THE DESIGN OF PREDICTABLE MULTI-CORE PROCESSORS WHICH SUPPORT TIME-TRIGGERED SOFTWARE ARCHITECTURES

A thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

By

Keith Florence Athaide

Embedded Systems Laboratory

Department of Engineering

University of Leicester

Leicester, United Kingdom

October 2010

The design of predictable multi-core processors which support time-triggered software architectures

Keith F. Athaide

ABSTRACT

Safety-critical systems – such as those used in the medical, automotive and aerospace fields – have a crucial dependence on the reliable functioning of one or more embedded processors. In such systems, a co-operative software design methodology can be used to guarantee a high degree of reliability; when coupled with a time-triggered architecture, this methodology can result in robust and predictable systems with a comparatively simple software design, low operating system overhead, easier testability, greater certification support and tight jitter control.

Nevertheless, the use of a co-operative design methodology is not always appropriate, since it may negatively affect system responsiveness and can add to the maintenance costs. Many alternatives have been researched and implemented over the past few decades to address such concerns, albeit by compromising on some of the benefits this architecture provides.

This thesis makes five main contributions to tackle the major obstacles to single-processor time-triggered co-operative designs:

- it proposes and describes the implementation of a novel multi-core processor with two capable software scheduler implementations that allow application software to be designed as for a single-core system;
- it describes the internalisation of these scheduler implementations into hardware which allows application software to use all available computing capacity;
- it describes a hardware technique to eliminate the variations in starting times of application software, thereby increasing the stability of applications;
- it describes the implementation of a hardware technique for sharing input/output resources amongst application software with increased determinism by leveraging the time-triggered nature of the underlying system;
- it describes the implementation of a predictable processor that supports purely co-operative software and is suitable for the secondary cores on a multi-core design (due to its small size).

Overall, the contributions of this thesis both increase system responsiveness and lessen the impact of seemingly innocuous maintenance activities.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the Almighty Triune God for His grace and support mediated through the one Church guided by the See of St. Peter. It is the restlessness of the heart as it seeks God that has led to the pursuit of knowledge in the philosophical, historical, ethical, medical and natural sciences.

I owe my deepest gratitude to my supervisor, Professor Michael J. Pont, for his guidance, patience and encouragement throughout my research; and also for his constant endeavours to be available to his students. I would also like to thank my technical supervisor, Dr. Devaraj Ayavoo, for his support and for all his words of encouragement; and his colleague, Dr. Zemian Hughes for lending an open ear to ludicrous thoughts and for intriguing ideas.

I would like to thank my sponsors for making this research possible: TTE Systems Ltd. for the grant of a studentship, the University of Leicester for the award of an Open Scholarship and the Department of Engineering for the allotment of departmental funding.

Further, I am sincerely grateful to my parish priest, Fr. Leon Pereira O.P., for his help during difficult times and to one of my housemates, Miss Isabel Lim Fong for being a friend. I would also like to thank my other housemates, Dr. Dénes Bisztray, Dr. Cicimol Alexander, Mr. Michael Furniss, Miss Réka Plugor and Miss Dimitrinka Atanasova for their understanding and tolerance of many eccentricities that escalated as time wore on. I extend the same thanks to my colleagues Mr. Syed Aley Imran Rizvi, Dr. Imran Sheikh, Mr. Muhammad Amir, Mr. Musharraf Ahmed Hanif and particularly Miss Farah Lakhani.

Last, but not least, I would like to thank my parents, Mr. Gregorio Taumaturgo Policarpo Francisco Ataide and Mrs. Clara Ditosa Da Cruz é Ataide and my siblings Mrs. Karen Marisa Kaur, Mr. Kevin Tonisavio Athaide and Miss Kate Caroline Athaide for always being there and for their prayers.

There are many others in the parish of Holy Cross Priory Church, in Leicester, in the United Kingdom, on earth, in purgatory and in heaven, too many to name here, to whom I also extend my sincere gratitude. May God bless you all.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Publications

Athaide, K.F. and Pont, M.J. "**Ameliorating Problems in System Integration with Time-Triggered Scheduling**", Proceedings of the 5th IEEE International Conference on Systems of Systems Engineering (SoSE '10) (Loughborough, UK, 22 – 24 June 2010).

