need three components: a task-queue, an EDF scheduler, and a server. We characterize the behavior of these components by variables and signals which are passed between them, as visualized in the block diagram in Figure 2.2. We now elaborate on this template component-wise.

#### Task-queue

The task-queue contains, in First-Come-First-Serve (FCFS) order, the arrived and not yet finished jobs of the task that is served by the server. Each task-queue has three associated values, namely

- an absolute deadline  $d$  that denotes its dynamic-priority as negotiated with the EDF scheduler,
- a non-negative capacity  $c$  that denotes how long the task-queue will execute before it is evicted out of the EDF-queue, and
- a re-insertion time  $r$  that denotes when the task-queue must be re-inserted into the EDF-queue, if it is not already in the EDF-queue.

If a new job arrives while the task-queue is empty, the task-queue generates a signal `TQNonEmpty`. An evicted non-empty task-queue is re-inserted in the EDF-queue no later than the re-insertion time  $r$ .

#### EDF scheduler

The EDF scheduler is common across all dynamic priority servers on that resource. It implements the standard EDF policy, but with the following two minor changes.

- The EDF-queue contains pointers to task-queues instead of pointers to individual jobs, sorted according to their deadlines. At any time, the task-queue which is in the EDF-queue and has the smallest deadline is scheduled. The scheduler maintains, in a variable denoted  $\delta$ , the length of time for which the task-queue executes.



- If either the task-queue becomes empty or if  $\delta$  equals the capacity of the task-queue  $c$ , the EDF scheduler performs the following operations
  - it evicts the task-queue from its EDF-queue,
  - it generates a signal TQRemoved,
  - it communicates the value of  $\delta$  to the server, and
  - it resets the value of  $\delta$  to 0.

### Server

The server algorithm is responsible for updating the task-queue parameters  $d$ ,  $c$  and  $r$ , such that appropriate supply and demand guarantees are satisfied. These updates occur only when either of the two signals, TQNonEmpty or TQRemoved, are generated.

With this template, a dynamic-priority server can be compactly specified by (a) the initial assignments to the task-queue parameters, and (b) how the server algorithm changes the task-queue parameters when the two signals TQNonEmpty and TQRemoved are generated.

### 2.3.2 Implementation of an SP-DBS

The capacity of the task-queue,  $c$ , must be increased at certain times by the server. Such an increase is referred to as a *replenishment*. Further, at certain times the task-queue is re-inserted into the EDF-queue with a specific capacity and a deadline. This is referred to as a *service request*. To implement replenishments and service requests, an SP-DBS with parameters  $(Q_s, P_s, D_s)$  enforces the following two rules.

1. If a service request is generated at a certain time  $x$ , then the deadline of the task-queue is set to at least  $z = x + D_s$ .
2. Subsequent to a service request at time  $x$ , let the task-queue be evicted after executing for a certain time  $y$ . Then the capacity of the task-queue can be replenished by the amount  $y$  at any time on or later than  $x + P_s$ .

To enforce these rules, an SP-DBS maintains the following three server variables<sup>4</sup>

<sup>4</sup>The server variables are specific to an SP-DBS and must be contrasted from task-queue variables which are common across all servers.

**Algorithm 1:** Server algorithm of an SP-DBS

---

```

1 At the start
2    $c' \leftarrow Q_s, r' \leftarrow 0, \eta \leftarrow \text{empty set}$  // Initialize server parameters
3    $c \leftarrow Q_s, d \leftarrow 0, r \leftarrow \infty$  // Initialize task-queue parameters
4
5 When TQNonEmpty is generated
6    $d \leftarrow \max\{d, \text{current time} + D_s\}$  // Generate a new service request
7    $r \leftarrow d - D_s$ 
8    $r' \leftarrow d - D_s$ 
9
10 When TQRemoved is generated
11   $c \leftarrow c - \delta$  // Update capacity from EDF scheduler
12   $\eta \leftarrow \eta \cup (r' + P_s, c' - c)$  // Add entry to future replenishment set
13   $c' \leftarrow c$ 
14  if  $c = 0$  then
15     $\eta' \leftarrow \{(u, v) \mid (u, v) \in \eta \wedge u \leq \text{current time}\}$ 
16    if  $\eta'$  is not empty then
17       $c \leftarrow c + \sum_{(u,v) \in \eta'} v$  // Replenish with all outdated entries in  $\eta$ 
18       $\eta \leftarrow \eta - \eta'$ 
19    else
20       $(u', v') = \arg \min_{(u,v) \in \eta} u$  // Replenish with oldest entry in  $\eta$ 
21       $d \leftarrow \max\{d, u' + D_s\}$ 
22       $c \leftarrow v'$ 
23       $\eta \leftarrow \eta - (u', v')$ 
24  if Task-queue is non-empty then
25     $r \leftarrow d - D_s$  // Generate a new service request
26     $r' \leftarrow d - D_s$ 

```

---

- $r'$  which denotes the time of the last service request,<sup>5</sup>
- $c'$  the task-queue capacity at  $t'$ ,
- a future replenishment set  $\eta$  composed of a set of tuples  $(u, v)$ , which denote that at time  $u$  the task-queue capacity can be replenished by  $v$ .

For the described template and server parameters, we detail the algorithm of an SP-DBS in Algorithm 1. We show that the proposed algorithm of an SP-DBS implements a valid DBS as it satisfies the two DBS properties.

**Theorem 2.3:** *SP-DBS satisfies DBS Properties I and II as defined in Definitions 2.2 and 2.3.*

Given that an SP-DBS is a DBS, we benefit from the demand and supply guarantees proved in Theorems 2.1 and 2.2. Both schedulability

---

<sup>5</sup>The value of  $r'$  exactly equals the task-queue re-insertion time  $r$ . However, for a soft variant (Section 2.3.4) and for composition of servers (Section 2.4) the two values differ.

and delay analyses for an SP-DBS follow from these guarantees. We illustrate the working of the algorithm for the following example.

**Example 2.2:** Consider two SP-DBSs:  $DBS_1$  and  $DBS_2$  with parameters (given as the tuple  $(Q_s, P_s, D_s)$ )  $(3, 6, 5)$  and  $(1, 3, 2)$ , respectively. Each server executes a stream of jobs. The tuples  $(a_i, C_i)$ , i.e., the arrival time and the execution demand, of the jobs arriving at  $DBS_1$  are  $\{(3, 3), (10, 2), (13, 2), (19, 2), (20, 1)\}$ . The corresponding tuples for  $DBS_2$  are  $\{(0, 3), (12, 1), (16, 2), (18, 1)\}$ .

Using Theorem 2.2, we can verify that these two SP-DBSs are schedulable. We illustrate how the jobs are served by the two servers in Figure 2.3. The arrival of jobs is shown with upward arrows with the execution time of the job (unknown to the server) depicted on top of the arrow. In the figure, we plot the execution pattern, the deadline  $d$  and the capacity  $c$  of both the servers over time. Dark boxes under the deadlines denote times when the task-queue is in the EDF-queue. We also plot the (input) arrival function  $R$ , the output arrival function  $R'$ , and the lower-bound on the output arrival function  $R \otimes \text{dbf}_s$  (Definition 2.3).

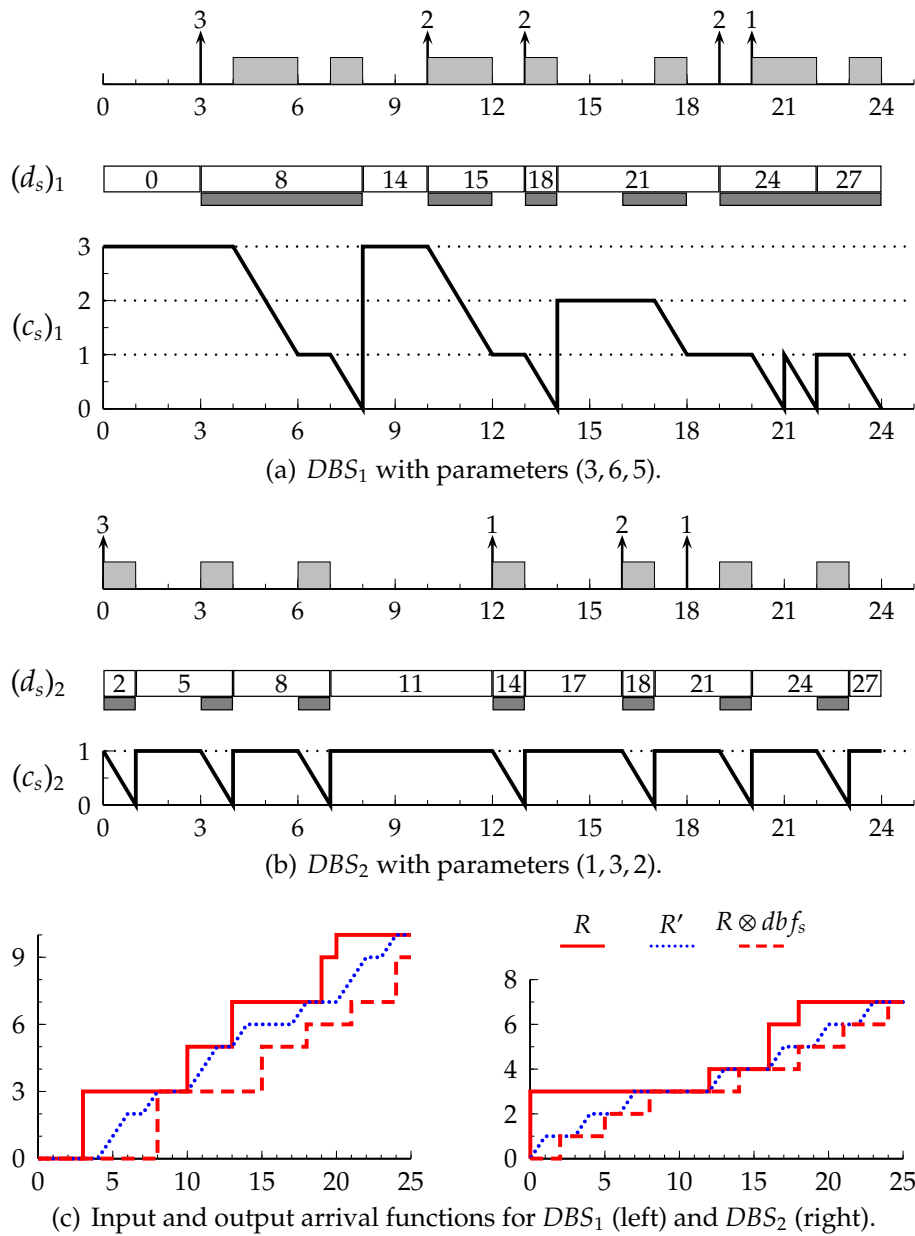
We now illustrate that the two DBS properties are satisfied in the example. The DBS Property I can be verified by checking (2.2) for each interval for the two servers. For instance, as  $((\text{dbf}_s)_1 + (\text{dbf}_s)_2)(5) = 5$ , schedulability in an interval of length 5 where the demand equals 5, will confirm DBS Property I. Indeed, this is true for the interval  $[3, 8]$ . To verify DBS Property II of SP-DBS, we observe that input and output arrival functions as satisfy  $R' \geq R \otimes \text{dbf}_s$ , for both the servers (Figure 2.3(c)). Additionally, it is revealing to compute the value of  $s$  for different values of  $t$  such that (2.3) is satisfied.

### 2.3.3 Implementation complexity

We characterize the overhead of the SP-DBS algorithm with the worst-case memory and timing complexities. We define a parameter  $C^{\min}$  as the smallest number such that the execution and arrival times of all jobs, and the parameter  $Q_s$  are integral multiples of  $C^{\min}$ . In other words,  $C^{\min}$  is the time granularity of all scheduling events.

#### Memory complexity

We ignore the constant memory overhead of maintaining the server variables  $c'$  and  $t'$ . We focus on the future replenishment set  $\eta$ , the size of which varies. It can be shown that at any point of time, the sum of the second element  $v$  across all tuples of  $\eta$ , does not exceed  $Q_s$ . Using this and the defined parameter  $C^{\min}$ , the number of entries in  $\eta$  is at most  $Q_s/C^{\min}$ .



**Fig. 2.3** Illustration of the SP-DBS algorithm for Example 2.2. Arrival of jobs are shown with upward arrows with their execution demands. The server deadline and capacity are shown over time. The area below the deadlines are shaded dark if the corresponding task-queue is in the EDF-queue. The (input) arrival function  $R$ , the output arrival function  $R'$ , and the lower-bound on the output arrival function,  $R \otimes dbf_s$  are also plotted.

### Timing complexity

The elements of  $\eta$  can be sorted by the first element of the tuple  $u$ . It can be shown that the tuples of such a sorted  $\eta$  are added and removed with a FCFS discipline. Thus, all operations on  $\eta$  happen in constant time. However, line 15 of Algorithm 1 operates on several entries of  $\eta$ . The number of entries in  $\eta$  depends on the number of service requests generated in the past (lines 7 and 25 of Algorithm 1) which have not yet been replenished. It can be shown that number of service requests until time  $t$  is at most  $R'(t)/C^{\min}$ . As  $R'(t) \leq \text{dbf}_s(t + D_s) \leq Q_s + (Q_s/P_s)t$ , the maximum number of service requests made to the EDF-queue grows linearly in time and is inversely related to the parameter  $C^{\min}$ .

To conclude, both the memory and timing complexities of the server algorithm are bounded, and inversely grow as the granularity of the jobs, defined in  $C^{\min}$ , falls. If the essential complexity of a very small  $C^{\min}$  is avoided, then SP-DBS has an efficient implementation.

### 2.3.4 Soft variant of SP-DBS

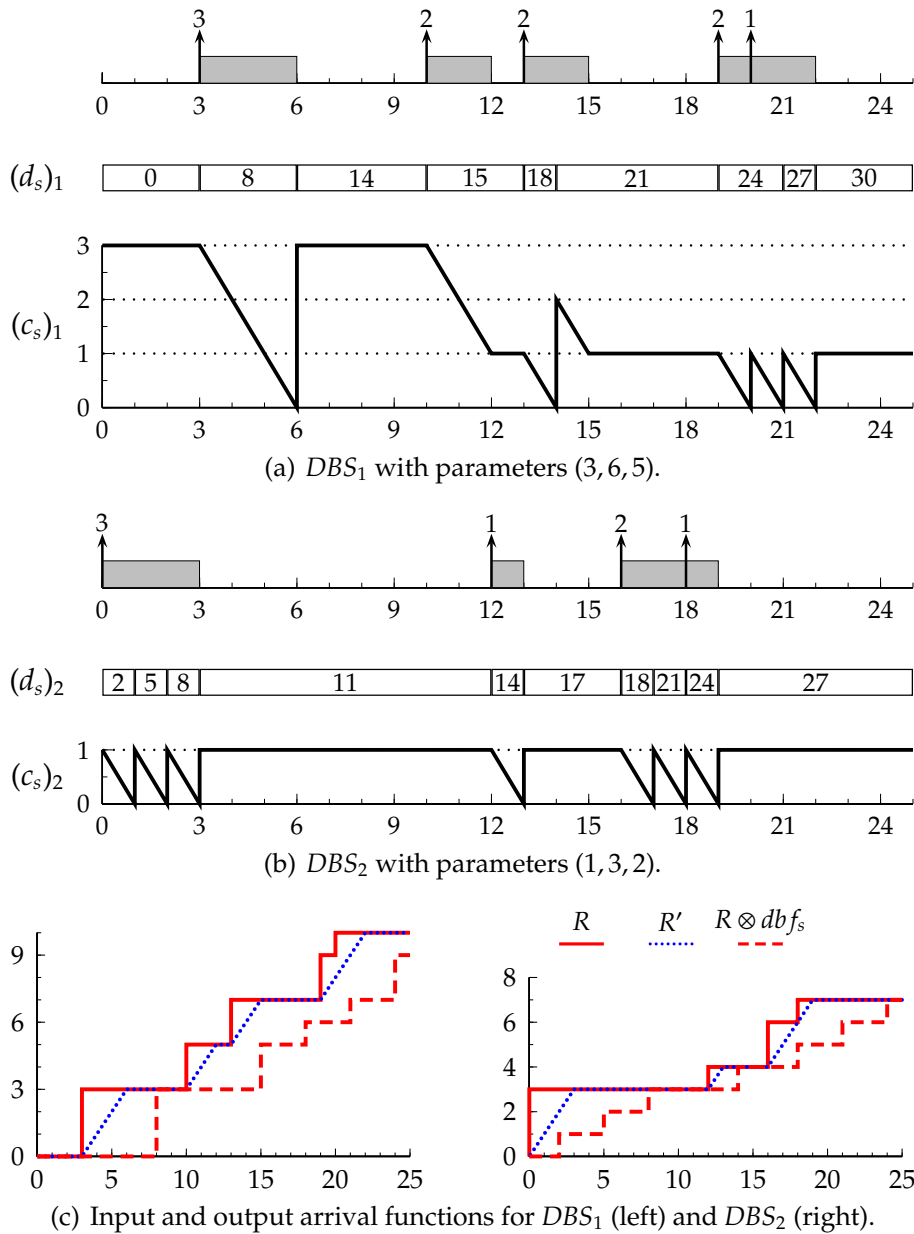
In existing research literature, soft<sup>6</sup> variants of dynamic-priority priority servers have been proposed. For instance, the original Constant Bandwidth Server proposed in [AB98] is a soft server. In such implementations, whenever the task-queue served by the server is non-empty, it is inserted into the EDF-queue. Both hard and soft variants have their relative merits and demerits as discussed in [APSL09].

The presented algorithm of SP-DBS is a hard variant. With the template of a dynamic-priority server presented in Section 2.3.1, differentiating between soft and hard variants is straight-forward. A soft variant requires a minor change: The re-insertion time  $r$  is ignored, and a non-empty task-queue is always in the EDF-queue. For the presented algorithm of the SP-DBS, it can be proved that with this change, the DBS Properties I and II still hold.

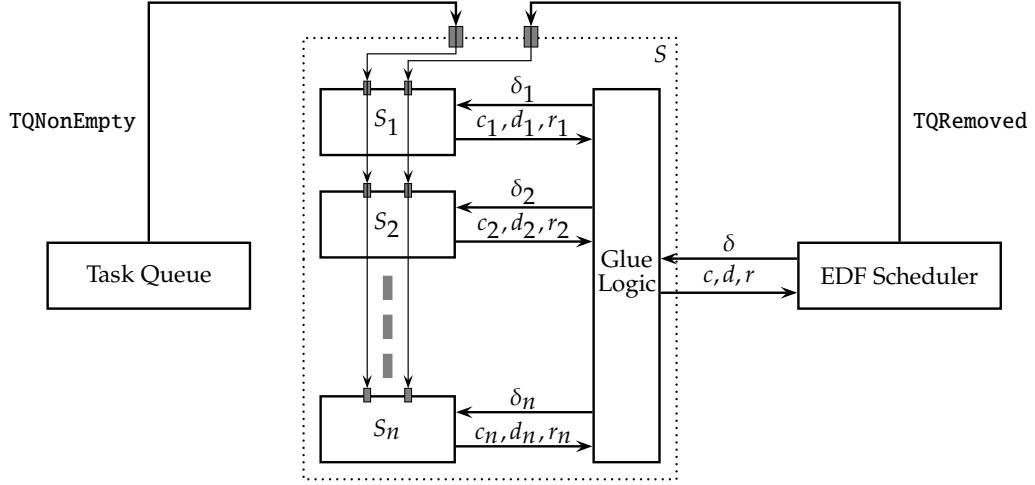
For Example 2.2 we illustrate the behavior of the soft variant of SP-DBS in Figure 2.4. Notice the contrast to the earlier hard-variant: The servers are now workload-conserving, and execute the jobs successively at the expense of larger server deadlines. The two DBS properties can be verified from this example.

---

<sup>6</sup>The word “soft” here is not used in the same sense as in soft real-time systems. Soft servers can be characterized by hard real-time properties.



**Fig. 2.4** Illustration of the soft variant of SP-DBS for Example 2.2. Arrival of jobs are shown with upward arrows with their execution demands. The server deadline and capacity are shown over time. Whenever a task-queue is non-empty, it is inserted into the EDF-queue. The (input) arrival function  $R$ , the output arrival function  $R'$ , and the lower-bound on the output arrival function  $R \otimes dbf_s$  are also plotted.



**Fig. 2.5** A template for the composition of dynamic-priority servers. The glue logic of the composed server translates the interface variables between the EDF scheduler and the constituent servers. The two signals are relayed to all servers.

## 2.4 Composition Operators for DBSs

We motivate the need for DBSs which further generalize SP-DBSs with the following example.

**Example 2.3:** Consider two periodic tasks  $E$  and  $F$  executing on the same processor. Task  $E$  has a period 1.5, a maximum jitter 0.5 that can be exhibited by a maximum of 3 consecutive jobs, a WCET 1, and a relative deadline 3. Task  $F$  has a period 6, WCET 1.5, and a relative deadline 2.

Employing (2.1) confirms that the two tasks are EDF-schedulable. However, they cannot be served by utilization-based servers. Furthermore, it can be shown that the tasks cannot be served by SP-DBSs. This schedulability-gap is because the demand bound function of task  $E$  is not a shifted-periodic function. Designing an over-provisioned SP-DBS for task  $E$  fails the schedulability test. This is illustrated in Figure 2.6(a). This motivates the design of a broader class of DBSs. Towards this end, we propose composition operations that operate on several SP-DBSs to form a single DBS. In the remainder of this section, we present two such composition operations, and illustrate their utility.

We first discuss a general template for operations on one or more DBSs. Let an operation on a set of DBSs,  $\mathbf{S} = \{S_1, S_2, \dots, S_n\}$ , compose them to form a single server,  $S$ . We refer to  $S$  as the *composed* server and  $S_1, S_2, \dots, S_n$  as the *constituent* servers. The composed server must provide the *glue logic* to interface the constituent servers with the EDF scheduler, as shown in Figure 2.5. The glue logic has two functions, (a) to translate the task-queue parameters as set by the constituent servers on to actual

task-queue parameters, and (b) to translate the variable  $\delta$  as provided by the EDF scheduler on to the constituent servers. All server algorithms and the glue logic are executed only when the signals `TQNonEmpty` and `TQRemoved` are raised.

For any composition operation, we need to prove that the composed server is a DBS if all constituent servers are DBSs. If so, furthermore, it is desirable that demand bound function of the composed server be a function of the demand bound functions of the constituent servers.

### 2.4.1 Min-Composition

We now discuss the *min-composition* operation which is formally defined as follows.

**Definition 2.4: (Min-Composition)** *A set of DBS  $\{S_1, S_2, \dots, S_n\}$  is said to be min-composed to form a server  $S$ , denoted as  $S = S_1 \wedge S_2 \wedge \dots \wedge S_n$ , if the glue logic of server  $S$  implements the following interface*

$$\begin{aligned} d &:= \max(d_1, d_2, \dots, d_n), \\ c &:= \min(c_1, c_2, \dots, c_n), \\ r &:= \max(r_1, r_2, \dots, r_n), \\ \delta_i &:= \delta, \quad \forall i \in \{1, 2, \dots, n\}. \end{aligned} \quad (2.7)$$

For the glue logic defined as above, we prove the following property of the composed server.

**Theorem 2.4:** *Let  $S := S_1 \wedge S_2 \wedge \dots \wedge S_n$ , then the server  $S$  is a DBS with a demand bound function  $\text{dbf}_s$  which is related to  $(\text{dbf}_s)_i$ , the demand bound function of server  $S_i$ ,  $i \in \{1, 2, \dots, n\}$ , as*

$$\text{dbf}_s := \min((\text{dbf}_s)_1, (\text{dbf}_s)_2, \dots, (\text{dbf}_s)_n). \quad (2.8)$$

The defined min-composition creates a composed server which has a  $\text{dbf}_s$  that is the minimum of the  $\text{dbf}_s$  of the constituent servers. We now illustrate the utility of this composition with Example 2.3. As noted, SP-DBSs cannot schedule the two tasks  $E$  and  $F$ , though the tasks are EDF-schedulable (Figure 2.6(a)). This schedulability-gap is attributed to the demand bound function of task  $E$ , which is not a shifted-periodic function. However, with min-composition we can design a server that exactly reserves the demand bound function of task  $E$ . Consider  $S_{E'} = S_1 \wedge S_2$ , where  $S_1$  and  $S_2$  are SP-DBSs with parameters  $(1, 1.5, 1.5)$  and  $(1, 1, 2)$ , respectively. As we show in Figure 2.6(b),  $S_{E'}$  has a  $\text{dbf}_s$  equal to the demand bound function of task  $E$ . Thus, min-composition bridges the schedulability-gap for this example.



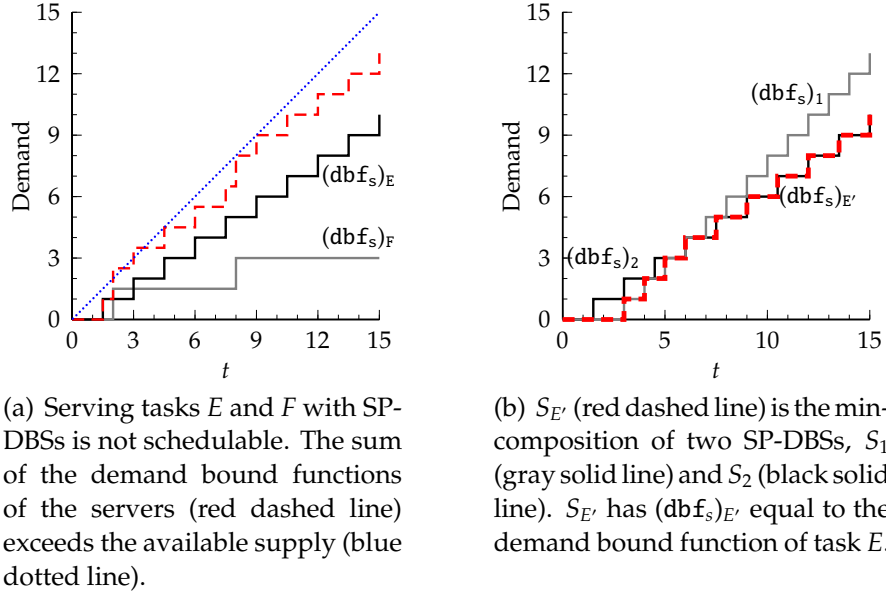


Fig. 2.6 Utility of min-composition for Example 2.3.

## 2.4.2 Left-Shift

We now discuss a second operation on a single DBS called *left-shift*. The operation is formally defined as follows.

**Definition 2.5: (Left-Shift)** A server  $S$  is said to implement left-shift by  $\tau$  of a DBS  $S_1$ , denoted as  $S := (\overset{\tau}{\leftarrow} S_1)$ , if the glue logic of  $S$  behaves as

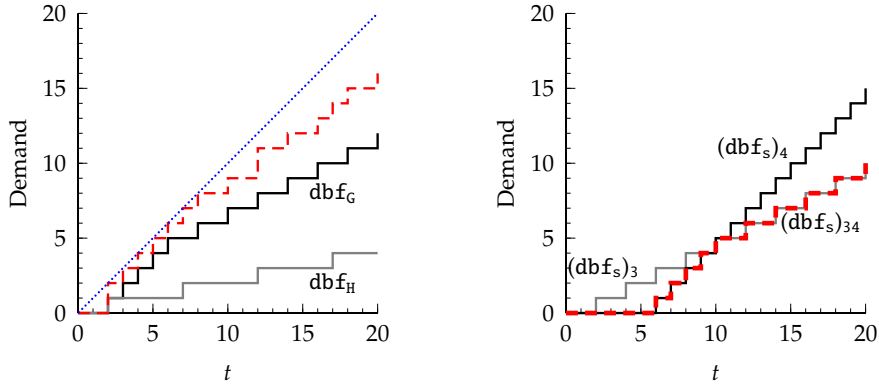
$$\begin{aligned}
 d &:= d_1 - \tau, \\
 c &:= c_1, \\
 r &:= \max(t, r_1 - \tau) \\
 \delta_1 &:= \delta.
 \end{aligned} \tag{2.9}$$

The following result characterizes the left-shift operation.

**Theorem 2.5:** Let  $S_1$  be a DBS with demand bound function  $(\text{dbf}_s)_1$ , such that  $(\text{dbf}_s)_1(t + \tau) \leq t, \forall t \geq 0$ . Then,  $S := (\overset{\tau}{\leftarrow} S_1)$  defines a DBS with a demand bound function  $\text{dbf}_s$  given as

$$\text{dbf}_s(t) = (\text{dbf}_s)_1(t + \tau), \quad t \geq 0. \tag{2.10}$$

The left-shift composition generates a server with a  $\text{dbf}_s$  which is a left-shifted version of the  $\text{dbf}_s$  of the constituent server. We illustrate the advantage of left-composition in the following example.



(a) Tasks  $G$  and  $H$  are EDF-schedulable: the sum of the demand bound functions (dashed red line) does not exceed the supply function (dotted blue line).

(b)  $DBS_{34}$  is the min-composition of  $DBS_3$  and  $DBS_4$ . Then  $(dbf_s)_{34}$  is left-shifted by 4 to equal the demand bound function of task  $G$ .

Fig. 2.7 Utility of left-shift for Example 2.4.

**Example 2.4:** Let two periodic tasks  $G$  and  $H$  execute on a processor. Task  $G$  has a period 2, relative deadline 2, a maximum jitter 1 that can be exhibited by a maximum of 4 consecutive jobs, and a WCET 1. Task  $H$  has a period 5, a relative deadline 2 and WCET 1.

The tasks are EDF schedulable as shown in Figure 2.7(a). As before, there exists no configuration of utilization-based servers or SP-DBSs that serves the tasks. Further, no min-composition of SP-DBS can serve the task set. This schedulability-gap is attributed to the demand bound function of task  $G$ . We now design a server that exactly reserves the demand bound function of task  $G$ . Let  $S_3$  and  $S_4$  be SP-DBS with parameters  $(1, 2, 2)$  and  $(1, 1, 6)$ , respectively. Let  $S_{34} := S_3 \wedge S_4$  and  $S_G := (\overset{4}{\leftarrow} S_{34})$ . Then, as shown in Figure 2.7(b),  $S_G$  has a demand bound function  $(dbf_s)_G$  exactly equal to the demand bound function of task  $G$ . Thus, hierarchical left- and min-composition bridges the schedulability-gap for this example.

## 2.5 Summary

Example 2.1 illustrated that SP-DBSs can bridge the schedulability-gap for tasks with deadlines less than periods. Such tasks are to be expected in CPSs where the *end-to-end* deadline of a task-chain may equal the task period. Examples 2.3 and 2.4 illustrated that composition of SP-DBSs can bridge the schedulability-gap for tasks with bursts and jitters. Such tasks are to be expected in distributed CPSs where interference on networks cause irregularities in arrival times of jobs. In addition, we showed that

the proposed algorithm of SP-DBS and the composition operations are efficient to implement. With these arguments, we claim to accomplish the said goal, i.e., to design a run-time manager to abstract the variability in the timing properties of one task from the analysis of all other tasks sharing the same resource.

From these results a design pattern emerges. First, we constructed a run-time manager that only absorbs a small class of variabilities, but is efficient to implement. Then we defined composition operations to modularly build more complex run-time managers which can absorb a wider class of variabilities. To be effective, such operations need to sufficiently broaden the class of efficient run-time managers.

## Appendix

### Proof of Theorem 2.1

This follows from the EDF-schedulability test of (2.1) shown in [BMR90] and the DBS Property I defined in Definition 2.2.  $\square$

### Proof of Theorem 2.2

Let  $R'$  be the output arrival function (defined in Appendix A.3) for the input arrival function  $R$  when served by a DBS characterized by  $\text{dbf}_s$ . Then, the maximum delay of any job  $d^{\max}$  is given as

$$d^{\max} = \text{Del}(R, R'). \quad (2.11)$$

From DBS Property II we know that, for every  $t \geq 0$ , there is some  $s \in [0, t]$  such that

$$R'(t) \geq R(s) + \text{dbf}_s(t - s).$$

This can be written as follows.

$$R'(t) \geq \inf_{0 \leq u \leq t} \{ R(u) + \text{dbf}_s(t - u) \}. \quad (2.12)$$

The R.H.S. is equal to  $R \otimes \text{dbf}_s$  from the definition of the  $\otimes$  operator (Appendix A.1). Substituting this in (2.11), the maximum delay is  $\text{Del}(R, R \otimes \text{dbf}_s)$ . Abstracting arrival functions to arrival curves we derive the required result.  $\square$

### Proof of Theorem 2.3

*DBS Property I:* Let  $\rho$  denote the set of all 3-tuples  $(x, y, z)$  interpreted as defined in Section 2.2.2. We describe two properties of the set  $\rho$  for an SP-DBS.

1. An SP-DBS requests for new resources in lines 6 and 25 of Algorithm 1. In either case, the reinsertion time  $r$  satisfies  $d - D_s$ , where  $d$  is the task-queue deadline. Thus for every  $(x, y, z) \in \rho$ ,  $y \geq x + D_s$ .
2. Future requests are based on the future replenishment set  $\eta$ . A new entry is added to  $\eta$  in line 12 of Algorithm 1. The entry added is  $(r' + P_s, c' - c)$  where  $r'$  is the time of the last request, and  $(c' - c)$  is the consumed budget since the last request. Thus, any consumed budget is replenished by the same amount  $P_s$  time units later. Further, the initial value of  $c'$  is  $Q_s$ . Thus, in any interval of length  $P_s$  the total resource requested is at most  $Q_s$ .

From these two properties of the set  $\rho$  we can show that the SP-DBS has a demand bound function given by (2.6).

*DBS Property II:* Let  $R(t)$  be the input arrival function of any stream of jobs served by an SP-DBS. Let  $R'(t)$  be the corresponding output arrival function. We need to show that

$$R'(t) \geq R(s) + \max \left\{ 0, \left( \left\lfloor \frac{t - s - D_s}{P_s} \right\rfloor + 1 \right) \times Q_s \right\} \quad (2.13)$$

If the task-queue is empty at  $t$ , we have  $R'(t) = R(t)$ . Then we can set  $s = t$  and satisfy (2.13). Let the task-queue be non-empty at time  $t$ , with a deadline  $d_t \geq t$ . We consider two cases.

*Case (a) with  $t < d_t$ :* There must be a service request at time  $s' = d_t - D_s$ , such that the service provided to the SP-DBS in  $[s', d_t]$  is non-zero. If at time  $s'$  a new task arrives at an empty task-queue, then we can set  $s = s'$  in (2.13). Else, the request was due to a depleted budget, i.e.,  $c = 0$  at  $s'$ . This implies that in the interval  $[s' - P_s, s']$  the task queue received the full budget of  $Q_s$ . At time  $s'' = s' - P_s$ , a service request is made. If this is because a new task arrives to an empty task-queue then we set  $s = s''$  in (2.13). Else, we repeat the above argument. If this process continues for  $n$  iterations, we have  $s + nP_s < t < s + nP_s + D_s$  and  $R'(t) - R(s) > nQ_s$ . Further as the task-queue is empty at  $s$ , we have  $R'(s) = R(s)$ . Then this  $s$  satisfies (2.13).

*Case (b) with  $t = d_t$ :* A request made by the server finishes just at the deadline. From DBS Property I we know that the server is schedulable. Thus, there exists some interval  $[s', t]$  in which the schedulability constraint is tight, i.e., the total resource requested and received by the SP-DBS in  $[s', t]$  is  $\text{dbf}_s(t - s')$ . If a new task arrives at  $s'$  to an empty task-queue, we can set  $s = s'$  and satisfy (2.13). Else, we follow the same argument as in the previous case, and identify  $s = s' - nP_s$  such

that a new task arrives to an empty task-queue. Then,  $R'(s) = R(s)$  and  $R'(t) - R'(s) = \text{dbf}_s(t - s)$ .  $\square$

### Proof of Theorem 2.4

To prove that the resultant server after composition is a DBS, we need to show that DBS Properties I and II are satisfied for the demand bound function as given in (2.8).