Athaide, K.F. Pont, M.J. and Ayavoo, D. "**Deploying a Time-triggered shared-clock architecture in a multiprocessor system-on-chip design**", Proceedings of the Fourth UK Embedded Forum (Southampton, UK, 9 – 10 September 2008), pp. 96-103. Published by the Institute of Engineering and Technology [ISBN: 978-0-86341-949-2].

Athaide, K.F. Pont, M.J. and Ayavoo, D. "**Shared-Clock Methodology for Time-Triggered Multi-Cores**", Proceedings of the 31st Communicating Process Architectures Conference (York, UK, 7 – 10 September 2008), pp. 149-162. Published by IOS Press [ISBN: 978-1-58603-907-3].

Athaide, K.F. Hughes, Z.M. and Pont, M.J. "**Towards a Time-Triggered Processor**", Poster presented at the third UK Embedded Forum (Durham, UK, 2 – 3 April 2007).

Patents

Pont, M.J., Athaide, K.F. and Ayavoo, D. (2007) "**Debug unit for use with time-triggered systems**" (filed UK, 11 May 2007: details awaited).

Chapter 1

Introduction

"*Simplicity is the ultimate sophistication*" – so articulated Leonardo da Vinci (1452 – 1519), a sentiment later expressed by Albert Einstein with, "*Everything should be made as simple as possible, but no simpler!*" (Einstein 1933). Yet, in our modern society technology gets ever more complex. In his acceptance speech for the 1980 ACM Turing Award, Hoare remarked that the price of reliability is the pursuit of utmost simplicity, and that it is a price most find "*hard to pay*" (Hoare 1981).

Consider the workflow of an average office worker whose day-to-day activities might involve data entry, envelope stuffing, sitting in meetings, etc. The simplest way to get through them is in a sequential, predefined manner (Sasso 1986; Schultz et al. 2003). Unfortunately, activities in a work day cannot be conducted sequentially due to competing requirements: meetings have to be arranged around all participants' schedules; the need to mail items arises at different points in the day; occurrence of emergencies and so on (Hudson et al. 2002). The worker must then jump from job to job to meet daily deadlines (Rouncejield et al. 1994).

Consequently, simplicity is lost because of the change in mindset required by each job interruption (O'Conaill et al. 1995; Schultz et al. 2003). This is in addition to the time required to prepare for and clean up after a job. For example, data entry would require pulling the keyboard closer and opening a spreadsheet program; envelope stuffing involves clearing space for papers,

envelopes and writing materials; meetings require walking to the meeting room and back; and so on. Increased interruptions lead to a loss in reliability due to variability in productivity and/or quality (Schultz et al. 2003; Roper et al. 2007; Haynes 2008) unless made up for during slack time (shortened lunch breaks, overtime, etc.) (Spira et al. 2005).

Additionally, multiple workers doing the same job must also arrange their work times to avoid arguments over company resources. For example, to prevent queues at the printer, workers may arrange to do envelope stuffing at different times. (Gordon et al. 1997)

A straightforward way to remedy the situation is to hire a worker for each job. However, this plainly costs the company in terms of salary even if productivity gains are made (Kim et al. 2004). On the other hand, a worker hired for a particular job does not have to be skilled to do every job and so could be paid less (Sasso 1986), perhaps resulting in a net salary decrease. The real solution, as it almost always is, is a compromise between the two extremes (Kremer et al. 1996).

Henry Ford, whose efforts were *“in the direction of simplicity”* in order to provide “*the very best service and the most convenient in use*”, pioneered by equalising and more than doubling daily wages and by employing workers according to their skill and physical handicaps; the former attracted the highest skill in the field and the latter increased production. His efforts also eliminated interruptions in the form of wasteful material and tool hunting with assembly lines. (Ford et al. 1922)

The office situation has parallels in the field of computing where a personal computer, assuming it has only one “brain” like an office with one worker, cannot nonchalantly spend five hours downloading a file before it allows a letter to be typed in a word processor. So the computer has to alternate very, very quickly between downloading the file and allowing the letter to be typed. Each alternation, like the office worker’s preparation when interrupted, requires memory to be re-allocated and re-loaded, the appropriate computing instructions to be re-fetched, etc. Also, like the office printer, both of the computer jobs will be “printing” to a common storage device and display device.

In this thesis, a solution to this loss of reliability in the field of *embedded computing* is explored by the application of a proven methodology in order to maintain programming simplicity: the exploration being inspired by the recent and rapid adoption of multi-core technology (Borkar 2007) and by the statement that “*the simple and elegant systems tend to be easier and faster to design and get right, more efficient in execution, and much more reliable*” (Dijkstra 1997). Colloquially, the task of keeping an office worker’s jobs simple and interruption-free (and hence reliable) is explored by hiring a superhuman worker with more than one brain.