*DBS Property I:* Consider any constituent server  $S_i$ . Since  $\delta_i = \delta$  whenever the task-queue executes, the server variable  $c$  of  $S_i$  is decreased. Furthermore from (2.7), for each request tuple  $(x_i, y_i, z_i)$  of  $S_i$  the corresponding request tuple  $(x, y, z)$  of  $S$  has  $x \leq x_i$ ,  $y \geq y_i$ , and  $z \leq z_i$ . Then, from the definition of  $\text{dbf}_s$  in Definition 2.1, the  $\text{dbf}_s$  of the composed server cannot exceed  $(\text{dbf}_s)_i$ .

*DBS Property II:* The property is to be shown for any constituent DBSs. To this end, we first define an algorithm of a DBS for a general  $\text{dbf}_s$ . Let  $R$  and  $R'$  denote the input and output arrival functions for some trace of jobs. Let  $t$  denote the current time. Whenever the task-queue is non-empty and not in the EDF-queue, a general DBS<sup>7</sup> will set the task-queue parameters as follows.

$$d = \min\{s \mid (R \otimes \text{dbf}_s)(s) > R'(t)\}, \quad (2.14)$$

$$c = (R \otimes \text{dbf}_s)(d) - R'(t), \quad (2.15)$$

$$r = \min\{s \mid (R \otimes \text{dbf}_s)(s) \geq R'(t)\}. \quad (2.16)$$

For the min-composition, let each constituent server be characterized by the above equations. Then, by applying the glue-logic of (2.7) the task-queue parameters as set by the composed server is given as below.

$$d = \max_{S_i \in S} (\min\{s \mid (R \otimes (\text{dbf}_s)_i)(s) > R'(t)\}), \quad (2.17)$$

$$c = \min_{S_i \in S} ((R \otimes (\text{dbf}_s)_i)(d) - R'(t)), \quad (2.18)$$

$$r = \max_{S_i \in S} (\min\{s \mid (R \otimes (\text{dbf}_s)_i)(s) \geq R'(t)\}). \quad (2.19)$$

As  $R$  and  $R'$  are the same for the all composed servers, the above equations simplify as follows.

$$d = \min\{s \mid (R \otimes \min_{S_i \in S} (\text{dbf}_s)_i)(s) > R'(t)\}, \quad (2.20)$$

$$c = (R \otimes \min_{S_i \in S} (\text{dbf}_s)_i)(d) - R'(t), \quad (2.21)$$

$$r = \min\{s \mid (R \otimes \min_{S_i \in S} (\text{dbf}_s)_i)(s) \geq R'(t)\}. \quad (2.22)$$

<sup>7</sup> This represents a hard-variant as the task-queue. A soft-variant requires setting  $r = t$ .

Hence, the composed server satisfies the DBS Property II with a  $\text{dbf}_s$  as defined in (2.8).  $\square$

### Proof of Theorem 2.5

To prove that the resultant server after composition is a DBS, we need to show that DBS Properties I and II are satisfied for the demand bound function as given in (2.10).

*DBS Property I:* Since  $\delta_1 = \delta$  whenever the task-queue executes, the server variable  $c$  of  $S_1$  is decreased. From the glue-logic of (2.9), for each request tuple  $(x_1, y_1, z_1)$  of  $S_1$  the corresponding request tuple  $(x, y, z)$  of  $S$  has  $x = x_1$ ,  $y = y_1 - \tau$ , and  $z = z_1 - \tau$ . Hence, the  $\text{dbf}_s(t)$  of the composed server cannot exceed  $(\text{dbf}_s)_1(t + \tau)$ .

*DBS Property II:* Consider the description of a generic algorithm of a DBS in the previous proof. By applying the glue-logic defined in (2.9), the task-queue parameters as set by the composed server satisfies the following conditions.

$$d = (\min\{s \mid (R \otimes (\text{dbf}_s)_1)(s) > R'(t)\}) - \tau, \quad (2.23)$$

$$c = \min_{S_1 \in S} ((R \otimes (\text{dbf}_s)_1)(d_1) - R'(t)), \quad (2.24)$$

$$r = \max(t, \max_{S_1 \in S} (\min\{s \mid (R \otimes (\text{dbf}_s)_1)(s) \geq R'(t)\}) - \tau). \quad (2.25)$$

Substituting  $\text{dbf}_s(t) = \text{dbf}_s(t + \tau)$  in the above equations satisfies (2.14) to (2.16). Hence, the composed server satisfies DBS Property II with  $\text{dbf}_s$  as defined in (2.10).  $\square$

# 3

## Shaping Real-Time Tasks to Minimize Peak-Temperature

### 3.1 Introduction

The computation of timing guarantees in CPSs must adapt to changing requirements of processor architectures used in such CPSs. In this chapter, we study one such requirement, namely system throttling to reduce peak-temperatures.

One of the primary design concerns in current mainstream processors is the high power density, which in turn translates to high on-chip temperatures. High temperatures are detrimental for two reasons. One, they affect the reliability of the processor. Two, at higher temperatures the leakage power consumption is higher, thereby causing a vicious cycle that may lead to *thermal run-aways*. Consequently, limiting the peak-temperature is an important design guideline.

Hardware cooling solutions remain the default approach to remove heat from electronic devices. However, the cost of packaging technology is increasing exponentially with rising power density: It is estimated to cost \$3 per watt of heat dissipated [SSH<sup>+</sup>03]. Furthermore, in mobile devices the constraints of volume and noise emission restrict the use of standard cooling infrastructure such as fans and ventilators [TT04]. Finally, in novel 3-dimensional stacking or fabrication of chips, conventional conductive cooling is less effective [CWZ04]. Consequently, it is now an established practice to supplement hardware solutions with run-time software solutions, which are broadly referred to as Dynamic Thermal Management (DTM) techniques [DM06]. DTM techniques

comprise of interventions including voltage (and frequency) scaling, clock-gating and execution-unit throttling.

Clearly, employing DTM techniques can have performance penalties, and thereby affect timing guarantees. For instance, if a hot processor is temporarily turned off when a job arrives, it may miss its deadline. This is particularly challenging if the input trace exhibits variability in job arrival patterns such as bursts of jobs arriving at unknown points in time. A run-time manager must anticipate such events, while simultaneously trying to aggressively minimize the temperature.

Most existing research which considers both temperature and real-time objectives, focuses on Dynamic Voltage Scaling (DVS). As noted by Brooks et al. in [BM01], for certain architectures the overhead of voltage scaling can be substantial. Instead they found throttling certain components of the architecture or disabling them entirely to be more effective in reducing temperatures. This is corroborated by the general push towards near-zero idle power to approximate the idealistic goal of energy-proportional computing [BH07].

Some research studies have considered non-DVS DTM techniques. Skadron et al. studied the use of control-theoretic techniques for triggering fetch-toggling in [SAS02]. This was extended to a predictive policy by Srinivasan et al. in [SA03]. For chip-multiprocessors (CMPs) Powell et al. proposed in [GPV04] the “heat-and-run” approach of executing tasks on a core until it needs cooling, and then migrating to cooler cores. A related approach is the adaptive workload-distribution algorithm proposed by Coskun et al. in [CRW07].

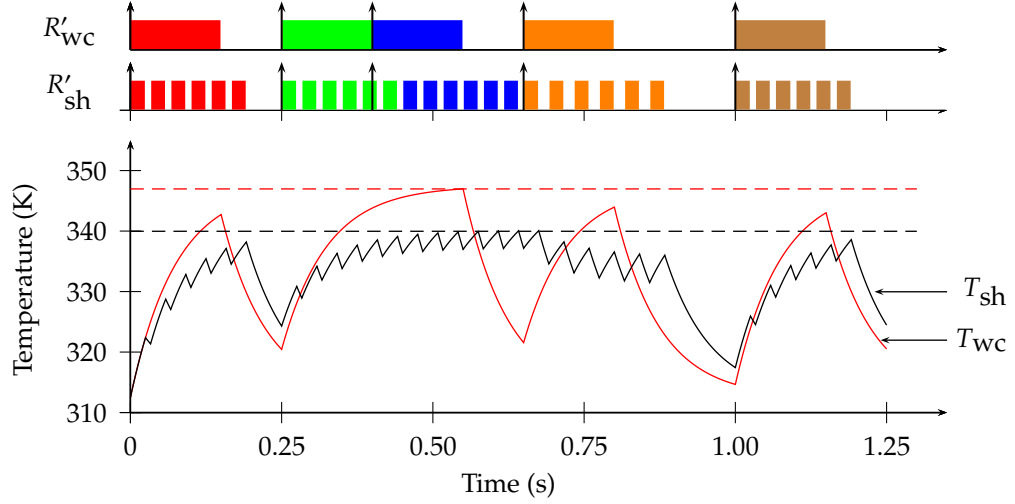
None of these studies focuses on guaranteeing hard real-time properties. Instead they optimize for the best-effort performance, for a given safe peak-temperature. In this chapter, our aim is to study the use of throttling as a DTM technique to simultaneously consider peak-temperature and the worst-case delay of any job. To this end, we advocate the use of *shapers* which originate from the networking domain [GGPS96]. We first illustrate their use with an example.

### 3.1.1 Motivating Example

**Example 3.1:** Consider a periodic task with period 0.25 s, execution time 0.15 s, a jitter 0.1 s, and relative deadline equal to period. Consider a specific trace of jobs of this task, with jobs arriving at times 0, 0.25, 0.4, 0.65, 1s. Let this trace be executed on a processor with thermal and power parameters (explained in Section 3.2) as listed in Table 3.1. Let the temperature of the processor at time 0 be 312.5 K.

For the specific trace of jobs in the above example, consider two





**Fig. 3.1** Motivating example for use of shapers to manage timing and peak-temperature objectives. In the plot of the output traces  $R'_{wc}$  and  $R'_{sh}$ , upward arrows denote job arrivals, colored blocks denote execution of the jobs, and blank areas denote times then the processor is idle.

traces with output arrival functions (defined in Appendix A.3) shown in Figure 3.1. The two output arrival functions are denoted  $R'_{wc}$  (for workload-conserving) and  $R'_{sh}$  (for shaped). In  $R'_{wc}$  the jobs are executed continuously until their completion. On the other hand, in  $R'_{sh}$  delays are inserted between “chunks” of the jobs. For both traces, we plot the temperature of the processor as a function of time. Whenever a job is executed the temperature rises, and whenever the processor is idle the temperature falls. As is evident from the figure, the inserted delays in  $R'_{sh}$  allow the processor to cool down intermittently. Consequently, the peak-temperature is higher for the output trace  $R'_{wc}$  (347 K as opposed to 340 K).

While it can reduce peak-temperature, insertion of delays must respect the deadlines of the jobs. In this particular example, deadlines of all jobs are satisfied in both the traces. In particular, for  $R'_{sh}$  the third job which arrives earlier with the maximum jitter, meets its deadline. Furthermore, the delays inserted between chunks of the jobs are variable. After the jitter in the third job, a larger delay is inserted between chunks of the fourth job, as no (negative) jitter is anticipated in the next job. This helps the processor to cool down further.

In conclusion, delays can be inserted between chunks of jobs, i.e., an input trace can be shaped, to reduce the peak-temperature in comparison to workload-conserving execution. However, such shaping should be carefully done. Deadlines of all jobs must be met. Furthermore, there are opportunities to anticipate future job arrival patterns and to exploit them to aggressively reduce the temperature. Nevertheless, the overhead

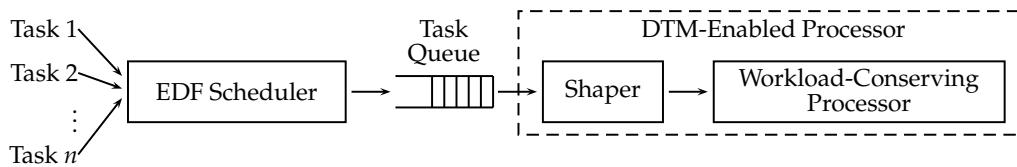


Fig. 3.2 Block diagram for the use of shapers for dynamic thermal management.

introduced by shaping must be managed.

### 3.1.2 Contributions

In this chapter, we propose the use of shapers to monitor job arrival times and execution demands, and to insert appropriate delays to reduce the peak-temperature of a processor. In particular, we recommend *leaky-bucket shapers* which can be efficiently implemented. Within the class of leaky-bucket shapers, we prove that a specific shaper, referred to as the *convex-hull shaper*, optimally minimizes the peak-temperature while meeting deadlines of all jobs. This optimality holds for the arrival curve representation of tasks, which can model variable job arrival patterns. We extend our results to non-zero timing and energy overheads in transitioning between the two states of the processor. We also consider shaping multiple tasks which are scheduled with an EDF scheduler.

In conclusion, the contribution of this chapter can be summarized in the block diagram of Figure 3.2. Multiple tasks are scheduled with an EDF scheduler. The resultant task-queue is interfaced with a shaper, which during run-time divides the jobs into smaller chunks and delays the effective arrival time of these chunks on to the processor. Equipped with such a shaper, a workload-conserving processor is enabled with the Dynamic Thermal Management (DTM) technique of throttling.

The rest of this chapter is organized as follows. We define the system model in Section 3.2. We derive the optimal convex-hull shaper and outline its implementation in Section 3.3. We extend the results to non-zero transition overheads and multiple streams of jobs in Section 3.4. We present evaluation results in Section 3.5. Finally we summarize in Section 3.6 and present the proofs of results in an appendix.

## 3.2 System Model

In this section, we define the processor, task and thermal models. We also outline the problem definition.

### 3.2.1 Processor Model

We consider a processor that executes jobs at a fixed frequency. When executing a job the processor is said to be in the *active mode* and consumes power  $P_{\text{act}}$ . When not executing any job the processor is said to be in the *idle mode* and consumes a lower power  $P_{\text{idl}}$ . The idle mode can be enforced, even when the task-queue is not empty, by techniques such as execution throttling [BM01] or fetch toggling [SSH<sup>+</sup>03]. The transition from active to idle modes has an overhead of  $t_{\text{tr}}$  time units, during which time the processor consumes the higher power  $P_{\text{act}}$ .

In current VLSI technology nodes, leakage power constitutes a significant fraction of the total power consumption (up to 40% [NSG<sup>+</sup>06]). As leakage power is highly sensitive to temperature changes, the power consumption must be considered as a function of the temperature. As argued in [LDSY07], a linear model works well in practice, i.e., the power consumption at a temperature  $T$  is given as

$$P(T) = \rho T + \omega, \quad (3.1)$$

for some positive parameters  $\rho$  and  $\omega$ . These parameters are different for the two processor modes, namely active and idle. We reference these parameters with sub-scripts *act* and *idl* to represent the mode of the processor. In conclusion, the processor model is given by the tuple  $\mathbf{P} = (\rho_{\text{act}}, \omega_{\text{act}}, \rho_{\text{idl}}, \omega_{\text{idl}}, t_{\text{tr}})$ .

### 3.2.2 Task Model

We model a task as a stream of jobs characterized by an arrival curve,  $\alpha$ , as defined in Appendix A.2. Each concrete trace of jobs is characterized by an input trace,  $R$ , that conforms to the arrival curve  $\alpha$ , as defined in Appendix A.2. Further, each task has a relative deadline  $D$ , which is the same for all jobs of that task. We assume that the deadlines of all jobs are met for a workload-conserving execution, i.e., in the absence of any shaping no job misses its deadline. From known results in Network Calculus [LBT01], this assumption is specified as the following relation

$$\alpha(\Delta - D) \leq \Delta, \quad \forall \Delta \geq D. \quad (3.2)$$

In conclusion, the task model is given by the tuple  $\tau = (\alpha, D)$ . We extend this model to consider multiple such tasks scheduled with an EDF scheduler in Section 3.4.2.

### 3.2.3 Thermal Model

For studies in architectural-level thermal management, heat flow is approximated by the Fourier's law of heat diffusion assuming that the

processor is a point source [HGV<sup>+</sup>06]. For such an approximation, the temperature of the processor,  $T$ , evolves according to the following differential equation

$$C \frac{dT}{dt} = -G(T - T_{\text{amb}}) + P(T), \quad (3.3)$$

where  $C$  and  $G$  are thermal model parameters,  $T_{\text{amb}}$  is the ambient temperature, and  $P(T)$  is the power consumed by the processor at temperature  $T$ . For the considered linear power model (3.1), the closed form solution to the differential equation (3.3), is given as

$$T(t) = T^\infty + (T(0) - T^\infty) \cdot e^{-a(t-t_0)}, \quad \forall t \geq 0, \quad (3.4)$$

where,

$$T^\infty = \frac{GT_{\text{amb}} + \omega}{G - \rho}, \quad (3.5)$$

$$a = \frac{G - \rho}{C}. \quad (3.6)$$

$T^\infty$  is referred to as the steady-state temperature, and  $a$  is referred to as the rate-constant. Both the parameters will be different for the two modes of the processor. We reference them with sub-scripts to represent the mode. In conclusion, the thermal model is given by the tuple  $\mathbf{T} = (T_{\text{act}}^\infty, a_{\text{act}}, T_{\text{idl}}^\infty, a_{\text{idl}})$ .

### 3.2.4 Problem Definition

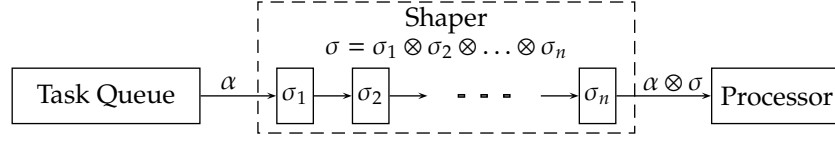
Given are a processor model  $\mathbf{P}$ , a task model  $\tau$ , and a thermal model  $\mathbf{T}$ . Under this assumption of (3.2), we are to design a run-time manager that chooses the mode of the processor (active or idle) at all times such that (a) all jobs complete within their deadlines, and (b) peak-temperature of the processor is minimized.

## 3.3 Optimal Leaky-Bucket Shaper

In this section, we assume negligible overhead when transitioning between the two processor modes. For this simplified setting, we derive the optimal shaper within the class of leaky-bucket shapers.

### 3.3.1 Introduction to Shapers

In the area of networking, traffic shaping is a well-studied technique to regulate flow of packets by buffering and delaying them [GGPS96].



**Fig. 3.3** Block diagram of serial composition of multiple shapers. The arrival curve at the input of the shaper is  $\alpha$  and at the output of the shaper is  $(\alpha \otimes \sigma)$  which does not exceed  $\sigma$ .

For instance, a shaper may ensure that the output stream has limited burstiness and thereby reduce the buffer-space required at a downstream node. When employing shapers for thermal management there is a slight difference. The temperature of the processor depends on how *long* the processor executes rather than how *many* jobs it executes. Hence, rather than the conventional approach of shaping based on the number of jobs (or packets), we consider shapers that shape the execution demand of jobs.

A shaper is characterized by a *shaping curve*, denoted  $\sigma$ . The shaper ensures that the trace of jobs at its output conforms to an arrival curve  $\sigma$ , independent of the input to the shaper. More specifically, the arrival curve at the output of a shaper is  $(\alpha \otimes \sigma)$  [Wan06], where  $\alpha$  is the arrival curve at the input and  $\otimes$  is as defined in Appendix A.1. A shaper is said to be *greedy* if it does not delay any job longer than is required to ensure the above condition.

Shapers can be composed serially, i.e., the output trace from one shaper can be the input trace of another shaper. So composing two shapers with shaping curves  $\sigma_1$  and  $\sigma_2$  results in a shaper with an effective shaping curve  $\sigma_1 \otimes \sigma_2$ . This is illustrated in the block diagram in Figure 3.3.

### 3.3.2 Leaky-Bucket Shapers

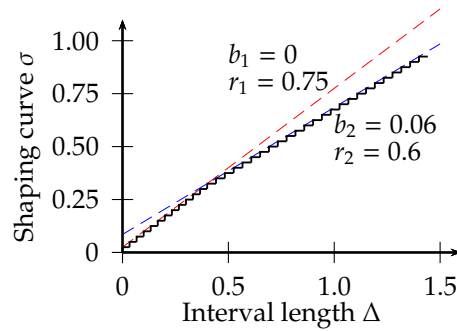
In this chapter, we restrict our study to leaky-bucket shapers, which are specific greedy shapers that can be efficiently implemented.

A *simple* leaky-bucket shaper is a greedy shaper, such that its shaping curve  $\sigma$  is given by three parameters, namely a *bucket-size*  $b$ , a *fill-rate*  $r$ , and a *leak-unit*  $u$ <sup>1</sup>, as

$$\sigma(\Delta) = \left\lfloor \frac{b + u + r\Delta}{u} \right\rfloor \times u. \quad (3.7)$$

For a *continuous approximation* of a simple leaky-bucket, i.e., in the limit as  $u \rightarrow 0$ , we have  $\sigma(\Delta) = b + r\Delta$ .

<sup>1</sup> A simple leaky-bucket shaper can be visualized as follows. Consider a bucket of size  $b$ , with an inlet filling the bucket at a constant rate  $r$ , and an outlet greedily leaking water at the granularity of  $u$ . When the bucket is full, the water from the inlet “overflows”.



**Fig. 3.4** Example shaping curve of a leaky-bucket shaper. The leak-unit for both shapers is  $u = 0.025$  s. Notice that the shaping curve, shown in a black solid line, is the minimum of the shaping curves of the two constituent simple leaky-bucket shapers, whose continuous approximations (increased by  $u$ ) are shown in dashed red and blue lines.

Serially composing simple leaky-bucket shapers, as shown in Figure 3.3, is particularly interesting. If two simple leaky-bucket shapers with shaping curves  $\sigma_1$  and  $\sigma_2$ , and the same leak-unit are composed, then the resultant shaping curve is given as  $\min(\sigma_1, \sigma_2)$ . As we will see in Algorithm 2, this simplified composition enables an efficient implementation. We refer to all shapers generated by serially composing simple leaky-bucket shapers as leaky-bucket shapers. The leak-unit of a leaky-bucket shaper equals the identical leak-units of its constituent simple leaky-bucket shapers. We illustrate the composition of simple leaky-bucket shapers with the following example.

**Example 3.2:** Consider two simple leaky-bucket shapers with  $b_1 = 0$ ,  $r_1 = 0.75$  s,  $b_2 = 0.06$  s,  $r_2 = 0.6$  s, and  $u_1 = u_2 = 0.025$  s. Serially composing these shapers generates a leaky-bucket shaper whose shaping curve is shown in Figure 3.4. This shaping curve is used to shape the trace of jobs in Example 3.1, as was illustrated in Figure 3.1.

### 3.3.3 Analysis with Leaky-Bucket Shapers

We now present analytical results which help us identify the optimal leaky-bucket shaper for the said problem definition. Recall that for optimality we need to meet the deadlines of all jobs and minimize peak-temperatures. In the following result we characterize worst-case delay when using shapers.

**Lemma 3.1:** Given a task with an arrival curve  $\alpha$ , that is shaped with a shaping curve  $\sigma$  and then executed on a dedicated processor. The maximum delay of any job of this task, denoted  $d^{\max}$ , is given by

$$d^{\max} = \text{Del}(\alpha(\Delta), \sigma(\Delta)) + \text{Del}(\sigma(\Delta), \Delta). \quad (3.8)$$

The above result is interpreted as follows. The maximum delay of the job is the sum of two delays, namely the maximum delay at the shaper and the maximum delay at the processor. These two delays are given by the two summands in (3.8). Each summand uses the Del operator which is as defined in Appendix A.3.

In the following result we characterize the worst-case temperature when using shapers by simulating a specified critical trace. This trace is characterized by its input arrival function derived from the arrival curve of the task and the shaping curve of the shaper.

**Lemma 3.2:** *Given a task with an arrival curve  $\alpha$ , that is shaped with a shaping curve  $\sigma$  and then executed on a processor. The worst-case temperature is the temperature of the processor after executing a critical trace of jobs with arrival function  $R^*(t)$  given as*

$$R^*(t) = (\alpha \otimes \sigma)(t_{\max}) - (\alpha \otimes \sigma)(t_{\max} - t), \quad t \in [0, t_{\max}], \quad (3.9)$$

where  $[0, t_{\max}]$  defines the time-domain of interest.

From the above two lemmas, we derive the optimal shaper in the following result.

**Theorem 3.1:** *Given is a task with an arrival curve  $\alpha$  and relative deadline  $D$ . Then the leaky-bucket shaper which minimizes the peak temperature of a processor executing this task and meets all deadlines has a shaping curve*

$$\sigma_{ch} = \text{ConvexHull}(\alpha(\Delta - D)), \quad (3.10)$$

where  $\text{ConvexHull}$  is the smallest convex envelope of all points in its argument.

We refer to the shaper defined in (3.10) as the *convex-hull shaper* and its shaping curve as the *convex-hull shaping curve*. A convex-hull shaper is a leaky-bucket shaper: Each piecewise-linear segment of its shaping curve can be realized by a simple leaky-bucket shaper with zero leak-unit. We illustrate this with an example.

**Example 3.3:** *Consider the periodic task from Example 3.1. The arrival curve of this task is shown in Figure 3.5(a). For the deadline of 0.25 s, the convex-hull shaping curve is shown in Figure 3.5(b). Note that the shaping curve is composed of two piecewise-linear segments which can be realized by composing two simple leaky-bucket shapers. The continuous approximation of the leaky-bucket shaper ( $u = 0$ ) shown in Figure 3.4 is the convex-hull shaper for this task.*

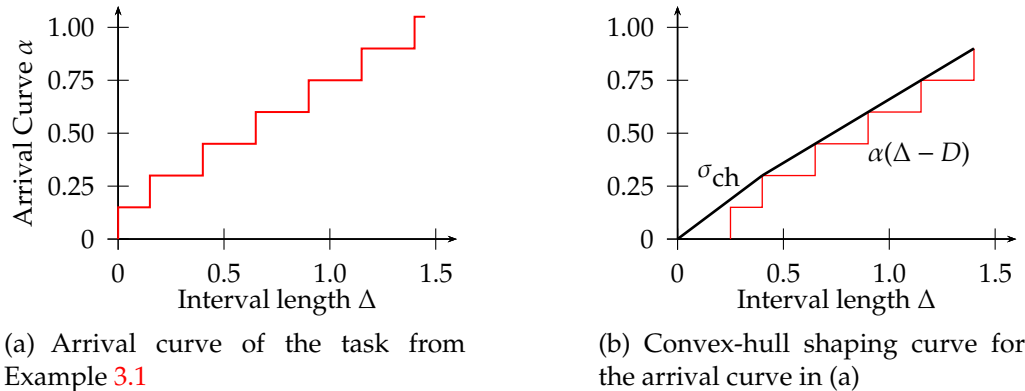


Fig. 3.5 Illustration of a convex-hull shaper.

### 3.3.4 Implementing the Convex-Hull Shaper

We motivated the exclusive focus on leaky-bucket shapers because they can be efficiently implemented. We present in Algorithm 2 our implementation of such a shaper. There are two requirements of the implementation: (a) access to a programmable timer which generates an interrupt when the timer underflows, and (b) interrupts when the task-queue becomes empty or becomes non-empty. Both these requirements are met for most CPSs.

As described in the algorithm, the shaper maintains a fill-value, denoted as  $f$ , for each constituent simple leaky-bucket shaper of the convex-hull shaper. These fill-values are modified in two ways: (a) they are increased at the constant-rate of  $r$  up to the maximum value of  $(b + u)$  (line 6), and (b) they are reduced by  $u$  whenever the processor transitions to the active mode (line 20). The shaper allows a task to run only when the fill-values of *all* the simple leaky-bucket shapers is greater than or equal to  $u$  (line 8). If this is not the case, then the length of time for which the processor is to be forced to be idle is computed directly from the fill-values (line 13). In addition, whenever the task-queue is empty the processor remains in the idle mode.

## 3.4 Extensions of Convex-Hull Shaper

In this section, we extend the results for the previous section to consider (a) non-zero transition overhead between the two modes of the processor, and (b) multiple streams of jobs scheduled with an EDF scheduler.



**Algorithm 2: Implementation a Convex-Hull Shaper.**


---

**Input:** Convex-hull shaper is composed of a set of simple leaky-bucket shapers  $\mathbf{S}$ . Each shaper  $S_i \in \mathbf{S}$  is given by bucket-size  $b_i$ , fill-rate  $r_i$ , and leak-unit  $u$ .

```

1 Procedure Initialize()
2    $t \leftarrow 0, t' \leftarrow 0$ 
3    $f_i \leftarrow b_i + u, \forall S_i \in \mathbf{S}$ 
4   Shaper()
5 Procedure Shaper()
6    $f_i \leftarrow \min(f_i + (t - t') \cdot r_i, b_i + u), \forall i \in \mathbf{S}$ 
7    $t' \leftarrow t$ 
8   if  $\min_{i \in \mathbf{S}} f_i \geq u$  then
9     active  $\leftarrow$  True
10    RunTask()
11  else
12    active  $\leftarrow$  False
13     $t_{\text{idle}} \leftarrow \max_{S_i \in \mathbf{S}} ((u - f_i) / r_i)$ 
14    Program timer with  $t_{\text{idle}}$ 
15    Put processor in idle mode
16 Procedure RunTask()
17   if task queue is empty then
18     Put processor in idle mode
19   else
20      $f_i \leftarrow f_i - u, \forall S_i \in \mathbf{S}$ 
21     Program timer with  $u$ 
22     Execute task
23 Interrupt Service Routine TimerUnderflow()
24   if active then
25     active  $\leftarrow$  False
26     Shaper()
27   else
28     active  $\leftarrow$  True
29     RunTask()
30 Interrupt Service Routine TQNonEmpty()
31   if active then
32     Shaper()
33 Interrupt Service Routine TQEmpty()
34   Put processor in idle mode

```

---

**3.4.1 Non-Zero Transition Overhead**

During the transition between modes, for  $t_{\text{tr}}$  time units the processor executes no jobs but consumes the power according to the active mode. Given this, we can model the transition overhead as part of the execution demand of the jobs. For a given leak-unit  $u$  and arrival curve  $\alpha$ , let  $\alpha_{\text{oh}}$  denote the *overhead-aware arrival curve* of the task. The leak-unit represents the smallest unit of time between transitions in modes of the processor. Then, the total overhead in time, for any interval of length  $\Delta$ , is at most

$\lceil (\alpha_{\text{oh}}/u) \rceil t_{\text{tr}}$ . From this, we have the following condition on  $\alpha_{\text{oh}}$ .

$$\alpha_{\text{oh}} \geq \alpha + \left\lceil \frac{\alpha_{\text{oh}}}{u} \right\rceil \cdot t_{\text{tr}}. \quad (3.11)$$

For  $\alpha_{\text{oh}}$  to be a valid arrival curve, it should be sub-additive [TCN00]. Enforcing this along with (3.11), we arrive at the following definition of the overhead-aware arrival curve.

$$\alpha_{\text{oh}} = \left\lceil \frac{\alpha}{u - t_{\text{tr}}} \right\rceil \cdot u. \quad (3.12)$$

We can now modify the definition of the convex-hull shaper, presented in (3.10), to be overhead-aware as follows.

$$(\sigma_{\text{ch}})_{\text{oh}} = \text{ConvexHull}(\alpha_{\text{oh}}(\Delta - D)). \quad (3.13)$$

By absorbing the transition overhead into the arrival curve we benefit from the validity of the established results. Indeed, optimality result of Theorem 3.1 holds for the shaping curve  $(\sigma_{\text{ch}})_{\text{oh}}$ .

### 3.4.2 Shaping Jobs from Multiple Tasks

So far, we have considered a single stream of jobs, when all jobs have the same deadline. We extend this to several streams of jobs, with different deadlines, scheduled with an EDF scheduler. The EDF policy is applied to all incoming jobs, and then the shaper works on the resultant queue, as illustrated in the block diagram in Figure 3.2.

Let a task-set  $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$  be given, where  $\tau_i$  has an arrival curve  $\alpha_i$  and a relative deadline  $D_i$ . For such a task-set we can obtain the demand bound function, which was introduced in Definition 2.1, as

$$\text{dbf}(\Delta) = \sum_{\forall \tau_i \in \tau} \alpha_i(\Delta - D_i). \quad (3.14)$$

We extend the definition of the convex-hull shaper from (3.10) to the following.

$$\sigma_{\text{ch}} = \text{ConvexHull}(\text{dbf}) \quad (3.15)$$

The results presented in Section 3.3 hold under this extension. In conclusion, EDF scheduler enables us to coalesce the different tasks into an equivalent task as characterized by the dbf, and we design the convex-hull shaper for this dbf.

Power parameters		Thermal parameters	
$\rho_{\text{idl}}$	0.1 W/K	$G$	0.3 W/K
$\omega_{\text{idl}}$	-25 W	$C$	0.03 J/K
$\rho_{\text{act}}$	0.1 W/K	$a$	0.1 ms
$\omega_{\text{act}}$	-11 W	$T_{\text{amb}}$	300 K

Tab. 3.1 Example processor parameters.

## 3.5 Experimental Results

We consider a processor with power parameters obtained from [YCTK10]. The thermal parameters are typical parameters sourced from [SSH<sup>+</sup>03]. The power and thermal parameters are listed in Table 3.1.

We consider a video-conferencing application that consists of three tasks: a video codec, an audio codec and a network process. All three tasks are periodic but can exhibit jitter. The timing properties of these tasks are listed in Table 3.2.

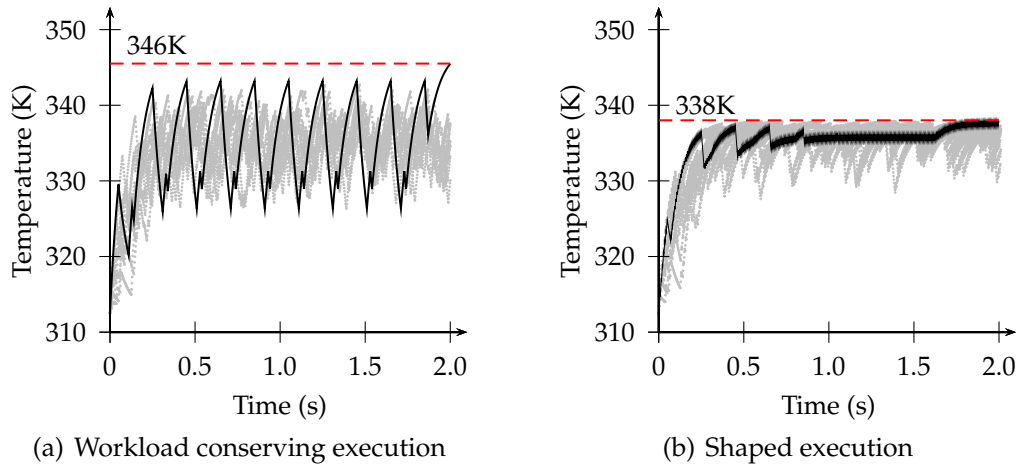
	Video	Audio	Network
Period	0.2	0.2	0.1
Jitter	0.05	0.05	0.03
WCET	0.06	0.03	0.02
Deadline	0.2	0.2	0.1

Tab. 3.2 Example task parameters of a video-conferencing application. All times are in seconds.

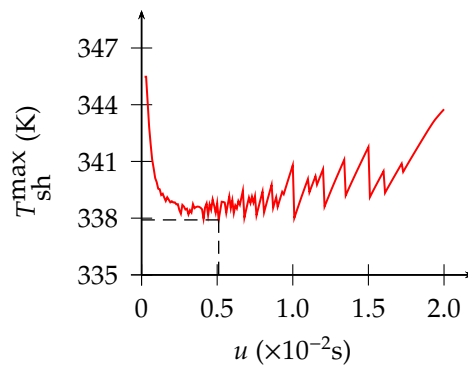
### 3.5.1 Computing the Optimal Shaper

We first compute the peak-temperature of the processor when executing the three tasks, for a workload-conserving execution, i.e., with no shaping. Employing Lemma 3.2, we obtain the peak-temperature as  $T_{\text{WC}}^{\text{max}} = 346$  K. We illustrate the validity of this bound by generating 20 random traces of jobs and plotting the evolution of temperature for these traces in Figure 3.6(a). We also plot the evolution of the temperature for the worst-case trace (that has the highest peak-temperature) as defined in (3.9).

We now design the shaping curve by finding the optimal convex-hull shaper. Note that in this example, we consider multiple tasks and have a non-zero transition overhead. Thus, we have to choose the right leak-unit  $u$  for the convex-hull shaper. To this end, we first compute the overhead-aware arrival curves  $\alpha_{\text{oh}}$ , then the demand bound function dbf, and finally the convex-hull shaper. For different values of  $u$  in the range  $[0.2, 2]$ s, we compute the peak-temperature using Lemma 3.2. The obtained values are plotted in Figure 3.7.



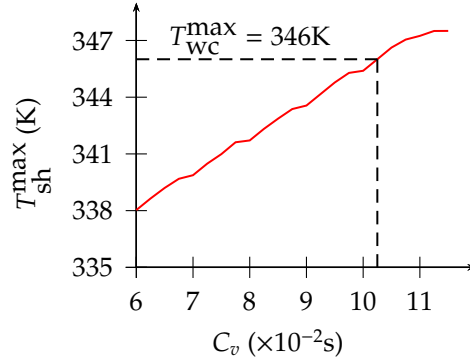
**Fig. 3.6** Peak-temperature for workload-conserving and shaped executions. The gray dots denote the temperature plots for 20 randomly generated traces. The dark lines denote the temperature plots for the worst-case trace as defined in Lemma 3.2.



**Fig. 3.7** Optimal choice of the leak-unit for the convex-hull shaper for the video-conferencing application.

It is observed that the peak-temperature is larger for too small and too large values of  $u$ . This is explained as follows. When  $u$  is too small, the processor switches modes often and incurs a large overhead. On the other hand, if  $u$  is too large then enforcing the large granularity of  $u$  can increase the overhead-aware arrival curve (3.12). From the plot, the optimal value of  $u = 0.51$  s with a peak-temperature  $T_{sh}^{\max} = 338$  K. We illustrate this bound by simulating 20 randomly generated traces and the worst-case trace from (3.9) as shown in Figure 3.6(b).

In conclusion, for the video conferencing application, with the obtained shaping curve we reduce the peak-temperature by 8 K, in comparison to the workload-conserving case.



**Fig. 3.8** Dependence of the peak-temperature on the WCET of the video codec task. For the given peak-temperature of 346K, shaping tasks can support a much higher WCET than workload conserving execution.

### 3.5.2 Exploiting the Lower Peak-Temperature

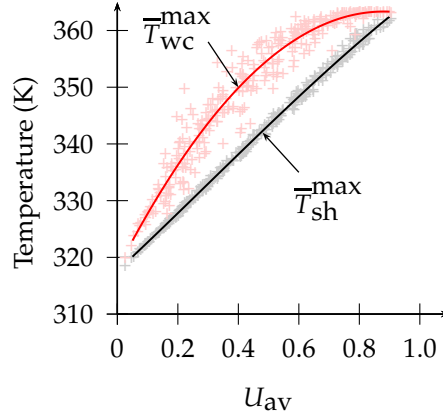
The advantage of shaping in lowering the peak-temperature can be translated to an improved workload supported for a given peak-temperature. Consider for instance that to support higher video resolution the WCET of the video codec task, denoted  $C_v$ , is to be increased. We identify the largest  $C_v$  that can be supported with shaping, such that the peak-temperature is smaller than the peak-temperature with no shaping  $T_{wc}^{\max} = 346$  K.

We vary  $C_v$  in the range  $[0.06, 0.12]$ s, and for each value obtain the optimal peak-temperature with a convex-hull shaper<sup>2</sup>. From the obtained plot shown in Figure 3.8 we see that we can increase  $C_v$  to 0.102 s with the peak-temperature  $T_{sh}^{\max} \leq 346$  K. This is a substantial increase, of more than two-thirds, compared to the original value of 0.06 s. In conclusion, shaping tasks can be used to support a significantly higher execution demand for a given bound on the peak-temperature, in comparison to workload-conserving execution.

### 3.5.3 Comparison for Randomly Generated Task-Sets

To quantify the advantage of shaping, independent of the task-set, we perform experiments with randomly generated task-sets. We consider two periodic tasks with randomly generated periods  $p_1$  and  $p_2$ , with mean values of 0.2s each, and jitters  $p_1/2$  and  $p_2/2$ , respectively. The WCETs of the two tasks,  $C_1$  and  $C_2$ , are randomly generated with mean values of  $p_1/4$  and  $p_2/4$ , respectively. We define the average utilization of a task-set as  $U_{av} = C_1/p_1 + C_2/p_2$ . We generate 500 such task-sets. For each case,

<sup>2</sup>For each value of  $C_v$ , we compute the overhead-aware arrival curves, the demand bound function, the optimal value of  $u$  (like in Figure 3.7), the convex-hull shaper, and then the peak-temperature from Lemma 3.2.



**Fig. 3.9** Advantage of shaping for randomly generated task-sets. Light points denote the peak-temperatures for 500 randomly generated task-sets. The dark lines denote the mean of the peak-temperatures, at each value  $U_{av}$ .

we obtain the peak-temperatures for the workload-conserving execution ( $T_{wc}^{\max}$ ) and for shaping with the optimal convex-hull shaper ( $T_{sh}^{\max}$ ). We plot  $T_{wc}^{\max}$  and  $T_{sh}^{\max}$  against  $U_{av}$  for the generated task-sets in Figure 3.9. We denote the mean values for a given  $U_{av}$  as  $\bar{T}_{wc}^{\max}$  and  $\bar{T}_{sh}^{\max}$ .

As expected, both  $\bar{T}_{wc}^{\max}$  and  $\bar{T}_{sh}^{\max}$  are increasing functions of the utilization  $U_{av}$ . In all cases,  $\bar{T}_{sh}^{\max} < \bar{T}_{wc}^{\max}$ , with a significant average difference of 8.8 K across all the task-sets.

We can alternatively interpret this improvement in performance terms for a given allowed peak-temperature  $T^{\max}$ . For the considered setup, if the cooling system is efficient enough to allow for a very high temperature  $T^{\max} > 360$  K, then shaping tasks is not important as workload conserving execution itself supports a large utilization. For very small temperature bounds  $T^{\max} < 330$  K, again shaping does not provide much benefit as both methods support only small utilizations. However, for moderate values of  $T^{\max}$  between 330 K and 360 K, shaping tasks leads to large improvements in the supported utilization. From our data, on average, shaping tasks substantially increases the supported utilization by over 40 % in comparison to the workload conserving execution.

Finally, with shaping an increase in utilization of the task-set comes at the cost of almost linear increase in the peak-temperature. In contrast  $\bar{T}_{wc}^{\max}$  is concave, reaching higher peak-temperatures for low utilizations.

From these experiments, we conclude that shaping of tasks is an effective and efficient method to implement dynamic thermal management for real-time tasks.

## 3.6 Summary

Leaky-bucket shapers are efficient run-time managers which monitor job arrival times, and insert appropriate delays such that the output trace conforms to a specific arrival curve. By extending such shapers to monitor and shape execution demands of jobs, rather than their number, they can be employed to implement the dynamic thermal management technique of throttling. We showed that the parameters of the leaky-bucket shapers can be carefully chosen to meet all timing constraints and minimize the peak-temperature. The experimental results numerically quantified the advantage of shaping over workload-conserving execution.

The design pattern identified in the previous chapter repeats. In this chapter, we composed simple leaky-bucket shapers to design a run-time manager, called the convex-hull shaper, which is both efficient to implement and yet can be tuned for the specific parameters of a given system. We thus conclude this part of the thesis by noting that modularly composing the run-time manager from efficient constituents is a strongly motivated design pattern to absorb variability in timing properties in CPSs.

## Appendix

### Proof of Lemma 3.1

The maximum delay of any job is the sum of two delays: (a) the maximum delay introduced by the shaper, and (b) the maximum delay in queuing and executing on the processor. From Network Calculus [LBT01], the maximum delay introduced by the shaper is given by  $\text{Del}(\alpha, \sigma)$ . Also, the arrival curve at the output of the shaper (and input of the processor) is given as  $\alpha' = \alpha \otimes \sigma \leq \sigma$ . Then, the worst-case delay on the processor is  $\text{Del}(\alpha', \beta) \leq \text{Del}(\sigma, \beta)$ , where  $\beta$  is the service curve of the processor. In this chapter, we assume a fully available processor, i.e.,  $\beta(\Delta) = \Delta$ . Thus, the maximum delay on the processor is  $\text{Del}(\sigma(\Delta), \Delta)$ .  $\square$

### Proof of Lemma 3.2

Given a single stream of jobs with an arrival curve and a dedicated processor executing it at a constant speed, [RYB<sup>+</sup>11] presents a technique to compute the worst-case temperature. In particular, it defines a critical trace of jobs which when simulated gives the worst-case temperature. For the arrival curve at the input of the processor equal to  $(\alpha \otimes \sigma)$  this trace is defined exactly as in (3.9).  $\square$

**Proof of Theorem 3.1**

From (3.8), any shaping curve  $\sigma$  that satisfies the deadline  $D$  is constrained as follows.

$$\begin{aligned} & \text{Del}(\alpha, \sigma) \leq D, \\ \Rightarrow & \quad \sigma(\Delta) \geq \alpha(\Delta - D), \quad \forall \Delta \geq 0. \end{aligned} \quad (3.16)$$

A shaper which is the composition of leaky-bucket shapers has a shaping curve which is a piecewise linear concave function. The smallest such shaping curve satisfying (3.16), is given as the defined shaping curve  $\sigma_{\text{ch}}$  in (3.10). Thus any other valid shaping curve  $\sigma'$  must satisfy  $\sigma \geq \sigma_{\text{ch}}$ . We need to show that the worst-case temperature with the shaping curve  $\sigma'$  cannot be smaller than with the shaping curve  $\sigma_{\text{ch}}$ . From Lemma 2 in [RYB<sup>+</sup>11] we know that the worst-case temperature cannot be lower with a higher arrival curve. With the shaping curve  $\sigma'$ , the arrival curve at the input of the processor is not smaller than the arrival curve with the shaping curve  $\sigma_{\text{ch}}$ . Thus, the worst-case temperature cannot be smaller with any other shaping curve that satisfies the deadline constraint.  $\square$



## **Part II**

# **Bounding Variability in Formal Analysis**



# 4

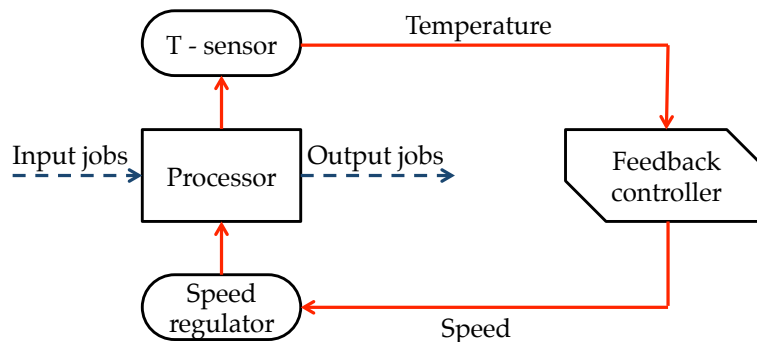
## Analysis of Temperature-Based Feedback Control of Speed

### 4.1 Introduction

As motivated in the previous chapter, managing on-chip temperature is a requirement in most modern processors. Such management affects the timing guarantees provided by the CPS. Consequently, both timing and temperature objective functions can be *simultaneously* considered in optimizing run-time managers such as the shapers from the previous chapter. Such run-time managers would dynamically adapt the thermal management while satisfying the timing requirements of executing jobs.

An alternate view is to deliberately isolate the two objectives of satisfying timing and temperature constraints. Such an isolation may be motivated for the following two reasons.

- High temperatures affect the physical safety of the CPS. If due to a software bug the execution demand of a task is larger than expected, the temperature constraint may be violated leading to an unavailable processor. This is particularly significant as *all* applications sharing the processor are stopped due to a physical failure on the processor. In such situations, a higher *criticality* might be accorded to satisfying the temperature constraint.
- The parameters which need to be modeled and analyzed for simultaneously considering temperature and timing objectives span widely. They consist of the physical properties of heat diffusion, the architectural properties of power generation, and the software



**Fig. 4.1** Block diagram of reactive speed scaling. The temperature of the processor is constantly sensed. The sensed temperature is the input to a feedback controller, which computes the speed the processor. The speed is then actuated by a speed regulator. Independently, the processor executes a stream of incoming jobs.

properties of execution demand of jobs. It is not uncommon to expect design-time models of these parameters to be inaccurate. In the presence of such uncertainty, *robustness* can be derived with the use of feedback control in the run-time manager. Since at all times the temperature can be sensed, it is more intuitive to design a feedback controller to exclusively consider the temperature objective.

The above two reasons, motivate assigning *primacy* to satisfaction of the temperature constraints with a feedback controller. In other words, independent of the executing tasks, we require that the temperature constraints must be satisfied. Indeed, the timing properties of the tasks will depend on the chosen controller. In this chapter, we study how to compute such timing properties.

We consider *temperature-based feedback control of speed*, which was first proposed by Wang and Bettati in [WB08]. This setup is depicted in the block diagram of Figure 4.1. A processor is executing a stream of jobs. The speed of the processor is regulated by a feedback controller whose input is the temperature of the processor. As the temperature rises, the controller will slow down the processor, and vice versa. Typically, a slower speed corresponds to a smaller power consumption, which can reduce the rate of temperature increase. If such a controller is carefully designed, the temperature of the processor can be guaranteed not to exceed a stipulated bound, independent of the jobs.

The setup in Figure 4.1 is an example of system where a run-time manager is designed to explicitly manage the temperature, in the presence of variability and uncertainty. For such a *given* run-time manager, to provide hard timing guarantees, we have to derive an efficient and accurate *analysis procedure*. Such an analysis is not straightforward for the following three reasons.

- The analysis must consider how the controller regulates the speed. For instance, in Intel Sandy Bridge processors, Turbo Boost runs the processor at a higher frequency. Given the thermal constraints, this lasts only for up to 30 – 60s [NRAW11].
- The analysis must consider when earlier jobs were executed. For instance, a new job arriving when the processor is idle may run at a higher or lower speed based on when and for how long the processor was busy earlier. Thus, the classical assumption of working with busy windows does not apply. This can be challenging if the jobs exhibit variability in arrival times and execution demands.
- Finally, the results will also depend on *initial conditions*, i.e., the temperature of the processor at time 0. For instance, timing guarantees may not be satisfied if the processor is too hot initially.

To conclude, the presence of a feedback controller to manage the processor temperature further complicates the challenge of computing timing guarantees.

### 4.1.1 Contributions

In this chapter, we identify an analysis procedure to compute hard timing guarantees for a processor with feedback control of speed executing a single stream of jobs in First-Come-First-Serve (FCFS) order. This procedure can consider variability in arrival times and execution demands of jobs as modeled by an arrival curve. Further, it can consider different initial temperatures of the processor. Finally, the procedure is efficient to enable design space exploration inherent in such multi-objective settings.

More specifically, we identify a *critical* trace of jobs, conforming to the given arrival curve. By simulating this trace on a *thermally-clipped* hypothetical model of the processor, we compute the bound on the maximum delay of any job. This result depends on a proposed *monotonicity principle* which holds under commonly satisfied assumptions on the power consumption, heat diffusion, and the feedback controller.

The defining characteristic of our result is that the critical trace of jobs does not depend on the thermal, power, or controller models, or the initial temperature of the processor. We also show that for a particular class of faulty temperature sensors, the critical trace does not depend on the error introduced by the sensors.

The rest of the chapter is organized as follows. We detail the system model and the problem statement in Section 4.2. In Sections 4.4, 4.5, and 4.6 we solve the problem for maximum, minimum, and any initial temperature

of the processor, respectively. We consider faulty temperature sensors in Section 4.7. We present experimental results in Section 4.8. We summarize in Section 4.9 and provide proofs of results in an appendix.

## 4.2 System Model

In this section, we detail the processor model, the thermal model, the control law, and the task model. We also define the problem statement.

### 4.2.1 Processor Model

We denote a processor with feedback control of speed as  $\mathbf{P}$ . We denote the speed, temperature and power consumption of  $\mathbf{P}$  at any time  $t$  as  $s(t)$ ,  $T(t)$  and  $P(t)$ , respectively.

The speed of the processor can be regulated to any speed belonging to a given set of allowed speeds, denoted as  $\mathbf{S}$ . If and only if there is no pending workload, the processor is said to be idle with a speed 0. We do not consider any time or energy overhead in switching between different speeds. The execution time of a job scales linearly with the processing speed. For instance, at twice the speed a job completes in half the time.

The power consumption of the processor is modeled a convex increasing function of the speed, and is given as

$$P(t) = \phi(s(t)). \quad (4.1)$$

The convexity of  $\phi$  can be interpreted thus: The total energy consumed to execute a certain number of processor cycles is monotonically non-decreasing with the speed. Typically,  $\phi(s)$  is of the form  $(p + s^q)$ , where  $p > 0$  is the power consumption of idle processor, and the exponent  $q$  is in the range  $[2, 3]$ . Indeed, such a function is convex.

To conclude, the processor model is given by the set of speeds  $\mathbf{S}$  and the convex increasing function  $\phi$ .

### 4.2.2 Thermal Model

As in the previous chapter, we assume that heat flow is approximated by the Fourier's law of heat diffusion assuming that the processor is a point source [HGV<sup>+</sup>06]. For such an approximation, the temperature of the processor,  $T$ , evolves according to the following differential equation

$$C \frac{dT}{dt} = -G(T(t) - T_{\text{amb}}) + P(t), \quad (4.2)$$

where  $C$  and  $G$  are thermal model parameters and  $T_{\text{amb}}$  is the ambient temperature. In the time interval  $[t_1, t_2]$ , if the speed of the processor is set to  $s$  and there is pending workload, then the closed form solution of the above differential equation is given as

$$T(t) = T^\infty(s) + (T(t_1) - T^\infty(s)) \cdot e^{-a(t-t_1)}, \quad \forall t \in [t_1, t_2], \quad (4.3)$$

where  $T^\infty(s)$  is the steady-state temperature at speed  $s$ , and  $a$  is the thermal rate-constant. These parameters are given as

$$T^\infty(s) = T_{\text{amb}} + \frac{\phi(s)}{G}, \quad (4.4)$$

$$a = \frac{G}{C}. \quad (4.5)$$

To conclude, the thermal model is given by the parameters  $G$ ,  $C$ , and  $T_{\text{amb}}$ .

### 4.2.3 Control Law for Speed Scaling

The control law is the relation used by the feedback controller to set the speed of  $\mathbf{P}$  as a function of its temperature. This law is specified by the function  $f$ , where

$$s(t) = f(T(t)). \quad (4.6)$$

We assume that  $f$  is monotonically non-increasing, i.e., the speed at higher temperatures cannot be higher. This models an intuitive class of control laws where speed scaling is used to limit the temperature.

Once the temperature of the processor is high and there is still pending workload, there are two options for the feedback controller. Firstly, it can alternate between two speeds such that the temperature oscillates within the given bound of a safe temperature. Alternatively, the feedback controller can maintain a constant speed such that the temperature saturates to the steady-state temperature at that speed. We assume that controller follows the latter option, as it is the commonly adopted for instance in the Intel® Sandy Bridge processors [NRAW11].

How does this assumption constrain the control law  $f$ ? Let the maximum temperature guaranteed by the controller be denoted  $T_{\text{peak}}$ . Then, once the processor reaches this temperature, it must execute at a constant speed, say  $s^*$ , such that the temperature does not either increase or decrease. Thus, we have the following condition on  $f$ .

$$f(T_{\text{peak}}) = s^*, \quad (4.7)$$

where,

$$T^\infty(s^*) = T_{\text{peak}}. \quad (4.8)$$

Given the monotonicity of the  $f$ , we have the following condition.

$$f(T) \geq s^*, \quad \forall T < T_{\text{peak}}. \quad (4.9)$$

From the above it follows that the  $s^*$  is the slowest-speed assigned by the controller and is thus denoted as  $s^{\text{min}}$ .

#### 4.2.4 Task Model

In this chapter, we consider a single stream of jobs executed on the processor in First-Come-First-Serve (FCFS) order. The execution demand of jobs is characterized by an arrival curve, denoted  $\alpha$ , and as defined in Appendix A.2. As noted earlier, the arrival curve can represent variability in job arrival times and execution demands of jobs. In contrast to the earlier chapters, execution demand is not expressed by the amount of time required for a job to complete execution. Instead, we use the number of processor cycles. This is necessary as a single job may execute at multiple processing speeds due to the speed scaling. The execution time of a job can be obtained from the execution demand and the speed of the processor. As an example, a job with an execution demand  $1 \times 10^8$  cycles will require 1 s to execute on a processor running at a constant speed of 100 MHz.

#### 4.2.5 Problem Statement

Given  $\mathbf{P}$  with a set of allowed speeds  $\mathbf{S}$ , the power consumption as a function of speed  $\phi$ , the thermal parameters  $G$ ,  $C$  and  $T_{\text{amb}}$ , a control law for speed scaling  $f$ , and the initial temperature  $T(0)$ . Given also a task with the arrival curve  $\alpha$  which is executed in FCFS order on  $\mathbf{P}$ . For some given  $t_{\text{hor}} > 0$ , we want to compute an upper-bound on the delay (or the response time) of any job that arrives in the interval  $[0, t_{\text{hor}}]$ , for any trace conforming to the arrival curve. This upper-bound is denoted as  $d^{\text{max}}$ .

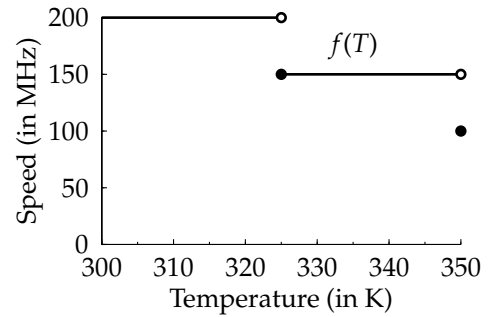
### 4.3 Illustrating Example

In this section, we will illustrate the working of a processor with feedback control of speed for an example trace of jobs, which is depicted in the block diagram of Figure 4.1.

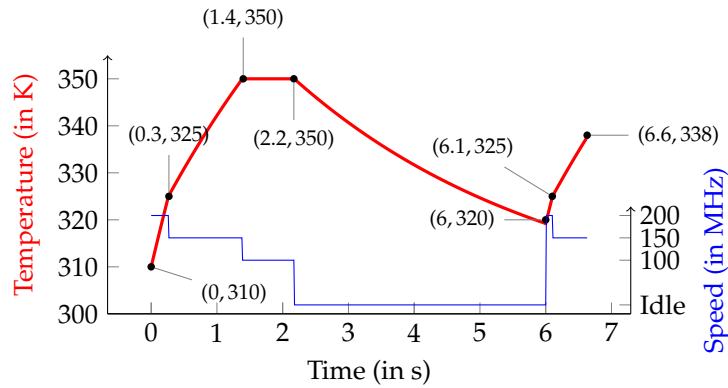
**Example 4.1: (Processor Model)** Consider a processor with thermal parameters  $G$ ,  $C$ , and  $T_{\text{amb}}$  and the function  $\phi$  as specified in Figure 4.2. The control law,  $f$ , is also plotted in the figure. With these parameters, the controller guarantees that the temperature of the processor does not exceed  $T_{\text{peak}} = 350 \text{ K} = T^{\infty}(s^{\text{min}})$ , where  $s^{\text{min}} = 100 \text{ MHz}$ .



$$\begin{aligned} \mathbf{S} &= \{0, 100, 150, 200\} \text{ MHz} \\ \phi(s) &= 2 + 12.5 \times \left( \frac{s}{100 \text{ MHz}} \right)^{2.3} \text{ W} \\ G &= 0.25 \text{ W/K} \\ C &= 1 \text{ J/K} \\ T_{\text{amb}} &= 292 \text{ K} \end{aligned}$$



**Fig. 4.2** Parameters of a processor with feedback control of speed. Note that the  $\phi(s)$  is a convex increasing function, and  $f(T)$  is a monotonically non-increasing function.



**Fig. 4.3** Illustration of working of a processor with feedback control of speed. The temperature of the processor is plotted in a thick red line, while the speed is plotted in a thin blue line. Times and temperatures at which the speed is changed are noted.  $J_1$  finishes at 2.2 s with a response time of 2.2 s, while  $J_2$  finishes at 6.6 s with a response time of 0.6 s.

**Example 4.2:** Consider two jobs  $J_1$  and  $J_2$  arriving at times 0 s and 6 s, respectively. The execution demands of the jobs are  $3 \times 10^8$  cycles and  $1 \times 10^8$  cycles, respectively.

We consider the execution of the jobs specified in Example 4.2 on the processor with parameters specified in Example 4.1. Let the temperature of the processor initially be  $T(0) = 310$  K. The temperature and the speed of the processor during the execution of the two jobs are plotted in Figure 4.3.  $J_1$  begins to execute at time 0 at the maximum speed of 200 MHz with the temperature rising quickly. At time 0.3 s as the temperature reaches 325 K, the speed is reduced to 150 MHz, according to the control law  $f$ . The temperature continues to rise, but a slower rate. Then at time 1.4 s as the temperature reaches 350 K, the speed is reduced to 100 MHz, according to the control law  $f$ . This speed ensures that the temperature remains at the upper-bound of 350 K, as  $T^\infty(100 \text{ MHz}) = 350$  K. Then at time 2.2 s,  $J_1$  completes execution. Subsequently, the processor remains idle and cools until  $J_2$  arrives at time 6 s. In the interval [6, 6.1] s the

processor runs at the highest speed of 200 MHz. Then, the speed is reduced to 150 MHz. Finally,  $J_2$  completes execution at time 6.6 s.

Clearly, on such a processor the delay of a job depends on what has happened earlier. For instance,  $J_1$  executes at the highest speed of 200 MHz for 0.3 s, while for  $J_2$  the corresponding time is only 0.1 s. This is because the temperatures at the start of the execution of the two jobs are different. This clearly depends on both the timing and the thermal parameters. For instance, if the processor cooled faster or if  $J_1$  arrived earlier, then  $J_2$  would be able to execute at the highest speed for longer.

This example illustrates that jobs of the same stream *interfere* with each other through the speed scaling. With variable job arrival times and execution demands, such interference must be carefully considered in computing the bound  $d^{\max}$ . In the next three sections, we will discuss how to effectively compute this bound for different values of the initial temperatures.

## 4.4 Analysis for Maximum Initial Temperature

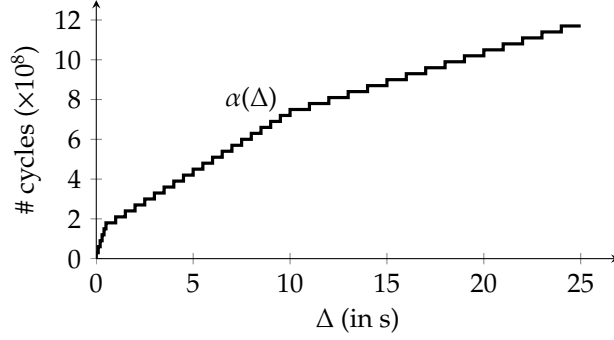
We analyze the problem for different values of the initial temperature of the processor, i.e.,  $T(0)$ . The temperature of the processor can vary in the range  $[T^\infty(0), T^\infty(s^{\min})]$ , where  $s^{\min}$  is the smallest speed actuated by the control law. In this section, we consider  $T(0) = T^\infty(s^{\min})$ , i.e., the maximum possible initial temperature. Starting from such a high temperature, the jobs are anticipated to suffer the largest delay. In the following result we compute the bound on the delay  $d^{\max}$  using the Del operator defined in Appendix A.3.

**Theorem 4.1:** *The worst-case delay when a stream of jobs with input arrival rate  $\alpha$  is executed on  $\mathbf{P}$  with  $T(0) = T^\infty(s^{\min})$  is given by*

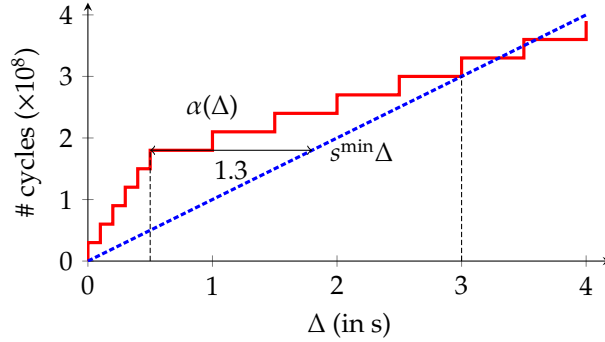
$$d^{\max} = \text{Del}(\alpha(\Delta), s^{\min}\Delta) \quad (4.10)$$

The above result is interesting. It says that for the specific case of the maximum initial temperature, the bound  $d^{\max}$  depends only on the task model (as given by  $\alpha$ ), and the slowest speed actuated by the feedback controller  $s^{\min}$ . In particular, it does not depend on how high the other available speeds are, how quickly the processor cools, or how carefully the control law is designed. We illustrate this analysis with an example.

**Example 4.3:** *Consider a task with an execution demand of  $0.3 \times 10^8$  cycles. The jobs arrive at three rates  $1 \text{ s}^{-1}$ ,  $2 \text{ s}^{-1}$ , and  $10 \text{ s}^{-1}$ . Such tasks can be modeled*



**Fig. 4.4** Arrival curve of the task specified in Example 4.3. The jobs of the tasks arrive at three different rates  $1 \text{ s}^{-1}$ ,  $2 \text{ s}^{-1}$ , and  $10 \text{ s}^{-1}$ .



**Fig. 4.5** Illustration of computation of  $d^{\max}$  for the maximum initial temperature for the task model of Example 4.3 and the processor model of Figure 4.2.

with leaky-buckets, as were used in modeling the shaping curves in the previous chapter. In this case, corresponding to the three rates, the capacities of the leaky-buckets in number of jobs are 15, 5, and 1, respectively.

The arrival curve of the task in the above example is shown in Figure 4.4. Consider the processor model described in Example 4.1. We illustrate the computation of  $d^{\max}$  for the maximum initial temperature in Figure 4.5. The largest horizontal distance between  $\alpha$  and  $s^{\min}\Delta$  is 1.3 s. This gives the value of  $d^{\max}$  according to (4.10).

To conclude, for the specific case of maximum initial temperature,  $d^{\max}$  is given by the relation between  $\alpha$  and  $s^{\min}$  as in (4.10).

## 4.5 Analysis for Minimum Initial Temperature

In this section, we consider the case of minimum initial temperature. The minimum temperature is attained when the processor is indefinitely idle and reaches the temperature  $T^{\infty}(0)$ . Thus, in this section, we compute  $d^{\max}$  for  $T(0) = T^{\infty}(0)$ . To this end, we first derive some useful monotonicity

principles and then use them to find a critical trace.

### 4.5.1 Monotonicity Principles

First, we define a *unit cycle*, denoted as  $u$ , as the smallest measure of the execution time of a job. Conveniently,  $u$  may be defined to be one processor cycle. We assume that the speed of the processor is constant during the execution of a unit cycle.

We present three results on the execution of a unit cycle on a processor with feedback control of speed.

**Lemma 4.1:** *Consider the execution of a unit cycle on  $\mathbf{P}$ . Increasing the temperature of the processor at the start of the execution of the unit cycle, cannot lead to (a) an earlier finish time of the unit cycle, and (b) a lower temperature of the processor at the finish time of the unit cycle.*

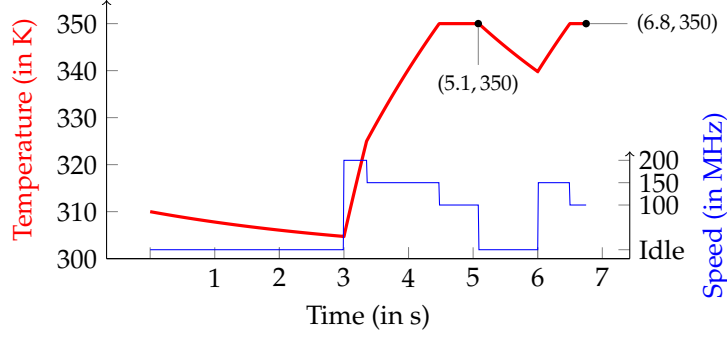
**Lemma 4.2:** *Consider the execution of a unit cycle on  $\mathbf{P}$ . Delaying the execution of the unit cycle cannot lead to an earlier finish time of the unit cycle.*

**Lemma 4.3:** *Consider the execution of a unit cycle on  $\mathbf{P}$ . Delaying the execution of the unit cycle cannot lead to a lower temperature of the processor measured at the finish time of the unit cycle in the delayed execution.*

From these results, we see that heating up the processor or adding idle time before the execution of a unit cycle cannot decrease either the finish time or the temperature at the finish time. Note that these results hold independent of the *values* of the power and temperature parameters, and the control law. However, they do require that the power consumption be a convex function of the speed, the temperature model be as described by the Fourier heat model, and the control law to be monotonically non-decreasing. Deriving from the above lemmas, we have the following crucial theorem.

**Theorem 4.2: (Monotonicity Principle)** *Consider any given trace of jobs served by  $\mathbf{P}$ . Delaying the arrival time of any job of the trace cannot lead to an earlier finish time of any subsequent job.*

To put the above result in perspective, consider the case of a constant speed processor serving a stream of jobs in FCFS order. It can be verified that delaying the arrival of any job will not lead to an earlier finish time of any subsequent job. In other words, delaying the arrival of a job does not decrease the interference the job has on any subsequent job. This desirable monotonic relationship between arrival time and interference holds even for processors with feedback control of speed, as confirmed



**Fig. 4.6** Illustration of the monotonicity principle. The temperature of the processor is plotted in a thick red line, while the speed is plotted in a thin blue line. The job model is from Example 4.2 and the processor parameters are as in Figure 4.2. The arrival time of  $J_1$  is delayed from 0 to 3 s. Due to the delay, both jobs  $J_1$  and  $J_2$  finish later than in the original case (Figure 4.3).

by the above theorem. This principle motivates the search for an effective analysis of the delay bound  $d^{\max}$ .