1.1 Embedded Computing

In computer science, the field of embedded computing studies *embedded processor cores*. These are like their desktop and server variants in that their behaviour can be modified by software; but are different in that they have a longer life cycle, are designed for harsher environments and have lower power consumption. Embedded processor cores are often included with mechanical

and other parts, forming an *embedded system*, the forerunner to tomorrow's cyber-physical system (Lee 2009).

"Embedded" is used to indicate that the presence of the system is unapparent to the end user rather than imposing constraints on physical size, though, undeniably, smaller sizes may facilitate the concealment. It also indicates that the system is part of a larger system or product whose primary functionality need not be as a computer, as in the case of an antilock braking system in a car (Pont 2002; Wolf 2002; Ganssle 2003). Such a system is designed to perform a small number of dedicated functions albeit with choices and different options, i.e. the end-user can make choices concerning functionality but cannot change the functionality of the system by adding/replacing software (Sachitanand 2002; Heath 2003).

Embedded systems have proliferated into our daily lives, being present in televisions, cars, aircraft, Automated Teller Machines (ATMs), etc., and even in such mundane products as ovens, toasters and dishwashers. Soon even engineers will be hard put to identify embedded systems due to the improbability of being able to squeeze in all the required features and intelligence into tiny devices. This can be seen already in contactless payment cards (Olsen 2007) or in car tyres that communicate wirelessly with the vehicle about road conditions, tyre inflation, temperature, etc (Ergen et al. 2009). These systems will soon be woven into everything we touch, including fabrics (Kim et al. 2009).

Already by 2000, 98% of the computing devices sold worldwide were embedded devices (Borriello et al. 2000). The number is expected to reach 16 billion for

this year (nearly 3 embedded devices per person on earth) and to exceed 40 billion in 2020 (Helmerich et al. 2005; ARTEMIS SRAWG 2006). In addition, the global market is expected to increase from £63.3 billion in 2008 to an estimated £77.3 billion by the end of 2013 (Joshi 2009); and expenditure on software research and development is expected to increase from £47.9 billion in 2002 to £109.0 billion in 2015 (Helmerich et al. 2005; Alves 2007).

1.2 Real-time embedded systems

Embedded systems are frequently required to have real-time behaviour, that is, they have to produce an action in response to a stimulus within a specified time interval, independently of how quickly the action is performed (Buttazzo 2002a). Outside of this interval, the result, even if correct, is marginally or completely useless (Audsley et al. 1990). In *safety-critical* real-time systems, safe performance or operation is essential and errors or malfunctions arising due to the failure of real-time behaviour can result in death, injury or illness, major economic loss, mission failure, environmental damage or property damage (Dimond et al. 2002). This description clearly fits systems used in industrial automation, medical equipment, nuclear power plants, avionics and automobiles, amongst others (Redmill 1992; Profeta III et al. 1996).

By their nature, safety-critical systems are expected to be as *dependable* as possible. Dependability is the trustworthiness of a computer system such that reliance can justifiably be placed on its behaviour (Laprie 1992) and subsumes reliability, availability, integrity, maintainability, etc. (Avižienis et al. 2004). It is an assessment shaped by social and psychological factors in addition to hard statistics.

This thesis concerns itself with two aspects of dependability in safety-critical systems; *reliability*, which is the ability to provide a required service, according to stated specifications, for a time under varying operating conditions (IEEE 1990; Laprie 1992); and, *maintainability*, which is the ease with which modifications can be made to correct faults, improve performance or other attributes, or adapt to a changed environment (IEEE 1990). Maintainability, like reliability, assumes that the environmental conditions declared by the system specifications always holds and is differentiated from *robustness* which indicates tolerance to unexpected conditions (IEEE 1990).

1.3 The time-triggered co-operative architecture

Among the myriad of ways to build real-time systems (Fidge 2002; Sha et al. 2004; Buttazzo 2005a), this thesis focuses on the simple, reliable time-triggered co-operative architecture (TTCA) (Pont 2001). It has been found to be a good match for a wide range of applications, such as automotive applications (Ayavoo et al. 2004; Short et al. 2005), wireless (ECG) monitoring systems (Phatrapornnant et al. 2006), various control applications (Edwards et al. 2004; Key et al. 2004; Bautista et al. 2005; Nghiem et al. 2006), data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont 2002).

The co-operative nature of TTCA simplifies software development and facilitates more emphatic guarantees of the quality of interaction with the environment; both, of which make it a reliable architecture. But this nature is simultaneously a big impediment that necessitates an accurate estimation of software execution times. Software non-determinism (from branching, variable

number of iterations and dependence on environmental events) and hardware non-determinism (from caches, direct-memory-access) hinder such estimations.

In addition, software that takes too long to execute can adversely affect system responsiveness, even if such software runs infrequently, e.g. software that runs every ten hours but takes two hours to execute stops the system from doing anything else for those two hours. This then imposes a severe constraint that all software must execute quickly in order to maintain system response times (Allworth 1981; Locke 1992).

In most cases, execution time is not a single value, but a range. In such cases, TTCA exposes fragility if the upper bound of the range is too liberal; leading to an excessive waste of resources. Conversely, conservative estimates potentially throw the entire system off temporally. While techniques have been developed to deal with such failures, they can be counterproductive due to increased complexity and/or the introduction of unknown states into the system (Locke 1992; Kalinsky 2001; Hughes et al. 2008).

TTCA has also been criticised for requiring exhaustive validation and testing during design and maintenance (Liu et al. 1995), both of which are time-consuming and costly. However, given that the cost of unanticipated changes to software increases exponentially as systems age (Griswold et al. 1993), perhaps this extra scrutiny is a blessing in disguise.

Another consequence of co-operative execution and software and hardware non-determinism is that only the software immediately after the triggering event can be guaranteed a start time with minimum variance. All other software is

affected by the cumulative variance of the execution times of all software that executed prior in the same occurrence of the triggering event (Kalinsky 2001).

For these reasons, TTCA has been relegated as an ideal – suitable for only the simplest systems. For everything else, compromises have been made. Commercial system software vendors have also moved on, so that when building a new design or updating an old one, commercial off-the-shelf and standard operating systems are frequently adopted that do not support cyclic scheduling (Liu et al. 1995; Xu et al. 2000).

1.4 Aims of the thesis

The aims of this thesis are to increase the number of *feasible* variants of the single-processor time-triggered co-operative architecture (TTCA) and to increase the temporal quality (i.e. reliability) of responses of TTCA to environmental stimuli. Increasing the number of feasible variants decreases the chance that maintenance will result in an unfeasible design while the use of TTCA will reduce the amount of time needed to understand the program in the maintenance activity.

These aims will be pursued by increasing the concurrency in the system whilst maintaining the design simplicity of software that expects a non-concurrent system.

1.5 Scope

This thesis possesses the following scope:

- The techniques operate on *one* particular software processing core in a real-time system. Any other cores are considered together with the

environment of the system to form the environment of the core. The core may be the sole core in a processor or may be one of several (possibly heterogeneous) cores in a multi-core processor.

- The software executing on the core is expected to run at a fixed rate, to complete execution before a certain time, to have a bounded execution time, and to behave in a highly predictable manner (Stankovic 1988; Stankovic et al. 1990; Spuri et al. 1995; Buttazzo 2005a).
- The transformation under consideration in this thesis will result in the core being converted into a network of cores (Figure 1.1) changing neither the software application programming interface (API) nor the input/output (I/O) interface with other hardware.