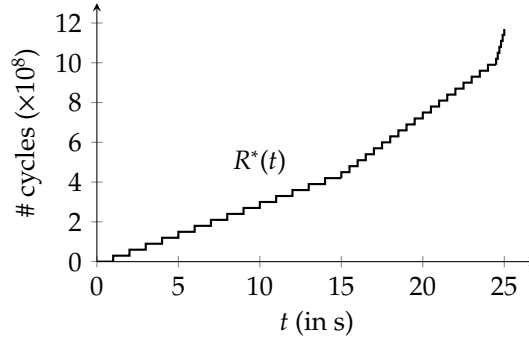
We illustrate the monotonicity principle for the processor model of Example 4.2 and the trace of two jobs specified in Example 4.1. In the example, two jobs  $J_1$  and  $J_2$  are executed immediately upon their arrival. Let us now delay the arrival of  $J_1$  by 3 s. With this modification, the evolution of the processor's temperature and speed are shown in Figure 4.6. For either job, the time and temperature when the job finishes is not smaller than in the earlier case (Figure 4.3). In particular,  $J_2$  starts at the same time in both cases, but finishes later when the arrival of  $J_1$  is delayed. Furthermore, delaying the arrival of  $J_1$  also increases the temperature of the processor at the finish time of  $J_2$  (from 338 K to 350 K). This illustrates the monotonicity principle.

## 4.5.2 Analysis for Minimum Initial Temperature

Given any trace of jobs, we can apply the monotonicity principle to increase the finish time of the *last* job by delaying the arrival of all previous jobs. However, for a given arrival curve of a task, such delaying must conform to the constraints of the arrival curve. We thus need to identify a *critical trace* which delays jobs as much as possible while conforming to the arrival curve. In the following, we define such a critical trace by its arrival function.

**Definition 4.1: (Critical Trace of Length  $l$ )** The critical trace of a task with an arrival curve  $\alpha$  and length  $l$  has an arrival function  $R^*$  given as

$$R^*(t) = \alpha(l) - \alpha(l - t), \quad t \in [0, l]. \quad (4.11)$$



**Fig. 4.7** Arrival function of the critical trace of the task described in Example 4.3 of length  $l = 25$  s. Notice the burst of jobs towards the end of the trace.

The critical trace of length  $l$  has jobs arriving in the interval  $[0, l]$ . In particular, the jobs are delayed such that the *burst* of jobs arrives towards the end of the trace. We illustrate this with an example. Consider the arrival curve shown in Figure 4.4. The critical trace of length  $l = 25$  s of this arrival curve is shown in Figure 4.7. Note that the burst of tasks is built up towards the end of the trace.

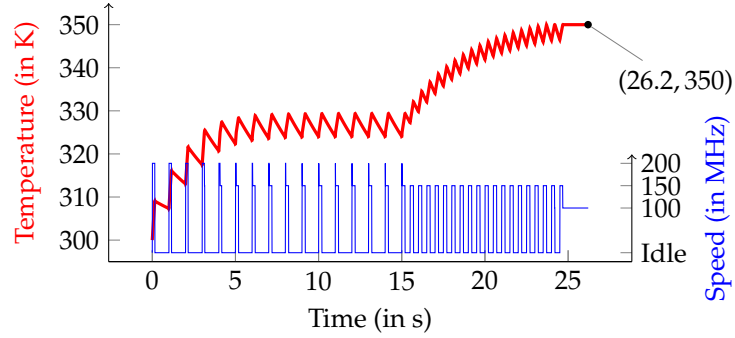
For the above definition of the critical trace, we show the following crucial result.

**Theorem 4.3:** *Let a task with arrival curve  $\alpha$  be executed on  $\mathbf{P}$  with the minimum initial temperature. The bound  $d^{\max}$  equals the delay of the last job of the critical trace of length  $l = t_{\text{hor}}$ .*

We interpret the above result. To compute the bound on delay, we need to simulate one specific trace, which is the critical trace of length equal to the given time horizon. Then the delay of the last job of that trace equals  $d^{\max}$ . Any trace of jobs which conforms to the arrival curve cannot have a job with a higher delay, arriving within the time horizon  $[0, t_{\text{hor}}]$ .

We illustrate the result for processor and task models of Examples 4.1 and 4.3, respectively. We plot arrival function  $R^*$  for the critical trace of length  $l = t_{\text{hor}} = 25$  s in Figure 4.7. The evolution of the temperature and speed of the processor when executing this critical trace is shown in Figure 4.8. The last job of the critical trace, which arrived at 25 s, finishes at 26.2 s. Thus, we derive the bound  $d^{\max} = 1.2$  s. Note how the burst of jobs arriving towards the end heats and slows the processor, and thereby delays the last job.

To conclude, we showed that a desirable monotonicity principle characterizes arrival time of a job and its interference on subsequent jobs, for a processor with temperature-based feedback control of speed. We employed this principle to show that simulating a defined critical trace of a task identifies the bound  $d^{\max}$ , for the minimum initial temperature.



**Fig. 4.8** Temperature (thick red) and speed (thin blue) of the processor when executing the critical trace shown in Figure 4.7. Notice how the burst of jobs arrive towards the end of the trace. The last job finishes at 26.2s with delay  $d^{\max} = 1.2$  s.

## 4.6 Analysis for Any Initial Temperature

Thus far, we have considered minimum and maximum initial temperatures. In either case, we applied very different principles to compute the delay bound  $d^{\max}$ . The natural question is what happens when the initial temperature is in between the two extremes. To this end, we first understand why the presented analysis cannot be extended to any initial temperature and then develop a modified analysis.

### 4.6.1 Critical Trace for Non-Minimum Temperature

We try to understand whether Theorem 4.3 extends for any initial temperature. In other words, does simulating the critical trace of length  $t_{\text{hor}}$  identify  $d^{\max}$ , independent of the initial temperature. Recall that in the critical trace the burst of jobs arrives towards the end of the trace. Intuitively, for a high initial temperature a burst of jobs arriving at the start of the trace may lead to larger delays and temperatures. In other words, a higher starting temperature may be expected to change critical trace. We illustrate this with an example.

**Example 4.4:** Consider a task which merges two periodic streams with periods 3s and 5s. The execution demand of jobs of both streams is  $0.75 \times 10^8$  cycles. The arrival curve for this task is shown in Figure 4.9.

For the task from the above example, we construct the arrival function  $R^*(t)$  of a critical trace of length  $l = 50$  s as defined in (4.11). We simulate the execution of this trace for the initial temperature  $T(0) = 330$  K and the time-horizon  $[0, 50]$ s. The corresponding plots of temperature and speed are shown in Figure 4.10(a). The value of  $d^{\max}$  computed using Theorem 4.3 is 0.96 s.

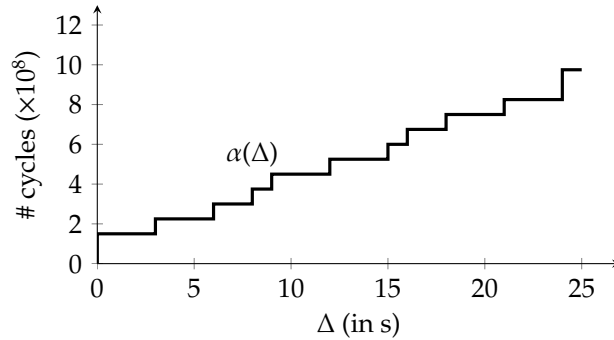


Fig. 4.9 Arrival curve of the task specified in Example 4.4.

Now consider a different trace where two jobs, each with an execution demand  $0.75 \times 10^8$  cycles, arrive at time 0. This conforms to the arrival curve as the task merges two independent periodic streams. The temperature and speed traces are plotted in Figure 4.10(b). The value of  $d^{\max}$  computed using Theorem 4.3 is 1.04 s. Clearly, the delay bound computed with the critical trace in Figure 4.10(a) is incorrect.

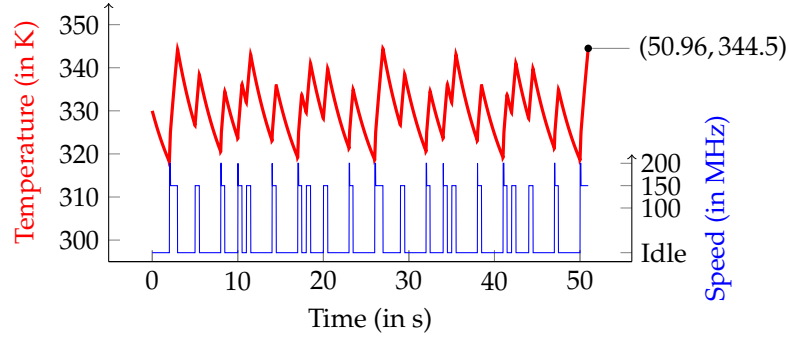
The burst of two jobs arriving at the same time also occurs in the critical trace at time 50 s. The temperature at the start of execution of these two jobs was about 320 K (Figure 4.10(a)). This is smaller than the initial temperature of 330 K. Thus, by delaying the burst in the critical trace, the high initial temperature of the processor does not coincide with the burst. In this example, this results in an incorrect delay bound.

The monotonicity principle of Theorem 4.2 holds independent of the initial temperature, i.e., delaying the arrival of a job can lead to a higher finish time of subsequent jobs. Consequently, a critical trace as defined in (4.11) must exhibit the worst-case delay and temperature. However, the *length* of the critical trace is not known. For the minimum initial temperature, the length of the critical trace is set equal to the given time horizon. For a higher initial temperature, the critical trace could be shorter. Indeed, for the example from above, the trace simulated in Figure 4.10(b) is a critical trace of length 0. Hence, one solution is to simulate the critical trace for every length  $l \in [0, t_{\text{hor}}]$  and identify the maximum delay of the last job across all the critical traces. This can be computationally expensive. In the remainder of this section, we present an abstraction to avoid this computation.

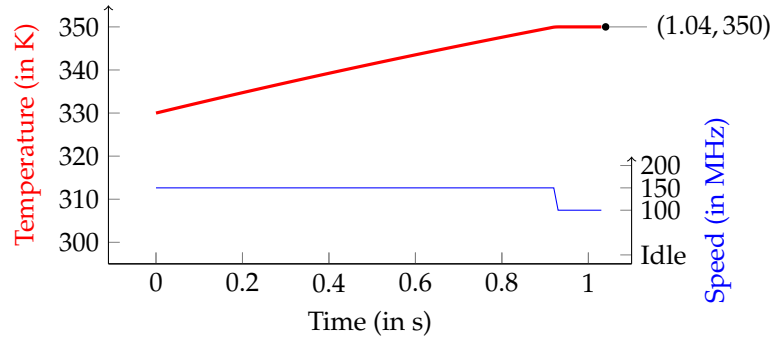
#### 4.6.2 Abstraction of a Thermally-Clipped Processor

We define a model of a hypothetical processor referred to as a *thermally-clipped processor*, denoted as  $\mathbf{P}_{\text{clip}}$ . In addition to the parameters of  $\mathbf{P}$ , defined in Section 4.2,  $\mathbf{P}_{\text{clip}}$  is characterized by a *clipping temperature*,





(a) Simulation of the critical trace of length 50 s for the arrival curve shown in Figure 4.9. The computed value of  $d^{\max} = 0.96$  s.



(b) Simulation of a trace of two jobs arriving at time 0. Each job has an execution demand of  $0.75 \times 10^8$  cycles. The computed value of  $d^{\max} = 1.04$  s.

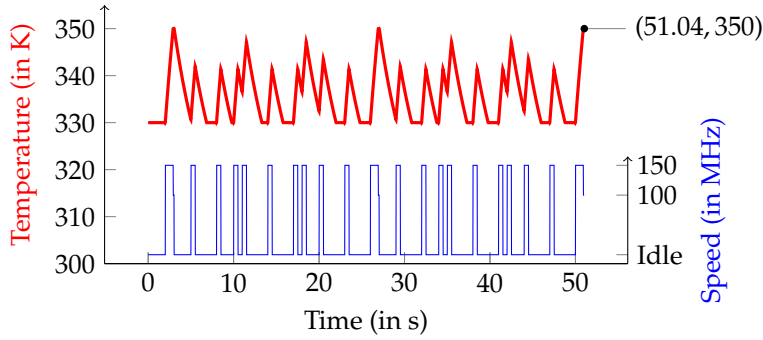
**Fig. 4.10** Invalidity of the critical trace for non-minimum temperature. The temperature of the processor is shown in a thick red line and the speed in a thin blue line. Starting from a higher temperature of 330 K, the critical trace defined in (4.11) has lower bounds on delay and temperature than another trace.

denoted as  $T_{\text{clip}}$ . The interpretation of this is as follows: Whenever the temperature of  $\mathbf{P}_{\text{clip}}$  falls below  $T_{\text{clip}}$ , it is forced to  $T_{\text{clip}}$ . In other words, the temperature of  $\mathbf{P}_{\text{clip}}$  is clipped to  $T_{\text{clip}}$  from below. The other parameters, namely the thermal model, power model, and the control law, all remain as before.

Let us consider some intuitive examples of thermally-clipped processors. For  $T_{\text{clip}} = T^{\infty}(0)$ , the corresponding  $\mathbf{P}_{\text{clip}}$  is identical to  $\mathbf{P}$ , as  $T^{\infty}(0)$  is the minimum temperature of the processor. For  $T_{\text{clip}} = T_{\text{peak}} = T^{\infty}(s^{\min})$ , the corresponding  $\mathbf{P}_{\text{clip}}$  is a constant-speed processor with a speed equal to  $s^{\min}$ .

Using this defined notion of thermal clipping, we present below the analysis of the bounds for any initial temperature.

**Theorem 4.4:** *Let a task with arrival curve  $\alpha$  be executed on  $\mathbf{P}$  with some initial temperature  $T(0)$ . The bound  $d^{\max}$  equals the delay of the last job of the critical trace of length  $t_{\text{hor}}$  when simulated on the corresponding  $\mathbf{P}_{\text{clip}}$  with clipping*



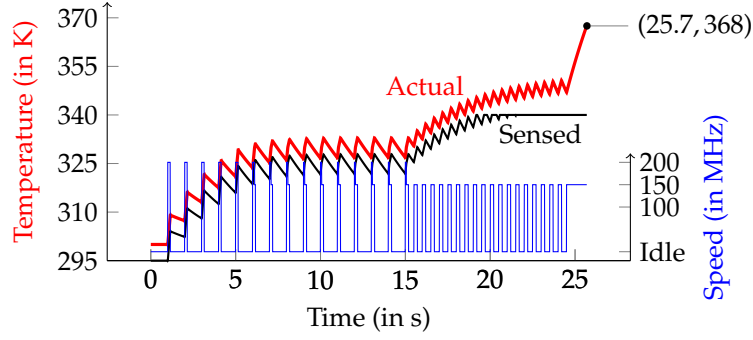
**Fig. 4.11** Analysis for the task model of Example 4.4 and processor model of Example 4.1. The processor has a clipping temperature of  $T(0) = 330$  K. The temperature of the processor is shown in a thick red line and the speed in a thin blue line. Notice how the temperature is clipped to 330 K. The computed bound is  $d^{\max} = 1.04$  s.

temperature  $T(0)$ .

We interpret the above result. As discussed, a critical trace of unknown length must be simulated on  $\mathbf{P}$  to identify  $d^{\max}$ . This uncertainty of the length of the critical trace is avoided by defining the abstraction of a thermally-clipped processor  $\mathbf{P}_{\text{clip}}$  with clipping temperature  $T(0)$ . On such a processor, simulating the critical trace of length  $t_{\text{hor}}$  identifies  $d^{\max}$ .

We illustrate this computation with the earlier example. Consider the task model of Example 4.4 and the processor model of Example 4.1. For the initial temperature  $T(0) = 330$  K and  $t_{\text{hor}} = 50$  s, the analysis of the delay bound using the above theorem is shown in Figure 4.11. Notice how the temperature is clipped from below to 330 K. The computed value of  $d^{\max} = 1.04$  s. Indeed, this delay bound holds true for the traces shown in Figures 4.10(a) and 4.10(b).

The result of Theorem 4.4 provides a unifying analysis for any initial temperature. For the case of minimum initial temperature,  $\mathbf{P}_{\text{clip}}$  and  $\mathbf{P}$  are identical and thus Theorems 4.3 and 4.4 are equivalent. For the case of the maximum initial temperature,  $\mathbf{P}_{\text{clip}}$  behaves identical to a processor working at the constant speed of  $s^{\min}$ . For such a processor, from known results in Network Calculus [LBT01], the worst-case delay is given by applying the  $\text{De1}$  operator to the arrival and service curves. This is identical to the result obtained in Theorem 4.1. To conclude, simulating the critical trace of length  $t_{\text{hor}}$  on the thermally-clipped processor is a unified approach to compute  $d^{\max}$  for any initial temperature.



**Fig. 4.12** Computing the worst-case delay with a faulty sensor with offset  $-5$  K and saturation at  $350$  K. The task model is from Example 4.3 and the processor model is from Example 4.1. The critical trace of length  $l = 25$  s is simulated. The actual and sensed temperature are shown in thick red and black lines, respectively. The speed is shown in a thin blue line. The computed bound on the delay is  $d^{\max} = 0.7$  s.

## 4.7 Faulty Temperature Sensors

Fabricating temperature sensors can be error-prone. The common errors are (a) offset: the sensed temperature is constantly higher or lower by a fixed value, and (b) saturation: the sensed temperature does not increase beyond a threshold. We model errors in the sensor by the transfer function  $\epsilon$ , where  $\epsilon(T)$  is the (erroneous) sensed temperature when the actual temperature is  $T$ . For instance, a sensor which has an offset of  $-5$  K and saturates at  $340$  K, has an transfer function

$$\epsilon(T) = \max(T - 5, 340). \quad (4.12)$$

Consider the analysis of  $\mathbf{P}$  with the minimum initial temperature as presented in Theorem 4.3. We examine if this analysis can be extended to consider faulty sensors. The following result identifies the class of error functions for which this is true.

**Theorem 4.5:** *If  $\epsilon$  is a monotonically non-decreasing function, Theorem 4.3 can be used to compute  $d^{\max}$ , where the controller works according to the erroneous sensed temperature.*

From the above result, for a large class of error functions, the approach of simulating the critical trace of length  $t_{\text{hor}}$  on the processor gives the bound on delay. In particular, the results applies for a sensor with an offset and/or saturation.

We illustrate this with an example. Consider a faulty sensor with an error function as in (4.12). Consider the processor and task models of Examples 4.1 and 4.4, respectively. For  $t_{\text{hor}} = 25$  s and the minimum initial temperature, we plot the temperature and speed traces for the critical trace in Figure 4.12. Indeed, the delay of the last job is the highest. Further, the

actual and sensed temperatures are maximum when the last job finishes. Given that the faulty sensor underestimates the actual temperature, the processor runs faster than it should nominally. Consequently,  $d^{\max}$  is smaller at 0.7 s, compared to 1.2 s for an accurate sensor (Figure 4.8).

To conclude, while the delay bound depends on the error function, the critical trace remains the same. Hence, the analysis approach of Theorem 4.3 extends to the case of faulty temperature sensors.

## 4.8 Experimental Results

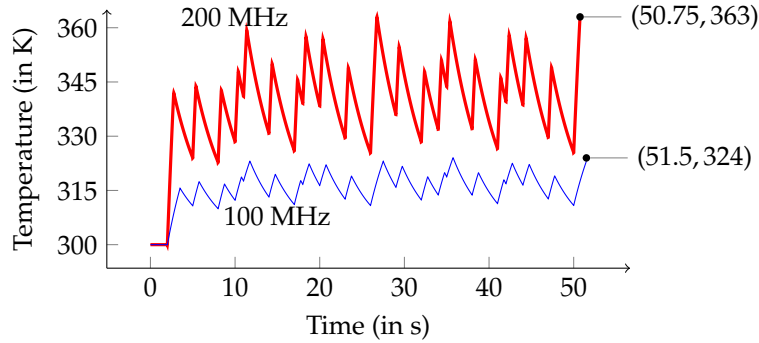
In this section, we will present experimental results that evaluate different aspects of a processor with feedback control of speed. First, we will numerically highlight the advantage of speed scaling. We will then illustrate the dependence of the computed bounds on the control law and the task model. In all experiments, the processor model is as specified in Example 4.1.

### 4.8.1 Advantage of Feedback Control of Speed

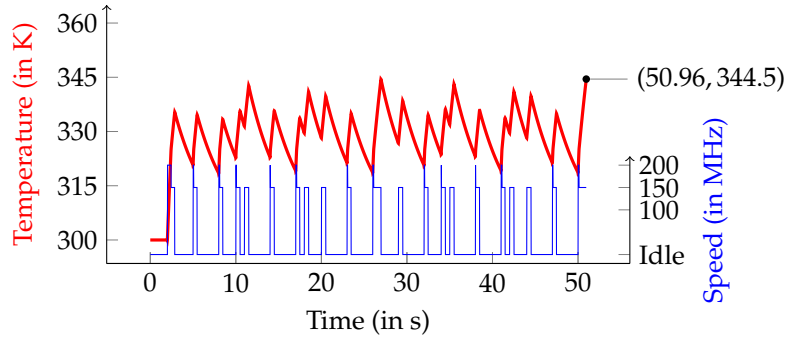
We first illustrate the advantage of feedback control of speed. Consider the task model of Example 4.4. We consider a horizon of  $[0, 50]$ s, and the minimum initial temperature. We are required to execute this task with delay and temperature constraints, namely (a) the relative deadline of each job is 1 s, and (b) the temperature must not exceed 350 K.

First, we consider two constant speed processors running at speeds 100 MHz and 200 MHz. Theorem 4.4 also applies to constant speed processors, and thus we simulate the critical trace of length 50 s for these two cases. The temperature and speed traces are plotted in Figure 4.13(a). For the processor running at 100 MHz, we have  $d^{\max} = 1.5$  s and  $T^{\infty}(100 \text{ MHz}) = 350$  K. At this slower speed, the temperature constraint is met while the delay constraint is not. On the other hand, for the processor running at 200 MHz, we have  $d^{\max} = 0.75$  s and  $T^{\infty}(200 \text{ MHz}) > 350$  K. Indeed, for the plotted trace the highest temperature of 363 K exceeds the temperature constraint. Thus, for the higher speed, the delay constraint is met while the temperature constraint is not.

Now we consider feedback control of speed with the control law shown in Figure 4.2. Note that the speeds used in this control law are between 100 MHz and 200 MHz. Constant execution at either of these speeds is unable to meet both delay and temperature constraints. This controller ensures that the temperature does not exceed  $T_{\text{peak}} = T^{\infty}(100 \text{ MHz}) = 350$  K. Thus, the temperature constraint is met. We simulate the critical trace and the obtained plots are shown in



(a) Temperature plots for two different speeds, namely 200 MHz (thick red line) and 100 MHz (thin blue line).



(b) Temperature plot in thick red line and speed plot in thin blue line.

**Fig. 4.13** Illustration of the advantage of feedback control of speed. Simulation of the critical trace shown in Figure 4.7 for the processor model of Figure 4.2. The constraints  $d^{\max} \leq 1$  s and  $T_{\text{peak}} \leq 350$  K are (a) not satisfied with constant speed processors, but are (b) satisfied by the feedback control of speed.

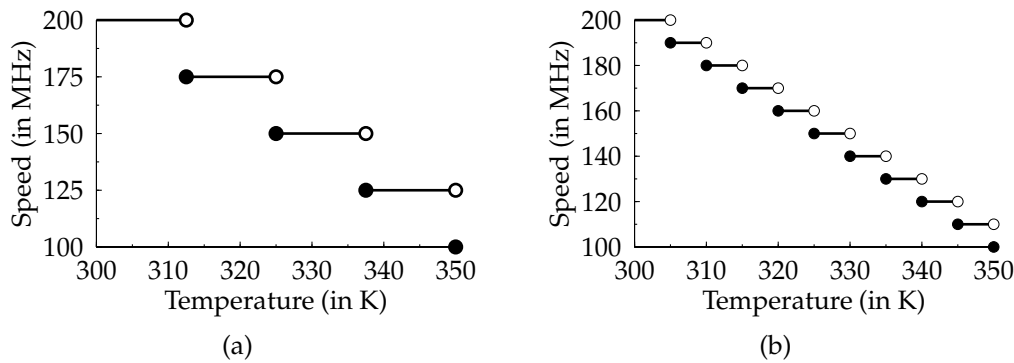
Figure 4.13(b). From Theorem 4.3, we have  $d^{\max} = 0.96$  s. Thus, both the given delay and temperature constraints are satisfied with the feedback control.

This example illustrates that speed scaling can be effectively used to trade-off the temperature and delay metrics. Furthermore, the controller satisfies the temperature constraint independent of the task model.

## 4.8.2 Finer Control Laws

The control law is a monotonically non-increasing function. In practice, the control law must be a piecewise constant function, where the discontinuities denote the temperatures where the speed of the processor are changed. A natural question then is with what granularity should the speeds be changed. We consider two control laws shown in Figure 4.14, which are *finer* than the original control law of Figure 4.2.

For the task model of Example 4.3, consider  $t_{\text{hor}} = 25$  s and the



**Fig. 4.14** Two control laws which are *finer* than the control law of Figure 4.2. In both laws the discontinuities are uniformly distributed.

minimum initial temperature. The value of  $d^{\max}$  for the control law of Figure 4.2 is 1.2 s. For the control laws of Figures 4.14(a) and 4.14(b) are 1 s and 1.06 s, respectively. Thus, the worst-case delay does not monotonically decrease with uniform reduction in granularity in the control law. To conclude, the control law must be carefully chosen to minimize  $d^{\max}$ .

### 4.8.3 Dependence of Optimal Control Law on the Task

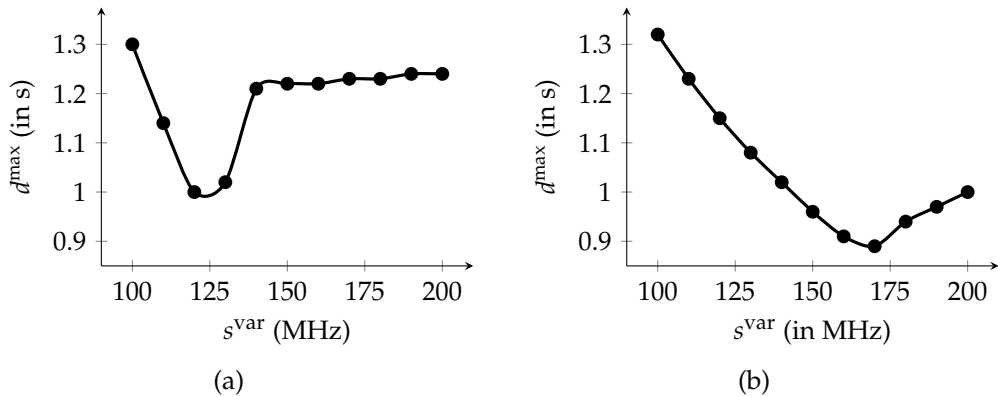
We continue the experimentation on the choice of the control law. We consider a control law parameterized by a variable  $s^{\text{var}}$  as

$$\begin{aligned} f(T) &= 200 \text{ MHz}, & T < 325 \text{ K} \\ &= s^{\text{var}}, & 325 \text{ K} \leq T < 350 \text{ K} \\ &= 100 \text{ MHz}, & T = 350 \text{ K}. \end{aligned}$$

We consider the task model of Example 4.3, with  $t_{\text{hor}} = 25$  s and the minimum initial temperature. We compute  $d^{\max}$  for different values of  $s^{\text{var}}$ . The obtained values are plotted in Figure 4.15(a). The optimal value of  $s^{\text{var}}$  is about 125 MHz.

Then we consider the task model of Example 4.4, with  $t_{\text{hor}} = 50$  s and the minimum initial temperature. The corresponding plots of  $d^{\max}$  for different values of  $s^{\text{var}}$  is shown in Figure 4.15(b). The optimal value of  $s^{\text{var}}$  is about 170 MHz.

The optimal values of the  $s^{\text{var}}$  are different for the two tasks. Indeed, the plots of Figure 4.15 eminently differ. This example highlights that the choice of the control law depends on the task model. In conclusion, the control law should be carefully chosen depending on given constraints and the task model. Such a choice can be exercised given the efficient analysis of simulating the critical trace. However, it remains an open problem.



**Fig. 4.15** Dependence of the bound on delay for different control laws for the task model of (a) Example 4.3 and (b) Example 4.4. The optimal control law is different for the two task models.

## 4.9 Summary

In the previous part of the thesis we considered the design of a run-time manager to absorb the variability and uncertainty in timing properties. In this chapter, we consider a given run-time manager which is designed for a specific objective of managing temperature. In particular, we consider temperature-based feedback control of speed. In the presence of such a controller, the challenge is to identify an analysis procedure to compute the timing guarantees. The analysis must consider the dynamic behavior of the controller in addition to variability in job arrival times and execution demands.

We solved this challenge by deriving the critical trace of jobs. By simulating this trace of jobs on the thermally-clipped model of the processor, we obtain bounds on the delay of any job and the temperature of the processor. Crucially, the critical trace of jobs does not depend on the thermal, power, or controller models, or the initial temperature of the processor. This result was derived based on the proposed monotonicity principle which holds under commonly satisfied assumptions on the model parameters.

We derive the following lesson from the results of this chapter. Identifying a critical trace of a system to derive the worst-case parameters is an effective solution in handling variability. Indeed, this approach goes back to the analysis of the critical instance for fixed-priority systems [LSD89]. The results of the chapter confirm that such an approach is applicable even for the apparently complicated temperature-based feedback control of speed. The difference however is that the bounds are not analytically derived, but computed by simulating the critical trace.

## Appendix

### Proof of Theorem 4.1

$s^{\min}$  is the slowest speed of  $\mathbf{P}$ . Thus, in any interval of length  $\Delta \geq 0$ , the minimum provided service is  $s^{\min} \cdot \Delta$ . Then, using the standard definition of Del from Appendix A, we arrive at the upper-bound of (4.10).

Tightness: For the specific input arrival function  $R = \alpha$ , during the execution of the first busy interval, the temperature remains  $T^\infty(s^{\min})$  and thus the speed remains  $s^{\min}$ .  $\square$

### Proof of Lemma 4.1

The control law  $f$  is monotonically non-increasing with the temperature of the processor. Thus, increasing the temperature cannot increase the speed of execution of the unit cycle and cannot decrease the finish time.  $\square$

### Proof of Lemma 4.2

We show this by a contradiction. Consider two schedules  $S_1$  and  $S_2$  starting at time 0, with the same initial temperature. In  $S_1$ , a unit cycle is executed from time 0, while in  $S_2$  the unit cycle is executed from some time  $s > 0$ . Let  $T_1$  and  $T_2$  denote the temperature traces for the two schedules  $S_1$  and  $S_2$ , respectively. Let  $P_1$  and  $P_2$  denote the power traces for the two schedules  $S_1$  and  $S_2$ , respectively. Let  $S_2$  finish the execution of the unit cycle earlier. Then there exists a time  $u > s$  such that the two schedules have executed the same workload in  $[0, u]$ . Also the speed at which the unit cycle is executed in  $S_1$  must be less than in  $S_2$ . From the convexity of the power function  $\phi$  we have the following condition on the power traces.

$$\int_0^u P_1(t)dt < \int_0^u P_2(t)dt. \quad (4.13)$$

Further,  $P_1$  is a constant function in  $[0, u]$  while  $P_2$  is a non-decreasing function in  $[0, u]$ . For the assumed thermal model, we have  $T_1(u) < T_2(u)$ . Then by the monotonicity of the control law, at time  $u$  the speed in  $S_1$  cannot be less than the speed in  $S_2$ . This is a contradiction. Hence, the unit cycle cannot finish earlier in  $S_2$ .  $\square$

### Proof of Lemma 4.3

Let  $S_1$  and  $S_2$  be the two schedules as defined in the previous proof. Let the unit cycle finish execution at times  $d_1$  and  $d_2$  in the two schedules, with  $d_2 \geq d_1$ . In  $S_1$ , the processor runs at a constant speed in  $[0, d_1]$  and



is idle from  $[d_1, d_2]$ . In  $S_2$ , the processor is idle in  $[0, s]$  and runs at a higher constant speed in  $[s, d_2]$ . Then, from Lemma 2 in [RYB<sup>+</sup>11] the temperature at time  $d_2$  cannot be higher for schedule  $S_1$ .  $\square$

### Proof of Theorem 4.2

We prove this by induction on the total number of unit cycles of all jobs of the considered trace.

*Induction Hypothesis:* Delaying the execution of any job or increasing the starting temperature of the processor will not lead to an earlier finish time of any job.

*Basis:* For one unit cycle, Lemmas 4.1 and 4.2 prove the induction hypothesis.

*Inductive Step:* If the hypothesis holds for all traces with execution demand  $c$  unit cycles, then it holds for all traces with execution demand  $(c + 1)$  unit cycles.

Consider a trace of jobs with total execution demand of  $(c + 1)$  unit cycles. Let  $S_1$  and  $S_2$  be two schedules of executing the trace on the processor. In  $S_1$  all jobs are executed without any delay, whereas in  $S_2$  some jobs are executed with delay. Consider the first unit cycle of the trace. If the execution of this unit cycle is not delayed in  $S_2$ , then both schedules  $S_1$  and  $S_2$  behave the same until the execution of the first unit cycle. From thereon, by the inductive hypothesis the property is satisfied for the smaller trace with  $c$  unit cycles. Let the execution of the first unit cycle be delayed in  $S_2$ . Let  $d_1$  and  $d_2$  denote the finish time of this unit cycle in the two schedules  $S_1$  and  $S_2$ , where  $d_2 \geq d_1$ . Two cases arise, depending on the arrival time of the second unit cycle of in the trace, denoted as  $s$ .

*Case (a):*  $s \geq d_2$ . From Lemma 4.3, we have  $T_1(d_2) \leq T_2(d_2)$ . Further, in the two schedules the processor is idle in  $[d_2, s]$ . Thus,  $T_1(s) \leq T_2(s)$ . From time  $s$ , by the inductive hypothesis the property is satisfied for the smaller trace  $c$  unit cycles.

*Case (b):*  $s < d_2$ . Modify  $S_1$  to form a new schedule  $S_3$  such that the second unit cycle of the trace is delayed to start executing from time  $d_2$ . From the induction hypothesis, from the second unit cycle onwards, no job of schedule  $S_1$  will finish any later than the same job in  $S_3$ . From Lemma 4.3, we have  $T_3(d_2) \leq T_2(d_2)$ . Thus, from the induction hypothesis, starting from time  $d_2$ , no job of  $S_2$  can finish before the same job from  $S_3$ . Hence, no job of  $S_2$  finishes before the corresponding job of  $S_1$ .  $\square$

**Proof of Theorem 4.3**

We show this with a contradiction. Let the last job of critical trace be denoted  $J^*$ . Let there exist a different trace with arrival function  $R$ , such that a job  $J'$  arriving at time  $t'$  has a higher delay than the delay of  $J^*$  in the critical trace. Applying Theorem 4.2 on the arrival function  $R$ , we can generate a new arrival function  $R_1$  without decreasing the delay of  $J'$  as follows.

$$R_1(t) = \alpha(t' - t'') - \alpha(t' - t), \quad t \in (t'', t'], \quad (4.14)$$

$$= 0, \quad t \in [0, t'']. \quad (4.15)$$

where  $t''$  is given as

$$R(t') = \alpha(t' - t''). \quad (4.16)$$

The modified arrival function  $R'$  in the interval  $[t'', t']$  is a suffix of the function  $R^*$ , i.e.,

$$R_1(t) = R^*(t + (t_{\text{hor}} - t')), \quad t'' \leq t \leq t'. \quad (4.17)$$

Consider the common parts of the two function, i.e.,  $R_1(t'' : t')$  and  $R^*((t_{\text{hor}} - (t' - t'')) : t_{\text{hor}})$ . While beginning the execution of this common part for function  $R_1$ , the pending workload is zero and the temperature of the processor is the minimum. Then, applying Theorem 4.2, we know that the finish time of a job for the function  $R_1$  cannot be any higher than the corresponding job of  $R^*$  (correspondence here is for the the common parts of the two arrival functions). This contradicts the assumption that  $J'$  has a higher delay than  $J^*$ .

*Tightness:* The arrival function  $R^*$  of the critical trace conforms to the arrival function  $\alpha$ , and thus characterizes a valid trace.  $\square$

**Proof of Theorem 4.4**

Like in the previous proof, we define the trace with arrival function  $R_1$  and compare the common parts of  $R_1$  and  $R^*$ . At the start of the common part, the temperature of the processor executing  $R_1$  is not more than  $T(0)$  which is the lower bound on the temperature of  $\mathbf{P}_{\text{clip}}$  when executing  $R^*$ . By repeatedly applying Lemma 4.1 whenever the temperature of  $\mathbf{P}_{\text{clip}}$  is forced to  $T(0)$  we can show the result.

*Tightness:* In the simulation of the critical trace on  $\mathbf{P}_{\text{clip}}$ , let  $s$  denote the latest time when the temperature of  $\mathbf{P}_{\text{clip}}$  is forced to  $T(0)$ . The temperature of  $\mathbf{P}_{\text{clip}}$  in  $s^-$  is at least  $T(0)$ . Thus, at time  $s$  the processor is idle. Then, all jobs arriving in the interval  $[\rho, t_{\text{hor}}]$  are executed in  $\mathbf{P}_{\text{clip}}$  starting with an

empty buffer and with no temperature clipping. This is equivalent to the simulation of a trace with the following arrival function  $R$  on  $\mathbf{P}$ .

$$R(t) = R^*(t + s) - R^*(s), \quad 0 \leq t \leq t_{\text{hor}} - \rho, \quad (4.18)$$

$$= R(\tau - s), \quad t > t_{\text{hor}} - \rho. \quad (4.19)$$

Then the delay of the last job of the trace with arrival function  $R$  on  $\mathbf{P}$  is the same as the last job of the critical trace on  $\mathbf{P}_{\text{clip}}$ . As  $R$  conforms to the arrival curve  $\alpha$  it characterizes a valid trace.  $\square$

### Proof of Theorem 4.5

Theorem 4.3 is satisfied if the control law  $f$  is monotonically non-increasing. Recall that  $f$  translates a sensed temperature to a speed, that is actuated. Given a temperature sensor that is faulty with a transfer function  $\epsilon$ , we can define a modified control law  $f'$  as

$$f'(T) = f(\epsilon(T)). \quad (4.20)$$

Given that  $f$  and  $\epsilon$  are monotonic functions,  $f'$  is also a monotonic function. By thus absorbing the sensor's transfer function to the control law, the validity of Theorem 4.3 can be established for faulty sensors with a monotonic transfer function.  $\square$



# 5

## Satisfiability Modulo Real-Time Calculus

### 5.1 Introduction

Cyber-physical systems are expected to evolve into large and distributed setups. Indeed, interconnecting multiple heterogeneous devices will enable new applications. Consider for instance interconnecting multiple cars on a highway [YLVZ04] or heterogeneous devices in a power distribution grid [PFR09].

With the increasing scale of CPSs, the resultant design problems also scale in complexity for several reasons. Firstly, for real-time CPSs timing guarantees of several interacting applications may have to be simultaneously considered. Furthermore, platform requirements such as availability of buffer-space or bounds on energy consumed at each processor must be considered. Finally, there may be several design knobs which influence the metrics of interest with intricate dependencies.

We illustrate this challenge and provide a solution strategy for a specific design problem which we refer to as the *speed assignment problem*. In this problem we consider a real-time system with multiple processors executing multiple applications. Each application has one or more tasks executing on different processors, where the tasks are interconnected with dataflow dependencies. The timing properties of all applications, such as the arrival patterns of events and the execution times of the tasks, are given. The design problem is to assign a *static* speed to each

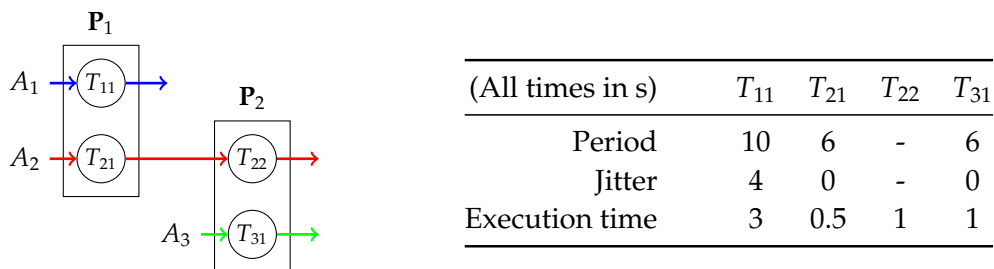


Fig. 5.1 Block diagram and parameters for Example 5.1.

processor from a given set of speeds.<sup>1</sup> This assignment is to be done under multiple conflicting constraints: (a) applications have end-to-end delay constraints, (b) processors have finite buffer-space for pending events, and (c) processors have a finite energy budget in a given interval of time. This is a discrete optimization problem where the number of designs grows polynomially in the number of processors and exponentially in the number of available speeds. For tens of processors and a few different speeds for each processor, the design space can be very large.

There are several examples of CPSs where such constraints arise. Timing constraints of distributed systems arise in the interconnection of Electronic Control Units (ECUs) in a modern automobile [LMSN04]. A Network-on-Chip (NoC) in a multi-processor system has both timing and buffer-space constraints [HM04]. In larger systems such as data-centers, the timing and energy consumption objectives are relevant [FWB07].

Assigning the speed of each processor provides effective design knobs to regulate timing, buffer-space, and energy metrics. Timing constraints may be met by increasing speeds. Energy constraints may be met by decreasing speeds. And buffer constraints may be met by carefully balancing speeds of different processors. However, such dependencies are not always intuitive. We illustrate this with an example.

### 5.1.1 Motivating Example

**Example 5.1:** Two processors  $P_1$  and  $P_2$  serve three applications  $A_1$ ,  $A_2$ , and  $A_3$  as shown in Figure 5.1.  $A_1$  and  $A_3$  comprise of a single task namely  $T_{11}$  and  $T_{31}$ , respectively.  $A_2$  comprises of two tasks, namely  $T_{21}$  and  $T_{22}$ , interconnected by a data dependency. A fixed priority scheduler is used on both processors with the priority ordering  $T_{11} > T_{21}$  on  $P_1$  and  $T_{22} > T_{31}$  on  $P_2$ . The timing properties of the tasks are specified in Figure 5.1.

In the above example, we study the effect of *halving the speed* of  $P_1$ ,

<sup>1</sup> In contrast to the previous chapter where the speed of a processor was dynamically changing, in this chapter we only consider a static speed. However, this static speed must be chosen from a given set of allowed speeds.

i.e., doubling the execution times of  $T_{11}$  and  $T_{21}$ . This change has the following non-intuitive consequences.

- The worst-case delay of application  $A_3$  increases from 2 to 4. On the contrary, we may expect no difference given that the only task of  $A_3$  is mapped on to  $P_2$ , whose speed remains unchanged.
- The worst-case buffer-space required at the input of  $T_{22}$  increases from 1 event to 3 events. On the contrary, we may expect that slowing the processor which provides input events to a task will reduce the required buffer-space.
- The worst-case energy consumption of  $P_2$  within intervals of certain lengths increases. On the contrary, we may expect no difference given that the speed of  $P_2$  remains unchanged.

The above example highlights that changing the speed of a processor can have non-intuitive effects on timing, buffer-space, and energy metrics. For larger systems with more processors and applications, such dependencies can be expected to be more intricate. Thus, the core of the speed assignment problem which is to understand how to assign the speed of each processor while satisfying the constraints, is a challenging problem.

The challenge motivates two responses. Firstly, we need a custom *theory* to systematically compute safe bounds on the end-to-end delay, buffer-space requirement, and energy consumption. Secondly, as the choice of speeds is not straightforward given that the constraints do not exhibit well-founded properties such as linearity or convexity, we need a generic design space *exploration engine*.

Given that the design space in the speed assignment problem can be very large, any solution strategy has to be evaluated by its computational efficiency. One approach in large optimization problems is to adopt a heuristic strategy such as gradient descent, simulated annealing, or evolutionary algorithms. However, none of these methods guarantees *optimality*, i.e., they may not find a speed assignment satisfying all the constraints even if there is one. This brings us to the core problem at hand: How can we reconcile the high complexity of finding a solution to the speed assignment problem with the need for an optimal solution?

For the said problem, a template of a solution comes from the domain of formal system verification, namely the use of a Satisfiability Modulo Theory (SMT) solver [BSST09]. An SMT solver combines a domain-specific theory solver with a SAT engine to exactly solve a given set of constraints. If the solver does not find a solution to the set of constraints, then a proof is furnished of why a solution does not exist. Thus, it

meets our requirements of providing an optimal solution by integrating a custom theory with a generic exploration engine.

However, the computational cost of the optimization is an open question. SMT solvers specify problems as extensions of a SATisfiability problem [MZ09]. Schaefer showed that even small instances of SAT problems (with 3 literals per clause) are NP-Hard [Sch78]. However, efficient computational heuristics have been developed which perform well in practice. The prime example is the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DLL62,DP60] which has been demonstrated to be efficient for a variety of domains [DMB11]. It remains to be seen if these techniques are effective for the speed assignment problem.

Two enabling principles for the design of an SMT solver are (a) interpretation and analysis of an *incomplete model* which formalizes a set of designs, and (b) computation of a *conflict clause* which formalizes non-satisfaction of constraints. In this chapter, we aim to explore if these principles can be derived for the speed assignment problem, and more generally for design problems in real-time CPSs.

### 5.1.2 Contributions

We formulate the speed assignment problem as a SATisfiability problem interpreted on the background theory of Real-Time Calculus [TCN00]. We solve this SAT problem by designing a custom Satisfiability Modulo Theory (SMT) solver. To this end, we define *abstract arrival and service curves* which generalize the arrival and service curves from Real-Time Calculus. We show that these abstract curves satisfy important monotonicity principles. These principles enable us to interpret incomplete models and generate conflict clauses, the two requisites for designing an SMT solver.

We implement a custom SMT solver with the OpenSMT [BPST10] solver framework which is specifically designed to interface a custom theory solver with the miniSAT solver [ES05]. For the theory solver for Real-Time Calculus, we use the Modular Performance Analysis (MPA) toolbox for MATLAB<sup>®</sup> [Wan06]. The specific nature of abstract curves enables a convenient design of the theory solver as an application over the MPA toolbox.

The key advantage of any SMT solver is the deep embedding of the theory solver within the exploration engine. This deep embedding has been shown to perform efficiently in practice, for several problems. The main result of this work is to confirm the computational success of SMT solvers in the considered case of the speed assignment problem, with RTC as the background theory. In particular, we show that for problem instances with large design spaces of about  $3 \times 10^{17}$  speeds assignments,



the solver computes a valid solution, if there is one, within hundreds of solver calls to the MPA toolbox. On a typical desktop computer, a solver call is executed within a couple of seconds.

The empirical results of this chapter motivate solving other design problems in CPSs with an SMT solver. For design problems with worst-case timing constraints, the groundwork for such a formulation is the use of the proposed abstract arrival and service curves. More generally, this chapter provides a template to use an SMT solver for design problems in CPSs which require hard guarantees.

The rest of the chapter is organized as follows. We formally specify the speed assignment problem in Section 5.2. We define abstract arrival and service curves in Section 5.3. In Section 5.4 we introduce the formal concepts in verification with SMT solvers. We apply the abstract curves for the specific problem of speed assignment in Section 5.5. We present experimental results for a large class of problems in Section 5.6. We summarize in Section 5.7 and provide proofs of results in an appendix.

## 5.2 System Model and Problem Definition

In this section, we define the processor and application models, and state the problem formulation.

### 5.2.1 Processor Model

We consider a set of processors denoted as  $\mathbf{P}$ , where the  $i$ th processor is denoted as  $\mathbf{P}_i$ . The speed of any processor  $\mathbf{P}_i$ , denoted as  $s_i$ , can be independently assigned from a set of allowed speeds denoted  $\mathbf{S}_i$ . This is a static speed assignment, i.e., the speed remains constant once chosen from the set of allowed speeds. The  $j$ th smallest speed in  $\mathbf{S}_i$  is denoted as  $\mathbf{S}_{ij}$ . We denote the speeds as multiples of the slowest speeds, which is normalized to 1. The power consumed by  $\mathbf{P}_i$  is modeled by a convex function of its speed, denoted as  $\phi_i(s_i)$ . When not executing any task,  $\mathbf{P}_i$  is said to be idle and consumes a fixed idle power  $\phi_i(0)$ .

### 5.2.2 Application Model

We consider a set of applications denoted as  $A$ , where the  $i$ th application is denoted as  $A_i$ . An application is represented by a directed acyclic graph (DAG), where nodes represent tasks and edges represent dataflow dependencies. We only consider linear task graphs, wherein each node has an in-degree and out-degree of 1. The  $j$ th task of  $A_i$  in the topological

order of the DAG is denoted as  $T_{ij}$ . Events arrive at the input of the first task of each application. Processing of each event is referred to as a job of that task. A task processes each event in First-Come-First-Server (FCFS) order, and subsequently generates an input event for its successor task, if any.

Each task  $T_{ij}$  is statically mapped to some processor given by a *mapping function*  $m(T_{ij})$ . Multiple tasks may be mapped to the same processor. In such a case, a fixed priority scheduler is used to arbitrate between tasks. The priority order for each processor is assumed to be given.

Each application  $A_i$  is characterized by an input arrival curve<sup>2</sup>  $\alpha_i = (\alpha_i^u, \alpha_i^l)$  which characterizes the execution demand of the events arriving at the input of task  $T_{i1}$ . Each task  $T_{ij}$  is characterized by the worst- and best-case execution times of its respective jobs, denoted as  $C_{ij}^u$  and  $C_{ij}^l$ , respectively. These execution times are specified for the normalized minimum speed of the corresponding processor. At other speeds, execution times scale linearly.

In the theoretical presentation in this chapter, we assume  $C^u = C^l = 1$  for each task. Including non-unit values of these terms is straightforward as discussed in [Wan06].

### 5.2.3 Constraints

We have three categories of constraints as specified below.

- *Delay constraints:* An application  $A_i$  may have an upper-bound on the end-to-end delay of any event, denoted as  $D_i$ . The end-to-end delay of an event is the time delay between its arrival at the first task of the application and the end of its processing at the last task of the application.
- *Buffer constraints:* Each task has an input buffer that queues pending events. A task  $T_{ij}$  may have an upper-bound on the buffer-space in terms of number of events, denoted as  $B_{ij}$ .
- *Energy constraints:* A processor  $P_i$  may have an upper-bound on the energy consumed for any interval of length  $\Delta \geq 0$  denoted as  $E_i^u(\Delta)$ .

### 5.2.4 Problem Statement

Given are processor and application models as described above. The problem is to assign the speed of each processor  $P_i$ , i.e., to identify  $s_i \in \mathbf{S}_i$ , such that all delay, buffer and energy constraints are satisfied. If for a

<sup>2</sup> Note that here we use the extended definition of arrival curves with both upper and lower curves. These are formally defined in Appendix A.2.

given problem instance no valid speed assignment exists, then this must be shown to be the case.

## 5.3 Abstract Arrival and Service Curves

We make a particular choice in the order of presentation in this chapter. We will first, in this section, define and analyze abstract arrival and service curves. Subsequently, in the next two sections, we will illustrate their utility for the specific problem of speed assignment. This ordering is influenced by the expectation that abstract arrival and service curves may be relevant in SMT formulations of other design problems, or in altogether different settings.

In this section, we will study the Greedy Processing Component (GPC) which is defined in Appendix A.4. The GPC forms the basic unit of analysis in Modular Performance Analysis (MPA). A GPC is characterized by two sets of equations. First, the output arrival and service functions  $\alpha'$  and  $\beta'$  are given in terms of the input arrival and service functions  $\alpha$  and  $\beta$ . Second, upper-bounds on the delay and buffer-space are given in terms of the input arrival and service functions. The equations formalizing them are shown in Appendix A.4.

### 5.3.1 Definition

In this section, we study an *abstract* component denoted as GPC. The abstraction arises out of simultaneously considering multiple *configurations* of the component. Each configuration of the GPC is a standard GPC. Two configurations can differ in their input arrival curves, or input service curves, or both. We denote  $X$  as the set of finitely many configurations, and  $x \in X$  as a particular configuration of a GPC. We now define the abstract curves.

**Definition 5.1: (Abstract Arrival Curve)** Let  $\alpha_x = (\alpha_x^u, \alpha_x^l)$  denote the arrival curve at the input (or output) of a GPC in the configuration  $x \in X$ . Then the input (or output) abstract arrival curve  $\underline{\alpha} = ((\alpha^u)^l, (\alpha^l)^u)$  satisfies the following constraints.

$$(\alpha^u)^l \leq \alpha_x^u, \quad \forall x \in X, \quad (5.1)$$

$$(\alpha^l)^u \geq \alpha_x^l, \quad \forall x \in X. \quad (5.2)$$

The upper abstract arrival curve  $(\alpha^u)^l$  is the lower-bound on the upper arrival curves across all configurations. Similarly, the lower abstract

arrival curve  $(\alpha^l)^u$  is the upper-bound on the lower arrival curves across all configurations.

**Definition 5.2: (Abstract Service Curve)** Let  $\beta_x = (\beta_x^u, \beta_x^l)$  denote the service curve at the input (or output) of a GPC in the configuration  $x \in X$ . Then the input (or output) abstract service curve  $\underline{\beta} = ((\beta^u)^l, (\beta^l)^u)$  satisfies the following constraints.

$$(\beta^u)^l \leq \beta_x^u, \quad \forall x \in X, \quad (5.3)$$

$$(\beta^l)^u \geq \beta_x^l, \quad \forall x \in X. \quad (5.4)$$

The upper abstract service curve  $(\beta^u)^l$  is the lower-bound on the upper service curves across all configurations. Similarly, the lower abstract service curve  $(\beta^l)^u$  is the upper-bound on the lower service curves across all configurations.

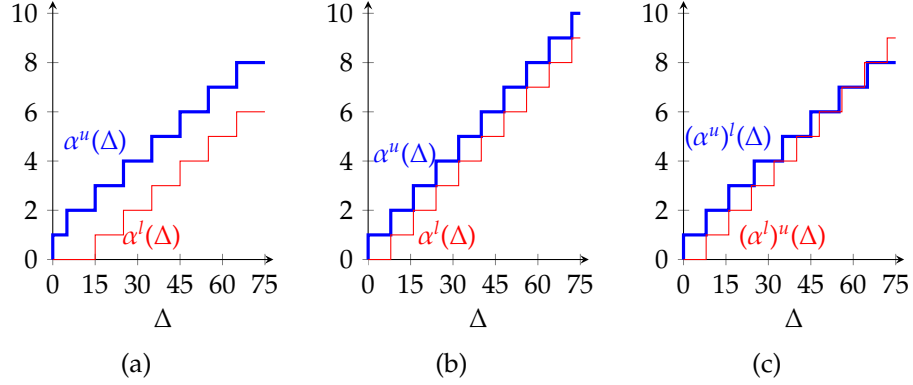
We refer to  $(\alpha^u)^l$  and  $(\beta^u)^l$  as upper curves, and  $(\alpha^l)^u$  and  $(\beta^l)^u$  as lower curves. It is instructive to expand the definition of the abstract curves. For instance, consider the upper abstract arrival curve  $(\alpha^u)^l$ . For each configuration  $x \in X$ , let  $R_x$  be a representative<sup>3</sup> arrival functions at the input of a GPC. Then,  $(\alpha^u)^l$  satisfying (5.1) is given as

$$(\alpha^u)^l(\Delta) = \min_{x \in X} \left\{ \sup_{t \geq 0} \{R_x(t + \Delta) - R_x(t)\} \right\}, \quad \forall \Delta \geq 0. \quad (5.5)$$

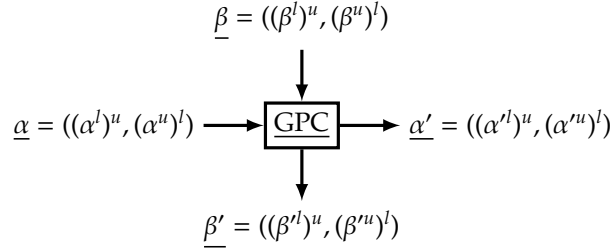
Similarly the other definitions can be expanded. Notice how we consider only *cross-bounds* in defining the curves. For instance in  $(\alpha^u)^l$ , the lower-bound across all configurations and the upper-bound across all times is considered. Thus, there are two distinct levels of abstraction. The inner-level abstracts events in time domain to the interval domain, while the outer-level abstracts the uncertainty in the configuration of the component. As we will see, these two abstractions play very different roles. While the inner abstraction is necessary to analyze worst-case (over time) parameters, the outer abstraction is necessary to analyze incomplete models in an SMT solver. We illustrate these curves with an example.

**Example 5.2:** Consider a periodic task in two configurations  $x_1$  and  $x_2$ . In configuration  $x_1$  the period, the jitter, and the execution time of each job are 10, 5, and 1. In configuration  $x_2$  the corresponding parameters are 8, 0, and 1, respectively.

<sup>3</sup> We assume that for each  $x \in X$ , the abstraction of  $\alpha_x^u$  is tight for given arrival function  $R_x$ , i.e.,  $\alpha_x^u(\Delta) = \sup_{t \geq 0} \{R_x(t + \Delta) - R_x(t)\}$ .



**Fig. 5.2** Illustration of abstract arrival curves for Example 5.2. The arrival curve for a GPC in configurations (a)  $x_1$  and (b)  $x_2$ . (c) The abstract arrival curve of the corresponding GPC.



**Fig. 5.3** Block diagram of the abstract Greedy Processing Component (GPC).

The arrival curve corresponding to the standard GPC for configurations  $x_1$  and  $x_2$  are shown in Figures 5.2(a) and 5.2(b), respectively. The abstract arrival curve according to Definition 5.1 is shown in Figure 5.2(c). Notice how the abstract upper curve contains parts of upper curves of both configurations. Also the abstract lower curve exceeds the abstract higher curve, which is not the case for the standard GPC.

A GPC can be visualized by the block diagram of Figure 5.3. Like in the standard GPC, we need to characterize a GPC by its input-output relations and the delay and buffer bounds. This is the focus of the rest of this section.

### 5.3.2 Input-Output Relations of GPC

The input-output relations specify the dependence of the output abstract curves  $\underline{\alpha}'$  and  $\underline{\beta}'$  on the input abstract curves  $\underline{\alpha}$  and  $\underline{\beta}$ .

**Theorem 5.1:** *The input and output abstract arrival and service curves of GPC are given by the following relations.*

$$(\alpha'^u)^l = \min\{((\alpha^u)^l \otimes (\beta^u)^l) \oslash (\beta^l)^u, (\beta^u)^l\}, \quad (5.6)$$

$$(\alpha'^l)^u = \min\{((\alpha^l)^u \oslash (\beta^u)^l) \otimes (\beta^l)^u, (\beta^l)^u\}, \quad (5.7)$$

$$(\beta'^u)^l = ((\beta^u)^l - (\alpha^l)^u) \bar{\oslash} 0, \quad (5.8)$$

$$(\beta'^l)^u = ((\beta^l)^u - (\alpha^u)^l) \bar{\otimes} 0. \quad (5.9)$$

The operators used in the above equations are defined in Appendix A.1. These equations mirror the corresponding equations for the standard GPC, as described in Appendix A.4. This is a significant property which can be interpreted as follows: The abstract arrival and service curves defined with only cross-bounds sufficiently describe input-output relations. For instance, to compute  $(\alpha'^u)^l$  we only need  $(\alpha^u)^l$ ,  $(\beta^l)^u$  and  $(\beta^u)^l$ . We do not need  $(\alpha^u)^u$ ,  $(\beta^l)^l$  or  $(\beta^u)^u$ , assuming these quantities are correspondingly defined. Thus, to propagate the uncertainty due to multiple configurations of a GPC, from its inputs to outputs, we only require the cross-bounds. This can be considered a *monotonicity principle*.

We illustrate the input-output relations for the Example 5.2. Let the task be executed on a resource that is a TDMA slot with period 8 and slot-size 1. For this setting, we compute the output arrival curves for the two configurations using the standard GPC equations. These are shown in Figures 5.4(a) and 5.4(b). Notice how the TDMA resource further widens the gap between the upper and lower curves. Then, we compute the output abstract arrival curve using (5.6) and (5.7). This is shown in Figure 5.4(c). Indeed, this curve satisfies (5.1) and (5.2). Again, the abstract upper curve contains parts of upper curves of both configurations. Also the abstract lower curve exceeds the abstract higher curve (first at  $\Delta = 100$ ).

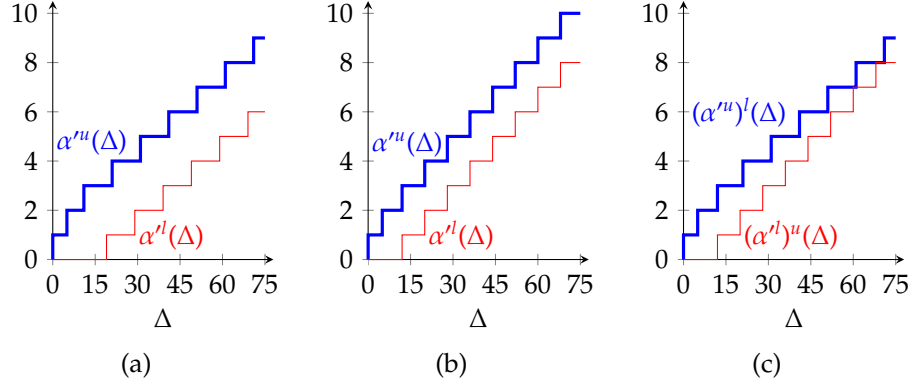
### 5.3.3 Delay and Buffer Bounds with GPC

Before we compute the delay and buffer bounds with GPC, we have to re-define them under the abstraction of multiple configurations. As we will see in the next two sections, these definitions are motivated by the application to analysis of incomplete models in an SMT solver.

**Definition 5.3: (Abstract Bounds)** Let  $\underline{d}_x^{\max}$  and  $\underline{b}_x^{\max}$  denote respectively the worst-case delay and buffer-space in a GPC in configuration  $x \in X$ . Then, the abstract bounds, denoted with underlines, satisfy the following constraints.

$$\underline{d}^{\max} \leq d_x^{\max}, \quad \forall x \in X, \quad (5.10)$$

$$\underline{b}^{\max} \leq b_x^{\max}, \quad \forall x \in X. \quad (5.11)$$



**Fig. 5.4** Illustration of abstract output arrival curves for Example 5.2. The output arrival curve for configuration (a)  $x_1$  and (b)  $x_2$ . (c) The abstract output arrival curve.

Notice that the abstract bounds consider the *weakest* bounds across all configurations, and thereby parallel the cross-bounds in the definition of the abstract curves. In the following result, we show how these bounds can be computed for a GPC.

**Theorem 5.2:** *The abstract bounds of a GPC are given by the following relations.*

$$\underline{d}^{\max} = \text{Del}((\alpha^u)^l, (\beta^l)^u), \quad (5.12)$$

$$\underline{b}^{\max} = \text{Buf}((\alpha^u)^l, (\beta^l)^u). \quad (5.13)$$

In the above equations, the Del and Buf operators are as defined in Appendix A.3. These equations mirror the corresponding equations for the standard GPC, as described in Appendix A.4. The abstract arrival and service curves defined with only cross-bounds sufficiently describe the abstract bounds. For instance, to compute  $\underline{d}^{\max}$  we only need  $(\alpha^u)^l$  and  $(\beta^l)^u$ . We do not need  $(\alpha^u)^u$  or  $(\beta^l)^l$ , assuming these quantities are correspondingly defined. Thus, to propagate the uncertainty due to multiple configurations of a GPC, from its inputs to the computed bounds, we only require the cross-bounds. This can be considered as another *monotonicity principle*.

The two stated monotonicity principles result in a practical advantage. They allow us to extend the computational abstractions and techniques developed for the standard GPC for the analysis of GPC. In particular, an arrival curve  $\alpha$  can be replaced by an abstract arrival curve  $\underline{\alpha}$ , and a service curve  $\beta$  can be replaced by an abstract service curve  $\underline{\beta}$ . The corresponding components of the two curves have the same properties. For instance, both  $\alpha^u$  and  $(\alpha^u)^l$  are sub-additive functions [Wan06]. Finally, a GPC with a single configuration is identical to a standard GPC.

To conclude, we defined abstract arrival and service curves. We showed how they are sufficient to characterize input-output relations and bounds of a GPC. In the next two sections, we will see how to apply this to the speed assignment problem.

## 5.4 Basics of a SMT Solver

In this section, we present the basic notations and concepts in the working of an SMT solver. We adapt these to suit the specific requirements of our problem. A complete and more formal description is available at [BM07, BSST09, Str10].

### 5.4.1 Syntax

A *language* comprises a *signature*  $\Sigma$  and a countable set of *variables*  $\mathcal{V}$ .  $\Sigma$  is a finite set of function symbols  $\Sigma_F$  and predicate symbols  $\Sigma_P$ . Symbols with zero arity are called constant symbols. A *term* is any (recursive) application of a function symbol on other function symbols and variables. An *atomic formula* is the application of a predicate symbol on a set of terms. A *clause* is the disjunction of a set of atomic formulas or their negations. The set of *quantifier free formulas* is the closure of all atomic formulas under the predicate operators of negation, conjunction, disjunction and implication. As a matter of convention, formulas are specified in the CNF template as conjunctions of clauses. In this chapter, we only consider formulas which are quantifier free, i.e., all variables are free of existential and universal quantifiers. In such cases, it is convenient to represent the variables as constant function symbols in  $\Sigma_F$ .

### 5.4.2 Semantics

A *model*  $M$  gives meaning to a language. It interprets the function and predicate symbols in  $\Sigma$  from a given domain of elements  $S$ . As an example, a function symbol  $f \in \Sigma_F$  with arity  $n$  is interpreted in  $M$  as some function  $f^M : S^n \mapsto S$ . The interpretation of a formula  $\varphi$  under  $M$  is denoted as  $\varphi^M$ . A *theory*  $\mathcal{T}$  is a set of models, i.e., a theory  $\mathcal{T}$  constrains the interpretation of the symbols in  $\Sigma$ . A formula  $\varphi$  is said to be *satisfiable* for a theory  $\mathcal{T}$ , if and only if, there exists a model  $M \in \mathcal{T}$  such that  $\varphi^M$  is  $\top$ .<sup>4</sup> This is denoted as  $M \models_{\mathcal{T}} \varphi$ . A clause  $c$  is said to be a *conflict clause* for a given formula  $\varphi$ , if and only if, for any model  $M$ ,  $\neg(c^M) \Rightarrow M \not\models_{\mathcal{T}} \varphi$ .

<sup>4</sup> We use  $\top$  and  $\perp$  as the propositional constants true and false.



A model is said to be *incomplete* if it does not interpret all symbols, and *complete* otherwise.<sup>5</sup> By additionally interpreting some symbols uninterpreted in  $M$ , we obtain a set of models  $\mathbf{M}$ . Any model  $M' \in \mathbf{M}$  is said to be a *refinement* of  $M$ . A formula  $\varphi$  is said to be  *$M$ -satisfiable* for a theory  $\mathcal{T}$ , if and only if, there exists a model  $M' \in \mathbf{M}$  such that  $M' \models_{\mathcal{T}} \varphi$ .

In the context of SMT solvers, it is useful to define additional symbols not in  $\Sigma$  which may be uninterpreted in a model. Such symbols help define abstractions of other function symbols. In this chapter, we will only consider additional propositional function symbols. A model  $M$  assigns (some of) these propositional variables from the set  $\{\top, \perp\}$ . Furthermore, all function symbols other than the propositional variables are implicitly interpreted by deductions in the theory  $\mathcal{T}$ . In this specific case, the propositional variables encode the design space.

### 5.4.3 Working of an SMT Solver

An SMT solver comprises of a SAT solver and a theory solver, which iteratively exchange information. We describe the interaction between these solvers in the  $i$ th iteration.

1. Using an algorithm such as the DPLL procedure [DLL62], the SAT solver computes a model  $M_i$ , i.e., it interprets some of the propositional variables.
2. The theory solver checks if a given formula  $\varphi$  is  $M_i$ -satisfiable. If provably  $\varphi$  is *not*  $M_i$ -satisfiable the solver returns `Unsat` and a conflict clause  $c_i$ . Otherwise, it returns `Sat` and no conflict clause.
3. If the theory solver returns `Sat` and  $M_i$  is complete, we have a solution to the problem. Otherwise, if the disjunction of all received conflict clauses is a tautology, then the problem is infeasible. If neither of these two terminating conditions are satisfied, the solver moves to the next iteration.

For an efficient SMT solver the theory solver must satisfy two properties. First, it must be able to verify that a formula is  $M$ -satisfiable for any incomplete model  $M$ . Second, it must be able to compute a conflict clause if a formula is not  $M$ -satisfiable. There are other desirable properties such as incrementality and back-tracability [BSST09] which we do not consider in this chapter.

<sup>5</sup>Alternatively and equivalently an incomplete model can be considered as a sub-set of the formulas which define a theory [BSST09].

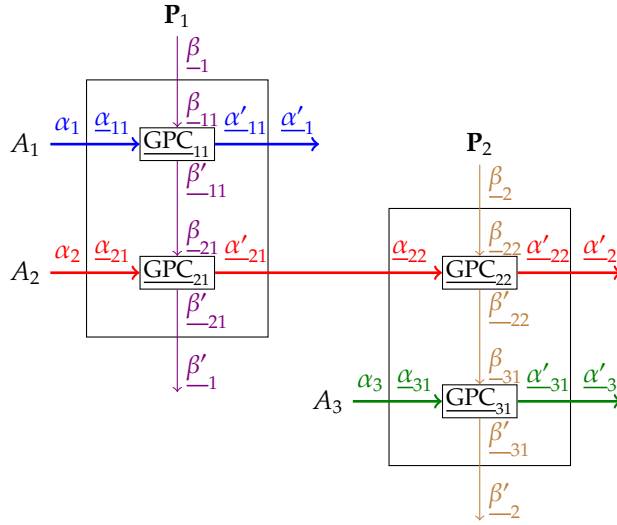


Fig. 5.5 MPA block diagram for Example 5.1. Each task is represented by a GPC. The interconnection defines equivalence between different input and output curves.

## 5.5 SMT Solver for Speed Assignment

In this section, we describe how to build the SMT solver for the speed assignment problem. In particular, we will describe the signature, the interpretation of incomplete models with abstract curves, and the generation of conflict clauses.

### 5.5.1 MPA Block Diagram

In the speed assignment problem, several tasks are executed on multiple processors. We model the execution of each task as a GPC with abstract input and output arrival curves. A GPC and its curves are referenced by sub-scripts matching that of the corresponding task. For instance, task  $T_{ij}$  has an abstract input arrival curve denoted as  $\alpha_{ij}$ . The output of a task is the input of a successor task, if any. Similarly for resources, the output service curve of a GPC is the input service curve of a lower priority GPC, if any. For Example 5.1, the relations between the different variables are illustrated in the block diagram in Figure 5.5. The only curves which are not abstract are the input arrival curves of each application, i.e.,  $\alpha_i$  for each application  $A_i \in A$ . All others are abstract curves whose computation we will describe later.

### 5.5.2 Signature and Model

Recall that the signature  $\Sigma$  comprises of function symbols  $\Sigma_F$  and predicate symbols  $\Sigma_P$ . The set of predicate symbols  $\Sigma_P$  includes the comparison

predicates = and  $\leq$ . The set of function symbols  $\Sigma_F$  includes the following.