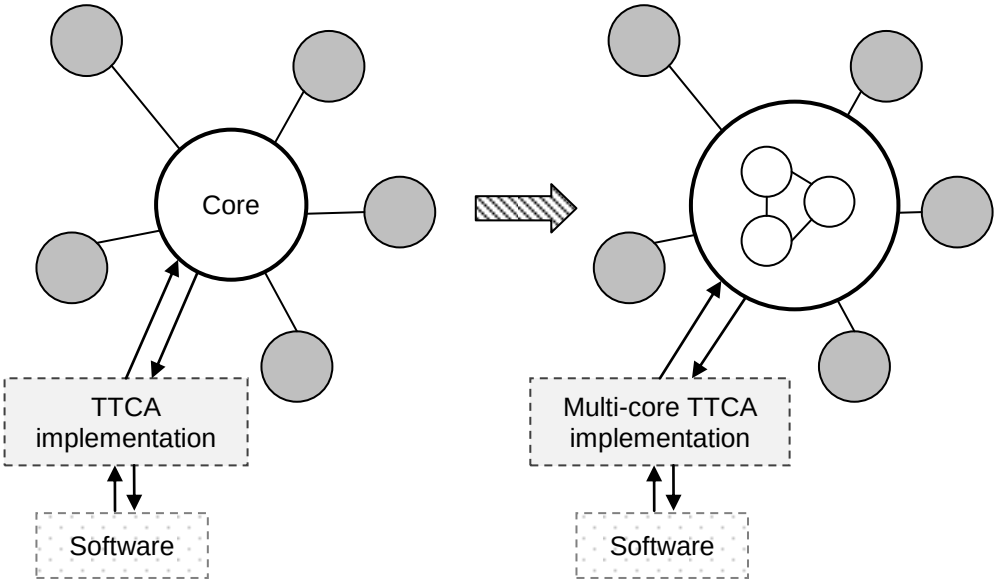

Figure 1.1: The transformation will replace a single-core processor with a multi-core one that preserves the I/O interface with other hardware and the software API.

1.6 Key contributions

This thesis makes the following contributions to increase the applicability of the time-triggered co-operative architecture:

- The first contribution is a novel processor for single-processor TTCA designs that are either unfeasible or impractical due to the presence of software with heavy utilisation and the presence of non-harmonic relationships in software execution rates. This has been achieved by increasing the number of available processing elements, without affecting the application software design. This contribution also has the benefit of reducing the probability of defects being introduced by the maintenance of feasible single-processor TTCA designs.
- The second contribution extends an existing TTCA hardware implementation to work with the new processor. This contribution results in a TTCA implementation that possesses zero scheduler overheads and the ability to execute heavily utilised or non-harmonically related software, in addition to better maintainability and a greater scope for future development.
- The third contribution uses existing software techniques to infuse jitter reduction capabilities into the non-concurrent TTCA hardware implementation and the concurrent version from the previous contribution. This contribution results in a TTCA implementation with zero task release jitter in addition to zero scheduler overheads, the ability to execute heavily or non-harmonically utilised software, better maintainability and a greater scope for future development.
- The fourth contribution is a hardware technique that allows peripherals to be accessed deterministically and concurrently without blocking whilst maintaining their consistency. This contribution eliminates the need for

the gateway tasks inherent in the TTCA implementations of the previous contributions.

- The fifth contribution is a processor core for purely co-operative software that does not suffer from jitter or lengthened latency due to the interrupt servicing, while continuing to provide the system scheduler with flexibility in schedule creation and in the order in which tasks are dispatched.

1.7 Thesis overview

Chapter 2 provides an overview of the software architecture for real-time systems, such as scheduling and resource sharing. The task model used for the hard real-time systems under consideration in this thesis is also described along with an account of latency and jitter and the reason they may give rise to problems in a system.

Chapter 3 provides an overview and execution model of TTCA along with the description of several software implementations and one hardware implementation. The problems that arise from use of TTCA are elaborated in Chapter 4 along with the solutions that have been proposed for these problems and the discrepancies that they possess.

Chapter 5 presents a novel multi-core processor to alleviate the problems of single-processor TTCA designs. The chapter also describes the wait-free loop-free inter-core communication scheme and the scheduler extensions that were made to allow standard TTCA application design to be used with the processor. Chapter 6 presents a case study that examines the maintenance effort required and the cost if a F16 flight system were initially deployed on a single-core

processor with an appropriate scheduler versus the multi-core processor described in Chapter 5.

Next, Chapter 7 incorporates the scheduler extensions of the multi-core processor into an existing TTCA hardware implementation in a successful attempt at achieving zero scheduler overhead and zero task release jitter. Chapter 8 then presents a case study that examines the migration of an existing co-operative system with many small communicating tasks to the presented multi-core platform and the effects this has on the tasks' execution properties.

Chapter 9 introduces a hardware technique to share resources that communicate with the environment, using a simple but novel technique to increase determinism in accessing these resources.

Finally, Chapter 10 concludes with a final summary, with a discussion of limitations and with the consideration of future work that may be performed.

1.8 Conclusions

The intuitive notion that productivity is increased by concentrating on a single task is not only demonstrative in the work place but is also reflected in software design. Particularly, a software developer's development productivity increases if software is written in the co-operative manner of execution. In the safety-critical field, co-operative execution can be coupled with the highly predictable time-triggered architecture to produce the time-triggered co-operative architecture (TTCA) that provides a simple and intuitive interface for application development.

Unfortunately, in the same way that an office worker is often impelled by daily deadlines to abandon simpler ways of working, so also is an embedded systems designer often impelled by real-time constraints to abandon a TTCA implementation. Alternative real-time designs are able to accommodate the constraints, but usually end up compromising on predictability or on the quality of environmental interactions or on the ease of application development.

This thesis, then, presents research aimed at accommodating some of the stricter real-time constraints on TTCA – expanding the range of designs that can be built on this architecture. The next chapter will examine, in greater detail, how the software for real-time systems is constructed and the various design choices involved.

Chapter 2

Software architecture of real-time system schedulers

2.1 Introduction

In real-time computing, the correctness of a system depends both on the logical result of a computation and on the time at which the results are communicated to the environment (Stankovic 1988; Ramamritham et al. 1994). Besides requiring that the logical result be computed before it is communicated, real-timeliness places no other constraint on the computation speed. Predictability is the foremost goal in these systems (Stankovic 1988; Stankovic et al. 1990; Buttazzo 2005a). These systems have a long life-cycle and a perennial uptime, i.e. no restarts are required unless necessitated by a maintenance or upgrade cycle (Koopman 1996).

The software design for these systems is concerned with enforcing the real-time constraints on the scanning of sensors and on the driving of actuators (non-reactive systems log data instead). The application design process leverages the support of a real-time operating system (RTOS) that may also: provide fault tolerance and distribution; and integrate time-constrained resource allocations and scheduling across a spectrum of resources, e.g. sensor processing, communications, etc. This chapter explores some of the design decisions of the RTOS components that manage the execution of application software and that manage the access to resources that connect to the environment.

The next two sections discuss the characteristics of a real-time system's interaction with its environment and the level of quality required from such interaction based on the system's classification. This is followed by a brief description of the hardware model of a typical real-time system and a more in-depth software model. The last few sections briefly examine resource management and scheduling.

2.2 Environmental interaction

Every reactive real-time system may be considered a control system: the system measures one or more environmental properties and responds (via real-time actions or responses) by algorithmically bringing one or more of these properties to desired states. The numerical or logical effects of the actions are constrained by the application design to an interval in which they must be used (the response interval) and to an interval in which the desired performance is obtained (the performance interval) as seen in Figure 2.1. The upper bound of the performance interval is the *deadline* of the response. (Audsley et al. 1990; Buttazzo 2005a)

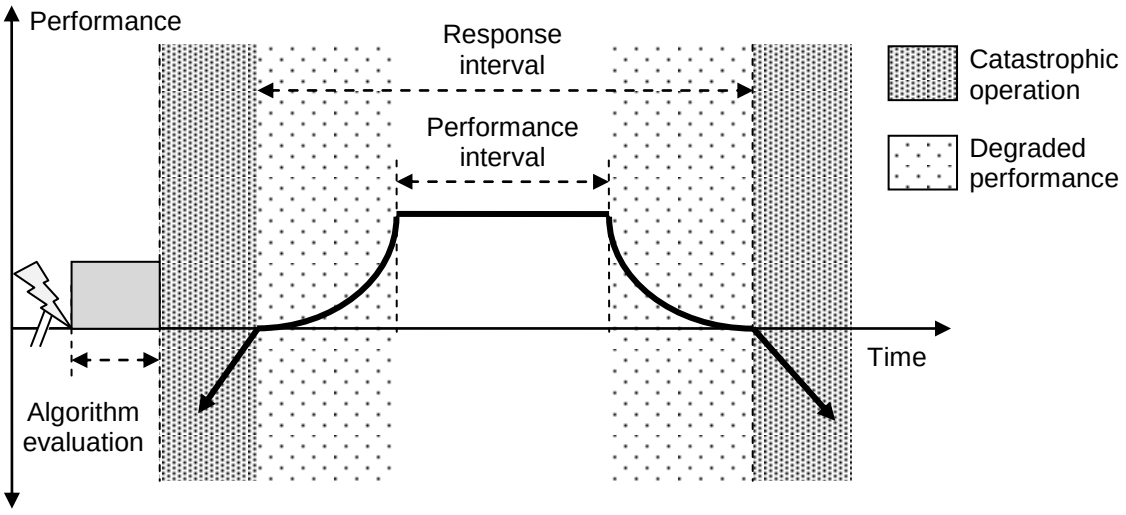

Figure 2.1: Effect of time on the quality of a real-time computation (Audsley et al. 1990)

It is quite common for the design of a real-time embedded response to have an infinitesimally small, fixed-position performance interval, implying that the response must be *time deterministic* i.e., for every timed input stream, a unique timed output stream will be provided (Henzinger et al. 2003; Kopetz 2008). In other words, the temporal and the logical properties of a response should be determinable in finite time through design analysis under all assumed conditions, i.e. that these properties should be *predictable*.