1. The set of abstract curves for each GPC as described in the construction of the MPA block diagram and the curve 0. These are constant symbols.
2. The set of operators used in RTC, including  $\otimes$ ,  $\oslash$ ,  $\overline{\otimes}$ ,  $\overline{\oslash}$ , Del and Buf. All these symbols have an arity of 2.
3. For each processor  $\mathbf{P}_i \in \mathbf{P}$ , two constant symbols  $s_i^{\min}$  and  $s_i^{\max}$ .
4. For each processor  $\mathbf{P}_i \in \mathbf{P}$ , the a function  $\phi_i$ . These symbols have an arity of 1.

In addition, we define some propositional variables which encode the design space. For every processor  $\mathbf{P}_i$  and for each speed  $\mathbf{S}_{ij} \in \mathbf{S}_i$ , we define a propositional variable  $p_{ij}$ . As we will see, these variables abstractly represent the speeds of the processors.

The semantics of the symbols in  $\Sigma$  are given by their interpretation in the model. A model assigns values to the symbols from a domain. For our custom theory, the symbols are of different *types* and thus there are multiple domains: (a) the domain of propositional constants  $\{\top, \perp\}$ , (b) the domain of non-negative scalars denoted  $\mathfrak{R}_+$ , and (c) the domain of non-negative non-decreasing functions denoted  $\mathfrak{R}_+^\infty$ .<sup>6</sup> In terms of these domains we can define the interpretation of each function. For instance, in a model  $M$ , the symbol  $\otimes^M : \mathfrak{R}_+^\infty \times \mathfrak{R}_+^\infty \mapsto \mathfrak{R}_+^\infty$ , and the symbol  $\phi_i^M : \mathfrak{R}_+ \mapsto \mathfrak{R}_+$ .

We are interested in only those models which conform to the theory of RTC. This applies the following conditions on the interpretations.

1. The RTC operators are as defined. Further the functions  $\phi_i$  for each processor  $\mathbf{P}_i \in \mathbf{P}$  is as given by the power consumption as a function of speed.
2. For each processor  $\mathbf{P}_i \in \mathbf{P}$ ,

$$s_i^{\min} = \mathbf{S}_{ik}, \quad \text{where } k = \max(\{x \mid (p_{ix} \mapsto \perp) \in M\} \cup \{0\}) + 1, \quad (5.14)$$

$$s_i^{\max} = \mathbf{S}_{ik}, \quad \text{where } k = \min(\{x \mid (p_{ix} \mapsto \top) \in M\} \cup \{\mathbf{S}_i\}). \quad (5.15)$$

This interpretation entails the following constraints on  $\mathcal{T}$ -valid propositional variables.

$$(p_{ij} \mapsto \perp) \in M \Rightarrow (p_{ik} \mapsto \perp) \in M, \quad \forall k < j, \quad (5.16)$$

$$(p_{ij} \mapsto \top) \in M \Rightarrow (p_{ik} \mapsto \top) \in M, \quad \forall k > j. \quad (5.17)$$

<sup>6</sup> To be precise, a theory over multiple domains is a *typed* first-order theory. However, in our case the domain  $\mathfrak{R}_+^\infty$  strictly generalizes propositional variables and non-negative scalars, forming a linear type hierarchy. In this case, with additional theory predicates, the domain can be uniformly considered to be  $\mathfrak{R}_+^\infty$ .

3. For each processor  $\mathbf{P}_i \in \mathbf{P}$ ,

$$(\beta_i^l)^u(\Delta) = s_i^{\max} \Delta, \quad (5.18)$$

$$(\beta_i^u)^l(\Delta) = s_i^{\min} \Delta. \quad (5.19)$$

4. The structural relations between abstract curves derived from the MPA block diagram, i.e., input and output relations of every GPC, and equivalence due to connections between any two GPCs.

From the above rules, we can conclude that a model  $M$  explicitly interprets only the propositional variables  $p_{ij}$  for every processor  $\mathbf{P}_i$  and speed  $\mathbf{S}_{ij}$ . All other interpretations are deduced from the above rules governing the theory. As the SAT solver computes  $M$ , it assigns (some of) the propositional variables which abstractly represent the design space of speeds of processors.

### 5.5.3 Analysis of Incomplete Models

We first identify the formula whose satisfiability we are interested in. Each constraint can be converted to a formula as described below.

- Delay constraints. For an application  $A_i$  with a single task if the delay constraint is  $D_i$ , then this is expressed by a formula  $(\varphi_d)_i$  given as

$$(\varphi_d)_i = \left( \underline{d}^{\max}(\text{GPC}_{i1}) \leq D_i \right). \quad (5.20)$$

For an application of multiple tasks this can be extended with known results from Network Calculus [LBT01] as

$$(\varphi_d)_i = \left( \text{Del} \left( \alpha_i^u, (\beta_{i1}^l)^u \otimes (\beta_{i2}^l)^u \otimes \dots \otimes (\beta_{in}^n)^u \right) \leq D_i \right), \quad (5.21)$$

where the tasks of  $A_i$  are  $\{T_{i1}, T_{i2}, \dots, T_{in}\}$ .

- Buffer constraints. For a task  $T_{ij}$  if the input buffer-space constraint is  $B_{ij}$ , then this is expressed by the formula  $(\varphi_b)_{ij}$  given as

$$(\varphi_b)_{ij} = \left( \underline{b}^{\max}(\text{GPC}_{ij}) \leq B_{ij} \right). \quad (5.22)$$

- Energy constraints. For a processor  $\mathbf{P}_i$  if the energy consumption constraint is  $E_i^u$ , then this is expressed by the formula  $(\varphi_E)_i$  given as

$$(\varphi_E)_i = \left( \underline{E}^u((\beta_i^u)^l, s_i^{\min}, \phi_i, \Delta) \leq E_i^u(\Delta) \right), \quad (5.23)$$

where,

$$\underline{E}^u(\beta, s, \phi, \Delta) = \beta(\Delta) \times (\phi(0) - \phi(s)) + \Delta \times \phi(s). \quad (5.24)$$

The overall formula  $\varphi$  is the conjunction of the formulas for each of the specified constraints.

To conclude, given an incomplete model  $M$ , we propagate the uncertainty in the speeds of the processors through the MPA block diagram with abstract curves defined by the GPC rules. Then, with the functions  $\underline{b}^{\max}$ ,  $\underline{d}^{\max}$ , and  $\underline{E}^u$ , we can check  $M$ -satisfiability of  $\varphi$ . Thus, the theory solver is equipped to return Sat or Unsat for any model  $M$ .

### 5.5.4 Generating Conflict Clauses

If the theory solver finds that  $\varphi$  is not  $M$ -satisfiable, then it must compute a conflict clause which encodes the reason for the non-satisfiability. This clause helps the SAT solver in future choices of models. Recall that the conflict clause  $c$  is the disjunction of literals, such that  $\neg(c^M) \Rightarrow M \not\models_{\mathcal{T}} \varphi$ . For higher efficiency, it is desirable that the conflict clause has fewer literals. Implementations of SMT solvers allow multiple conflict clauses to be returned. Thus, we compute a separate conflict clause for non-satisfiability of the formula corresponding to each constraint.

In our case, the conflict clause contains only the propositional variables  $p_{ij}$  which are explicitly interpreted in the model  $M$ . A propositional variable in a conflict clause must have a polarity based on its interpretation in  $M$ . For instance, if  $(p_{ij} \mapsto \top) \in M$ , then only  $p_{ij}$  can be a literal in the conflict clause, and if  $(p_{ij} \mapsto \perp) \in M$ , then only  $\neg p_{ij}$  can be a literal in the conflict clause. Thus, generating the conflict clause is equivalent to finding a sub-set of propositional variables which are interpreted in  $M$ .

To find the required sub-set we devise a compositional strategy. To this end, we define *conflict functions*  $\mathbf{C}(\cdot)$  which map an abstract curve to a sub-set of the propositional variables. First we define *input-output* conflict functions, i.e., conflict functions of output abstract curves of a GPC in terms of the conflict functions of its input abstract curves. In the following relations we drop the subscripts indicating the index of the GPC.

$$\mathbf{C}((\alpha'^u)^l) = \mathbf{C}((\alpha^u)^l) \cup \mathbf{C}((\beta^u)^l) \cup \mathbf{C}((\beta^l)^u) \quad (5.25)$$

$$\mathbf{C}((\alpha'^l)^u) = \mathbf{C}((\alpha^l)^u) \cup \mathbf{C}((\beta^u)^l) \cup \mathbf{C}((\beta^l)^u) \quad (5.26)$$

$$\mathbf{C}((\beta''^u)^l) = \mathbf{C}((\alpha^l)^u) \cup \mathbf{C}((\beta^u)^l) \quad (5.27)$$

$$\mathbf{C}((\beta'^l)^u) = \mathbf{C}((\alpha^u)^l) \cup \mathbf{C}((\beta^l)^u) \quad (5.28)$$

In addition to the compositional relations, we also have *boundary* conflict functions. For the applications, the input arrival curves are given and do not depend on the model. Thus, for every application we have an

empty conflict function, i.e.,

$$\mathbf{C}(\alpha_i^l) = \mathbf{C}(\alpha_i^u) = \{\}, \quad \forall A_i \in A. \quad (5.29)$$

For the processors, the input abstract service curves depend on the speed of the processor as given in (5.18) and (5.19). Thus, the conflict functions include the propositional variables as given below.

$$\mathbf{C}((\beta_i^l)^u) = \bigcup_{(p_{ij} \mapsto \top) \in M} p_{ij}, \quad \forall \mathbf{P}_i \in \mathbf{P}, \quad (5.30)$$

$$\mathbf{C}((\beta_i^u)^l) = \bigcup_{(p_{ij} \mapsto \perp) \in M} p_{ij}, \quad \forall \mathbf{P}_i \in \mathbf{P}. \quad (5.31)$$

Given the above conflict functions, we are now ready to derive the conflict clauses for the non-satisfaction of each constraint. To this end, we extend the notion of conflict functions to formulas. For instance,  $\mathbf{C}((\varphi_d)_i)$  is the set of propositional variables which form the conflict clause if  $(\varphi_d)_i$  is not  $M$ -satisfiable.

**Theorem 5.3:** *For the formulas corresponding to delay, buffer and energy constraints, the conflict functions are as given below.*

$$\mathbf{C}((\varphi_d)_i) = \bigcup_{\forall T_{ij} \in A_i} \mathbf{C}((\beta_{ij}^l)^u), \quad (5.32)$$

$$\mathbf{C}((\varphi_b)_{ij}) = \mathbf{C}((\alpha_{ij}^u)^l) \cup \mathbf{C}((\beta_{ij}^l)^u), \quad (5.33)$$

$$\mathbf{C}((\varphi_E)_i) = \left( \bigcup_{\forall m(T_{ab})=\mathbf{P}_i} \mathbf{C}((\alpha_{ab}^u)^l) \right) \cup \mathbf{C}((\beta_i^u)^l) \quad (5.34)$$

In the above result, the conflict functions of each formula is given by the union of the conflict functions of different curves. By recursively substituting the conflict functions as defined in input-output relations (5.25) to (5.28) and the boundary conflict functions (5.29) to (5.31), we arrive at the conflict clauses.

This concludes the presentation of the SMT solver. To summarize, we described the function and predicate symbols, and their interpretation by a model under the theory of RTC. In particular, we used abstract curves and interpreted them using the relations of GPC. The design space of speeds of the processor are encoded in propositional variables which are assigned by the SAT solver. Conflict clauses are computed for each non-satisfiable constraint using the defined conflict functions.

## 5.6 Experimental Results

In this section we will discuss our implementation of the SMT solver and then present experimental results.

### 5.6.1 Implementation of the Solver

We implemented the SMT solver for the theory of RTC by interfacing the OpenSMT solver [BPST10] and the Modular Performance Analysis toolbox [Wan06]. The OpenSMT solver from University of Lugano is designed specifically for supporting custom theories by providing a standard interface schema with an efficient SAT engine based on miniSAT [ES05]. On the other side, the theory solver is built as an application over the MPA toolbox. This is possible because of the definition of the abstract curves and their monotonicity principles.

### 5.6.2 Illustrating Example

We begin with a small example where we detail the interaction between the SAT solver and the theory solver.

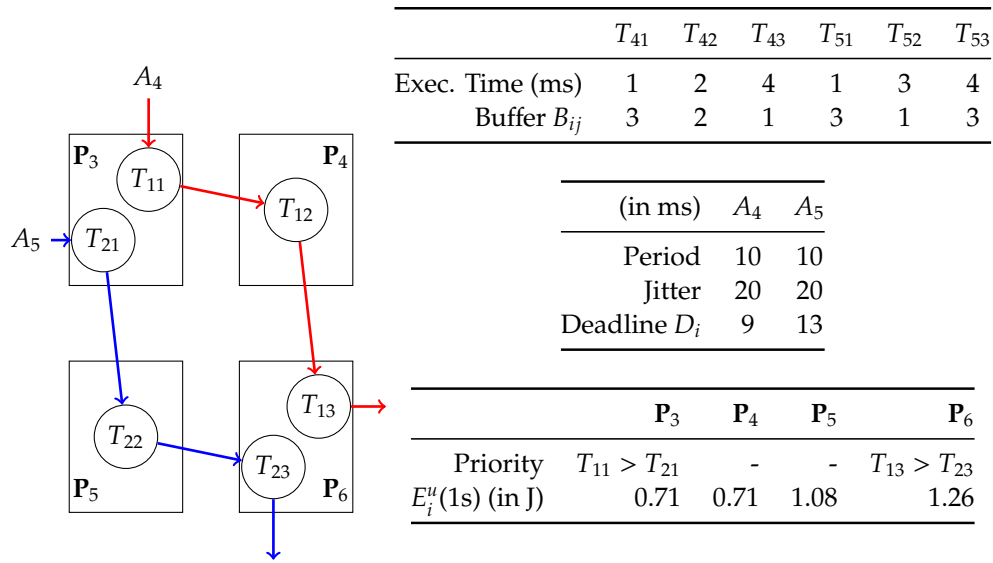
**Example 5.3:** *Two applications  $A_4$  and  $A_5$  with three tasks each execute on four processors  $P_3, P_4, P_5$  and  $P_6$ . The block diagram in Figure 5.6(a) shows the mapping of tasks to processors. The application and task parameters are shown in Figure 5.6(a). Each processor has the same set of speeds  $\mathbf{S} = (1, 1.5, 2)$  and power function  $\phi(s) = (s^{2.5} + 0.5)W$ . Each processor has a periodic energy constraint with period of 1s. The values of  $E_i^u(1s)$  are shown in Figure 5.6(a).*

The design space of the speed assignments is of size  $3^4 = 81$ . By exhaustive search we found that none of the 81 speed assignments satisfies all the constraints, i.e., the problem is unsatisfiable. We will now demonstrate how the SMT solver comes to the same conclusion.

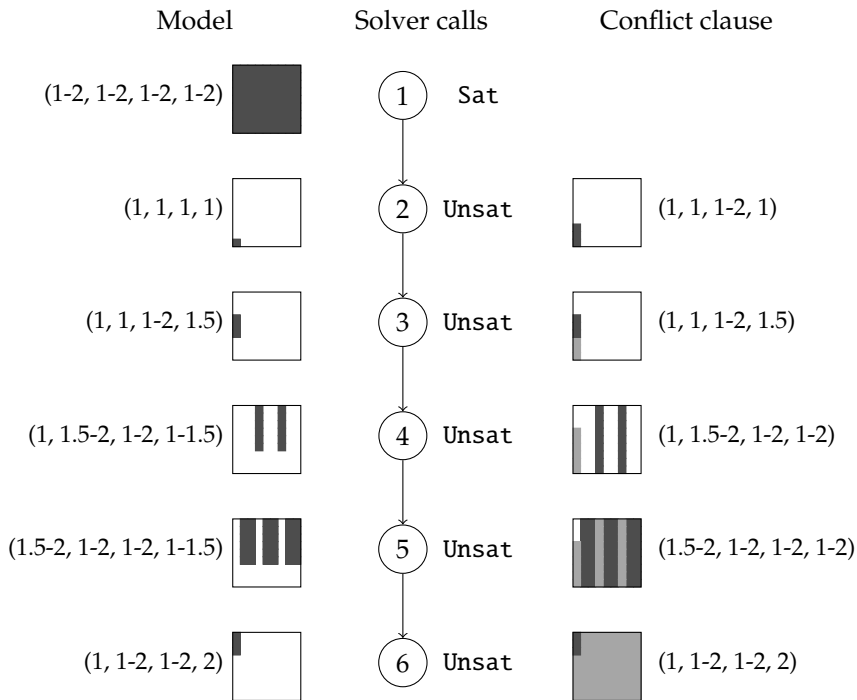
To visualize the interaction between the SAT and theory solvers, we represent the model and the conflict clause on a square of cells representing the design space of the 81 speed assignments. A model  $M$  is represented by all speed assignments which are obtained under refinement of  $M$ . A conflict clause  $c$  is represented by all speed assignments which entail the conflict clause.

We show the details of the solver calls in Figure 5.6(b). The following observations can be made from this example.

- In 6 solver calls, the solver is able to conclude that none of the 81 speed assignments satisfies all the constraints.



(a) Block diagram and parameters for Example 5.3.



(b) The interaction between the SAT solver and the theory solver for Example 5.3. The model and conflict clause are represented by a box of 81 cells representing the 81 speed assignments. For a model  $M$ , a cell is shaded black if the corresponding speed assignment is  $M$ -satisfiable. For each solver call, the theory solver either returns Sat or Unsat. If it returns Unsat, it generates one or more conflict clauses. For each generated conflict clause  $c$ , a cell is shaded black if the corresponding speed assignment entails  $c$ . Also a cell is shaded gray if it was shaded black in any of the earlier solver calls.

Fig. 5.6 Illustration of interaction between SAT and theory solvers for Example 5.3



- In the first solver call, the theory solver returns Sat for the infeasible problem instance. This illustrates that the abstraction may not always be tight and may require refinement for an accurate analysis.
- In many solver calls, the model represents multiple speed assignments. Such incomplete models are analyzed by the theory solver. This is the first source of efficiency.
- In solver calls 2, 4 and 5, the generated conflict clause covers speed assignments not in the model. In other words, the theory solver is able to conclude that other designs not represented by the model are also infeasible. This is the second source of efficiency.
- In the last solver call, the solver concludes that there is no feasible speed assignment as the disjunction of all generated conflict clauses is a tautology.
- Finally, the process also generates a *proof* for in-feasibility. In particular, it identifies the constraints which are not satisfied in the different solver calls and their implication on the feasibility of designs in the design space.

### 5.6.3 Larger Problem Instances

In the previous example we noted the following two sources of efficiency in the use of an SMT solver.

- (a) The interpretation and analysis of incomplete models.
- (b) The generation of conflict clauses which generalize beyond the model.

To understand the individual contribution of these two sources, we consider three variants of the solver.

$S_{AB}$ : This solver is as described in this chapter.

$S_A$ : In this solver, we only interpret and analyze complete models. Whenever the SAT solver generates an incomplete model, the theory solver trivially generates the output Sat. For complete models, the system is analyzed and conflict clauses are generated as described. To build such a solver, we do not need the notion of abstract curves developed in Section 5.3.

$S_B$ : In this solver, we analyze incomplete models, but compute trivial conflict clauses which include all propositional variables in the

#	Type	$S_{AB}$	$S_A$	$S_B$	Random
1	Sat	64	9	38	38
2	Sat	163	108	4509	122
3	Sat	77	30	41	278
4	Sat	153	434	406	980
5	Sat	289	12195	839	3166
6	Sat	259	1067	552	40771
7	Sat	366	405	3122	>50k
8	Sat	409	1367	4269	>50k
9	Sat	521	>50k	7931	>50k
10	Unsat	1	>50k	1	>50k
11	Unsat	80	>50k	98	>50k
12	Unsat	306	>50k	852	>50k
13	Unsat	416	>50k	6210	>50k

**Tab. 5.1** Number of solver calls until termination for the different variants of the solver for different problem . Each problem instance has about  $4 \times 10^{17}$  speed assignments. When the solver calls exceeded 50,000 the problem was terminated.

model. In other words, the conflict clause does not generalize beyond the given model. To build such a solver, we do not need the notion of conflict functions developed in Section 5.5.4.

Note that all three variants of the solvers are sound. However, we expect their performance to be different. To evaluate this we consider 13 problem instances with 25 different processors and 5 distinct speed levels for each processor. For each problem instance, the design space contains  $5^{25} \approx 3 \times 10^{17}$  different speed assignments. We consider execution of 10 – 25 applications of 3 – 5 tasks each. The binding of the tasks and their properties are randomly chosen. The constraints on delay, buffer-space and energy are also randomly chosen. For each of the 13 problem instances, we execute the three variants of the solvers multiple times. In addition, we also use a “Random” solver which picks up a random speed assignment and checks its feasibility. If the speed assignment is feasible, it terminates. The average number of solver calls<sup>7</sup> for the four solvers are tabulated in Table 5.1.

We make the following observations from these results.

- $S_{AB}$  solves the problem (both for Sat and Unsat instances) within a few hundred solver calls. On an average, a solver call takes about 2 seconds on a desktop computer with 4GiB RAM and a 2.0GHz processor. Thereby, most problems are solved within 20 minutes. Given the large size of the considered design spaces, the computational efficiency of  $S_{AB}$  is encouraging.

<sup>7</sup> For variant  $S_A$  a solver call is accounted only for complete models.

- The comparison between  $S_A$  and  $S_B$  is inconclusive. In problem instances where a large fraction of the design space is feasible,  $S_A$  performs very well and sometimes better than  $S_{AB}$ . This can be attributed to the way  $S_A$  works: A solver call is initiated only for complete models. If the design space is largely feasible this approach can arrive at a solution earlier than in the incremental approach of  $S_{AB}$  and  $S_B$ . In most other problem instances, especially ones which are infeasible,  $S_A$  performs significantly worse than  $S_B$ .
- $S_{AB}$  performs much better than both  $S_A$  and  $S_B$  for most problem instances. This demonstrates that analysis of incomplete models and the generation of conflict clauses are complementary contributions towards efficient exploration of the design space.

To conclude, we illustrated how the designed solver works for the Example 5.3. With experimental results for large design spaces we confirmed the computational efficiency of the solver. We also verified that both the analysis of incomplete models and the generation of conflict clauses contribute to the efficiency of the solver.

## 5.7 Summary

The design of distributed real-time CPSs necessitates finding feasible solutions to large constrained problems. The speed assignment problem with timing constraints of applications, and buffer-space and energy constraints of the processors, does not admit trivial solution strategies. We proposed the use of SMT solvers to optimally solve such a problem efficiently. If the problem is infeasible, the solver provides a proof which can direct a subsequent design iteration.

The key enabler was the definition and analysis of abstract arrival and service curves. These abstract curves quantify the uncertainty in timing properties due to simultaneous analysis of multiple designs. By defining the GPC we propagated this uncertainty to output curves and computed bounds. Enabled with this, we can analyze incomplete models and generate conflict clauses.

In the previous chapter we showed how the abstractions of a thermally-clipped processor and a critical trace can analyze timing guarantees in the presence of large run-time variability. In this chapter, the abstract arrival and service curves enabled the representation and analysis of multiple designs. From this part of the thesis, we conclude that *abstraction* forms an effective strategy in analysis of complex real-time CPSs.

## Appendix

### Proof of Theorem 5.1

Consider the following equation.

$$(\alpha'^u)^l = \min \left\{ ((\alpha^u)^l \otimes (\beta^u)^l) \oslash (\beta^l)^u, (\beta^u)^l \right\}. \quad (5.35)$$

To show the above equation, we need to show the following property where  $X$  is the set of configurations of the GPC.

$$\min_{x \in X} \alpha'_x{}^u \geq (\alpha'^u)^l \quad (5.36)$$

$$\Rightarrow \min_{x \in X} ((\alpha_x^u \otimes \beta_x^u) \oslash \beta_x^l) \geq \left( \left( \min_{x \in X} \alpha_x^u \right) \otimes \left( \min_{x \in X} \beta_x^u \right) \right) \oslash \left( \max_{x \in X} \beta_x^l \right). \quad (5.37)$$

We need to expand the definition of the operators as follows.

$$((f \otimes g) \oslash h)(\Delta) = \sup_{0 \leq \lambda} \left\{ \inf_{0 \leq \mu \leq \lambda + \Delta} \{f(\mu) + g(\lambda + \Delta - \mu)\} - h(\lambda) \right\} \quad (5.38)$$

Clearly, this function is monotonically non-decreasing in the functions  $f$  and  $g$ , and non-increasing in the function  $h$ . Thus, (5.37) holds.

Similarly, based on the monotonicity properties of the operators  $\otimes$ ,  $\oslash$ ,  $\bar{\otimes}$  and  $\bar{\oslash}$  (as defined in Appendix A) we can show the relations of Theorem 5.1.  $\square$

### Proof of Theorem 5.2

Consider the delay bound.

$$\underline{d}^{\max} = \text{Del}((\alpha^u)^l, (\beta^l)^u). \quad (5.39)$$

To show the above equation, we need to show the following property where  $X$  is the set of configurations of the GPC.

$$\min_{x \in X} d_x^{\max} \geq \underline{d}^{\max} \quad (5.40)$$

$$\Rightarrow \min_{x \in X} (\text{Del}(\alpha_x^u, \beta_x^l)) \geq \text{Del} \left( \min_{x \in X} \alpha_x^u, \max_{x \in X} \beta_x^l \right). \quad (5.41)$$

We need to expand the definition of the Del operator as follows.

$$\text{Del}(f, g) = \sup_{\lambda \geq 0} \{ \inf \{ \tau \geq 0 : f(\lambda) \leq g(\lambda + \tau) \} \}. \quad (5.42)$$

Clearly, the above function is monotonically non-decreasing in the function  $f$  and non-increasing in the function  $g$ . Thus, (5.41) holds.

Similarly, the buffer bound can be proved based on the monotonicity of the Buf function as defined in Appendix A.  $\square$

**Proof of Theorem 5.3**

Consider the formula  $(\varphi_d)_i$  corresponding to the delay bound of application  $A_i$ . Let the considered model be  $M$ . If for the given model,  $(\varphi_d)_i^M$  is  $\perp$ , then we have the following condition.

$$\text{De1} \left( (\alpha_i^u)^M, ((\beta_{i1}^l)^u)^M \otimes ((\beta_{i2}^l)^u)^M \otimes \dots \otimes ((\beta_{in}^l)^u)^M \right) > D_i, \quad (5.43)$$

where the tasks of  $A_i$  are  $\{T_{i1}, T_{i2}, \dots, T_{in}\}$ . From the previous proof we know that the De1 is monotonically non-decreasing in the first parameter and monotonically non-increasing in the second parameter, and  $\otimes$  is monotonically non-decreasing in both operands. Then, we have the following condition.

$$\left( \alpha_i^u \geq (\alpha_i^u)^M \right) \wedge \left( (\beta_{i1}^l)^u \leq ((\beta_{i1}^l)^u)^M \right) \wedge \dots \wedge \left( (\beta_{in}^l)^u \leq ((\beta_{in}^l)^u)^M \right) \Rightarrow \neg ((\varphi_d)_i)^M. \quad (5.44)$$

Now we describe how to expand the L.H.S. of the above entailment. In particular, we will show how the conflict functions on abstract curves relate to the L.H.S.

1.  $(\alpha_i^u \geq (\alpha_i^u)^M)$  is a tautology as the input arrival curve of the application is independent of  $M$ . This leads to the empty conflict function of  $\alpha_i^u$  as in (5.29).
2. If for some  $j$ ,  $\beta_{ij}$  is the full resource availability of some processor  $\mathbf{P}_a$ , then we have the following condition.

$$(\beta_a^l \leq (\beta_a^l)^M) \Rightarrow ((\beta_{ij}^l)^u \leq ((\beta_{ij}^l)^u)^M). \quad (5.45)$$

Further the L.H.S. can be expanded as follows.

$$\bigwedge_{\{p_{ab} \mid (p_{ab} \mapsto \top) \in M\}} p_{ab} \Rightarrow (\beta_a^l \leq (\beta_a^l)^M). \quad (5.46)$$

Hence, the conflict function on  $\beta^l$  is as defined in (5.30).

3. If for some  $j$ ,  $\beta_{ij}$  is the remaining service curve of some  $\underline{\text{GPC}}_{ab}$ , then we have the following condition.

$$\left( (\beta_{ab}^l)^u \leq ((\beta_{ab}^l)^u)^M \right) \Rightarrow \left( (\beta_{ij}^l)^u \leq ((\beta_{ij}^l)^u)^M \right). \quad (5.47)$$

Given the input-output relation of a  $\underline{\text{GPC}}$  in Theorem 5.1, we can expand the L.H.S. of the above entailment as follows.

$$\left( (\beta_{ab}^l)^u \leq ((\beta_{ab}^l)^u)^M \right) \wedge \left( (\alpha_{ab}^u)^l \geq ((\alpha_{ab}^u)^l)^M \right) \Rightarrow \left( (\beta_{ab}^l)^u \leq ((\beta_{ab}^l)^u)^M \right). \quad (5.48)$$

The above operation from the outputs of a  $\underline{\text{GPC}}$  to its inputs, is formalized in the relation of the conflict functions in (5.28). Similarly, we recursively apply (5.25) to (5.28) to expand the entailment.

Thus, the entailment of (5.44) is expanded using the rules on the conflict functions to arrive at a grounded formula on the predicates interpreted in  $M$ . This formula is given by the conjunction of the predicates, with appropriate polarity, in the conflict function of  $(\varphi_d)_i$  as defined in (5.32).

Similarly, we can prove the correctness of the conflict functions of formulas corresponding to the buffer and energy bounds.  $\square$

## **Part III**

# **Exporting Variability through Richer Guarantees**





# 6

## The Settling-Time Metric

### 6.1 Introduction

In some real-time CPSs, jobs may be expected to miss their deadlines, inevitably. Neither an adaptive run-time manager nor a very accurate analysis may prevent such deadline misses. Often such a system is considered to be unschedulable and deemed unsuitable for an application which requires hard timing guarantees. We argue that such systems merit greater attention and that richer guarantees must be defined to quantify their behavior. To this end, we propose the settling-time metric in this chapter.

The focus on richer guarantees is motivated by the satisfaction of two assumptions: One, if the deadline misses are only seldom observed, and two, if the application can tolerate a certain number of deadline misses and still guarantee correct functionality. We discuss these two reasons briefly.

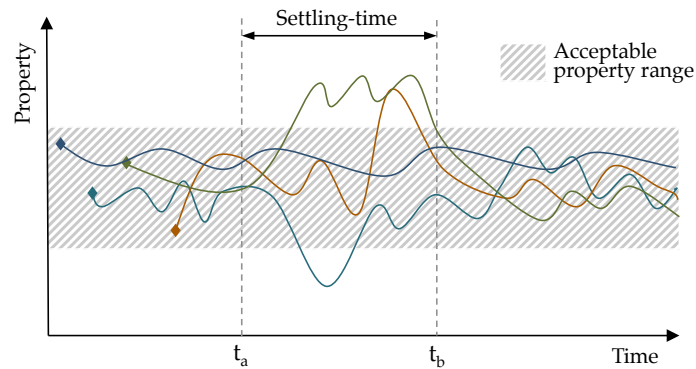
- Deadline misses may seldom be observed in CPSs where the timing models exhibit a dual nature: They conform to a *nominal* model at most times, which is violated when certain *rare-events* occur. An example is the dependence of execution time on the cache contents. The execution time of a job can be exceptionally higher with a “cold cache”, which is encountered during the first instance of the task. Consequently, it is plausible that the first few jobs miss their deadlines while the others do not. Another example of a rare-event is the interference from an infrequent task such as a failure recovery mechanism or a garbage collector.

- A class of real-time applications can tolerate a certain number of deadline misses and still guarantee correct functionality. As an example consider networked control systems (NCSs) where a controlled system, also called a plant, cannot be guaranteed to be *stable* if the (end-to-end) delay between sensing and actuating always exceeds a specified relative deadline. However, the plant can remain stable even if the delays of some of the control signals exceed the deadline. Indeed, it has been shown that the patterns of deadline hits and misses which guarantee a particular type of stability, called exponential stability, form a regular language [WA07].

In both the above cases, it is essential that we provide *hard* guarantees. For instance, however seldom, the WCET with a cold cache must be analyzed and bounded. Similarly, for a NCS the worst-case pattern of deadline hits and misses must be identified and checked to guarantee plant stability. Thus, while we aim to extend the schedulability guarantee, we cannot lose the ability to provide hard timing guarantees to verify the resultant CPSs. This invalidates the use of stochastic approaches which guarantee metrics such as average failure rates [BBB03], consider probabilistic execution times [TDS<sup>+</sup>95], or probabilistic arrival times [Jia06].

A closely related stream of research is the study of *overload* in real-time scheduling. An overload is said to occur when the system capacity is inadequate to meet the demand, and can result in deadline misses. Overloads are not anticipated but may occur exceptionally during runtime. Different metrics have been proposed to quantify performance during an overload. One approach is to maximize the cumulative value function [BPB<sup>+</sup>00, BSS95], where the value function maps the finish time of a job to a certain value or reward. Other works have considered the *firm model* wherein a job finishing after its deadline has zero value. For the firm model the effective processor utilization (EPU) has been studied [BH97]. The EPU is the fraction of time the resource spends executing jobs which finish on or before their deadlines. Another metric is the  $(m, k)$ -firm guarantee, which specifies that at least  $m$  jobs out of any  $k$  consecutive jobs must meet their deadlines [HR95]. This was generalized to consider more intricate patterns in [BBL01]. Different schedulers have been studied to maximize these proposed metrics.

A constant factor in these and related works is the assumption that the overload persists. For instance, an overload caused by a higher than expected execution time of a task is assumed for *all* jobs of that task. Under this setting, the challenge is to *monitor* the incidence of overload, and if an overload occurs to *adapt* the schedule to maximize the metric of interest.



**Fig. 6.1** Illustration of settling-time. Different traces of a property of interest are plotted over time. A rare-event occurs at time  $t_a$ . In response, in some traces, the property moves outside the acceptable range. However, after time  $t_b$ , for all the traces the property is again within the acceptable range. The settling-time then is  $(t_b - t_a)$ .

In contrast, we interpret overload as caused due to rare-events. The rare-events and their effects are assumed to be transient. In the absence of rare-events the system is not under overload and works nominally, wherein no deadlines are missed. If a rare-event occurs, deadlines may be missed, but only until a certain time. Two rare-events must be separated by a minimum inter-arrival time. In other words, we argue for the deliberate separation of the timing model into two parts: a nominal model which is satisfied most of the time, and a rare-event model which can occur infrequently but with bounded minimum inter-arrival times. This careful separation can isolate the rare and transient conditions under which deadlines may be missed. As discussed, we believe a transient model better models the platform and application variabilities in real-time CPSs.

Our viewpoint is motivated from control systems, where the effect and response to incident disturbances are studied. More specifically a controlled system, also called a plant, is designed to remain in a desirable *stable* state. If certain disturbances arise, the plant may be evicted out of the stable state. Then it is of interest to understand how *long* the plant would take to return to its stable state. This duration is referred to as the *settling-time*. For a generalized setting, we visualize the notion of settling-time in Figure 6.1. Indeed, the settling-time depends on both the plant dynamics and the disturbance itself. Furthermore, in the presence of variability, the computation of settling-time must consider all possible system traces.

### 6.1.1 Contributions

Deriving from this analogy to a control system, we define settling-time for a real-time system as the longest time for which jobs can miss their deadlines after the occurrence of a rare-event. A rare-event could be because of higher than expected execution demand or lower than expected resource availability. We characterize both these classes of rare-events within the proposed Rare Events with Settling-Time (REST) framework. In the REST framework, execution demand and resource availability are modeled in the abstract interval domain proposed in Network Calculus [LBT01], which can adequately model variability.

For the REST framework, we provide an analytical method to derive the settling-time for a single task. We also compute overshoot metrics such as the maximum tardiness and the maximum number of jobs which can miss their deadline after the occurrence of a rare-event.

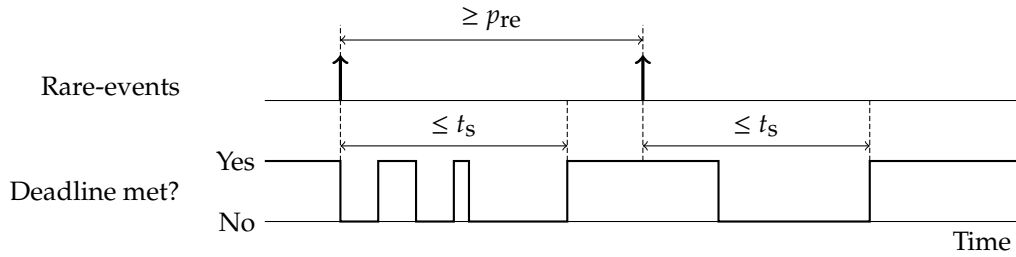
We extend the analysis to multiple tasks scheduled with fixed-priority and EDF schedulers. We prove that the EDF scheduler optimally minimizes settling-time, independent of the time and properties of a rare-event. This extends the known optimality of the EDF scheduler under schedulability [LL73], but it counters the observed poor performance of EDF scheduler under persistent overload [Loc86]. We also show that the Rate-Monotonic (RM) priority assignment is the optimal choice for periodic implicit deadline task-sets, again extending the known optimality under schedulability. However, this optimality does not extend to task-sets with explicit deadlines.

The rest of the chapter is organized as follows. In Section 6.2 we formally define rare-events and settling-time. In Section 6.3 we compute the settling-time for the defined rare-event model for a single task. We extend this to multiple tasks scheduled with an EDF scheduler or a fixed-priority scheduler in Section 6.4. Finally, we summarize in Section 6.5 and include proofs of the results in an appendix.

## 6.2 Rare-Events with Settling-Time (REST)

In this section, we formally define rare-events and settling-time. We refer to this as the rare-events with Settling-Time (REST) framework.

In the REST framework, the timing properties of the CPS are specified by two models, namely a *nominal model* and a *rare-event model*. The nominal model characterizes the standard parameters such as bounds on inter-arrival times of a task and its execution time. In our specific case, the nominal model specifies the execution demands of tasks with



**Fig. 6.2** Definition of the REST framework. Rare-events can occur with a minimum inter-arrival distance of  $p_{re}$  time units. The deadlines of jobs may be missed only up to  $t_s$  time units after the start of the rare-event.

an arrival curve denoted as  $\alpha$ , and the resource availability with a service curve denoted as  $\beta$ . These are as defined in Appendix A.2. We assume that the relative deadlines of all jobs of a task are the same, denoted as  $D$ . This model is nominal in the sense that it is satisfied *often*. However, under some exceptional conditions, a rare-event can occur which deviates the timing properties from the nominal model, and is characterized by the rare-event model. We will subsequently characterize different kinds of rare-events. Two rare-events must be separated by at least  $p_{re}$  time units.

Under exclusive conformance of the nominal model, all desired timing properties are satisfied. In this chapter, we only consider the property of all jobs meeting their respective deadlines. Thus, if the nominal model is always satisfied, all jobs meet their deadlines. However, when and after a rare-event occurs, one or more jobs may miss their deadlines. The *settling-time* denoted  $t_s$  is the maximum length of time up to which jobs can miss their deadlines after the occurrence of a rare-event. The REST framework is visualized in Figure 6.2.

For given nominal and rare-event models, if  $t_s = 0$  then we say that the real-time system is *unconditionally stable*. On the other hand, if  $t_s \geq p_{re}$  then we say that the real-time system is *unstable*. In interesting cases, we expect *conditionally stable* systems where  $t_s < p_{re}$ . The aim in this chapter is to compute  $t_s$  for such cases. With this assumption, it suffices to focus on the effect of a *single* rare-event on the settling-time. We will follow this convention throughout the rest of the chapter. In the remainder of this section, we define two specific kinds of rare-events, namely *demand overflow* and *supply shortage* rare-events.

### 6.2.1 Demand Overflow

A demand overflow rare-event is a rare-event where the execution demand of a task exceeds its nominal model. This could be because of two reasons: (a) a job may arrive with a larger than expected execution

demand, or (b) additional unexpected jobs may arrive. Rare-events are *instantaneous*, i.e., all jobs exceeding their execution demand arrive at the same time instant and/or all additional jobs arrive at the same time instant. Recall also that we only analyze the incidence of a single rare-event. Finally, we assume that any additional jobs caused by the rare-event of a task have the same relative deadline as the other jobs of that task.

A demand overflow rare-event is characterized by an execution demand, denoted as  $R_{re}$ , which is the upper-bound on the additional cumulative execution demand introduced by the rare-event. Consider a task with an arrival curve  $\alpha$  and a demand overflow rare-event with execution demand  $R_{re}$ . Let a rare-event occur at time  $t^*$ . Let  $R(t)$  and  $\widehat{R}(t)$  denote the nominal and cumulative (sum of nominal and rare-event) arrival functions of the task.<sup>1</sup> Then,  $\widehat{R}(t)$  must satisfy the following conditions.

$$\widehat{R}(t) = R(t), \quad 0 \leq t \leq t^*, \quad (6.1)$$

$$\leq R(t) + R_{re}, \quad t > t^*. \quad (6.2)$$

Just as  $R(t)$  is abstracted by the arrival curve  $\alpha(\Delta)$ , we can abstract  $\widehat{R}(t)$  by the *rare-event-aware arrival curve* denoted  $\widehat{\alpha}$ . This is formalized as follows.

$$\widehat{R}(t + \Delta) - \widehat{R}(t) \leq \widehat{\alpha}(\Delta), \quad \forall t, \Delta \geq 0. \quad (6.3)$$

In the following, we show how to compute a valid  $\widehat{\alpha}$ .

**Theorem 6.1:** *The rare-event-aware arrival curve  $\widehat{\alpha}$  is given in terms of the nominal arrival-curve  $\alpha$  and the rare-event execution demand  $R_{re}$  as*

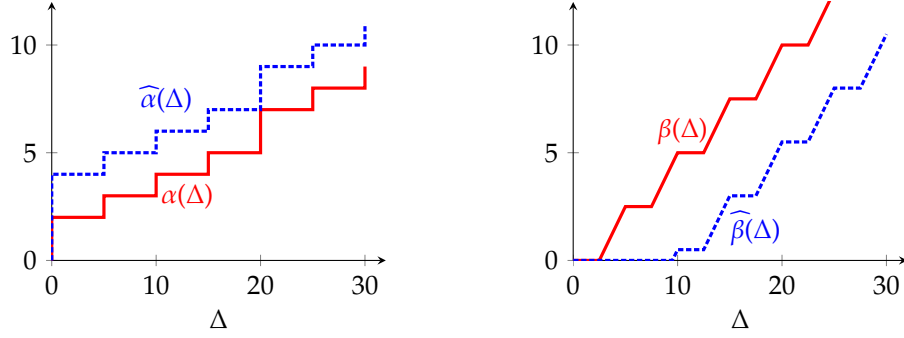
$$\widehat{\alpha} = \alpha + R_{re}. \quad (6.4)$$

As in other chapters, the abstraction of arrival functions to arrival curves enables a compact representation. In particular, (6.4) is defined independent of *when* the rare-event occurs. We illustrate this with an example.

**Example 6.1:** *Consider a periodic task with period and relative deadline of 5. Nominally, every fourth job has a WCET of 2 and every other job has a WCET of 1. In addition, the task can experience a demand overflow rare-event, whereupon 4 additional jobs with WCET 0.5 arrive.*

For the above example, we plot the arrival curve of the task in Figure 6.3(a).

<sup>1</sup> We make a particular choice with notations. We denote nominal values, such as arrival and service functions and curves, by their standard notations. The rare-event-aware arrival and service functions and curves are denoted with a hat on top.



(a) A demand overflow rare-event from Example 6.1. The nominal arrival curve is shown in a solid red line, and the rare-event-aware arrival curve is shown in a dashed blue line.

(b) A supply shortage rare-event from Example 6.2. The nominal service curve is shown in a solid red line, and the rare-event-aware service curve is shown in a dashed blue line.

Fig. 6.3 Examples of demand overflow and supply shortage rare-events.

## 6.2.2 Supply Shortage

Deviation from the nominal model, and subsequent deadlines misses, can also be due to a reduced resource availability. For instance, if a higher priority task has a demand overflow rare-event, it can reduce the resource available to a lower priority task. In such cases, the lower priority resource has a supply shortage rare-event. This rare-event is said to occur at the same time instant when the causer demand overflow rare-event occurs.

A supply shortage rare-event is characterized by a shortage, denoted as  $C_{re}$ , which is the upper-bound on the reduction in the available service of the resource. Consider a resource with a resource curve  $\beta$  and a supply shortage rare-event with shortage  $C_{re}$  at time  $t^*$ . Let  $C(t)$  and  $\widehat{C}(t)$  denote the nominal and reduced (nominal minus the effect of the rare-event) service functions of the resource. Then,  $\widehat{C}(t)$  must satisfy the following conditions.

$$\widehat{C}(t) = C(t), \quad 0 \leq t \leq t^*, \quad (6.5)$$

$$\geq (C(t) - C_{re})^\uparrow, \quad t > t^*, \quad (6.6)$$

where

$$f^\uparrow(t) = \max_{s \in [0, t]} f(s).$$

Just as  $C(t)$  is abstracted by the service curve  $\beta(\Delta)$ , we can abstract  $\widehat{C}(t)$  by the *rare-event-aware service curve* denoted  $\widehat{\beta}$ . This is formalized as follows.

$$\widehat{C}(t + \Delta) - \widehat{C}(t) \geq \widehat{\beta}(\Delta), \quad \forall t, \Delta \geq 0. \quad (6.7)$$

In the following, we show how to compute a valid  $\widehat{\beta}$ .

**Theorem 6.2:** *The rare-event-aware service curve  $\widehat{\beta}$  is given in terms of the nominal service-curve  $\beta$  and the rare-event shortage  $C_{re}$  as*

$$\widehat{\beta} = (\beta - C_{re})^\uparrow. \quad (6.8)$$

We illustrate this with an example.

**Example 6.2:** *Consider a TDMA resource of slot length 2.5 reserved in a period of 5. Let a low-level processor interrupt the working of the TDMA cycle for a total of 7 time units. The maximum shortage in this time is  $C_{re} = 4.5$  time units.*

For the above example, we plot the service curve of the resource in Figure 6.3(b).

## 6.3 Analysis of the REST model for a Single Task

In this section, we present the analysis of the REST framework for a single task and resource. We compute the defined settling-time metric for both the proposed models of rare-events. Further, we will define and compute two more metrics which characterize the behavior during the settling-time.

In the analysis we consider a single rare-event which could be either a demand overflow or a supply shortage. To homogenize the presentation for both kinds of rare-events, we always use the rare-event-aware curves  $\widehat{\alpha}$  and  $\widehat{\beta}$ . For a demand overflow rare-event we set  $\widehat{\beta} = \beta$ . Similarly, for a supply shortage rare-event we set  $\widehat{\alpha} = \alpha$ .

### 6.3.1 Computation of Settling-Time

We begin by defining a function which will be subsequently used to express the settling-time.

**Definition 6.1: (Function TS)** *Function TS with three arguments  $\alpha$ ,  $\beta$ , and  $D$  is defined as*

$$\text{TS}(\alpha, \beta, D) = \sup\{\Delta \geq 0 : \alpha(\Delta - D) > \beta(\Delta)\}. \quad (6.9)$$



In words,  $TS(\alpha, \beta, D)$  is the smallest argument after which  $\alpha$  shifted to the right by  $D$ , is always less than  $\beta$ . If  $\alpha$  is the arrival curve of a task and  $D$  is the relative deadline of all jobs of that task, then shifting  $\alpha$  to the right yields the demand bound function, as defined in Definition 2.1. Thus,  $TS$  can be considered the maximum interval length after which the demand bound function does not exceed the service curve.

With the above definition we can now present the result on computing the settling-time, which is denoted as  $t_s$ .

**Theorem 6.3: (Settling-Time of a Single Task)** *The settling-time of a task, denoted as  $t_s$ , with arrival curve  $\widehat{\alpha}$  and deadline  $D$ , served by a resource with service curve  $\widehat{\beta}$  is given by*

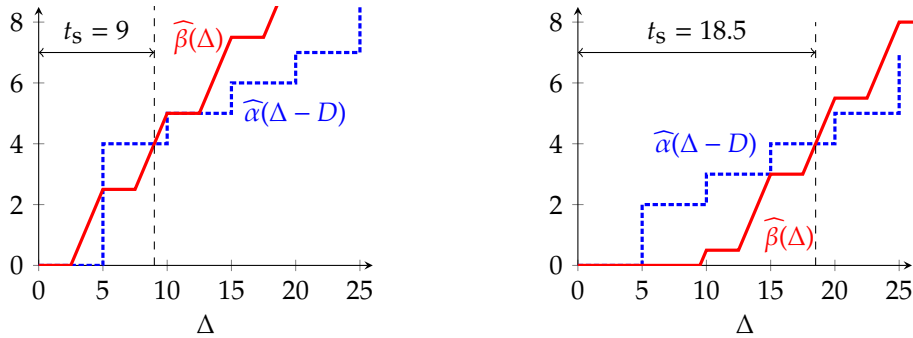
$$t_s = TS(\widehat{\alpha}, \widehat{\beta}, D). \quad (6.10)$$

From the above result, settling-time is derived from the simple  $TS$  operation on the rare-event-aware curves and the relative deadline. Furthermore, the computation is homogeneous for both kinds of rare-events. Finally, the computation is tight for the given abstraction of arrival and service curves. In other words, if every arrival function modeled by  $\widehat{\alpha}$  and every service function modeled by  $\widehat{\beta}$  can be observed, then there is a trace that has a deadline miss  $t_s$  time-units after a rare-event.

We illustrate this computation for the two earlier examples. Consider the task model from Example 6.1 and the resource model from Example 6.2. We analyze two different cases, one with the demand overflow rare-event specified in Example 6.1 and the other with supply shortage rare-event specified in Example 6.2. For both these cases, we compute the settling-time according to (6.10). This is illustrated in Figure 6.4.

### 6.3.2 Computation of Overshoot

Apart from settling-time, control engineers are interested in other metrics as well. For instance, it is often useful to characterize by how far the property deviates from the acceptable range after a disturbance. This is referred to as the *overshoot*. For a real-time CPS, overshoot may be measured in different ways. We propose two metrics, namely the maximum tardiness of all jobs and the maximum number of jobs which can miss their deadlines after a rare-event.



(a) Settling-time for the demand overflow rare-event detailed in Example 6.1.

(b) Settling-time for the supply shortage rare-event detailed in Example 6.2.

**Fig. 6.4** Computation of settling-time for demand overflow and supply shortage rare-events. The shifted arrival curve is shown in dashed blue lines, and the service curve is shown in a solid red line.

### Maximum Tardiness

For a job that misses its deadline, tardiness is the difference between its finishing time and its absolute deadline. The maximum tardiness for the REST framework, denoted as  $\widehat{T}$ , can be derived as follows.

**Theorem 6.4: (Maximum Tardiness)** *The maximum tardiness, denoted as  $\widehat{T}$ , of a task with arrival curve  $\widehat{\alpha}$  and deadline  $D$  which is served by a resource with service curve  $\widehat{\beta}$  is given by*

$$\widehat{T} = \widehat{D} - D, \quad (6.11)$$

where  $\widehat{D}$  is the worst-case response time of any job in the presence of a rare-event given as

$$\widehat{D} = \text{Del}(\widehat{\alpha}, \widehat{\beta}). \quad (6.12)$$

In the above equation the Del operator is as defined in Appendix A.3. In words, the worst-case delay of a job with a rare-event is the maximum horizontal distance between the two curves  $\widehat{\alpha}$  and  $\widehat{\beta}$ . Then, tardiness is simply the difference between this delay and the given relative deadline of the task.

### Maximum Number of Deadline Misses

Another metric of interest is the maximum number of jobs which miss their deadlines during the settling-time, denoted as  $\widehat{N}$ . So far, our model

of the task, given by the arrival curve, specifies how long jobs need to execute but not how *many* jobs need to execute. For instance, a large execution demand could either be a single long job or be broken into many small jobs. To compute  $\widehat{N}$ , we need bounds on the number of jobs that arrive both nominally and those due to rare-events. We define them as follows.

1. The nominal arrival curve of the task  $\alpha$  is decomposed as

$$\alpha(\Delta) = \gamma(\bar{\alpha}(\Delta)), \quad (6.13)$$

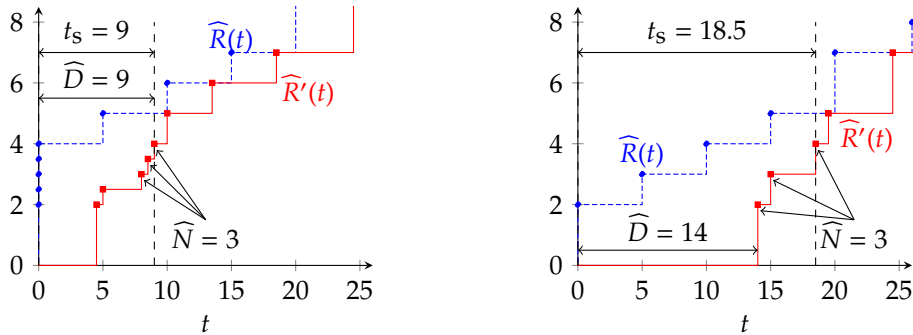
where  $\gamma(n)$  is the maximum execution demand of any  $n$  consecutive jobs, and  $\bar{\alpha}(\Delta)$  is the maximum number of jobs that can arrive in any interval of length  $\Delta$  [Wan06].

2. A demand overflow rare-event is characterized by  $N_{re}$  which is the upper-bound on the number of additional jobs which can arrive. As before, the cumulative execution demand due to the rare-event cannot exceed the parameter  $R_{re}$ . For a homogeneous presentation, for a supply shortage rare-event we set  $R_{re} = N_{re} = 0$ .

Given these additional parameters, computing  $\widehat{N}$  is not immediate. For instance,  $(\bar{\alpha}(t_s) + N_{re})$  is not a correct value of  $\widehat{N}$ . It specifies the maximum number of jobs which *arrive* within the settling-time. Instead, we are interested in the number of jobs which *finish* within the settling-time. Furthermore, some jobs finishing within the settling-time may meet their deadlines, and must not be counted in  $\widehat{N}$ . Our solution to this problem, is to define a *critical trace*. Then by simulating this trace and observing the number of jobs missing their deadlines we obtain  $\widehat{N}$ . We first define this critical trace.

**Definition 6.2: (Critical Trace)** *The critical trace is a specific trace of a REST framework, which defines a sequence of job arrivals, the execution demand of each job, and the service function as follows.*

1. The rare-event occurs at time 0.
2. Within any interval  $[0, t)$ , exactly  $(\bar{\alpha}(t) + N_{re})$  jobs (including nominal and rare-event jobs) arrive.
3. For any  $n \geq 0$ , the cumulative execution demand of the first  $(n + N_{re})$  jobs (including nominal and rare-event jobs) is exactly  $(\gamma(n) + R_{re})$ .
4. If multiple jobs arrive at the same time, then the jobs are queued and are executed in decreasing order of their execution demands.



(a) Computation of  $\widehat{D}$  and  $\widehat{N}$  for the demand overflow rare-event detailed in Example 6.1.

(b) Computation of  $\widehat{D}$  and  $\widehat{N}$  for the supply shortage rare-event detailed in Example 6.2.

**Fig. 6.5** Computation of overshoot for demand and supply rare-events. The input arrival function is shown in dashed blue lines, and the output arrival function is shown in a solid red line. Arriving jobs are marked with a circle on the input arrival function, while finishing jobs are marked with a square on the output arrival function.

5. The total service available in any interval  $[0, t)$  is exactly  $\widehat{\beta}(t)$ .

In words, the critical trace is the input trace when the rare-event occurs at time 0, the jobs arrive as early as possible and with the highest accumulated workload since time 0, and the resource provides the lowest accumulated service since time 0. In the following result we show that this trace leads to the highest number of deadline misses.

**Theorem 6.5: (Maximum Number of Deadline Misses)** *The maximum number of jobs which miss their deadline after the occurrence of a rare-event is equal to the number of jobs which miss their deadline for the critical trace defined in Definition 6.2.*

With a rare-event at time 0 in the critical trace, deadlines are missed only in the interval  $[0, t_s]$ . Hence, the critical trace is simulated only in this interval. Interestingly, the worst-case delay  $\widehat{D}$  is also obtained as the largest response-time of a job in the critical trace. We illustrate this with an example.

For the rare-events detailed in Examples 6.1 and 6.2, we show the input arrival functions  $\widehat{R}(t)$  of the critical traces in Figure 6.5. The critical traces also identify input service functions  $\widehat{C}(t)$ , which are not shown. We simulate the critical traces and obtain the output arrival functions  $\widehat{R}'(t)$  as defined in Appendix A.3. The output arrival functions are also shown in Figure 6.5. Notice that in either case, the rare-event occurs at time 0, jobs arrive as early as possible, and cumulative service function is as

small as possible. With these simulations, we compute  $\widehat{D}$  and  $\widehat{N}$ . Notice that for both examples, not all jobs arriving within the settling-time miss their deadlines. Furthermore, for the demand overflow rare-event shown in Figure 6.5(a), not all jobs finishing within the settling-time miss their deadlines.

## 6.4 Analysis of the REST model for Multiple Tasks

Thus far, we analyzed the REST framework for a single task. As this task with modeled with arrival curves and the resource by service curves, the analysis can be composed modularly to analyze larger systems as proposed in Modular Performance Analysis (MPA) [Wan06]. However, it is instructive to explicitly analyze multiple tasks scheduled by common schedulers. In particular, in this section we consider the fixed priority and EDF schedulers.

In both cases, we consider a task-set  $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ . Each task  $\tau_i$  has a nominal arrival curve  $\alpha_i$  and a relative deadline  $D_i$ . We consider a single rare-event, i.e., either one of these tasks have a demand overflow rare-event, or the resource has a supply shortage rare-event. The settling-time is defined as the maximum of the settling-time of all tasks.

### 6.4.1 Fixed Priority Scheduler

We assume a preemptive fixed priority scheduler where the priority of task  $\tau_i$  is higher than that of  $\tau_j$  if and only if  $i < j$ . In the following result, we adapt Theorem 6.3 to consider such a fixed priority scheduler.

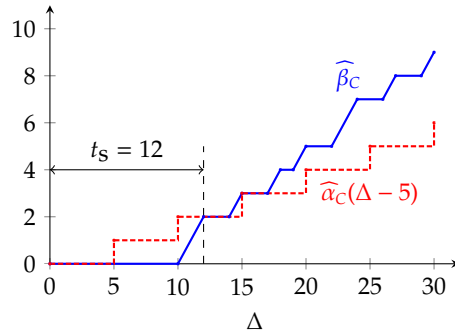
**Theorem 6.6:** *Consider a REST framework with a resource with service curve  $\widehat{\beta}$ . Under a preemptive fixed-priority scheduling policy, the resource serves the task-set  $\tau$ . Then, the settling-time is given as*

$$t_s = \max_{i \in \{1, 2, \dots, n\}} (t_s)_i$$

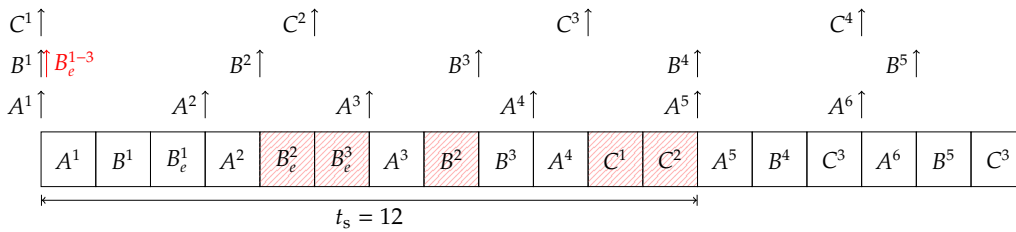
where,

$$(t_s)_i = \text{TS} \left( \widehat{\alpha}_i, \left( \widehat{\beta} - \sum_{j < i} \widehat{\alpha}_j \right)^\uparrow, D_i \right). \quad (6.14)$$

The above result is a direct extension of Theorem 6.3. The settling-time is the maximum of the settling-times of all tasks. For each task, we



(a) The computation of settling-time for task C for Example 6.3. The shifted arrival curve is shown in a dashed red line. The service curve is shown in a solid blue line.



(b) An example trace where deadlines are missed for the longest period of time, which is 12. Job arrivals are shown with upward arrows. The super-scripts on the task names indicate the job index. Rare-events are referenced with a sub-script  $e$ . Each job executes for 1 time unit. Jobs which miss their deadline are shown with a shaded box.

Fig. 6.6 Computation of settling-time for the task-set in Example 6.3 for a fixed priority scheduler.

compute the effective service curve of that task using known results from Modular Performance Analysis (MPA) [Wan06]. Note that the settling-time of a task does not depend on the properties of all tasks with a lower priority. This gives an inherent degree of isolation amongst the tasks. We illustrate this with the following example.

**Example 6.3:** Consider three periodic tasks  $A$ ,  $B$ , and  $C$ , with periods and relative deadlines 3, 4, and 5, respectively. The WCET of each task is 1. Consider the priority assignment  $A > B > C$ . Task  $B$  exhibits a demand overflow rare-event where up to 3 additional jobs can arrive with WCET of 1.

For the above example, we can verify that all deadlines are met for the nominal models of the tasks. With the demand overflow rare-event on task  $B$ , we do not expect any deadline misses for the higher priority task  $A$ . But jobs of both tasks  $B$  and  $C$  can miss their deadlines. We apply Theorem 6.6 for this example. The resultant computation reveals that task  $C$  has the highest settling-time of 12. We illustrate this computation in Figure 6.6(a). In Figure 6.6(b), we show an example trace where the a deadline is missed 12 time-units after the rare-event.

### 6.4.2 EDF Scheduler

Now we consider an Earliest Deadline First (EDF) scheduler, which uses the absolute deadlines of each job to prioritize the execution. The settling-time is given by the following result.

**Theorem 6.7:** *Consider a REST framework with a resource with service curve  $\widehat{\beta}$ . Under the EDF policy, the resource serves the task-set  $\tau$ . Then, the settling-time is given as*

$$t_s = \text{TS} \left( \sum_{\tau_i \in \tau} \widehat{\alpha}_i(\Delta - D_i), \widehat{\beta}, 0 \right). \quad (6.15)$$

The first argument of the function TS in (6.15) is the demand bound function of the task. Again, the settling-time can be interpreted as the maximum interval length at which the demand bound function exceeds the service curve.

In contrast to the fixed priority scheduler, for the EDF scheduler we do not compute task-specific settling-times. This lack of task-specific bounds has been observed as the “domino effect” where persistent overload can lead to a larger number of deadline misses with an EDF scheduler [Loc86], and perform worse than even “random” scheduling [JLT85].

We illustrate this computation for Example 6.3 with an EDF scheduler. As shown in Figure 6.7(a), the settling-time is 7. We plot in Figure 6.7(b) an example trace where a deadline is missed 7 time-units after the rare-event.

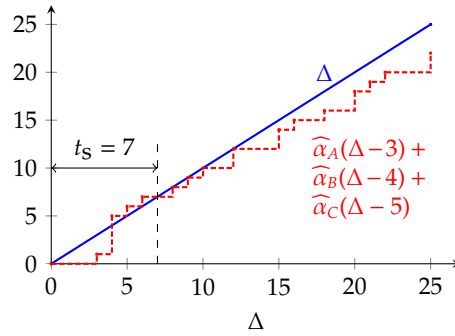
### 6.4.3 Settling-Time as a Scheduling Metric

In real-time CPSs, the primary question has been whether a task-set is schedulable or not. Indeed a clear notion of optimality under schedulability is defined. The optimality of the EDF scheduler is well known [LL73].

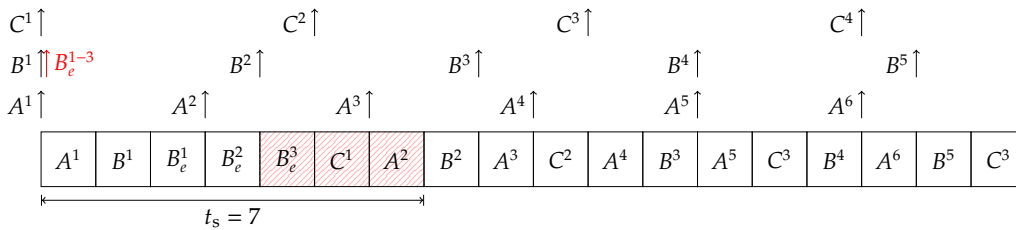
In the presence of rare-events which cause deadline misses, schedulability is not guaranteed. However, two non-schedulable systems can be compared by their settling-times, where a lower settling-time is preferable. Thus, settling-time can be a useful scheduling metric.

### Optimal Scheduling Algorithm

With this backdrop, it is interesting to ask whether there is a scheduling algorithm which is optimal in terms of minimizing the settling-time. We restrict our attention to schedulers which are agnostic of the occurrence



(a) The computation of settling-time. The demand bound function is shown in a dashed red line. The service curve of the full resource is shown in a solid blue line.



(b) An example trace where deadlines are missed for the longest period of time, which is 7. Job arrivals are shown with upward arrows. The super-scripts on the task names indicate the job index. Rare-events are referenced with a sub-script  $e$ . Each job executes for 1 time unit. Jobs which miss their deadline are shown with a shaded box.

**Fig. 6.7** Computation of settling-time for the task-set in Example 6.3 with an EDF scheduler.

of rare-events. In other words, the scheduler does not monitor if a rare-event occurred and adapt accordingly. Within this class of schedulers, we show in the following result that the EDF scheduler is optimal.

**Theorem 6.8:** *Under the REST framework, we are given a task-set with the arrival curves and relative deadlines of each task, the service curve of the resource, and the characteristics of a rare-event. Then, scheduling the tasks with an EDF scheduler optimally minimizes the settling-time.*

The above result is positive. Under conformance to the nominal models, the EDF scheduler meets all deadlines with the minimal required resource utilization. In addition, when rare-events occur, the EDF scheduler optimally minimizes the settling-time. Indeed, this optimality holds independent of when the rare-event occurs, whether it is a demand overflow or a supply shortage rare-event, or what its parameters are. This result is illustrated in Figures 6.6 and 6.7 for the task-set in Example 6.3. The settling-time for the EDF scheduler is smaller than that with the fixed-priority scheduler.



Priority	$t_S$	$(t_S)_A$	$(t_S)_B$	$(t_S)_C$
A > B > C	12	0	6	12
A > C > B	14	0	14	0
B > A > C	12	7	0	12
B > C > A	14	14	0	6
C > A > B	14	0	14	0
C > B > A	14	14	5	0

**Tab. 6.1** Settling-time for different priority assignments for task-set in Example 6.3.

### Optimal Priority Assignment

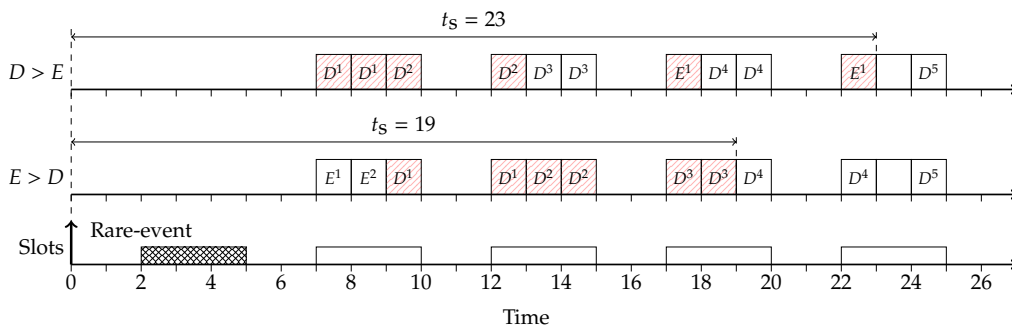
In certain CPSs it is desirable to immunize certain tasks from potential demand overflow rare-events in other tasks, even at the expense of a larger system-wide settling-time. As we observed, this is possible by carefully assigning priorities for a fixed-priority scheduler: a task is not affected by demand overflow rare-events of lower priority tasks. Hence, it is interesting to identify the effect of priority assignments on the settling-time.

We illustrate this with the task-set of Example 6.3. For different priority assignments, we verify that the task-set is schedulable for the nominal models. Then, for each priority assignment, we identify the task-wise settling-times using Theorem 6.6. We list the obtained values in Table 6.1.

The (joint) smallest value of the settling time is obtained for the Rate-Monotonic (RM) priority assignment ( $A > B > C$ ), i.e., tasks with shorter periods have higher priorities. This prompts the question if the RM priority assignment yields the fixed-priority scheduler with the smallest settling-time. Indeed, we show this is true for periodic tasks with implicit deadlines, i.e., with relative deadlines equal to periods. To this end, we first derive the following result on how to change priorities of jobs without increasing the settling-time.

**Lemma 6.1:** *Given is a task-set of periodic tasks with implicit deadlines. The settling-time is computed for some demand overflow or supply shortage rare-event. For a given priority assignment, let  $\tau_a$  and  $\tau_b$  be two tasks with consecutive priorities, such that  $\tau_a$  has the higher priority. If the period of  $\tau_a$  is larger than that of  $\tau_b$ , then swapping the priorities of  $\tau_a$  and  $\tau_b$  will not increase the settling-time.*

This result can be verified for the settling-times listed in Table 6.1. For instance the result holds for the two pairs of priority assignments: (a)  $A > B > C$  and  $A > C > B$ , and (b)  $B > A > C$  and  $B > C > A$ . Indeed, starting from any priority assignment with a series of swaps, as discussed



**Fig. 6.8** Computation of settling-time for two priority assignments of the task-set in Example 6.4. The first TDMA slot is unavailable because of the supply shortage rare-event at time 0. The first jobs of both periodic tasks arrive at time 0. The job index is shown with a super-script. Jobs missing deadlines are shown with dashed red lines. The priority assignment  $E > D$  has the smaller settling-time.

in the lemma, we can reach the RM priority assignment. Hence, the RM priority assignment is guaranteed to be optimal as formalized in the following result.

**Theorem 6.9:** *Under the REST framework, we are given a periodic task-set with implicit deadlines, the service curve of the resource, and the characteristics of a rare-event. The tasks are scheduled with a fixed priority scheduler. Then the Rate Monotonic (RM) priority assignment optimally minimizes the settling-time.*

The above result does not extend to periodic tasks with explicit deadlines, i.e., relative deadlines which are different from the respective periods. This is shown in the following example.