In some literature, the desired time determinism is ambiguously referred to as “predictability” and *determinism* is just as ambiguously used to refer to the degree to which a system’s exact execution sequence can be predicted ahead of time, i.e. *execution determinism* (Locke 1992; Stankovic et al. 1993; Bate 1998). Execution determinism has been found sufficient, though not necessary, to achieve time determinism (Locke 1992; Stewart 2001; Henzinger 2008). On the other hand, undisclosed implementation details that affect observable system behaviour or *observable implementation determinism* is crucial for reliable behaviour and is often given the sole importance (Engel et al. 2004; Henzinger 2008).

The range of the performance interval indicates the tolerated *variance* or *jitter* in the real-time action while the position of the response interval indicates the tolerated *latency*. Increases in the latency of responses inversely affect the system responsiveness and may degrade control performance and cause system instability (Kim et al. 1997). Depending on the application, jitter may also seriously impact performance: it may cause instability due to a variable sampling period (Locke 1992); it may introduce errors significant enough to render a sampled signal meaningless (Cottet et al. 1999); it may need to be

nearly eliminated for specialised I/O devices requiring precise timing relationships between inputs and outputs (Locke 1992); and, it may complicate and delay fault detection and recovery (Lin et al. 1996). For analytical purposes, in addition to the definition above, jitter can be defined as the unpredictable and irregular deviation in the latency of a response (Figure 2.2).

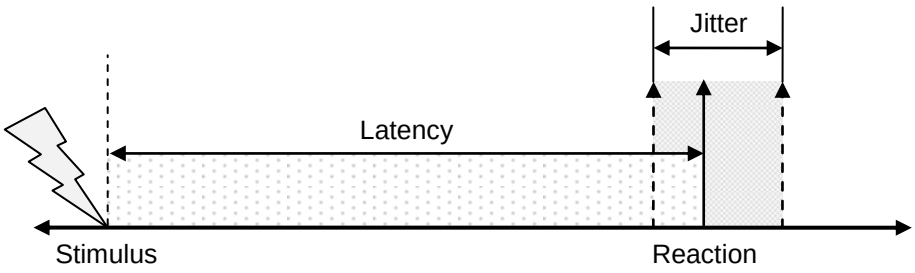

Figure 2.2: Latency and jitter of a response

In real-time computing systems latency and jitter may be caused by underlying hardware such as the oscillator hardware (Schossmaier et al. 1999), memory caches (Mueller et al. 1993; Basumallick et al. 1994; Schneider 2000), direct memory access hardware (Thiele et al. 2004; Pitter et al. 2007), variable event servicing times (Jeffay et al. 1993; Berg et al. 2004), etc. (Sanfridson 2000).

Latency and jitter are key factors of the quality of the system response mentioned in the aims in Section 1.4. The tolerance to these factors varies depending on the application, for example, the jitter may be allowed to tend to the response interval or may be required to tend to zero.

2.3 Classifications

The intervals in Figure 2.1 can be used to make a few distinctions: for a *hard* real-time action, both the response and performance intervals are finite and usually the same, while for a *soft* real-time action, the response interval is

infinitely large[1]. A hard real-time action provides no value to a system when it fails to meet its deadline (Buttazzo 2005a) and is *safety-critical* when the failure can result in death, injury or illness, major economic loss, mission failure, environmental damage or property damage (Dimond et al. 2002). A *firm* real-time action has a performance interval smaller than a finite response interval, tolerates responses anywhere in the response interval (with possible planned degraded performance) and cancels overly long calculations according to the dynamic quality requirements (Goossens et al. 1997; Laplante 2004). In spite of the type of action, a deadline always exists after which catastrophic, zero or mediocre performance is obtained.

A real-time system is normally classified on the maximum criticality of its actions. That is, a real-time system performing even one hard real-time action is a hard real-time system even if it performs soft or firm real-time actions. A real-time system may also have non-real-time software, for example, software that updates a display screen.