**Example 6.4:** *Consider a TDMA resource with slot size 3 and period 5. This resource has a supply shortage where it is unavailable for 5 time-units, i.e., a rare-event with  $C_{re} = 3$ . Let this resource schedule two periodic tasks D and E with periods and equal to 6 and 25, respectively. The relative deadlines of the tasks are 6 and 20, respectively. The WCET of both tasks is 2.*

For the above example consider the two possible priority assignments, (a)  $D > E$ , and (b)  $E > D$ . The settling-times computed using Theorem 6.6 for these priority assignments are 23 and 19, respectively. We plot the critical traces for these two priority assignments in Figure 6.8. As D has the smaller period and a smaller deadline, the RM priority assignment is  $D > E$ . This assignment has the higher settling-time. For this example the RM priority assignment is equivalent to the Deadline Monotonic (DM) priority assignment which assigns the higher priority to the task with the smaller relative deadline. Thus, the RM and DM priority assignments do not optimally minimize the settling-time for periodic task-sets with explicit deadlines.

To conclude, for the REST framework it is pertinent to consider settling-time as a scheduling metric. The optimality of the EDF scheduler for this metric generalizes its optimality under schedulability. This is a positive result and counters the observed poor performance of EDF under persistent overload. For periodic tasks with implicit deadlines the RM priority assignment is optimal within the class of fixed-priority schedulers.

## 6.5 Summary

We proposed settling-time as a metric to quantify timing properties of CPSs which are unschedulable. Such a metric requires a careful separation of nominal models from rare-event models. We argued that this separation is motivated by the variability in both the platforms and applications in CPSs. With such a dual model, settling-time is the duration for which the deadlines may not be met after a rare-event. In other words, there is outage in providing the timing guarantees, but for a bounded duration of time.

We showed how to compute the settling-time for a single task, and for multiple tasks scheduler with EDF and fixed-priority schedulers. We found the EDF scheduler to optimally minimize the settling-time. For fixed-priority schedulers, Rate Monotonic (RM) priority assignment is optimal for periodic tasks with implicit deadlines.

## Appendix

### Proof of Theorem 6.1

Let the demand overflow rare-event occur at some time  $t^*$ . Let the nominal and rare-event aware arrival functions be  $R(t)$  and  $\widehat{R}(t)$  respectively. Then,  $\widehat{R}(t)$  satisfies the constraints (6.1) and (6.2). Then, we have the following conditions.

$$\widehat{R}(t + \Delta) - \widehat{R}(t) = R(t + \Delta) - R(t) \quad t^* \notin [t, t + \Delta], \quad (6.16)$$

$$\leq R(t + \Delta) - R(t) + R_{re} \quad \text{else.} \quad (6.17)$$

Given that  $R(t + \Delta) - R(t) \leq \alpha(\Delta)$  for any  $t, \Delta \geq 0$ , the R.H.S. of the above constraints is upper-bounded by  $\alpha(\Delta) + R_{re}$ . Thus,  $\widehat{\alpha} = \alpha + R_{re}$ .  $\square$

### Proof of Theorem 6.2

A supply shortage rare-event with shortage  $C_{re}$  can be thought of as a reduction in supply due to a higher priority task with an arrival curve  $\alpha$

given as below.

$$\alpha(\Delta) = C_{re}, \quad \Delta > 0. \quad (6.18)$$

Then, from Modular Performance Analysis (MPA) [Wan06] we know that the available service curve is given as follows.

$$\widehat{\beta}(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{\beta(\lambda) - \alpha(\lambda)\}, \quad \forall \Delta \geq 0. \quad (6.19)$$

$$= (\beta - C_{re})^\uparrow. \quad (6.20)$$

□

### Proof of Theorem 6.3

We prove this with a contradiction. Let a rare-event occur at time  $t^*$ . Let a job finishing at time  $t > t^* + t_s$  miss its deadline, where  $t_s$  is as defined in (6.10). Let  $\widehat{R}$  and  $\widehat{C}$  denote any rare-event-aware arrival and service functions. Let  $u$  denote the latest time before  $t$  when the task-queue was empty.

*Case (a):  $u > t^*$  and  $\widehat{C}(u) > \widehat{C}(t^*)$ .* The empty buffer after  $t^*$  marks the end of the effect of a demand-overflow rare-event. The increase in the service function after  $t^*$  marks the end of the effect of supply-shortage rare-event. Thus, in every interval after  $u$ , the nominal arrival and service curves apply, and no job can miss its deadline.

*Case (b):  $u \leq t^*$  or  $\widehat{C}(u) = \widehat{C}(t^*)$ .* Define  $u' = \min(u, t^*)$ . In the busy-interval<sup>2</sup>  $[u', t]$  the accumulated service provided by the resource satisfies  $\widehat{C}(t) - \widehat{C}(u') \geq \widehat{\beta}(t - u')$ . Thus, the output arrival function  $\widehat{R}'$  at time  $t$  is given as below.

$$\widehat{R}'(t) = \widehat{R}(u') + \widehat{C}(t) - \widehat{C}(u') \quad (6.21)$$

$$\geq \widehat{R}(u') + \widehat{\beta}(t - u'). \quad (6.22)$$

Further, the arrival function at time  $t$  satisfies the following.

$$\widehat{R}(t - D) \leq \widehat{R}(u') + \widehat{\alpha}(t - u' - D). \quad (6.23)$$

Combining (6.22) and (6.23) we have the following condition.

$$\widehat{R}'(t) - \widehat{R}(t - D) \geq \widehat{\beta}(t - u') - \widehat{\alpha}(t - u' - D). \quad (6.24)$$

From the definition of  $t_s$  and because  $t - u' \geq t - t^* > t_s$ , the R.H.S. of the above equation is positive. Thus,  $\widehat{R}'(t) > \widehat{R}(t - D)$  which contradicts the supposition that the job finishing at  $t$  has a delay larger than  $D$ . □

<sup>2</sup>An interval is said to be busy if there are pending jobs in the task-queue throughout the interval.

**Proof of Theorem 6.4**

This follows from the definition of Del from Appendix A and the definition of tardiness.  $\square$

**Proof of Theorem 6.5**

Let  $\widehat{R}^*$  and  $\widehat{C}^*$  denote the rare-event-aware arrival and service functions of the critical trace defined in Definition 6.2. Let a rare-event occur at time  $t^*$ . Let  $\widehat{R}$  and  $\widehat{C}$  be any rare-event-aware arrival and service functions, respectively. Let  $\widehat{R}'$  be the corresponding rare-event-aware output arrival function. Let the last job missing its deadline finish at time  $t > t^*$ . Let  $u$  be the latest time, before  $t$ , when the task-queue was empty. We study the interval  $[u, t]$ . From the definition of the critical trace, we have the following conditions.

$$\widehat{R}(s) - \widehat{R}(u) \leq \widehat{R}^*(s - u), \quad \forall s \in [u, t], \quad (6.25)$$

$$\widehat{C}(s) - \widehat{C}(u) \geq \widehat{C}^*(s - u), \quad \forall s \in [u, t]. \quad (6.26)$$

Starting from an empty task-queue at time  $u$ , the arrival function  $\widehat{R}$  does not exceed the arrival function in the critical trace, and the service function  $\widehat{C}$  does not fall below the service function of the critical trace. Thus, the number of deadline misses cannot exceed that in the critical trace.  $\square$

**Proof of Theorem 6.6**

For each task  $\tau_i$  we can compute the effective rare-event-aware arrival and service curves. From MPA, the effective service curve of a fixed-priority task  $\tau_i$  is given as

$$\widehat{\beta}_i(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \left\{ \widehat{\beta}(\lambda) - \sum_{j \leq i} \widehat{\alpha}_j(\lambda) \right\}. \quad (6.27)$$

Given that the service curves are super-additive and arrival curves are sub-additive [LBT01], the above equation can be simplified as follows.

$$\widehat{\beta}_i = \left( \widehat{\beta} - \sum_{j \leq i} \widehat{\alpha}_j \right)^\uparrow. \quad (6.28)$$

Then, by applying Theorem 6.3 we arrive at (6.14).  $\square$

**Proof of Theorem 6.7**

We show this with a contradiction. Let a rare-event occur at time  $t^*$  and let a job finishing at time  $t > t^* + t_s$  miss its deadline, where  $t_s$  is as defined

in (6.15). Let  $u$  denote the latest time, before  $t$ , when the task-queue was empty.

*Case (a):  $u > t^*$  and  $\widehat{C}(u) > \widehat{C}(t^*)$ .* The empty buffer after  $t^*$  marks the end of the effect of a demand-overflow rare-event. The increase in the service function after  $t^*$  marks the end of the effect of supply-shortage rare-event. Thus, in every interval after  $u$ , the nominal arrival and service curves apply, and no job can miss its deadline.

*Case (a):  $u \leq t^*$  or  $\widehat{C}(u) = \widehat{C}(t^*)$ .* Define  $u' = \min(t^*, u)$ . The cumulative execution demand of all jobs with arrival time and deadline within  $[u', t]$  is bounded by  $\sum_{\tau_i \in \tau} \widehat{\alpha}_i(t - u' - D_i)$ . Since the job finishing at time  $t$  misses its deadline we have the following condition.

$$\sum_{\tau_i \in \tau} \widehat{\alpha}_i(t - u' - D_i) > \widehat{\beta}(t - u). \quad (6.29)$$

As  $t - u' \geq t - t^* > t_s$ , the above equation contradicts the definition of  $t_s$  in (6.15).  $\square$

### Proof of Theorem 6.8

Let  $(t_s)_{\text{EDF}}$  denote the settling-time with the EDF scheduler. Consider the specific trace of jobs with  $\widehat{R}_i = \widehat{\alpha}_i$  for each task  $\tau_i \in \tau$ , and  $\widehat{C} = \widehat{\beta}$ . Then, the cumulative execution demand of all jobs with arrival and deadline within  $[0, (t_s)_{\text{EDF}}]$  is greater than the available service. A scheduler that is agnostic to the occurrence of rare-events, cannot selectively drop certain tasks. Thus, independent of the scheduler, some job will miss its deadline at or later than time  $(t_s)_{\text{EDF}}$ . Thus, no scheduler can have a settling-time less than that of the EDF scheduler.  $\square$

### Proof of Lemma 6.1

We can equivalently write (6.14) as

$$(t_s)_i = \text{TS} \left( \sum_{j \leq i} \widehat{\alpha}_j, \widehat{\beta}, D_i \right). \quad (6.30)$$

When the priorities of two tasks with consecutive priorities are swapped, the settling-times of all other tasks remain unchanged. Thus, to show that the settling-time does not increase after the priority swap, we only need to show that the settling-time of the lower priority task, amongst the two considered tasks, does not increase. In other words, we need to show that the settling-time of  $\tau_b$  in the original priority assignment is not smaller than the settling-time of  $\tau_a$  with the swapped priorities. In either case, the first two arguments of TS in (6.30) are the same. Further, as the period of

---

$\tau_a$  and its implicit deadline are greater than that of  $\tau_b$ , the third argument of TS is higher for the swapped case. It is clear to see that the function TS is monotonically non-increasing w.r.t. the third argument. Hence, the settling-time cannot increase due to the swapping of priorities.  $\square$

**Proof of Theorem 6.9**

Given any priority assignment, we can iteratively apply Lemma 6.1 to change the priorities to match the RM priority assignment without increasing the settling-time.  $\square$





# 7

## Conclusions

### 7.1 Major Findings

The aim of this thesis was to identify the inherent complexity in providing hard real-time guarantees for Cyber-Physical Systems (CPSs) and to demonstrate effective solution strategies. To this end, we had the following major findings through the thesis.

**Variability in the timing models of CPSs.** We argued that the timing analysis of CPSs is more challenging than that of embedded systems. CPSs are large, distributed, general-purpose, cross-layer and federated systems. These properties of CPSs introduce variability in the timing models which are essential for guaranteeing timing properties. We illustrated this variability with several examples in this thesis.

- In the design of Demand Bound Server (DBS) we highlighted the challenge of timing isolation amongst multiple tasks in a distributed and federated CPS. The distributed nature introduces timing artifacts such as jitter and burst in the input streams, while the federated nature forces multiple applications of different criticality levels to interfere.
- In the design of cool-shapers and in the analysis of feedback controlled speed scaling, we highlighted the effect of actively managing temperature of general-purpose processors. Using either system throttling or speed scaling to manage temperature of the processor introduces variability in the resource availability as dictated by the heat generation and diffusion properties.

- In the presentation of the SMT solver we underscored the importance of simultaneously analyzing multiple designs in efficiently optimizing large CPSs. This multiplicity translates to models which express additional variability due to differences in the considered designs.
- In the proposal for the metric of settling-time, we presented the need for cross-layer objectives in CPSs. The abstraction of worst-case delays do not always tightly express the cross-layer objectives. Instead, guarantees which represent the variability due to rare timing events, can be more expressive.

**Templates of different solution strategies.** We proposed multiple solution strategies to handle variability in timing models of CPSs.

- In Part I of the thesis we presented the run-time managers DBS and cool-shaper which can dynamically monitor and adapt to variability. The commonality here was that both run-time managers were designed modularly by composing efficient constituent units. The composed run-time manager has a richer set of behaviors than that of the class of constituent units. We refer to this as the *behavioral composition*, which is summarized by the popular thesis of Gestalt theory that says, “*The whole is greater than the sum of the parts*”.
- In Part II of the thesis we defined analysis techniques for feedback controlled speed scaling and multiple designs embedded within a Satisfiability Modulo Theory (SMT) solver. The commonality here was the effectiveness of abstraction in representing and analyzing variability. In particular, the abstraction in the interval domain was employed to represent both critical traces for a processor with speed scaling and the properties of incomplete models within iterations of an SMT solver.
- In Part III of the thesis we argued for the careful separation between the nominal behavior and the incidence of certain rare-events. Then, timing guarantees were provided predicated on whether rare-events occurred within a time interval of length defined as the settling-time. Such a guarantee better represents the irregular timing requirements in CPSs such as stability with a network control system.

**Wide applicability of the interval domain abstraction.** Throughout the thesis, the presented solution strategies benefited from the applicability and suitability of the abstraction of curves in the interval domain, namely the arrival and service curves.

- By representing the demand bound function as the shifted version of an arrival curve, we motivated and analyzed DBSs and cool-shapers. In particular, the class of curves which is the minimum of multiple leaky-bucket shaping curves arose in both the design of min-composition of SP-DBS and the optimal convex-hull cool-shaper.
- Arrival and service curves were shown to be relevant in peak temperature calculation first in [RYB<sup>+</sup>11]. We extended this to consider the two common dynamic thermal management (DTM) techniques of system throttling and speed scaling. In both cases, we found the expressive power of arrival and service curves adequate to analyze the worst-case properties.
- By extending the curves to abstract curves which represent multiple designs in an SMT solver, we showed how to model uncertainty in the speed of processors. This is enabled by monotonicity principles satisfied by the arrival and service curves and the operators of min/max-plus algebra.
- Finally, we showed that by simulating critical traces defined by the arrival and services we can identify richer guarantees such as settling-time and number of deadline misses subsequent to a rare-event.

## 7.2 Outlook

We now summarize some of the promising directions of future work and certain reservations which emanate from the findings in this thesis.

Structural composition has been a common motif in the design of systems: Larger and more complex systems are built by composing smaller systems. In real-time systems, an influential example of structural composition is the use of time-triggered architectures [KB03]. The DBS and the cool-shaper are examples of an alternate kind of composition where the result of composition is not an enlargement, but an enriching of behavior. It is a relevant question whether behavioral composition has examples in other settings.

Feedback control is a backbone in the design of majority of engineering systems, and benefits from a good base of developed theory. In the timing analysis of feedback controlled speed scaling we illustrated that worst-case properties can be derived in the presence of such control. It is an open question if such an analysis can be extended to other settings. First examples are the feedback control of resource availability based on

the currently buffered queue of pending events or the remaining battery capacity.

SMT solvers represent a practical success of formal methods. We showed that this extends to the theory of Real-Time Calculus for the specific speed assignment problem. A natural question is its relevance in other design problems, such as configuration of schedulers (priority or bandwidth assignments) and binding decisions (tasks to processors or data to memory banks).

Cross-layer research is a defining forcing function in the development of CPSs. How to express and analyze the impact of timing properties of software blocks on higher system-level goals is an interesting research question. First examples of such goals are stability of a control plant and the information sensed with a deployed wireless sensor network.

In spite of the above positives, there is ground for expressing reasonable skepticism. Providing hard timing guarantees critically depends on the availability of data to populate required models such as the arrival and service curves. In this thesis, we did not focus on this challenge. Furthermore, the very properties which complicate analysis of CPSs may lower the relevance of worst-case analysis: In contrast to embedded systems which are carefully designed with formal techniques, large and open CPSs may be designed with engineering rules-of-thumb. In such CPSs, the certification of guaranteed properties may be more of an engineering and even societal challenge, than a theoretical challenge.

# A

## Real-Time Calculus

### A.1 Min-Plus/Max-Plus Algebra

The min-plus convolution  $\otimes$  and the min-plus deconvolution  $\oslash$  of two functions  $f$  and  $g$  are defined as:

$$(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\}, \quad (\text{A.1})$$

$$(f \oslash g)(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{f(\Delta + \lambda) - g(\lambda)\}. \quad (\text{A.2})$$

The max-plus convolution  $\bar{\otimes}$  and the min-plus deconvolution  $\bar{\oslash}$  of two functions  $f$  and  $g$  are defined as:

$$(f \bar{\otimes} g)(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\}, \quad (\text{A.3})$$

$$(f \bar{\oslash} g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{f(\Delta + \lambda) - g(\lambda)\}. \quad (\text{A.4})$$

Let  $f^l \leq f \leq f^u$  and  $g^l \leq g \leq g^u$ . Then, the operators satisfy the following monotonicity principles.

$$f^l \otimes g^l \leq f \otimes g \leq f^u \otimes g^u \quad (\text{A.5})$$

$$f^l \oslash g^u \leq f \oslash g \leq f^u \oslash g^l \quad (\text{A.6})$$

$$f^l \bar{\otimes} g^l \leq f \bar{\otimes} g \leq f^u \bar{\otimes} g^u \quad (\text{A.7})$$

$$f^l \bar{\oslash} g^u \leq f \bar{\oslash} g \leq f^u \bar{\oslash} g^l \quad (\text{A.8})$$

## A.2 Arrival and Service Functions and Curves

The execution demand of a trace of jobs can be described using an *arrival function*  $R(t)$  which denotes the cumulative execution demand of all jobs that arrive in the interval  $[0, t)$ . The availability of a resource can be described using a *service function*  $C(t)$  which denotes the cumulative resource available in the interval  $[0, t)$ .

It is often convenient to use the same *units* to describe arrival and service functions. For instance, on a bus both  $R$  and  $C$  can have the number of packets as their units. On a processor, both  $R$  and  $C$  can have the number of processing cycles as their units.

While the arrival function  $R(t)$  describes one concrete trace of jobs, an *arrival curve*  $\alpha(\Delta) = (\alpha^u(\Delta), \alpha^l(\Delta))$  represents a family of arrival functions defined as below.

$$\alpha^l(\Delta) \leq R(t + \Delta) - R(t) \leq \alpha^u(\Delta), \quad \forall t, \Delta \geq 0. \quad (\text{A.9})$$

While the service function  $C(t)$  describes one concrete trace of resource availability, a *service curve*  $\beta(\Delta) = (\beta^u(\Delta), \beta^l(\Delta))$  represents a family of arrival functions defined as below.

$$\beta^l(\Delta) \leq C(t + \Delta) - C(t) \leq \beta^u(\Delta), \quad \forall t, \Delta \geq 0. \quad (\text{A.10})$$

In Chapters 2 to 4 and 6 we use the arrival curve  $\alpha$  and the service curve  $\beta$  to only denote the upper-arrival curve  $\alpha^u$  and the lower-service curve  $\beta^l$ , respectively. In Chapter 5 we use the extended definition with the tuple of upper and lower curves.

## A.3 Workload Conserving Resource

A workload conserving resource is one where any pending jobs are executed whenever the resource is available. The executed trace of jobs can be described by an *output arrival function*  $R'(t)$  which denotes the cumulative execution demand received by the trace of jobs in the interval  $[0, t)$ . If  $R(t)$  is the arrival function of jobs executed by a workload conserving resource with service function  $C(t)$ , then the output arrival function  $R'(t)$  is given as:

$$R'(t) = \inf_{0 \leq u \leq t} \{R(u) + C(t) - C(u)\}. \quad (\text{A.11})$$

The remaining resource availability of a workload conserving resource after executing a trace of jobs can be described by an *output service function*  $C'(t)$  which denotes the cumulative resource availability in the interval

$[0, t)$ . If  $R'(t)$  is the output arrival function of jobs executed by a workload conserving resource with service function  $C(t)$ , then the output service function  $C'(t)$  is given as:

$$C'(t) = C(t) - R'(t). \quad (\text{A.12})$$

To distinguish between  $R(t)$  and  $R'(t)$ , the former is described as the *input* arrival function, and the latter is the *output* arrival function. Similarly,  $C(t)$  and  $C'(t)$ , are described as input and output service functions.

For given input and output arrival functions, we can compute the maximum delay suffered by any job, denoted as  $d^{\max}$ , as:

$$d^{\max} \leq \text{Del}(R, R'), \quad (\text{A.13})$$

where

$$\text{Del}(f, g) = \sup_{\lambda \geq 0} \{\inf\{\tau \geq 0 \mid f(\lambda) \leq g(\lambda + \tau)\}\}. \quad (\text{A.14})$$

The Del of two functions can be visualized as the maximum horizontal distance between the functions.

We can also compute the maximum number of jobs buffered at any time, denoted as  $b^{\max}$ , as:

$$b^{\max} \leq \text{Buf}(R, R'), \quad (\text{A.15})$$

where

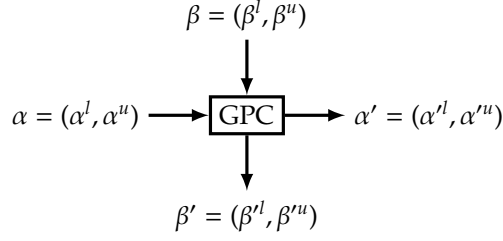
$$\text{Buf}(f, g) = \sup_{\lambda \geq 0} \{f(\lambda) - g(\lambda)\}. \quad (\text{A.16})$$

The Buf of two functions can be visualized as the maximum vertical distance between the functions.

## A.4 Greedy Processing Component

A Greedy Processing Component (GPC) is an abstraction of a workload conserving processor to enable Modular Performance Analysis (MPA) [Wan06]. In particular, it relates the arrival and service curves which abstract the arrival and service functions, respectively, at the inputs and outputs of a workload conserving processor. The block diagram of a GPC is shown in Figure A.1. The output arrival curve is denoted as  $\alpha'$  and the output service curve is denoted as  $\beta'$ .

For some concrete trace if  $R(t)$ ,  $C(t)$ ,  $R'(t)$  and  $C'(t)$  denote the input arrival function, the input service function, the output arrival function,



**Fig. A.1** Block diagram of the Greedy Processing Component (GPC). The outputs are given in terms of the inputs as given in (A.21) to (A.24).

and the output service function, then the arrival and service curves bound these quantities as:

$$\alpha^l(\Delta) \leq R(t + \Delta) - R(t) \leq \alpha^u(\Delta), \quad (\text{A.17})$$

$$\beta^l(\Delta) \leq C(t + \Delta) - C(t) \leq \beta^u(\Delta), \quad (\text{A.18})$$

$$\alpha'^l(\Delta) \leq R'(t + \Delta) - R'(t) \leq \alpha'^u(\Delta), \quad (\text{A.19})$$

$$\beta'^l(\Delta) \leq C'(t + \Delta) - C'(t) \leq \beta'^u(\Delta). \quad (\text{A.20})$$

To distinguish between  $\alpha(\Delta)$  and  $\alpha'(\Delta)$ , the former is described as the *input* arrival curve, and the latter is the *output* arrival curve. Similarly,  $\beta(\Delta)$  and  $\beta'(\Delta)$ , are described as input and output service curves.

The output curves are related to the input curves with the following relations [TCN00].

$$\alpha'^u = \min\{(\alpha^u \otimes \beta^u) \otimes \beta^l, \beta^u\}, \quad (\text{A.21})$$

$$\alpha'^l = \min\{(\alpha^l \otimes \beta^u) \otimes \beta^l, \beta^l\}, \quad (\text{A.22})$$

$$\beta'^u = (\beta^u - \alpha^l) \bar{\otimes} 0, \quad (\text{A.23})$$

$$\beta'^l = (\beta^l - \alpha^u) \bar{\otimes} 0. \quad (\text{A.24})$$

Also bounds on the maximum delay and buffer-space are given with the following relations.

$$d^{\max} = \text{Del}(\alpha^u, \beta^l) \quad (\text{A.25})$$

$$b^{\max} = \text{Buf}(\alpha^u, \beta^l). \quad (\text{A.26})$$



# Bibliography

- [AB98] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13. IEEE, 1998.
- [ABD<sup>+</sup>95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.
- [APSL09] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari. Resource reservations for general purpose applications. *Industrial Informatics, IEEE Transactions on*, 5(1):12–21, 2009.
- [BBB03] A. Burns, G. Bernat, and I. Broster. A probabilistic framework for schedulability analysis. In *Embedded Software*, pages 1–15. Springer, 2003.
- [BBL01] G. Bernat, A. Burns, and A. Liamsi. Weakly hard real-time systems. *Computers, IEEE Transactions on*, 50(4):308–321, 2001.
- [BCC<sup>+</sup>98] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, et al. Recommendations on queue management and congestion avoidance in the internet. 1998.
- [BH97] S. K. Baruah and J. R. Haritsa. Scheduling for overload in real-time systems. *Computers, IEEE Transactions on*, 46(9):1034–1039, 1997.
- [BH07] L. A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.
- [BLS10] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 13–22. IEEE, 2010.
- [BM01] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182. IEEE, 2001.

- [BM07] A. R. Bradley and Z. Manna. *The calculus of computation: decision procedures with applications to verification*. Springer, 2007.
- [BMR90] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.
- [BPB<sup>+</sup>00] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46(4):305–325, 2000.
- [BPST10] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The opensmt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153. Springer, 2010.
- [Bra11] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.
- [BSS95] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 90–99. IEEE, 1995.
- [BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
- [C<sup>+</sup>97] A. E. E. Committee et al. *Avionics Application Software Standard Interface*. Aeronautical Radio, 1997.
- [CBS00] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 295–304. IEEE, 2000.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
- [CRW07] A. K. Coskun, T. S. Rosing, and K. Whisnant. Temperature aware task scheduling in mpsoCs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1659–1664. EDA Consortium, 2007.
- [CWZ04] J. Cong, J. Wei, and Y. Zhang. A thermal-driven floorplanning algorithm for 3d ics. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 306–313. IEEE, 2004.

- [DL97] Z. Deng and J.-S. Liu. Scheduling real-time applications in an open environment. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 308–319. IEEE, 1997.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [DM06] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. *ACM SIGARCH Computer Architecture News*, 34(2):78–88, 2006.
- [DMB11] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.
- [EAL07] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 129–138. IEEE, 2007.
- [EBA<sup>+</sup>11] H. Esmailzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.
- [ES05] N. Een and N. Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.
- [FWB07] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH Computer Architecture News*, 35(2):13–23, 2007.
- [GB95] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.
- [GGPS96] L. Georgiadis, R. Guérin, V. Peris, and K. N. Sivarajan. Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking (TON)*, 4(4):482–501, 1996.
- [Gmb08] A. GmbH. ait worst-case execution time analyzers, 2008.
- [GPV04] M. Goma, M. D. Powell, and T. Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In *ACM SIGARCH Computer Architecture News*, volume 32, pages 260–270. ACM, 2004.

- [HGV<sup>+</sup>06] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: A compact thermal modeling methodology for early-stage vlsi design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(5):501–513, 2006.
- [HHJ<sup>+</sup>05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis—the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.
- [HM04] J. Hu and R. Marculescu. Application-specific buffer space allocation for networks-on-chip router design. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 354–361. IEEE Computer Society, 2004.
- [HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *Computers, IEEE Transactions on*, 44(12):1443–1451, 1995.
- [Jia06] Y. Jiang. A basic stochastic network calculus. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 123–134. ACM, 2006.
- [JLT85] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *RTSS*, volume 85, pages 112–122, 1985.
- [Joh98] L. A. Johnson. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk*, October, 1998.
- [KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
- [LB00] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Real-Time Technology and Applications Symposium, 2000. RTAS 2000. Proceedings. Sixth IEEE*, pages 166–175. IEEE, 2000.
- [LBT01] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer, 2001.
- [LCB00] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 217–226. IEEE, 2000.
- [LDSY07] Y. Liu, R. P. Dick, L. Shang, and H. Yang. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1526–1531. EDA Consortium, 2007.

- [Lee08] E. A. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.
- [LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
- [LMSN04] E. K. Liebemann, K. Meder, J. Schuh, and G. Nenninger. Safety and Performance Enhancement: The Bosch Electronic Stability Control (ESP). In *Proceedings of the SAE Convergence Congress & Exposition On Transportation Electronics*, 2004.
- [LMW96] Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 254–263. IEEE, 1996.
- [Loc86] C. Locke. *Best-effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, 1986.
- [LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.
- [MLBC04] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 211–218. IEEE, 2004.
- [MZ09] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.
- [NRAW11] A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power management architecture of the 2nd generation Intel® Core microarchitecture. 2011.
- [NSG<sup>+</sup>06] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz. The implementation of a 2-core, multi-threaded itanium family processor. *Solid-State Circuits, IEEE Journal of*, 41(1):197–209, 2006.
- [NSSLW05] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.
- [PFR09] M. Pipattanasomporn, H. Feroze, and S. Rahman. Multi-agent systems in a distributed smart grid: Design and implementation. In

- Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES*, pages 1–8. IEEE, 2009.
- [Rat91] E. P. Rathgeb. Modeling and performance comparison of policing mechanisms for atm networks. *Selected Areas in Communications, IEEE Journal on*, 9(3):325–334, 1991.
- [RLSS10] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM, 2010.
- [RYB<sup>+</sup>11] D. Rai, H. Yang, I. Bacivarov, J.-J. Chen, and L. Thiele. Worst-case temperature analysis for real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.
- [SA03] J. Srinivasan and S. V. Adve. Predictive dynamic thermal management for multimedia applications. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 109–120. ACM, 2003.
- [SAS02] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 17–28. IEEE, 2002.
- [SB94] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11. IEEE, 1994.
- [SB96] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.
- [SB09] S. Sriram and S. S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2009.
- [Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *STOC*, volume 78, pages 216–226, 1978.
- [SL04] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 57–67. IEEE, 2004.
- [SLMR05] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar. Opportunities and obligations for physical computing systems. *Computer*, 38(11):23–31, 2005.
- [SSH<sup>+</sup>03] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture.

- In *ACM SIGARCH Computer Architecture News*, volume 31, pages 2–13. ACM, 2003.
- [Str10] O. Strichman. *Decision procedures: an algorithmic point of view*. Springer, 2010.
- [Tak12] H. Takada. Automotive embedded systems, June 2012.
- [TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104. IEEE, 2000.
- [TDS<sup>+</sup>95] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Real-Time Technology and Applications Symposium, 1995. Proceedings*, pages 164–173. IEEE, 1995.
- [TT04] F. Tan and C. Tso. Cooling of mobile electronic devices using phase change materials. *Applied thermal engineering*, 24(2):159–169, 2004.
- [WA07] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *Hybrid Systems: Computation and Control*, pages 601–613. Springer, 2007.
- [Wan06] E. Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2006.
- [WB08] S. Wang and R. Bettati. Reactive speed control in temperature-constrained real-time systems. *Real-Time Systems*, 39(1-3):73–95, 2008.
- [YCTK10] C.-Y. Yang, J.-J. Chen, L. Thiele, and T.-W. Kuo. Energy-efficient real-time task scheduling with temperature-dependent leakage. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 9–14. European Design and Automation Association, 2010.
- [YLVZ04] X. Yang, L. Liu, N. H. Vaidya, and F. Zhao. A vehicle-to-vehicle communication protocol for cooperative collision warning. In *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, pages 114–123. IEEE, 2004.
- [ZBP01] W. Zhang, M. S. Branicky, and S. M. Phillips. Stability of networked control systems. *Control Systems, IEEE*, 21(1):84–99, 2001.





# List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

P. Kumar, J-J. Chen and L. Thiele. **Demand Bound Server: Generalized Resource Reservation for Hard Real-Time Systems.** In *Proceedings of the 11th International Conference on Embedded software, EMSOFT 2011*. Taipei, Taiwan, October 2011. (Chapter 2)

P. Kumar and L. Thiele. **Cool Shapers: Shaping real-time tasks for improved thermal guarantees.** In *Proceedings of 48th Design Automation Conference, DAC 2011*. San Diego, USA, June 2011. (Chapter 3)

P. Kumar and L. Thiele. **Timing Analysis on a Processor with Temperature-Controlled Speed Scaling.** In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2012*. Beijing, China, April 2012. (Chapter 4)

P. Kumar, D. Chokshi and L. Thiele. **A Satisfiability Approach to Speed Assignment for Distributed Real-Time Systems.** In *Proceedings of the 2013 Design, Automation & Test in Europe (DATE)*. Grenoble, France, March 2013. (Chapter 5)

P. Kumar and L. Thiele. **Quantifying the Effect of Rare Timing Events with Settling-Time and Overshoot.** In *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012*. San Juan, Puerto Rico, December 2013. (Chapter 6)

The following list includes publications that are not part of this thesis.

P. Kumar and L. Thiele. **Thermally Optimal Stop-Go Scheduling of Dataflow Graphs.** In *Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC 2011*. Yokohama, Japan, January 2011.

P. Kumar and L. Thiele. **End-to-end Delay Minimization in Thermally Constrained Distributed Systems.** In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems, ECRTS 2011*. Porto, Portugal, July 2011.

P. Kumar, J-J. Chen, A. Schranzhofer, L. Thiele and G. Buttazzo. **Real-Time Analysis of Servers for General Job Arrivals.** In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Algorithms, RTCSA 2011*. Toyama, Japan, August 2011.

P. Kumar and L. Thiele. **System-Level Power and Timing Variability Characterization To Compute Thermal Guarantees.** In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011*. Taipei, Taiwan, October 2011.

P. Kumar, D Goswami, S Chakraborty, K Lampka, A Annaswamy and L Thiele. **A Hybrid Approach to Cyber-Physical Systems Verification.** In *Proceedings of the 49th Design Automation Conference (DAC)*. San Francisco, USA, June 2012.

P. Kumar, N Stoimenov and L Thiele. **An Algorithm for Online Reconfiguration of Resource Reservations for Hard Real-Time Systems.** In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*. Pisa, Italy, July 2012.

P. Huang, P. Kumar, N Stoimenov and L Thiele. **Interference Constraint Graph - A New Specification for Mixed-Criticality Systems.** In *Proceedings of the IEEE 18th International Conference on Emerging Technologies & Factory Automation, ETFA 2013*. Cagliari, Italy, September 2013.

F Santy, G Raravi, G Nelissen, V Nelis, P Kumar, J Goossens, E Tovar. **Interference Constraint Graph - A New Specification for Mixed-Criticality Systems.** In *Proceedings of the 21st International Conference on*

*Real-Time Networks and Systems, RTNS 2013.* Sophia Antipolis, France, October 2013.

P. Kumar, H Yang, I Bacivarov and L Thiele. **COOLIP: Simple yet Effective Job Allocation for Distributed Thermally-Throttled Processors.** In *Proceedings of the 2014 Design, Automation & Test in Europe (DATE)*. Dresden, Germany, March 2014.

N Dhruva, P Kumar, G Giannopoulou and L Thiele. **Computing a Language-Based Guarantee for Timing Properties of Cyber-Physical Systems.** In *Proceedings of the 2014 Design, Automation & Test in Europe (DATE)*. Dresden, Germany, March 2014.