Hard real-time systems include devices that must shut down costly transformers before lightning strikes in power lines destroy them (Engblom 2002); engine control units that prevent too early or too late a combustion which can either destroy the engine or result in lower fuel economy and power (Bober et al. 2009); and other such control systems designed to expect determinism without which performance suffers degradation and may even lead to instability in the system (Marti et al. 2001). Soft real-time systems include DVD players,

[1] Performance of soft real-time actions may degrade continuously outside the performance interval, even turning negative, but even so, the consequences will not be catastrophic.

multimedia systems, monitoring apparatuses, telecommunication networks, mobile robotics, etc. Firm real-time systems include video conferencing devices and some database management systems.

2.4 Hardware model

The embedded processor cores mentioned in Section 1.1 perform calculations and interact with the environment through helper non-programmable hardware cores (serial ports, analogue interfaces, etc.) called *peripherals*. They may be placed with these peripherals onto a single integrated-circuit die to form a *microcontroller* or placed alone to form a *microprocessor*; when placed into suitable packaging, a *chip* emerges. The terms microcontroller and microprocessor are, however, gradually being used less than the term system-on-chip (SoC), owing to the scale of today's chips compared to the past (Bjerregaard et al. 2006).

Real-time systems with multiple microcontrollers, i.e. distributed systems, are common in automotive systems where cars may have between 20 and 100 electronic control units (Charette 2009; Ebert et al. 2009) and may even have two communication media (Turley 1999). However, they have their disadvantages such as an increased amount of hardware, wiring, points of failure, etc (Leen et al. 1999). Progressive miniaturisation, from multi-processors to multi-chip modules to chip-level multi-processors (CMP), aimed to eliminate these disadvantages while also providing higher bandwidth, lower latency, greater energy efficiency and more reliable communication (Multicore Association 2008). There is also a drive to replace the electronic control units with CMPs (Bergenhem 2007; Obermaisser et al. 2009).

CMPs have also been termed multi-cores (Wolf et al. 2008), where what was previously termed a “processor” is termed a “core”. “Processor” may instead be used to group processors in a heterogeneous CMP so that the CMP can be referred to as containing homogeneous “processors” which further contain heterogeneous “cores” (Duller et al. 2005). A CMP with peripherals may still be referred to as a microcontroller (Martin 2009) but where the SoC terminology is prevalent and the cores especially heterogeneous, it is referred to as a multi-processor system-on-chip (MPSoC) (Wolf et al. 2008).

The early CMP designs were one integrated-circuit versions of the prevailing multi-processor designs of the time – the shared-memory multi-processors – and leveraged many of the same techniques (Peng et al. 2007). More recently, the number of processors found in such a design has increased into the tens and hundreds (Borkar 2007) and inter-core communication has shifted from simple interconnects to more complex networks on the chip (Ascia et al. 2005; Bjerregaard et al. 2006). For distinction, the former group is identified simply as multi-core while the latter is identified as massively multi-core or many-core (Borkar 2007).

In this thesis, the “multi-core” terminology is adhered to, which has become colloquial usage even among non-technical users.

2.5 Software development model

As mentioned in Section 1.1, the embedded processor cores allow their behaviour to be modified by software, that is, these cores *execute* software by reading sequences of bits or *instructions* from memory hardware and interpreting them. The instructions indicate the calculations to perform and the

patterns of communication with other software processing cores and peripherals. The peripherals either interact (by sensing or actuating) with the environment directly or control another piece of hardware that does the interaction.

Working from a set of human-level specifications, software developers describe calculation algorithms at a comfortable level of abstraction (for example, a textual or graphical programming language) which is then converted by a development software artefact called a compiler into instructions. Algorithms are designed as a series of *tasks* that work together to accomplish a particular goal-oriented *job*. A group of common jobs then constitute a software *application*, several of which may be running on one core. Tasks contain sequential instructions and are meant to run concurrently with other tasks. In addition to task planning, the design process will also involve picking a suitable RTOS. The RTOS is responsible for *task scheduling*, i.e. deciding the time and order for task executions.

For example, a word processing application has jobs for editing and printing amongst others. Editing requires tasks capable of positioning the cursor, highlighting text, saving the text, etc while printing requires tasks capable of checking the printer status, pagination, etc.

In a hard real-time system, some tasks may require the maintenance of the real-time constraints mentioned in Section 2.2. These hard real-time tasks are those that use peripherals connected (directly or indirectly) with the environment, such as an analogue-to-digital converter, a pin driving a relay, etc;

and those that are required to precede the aforementioned tasks. From this, it follows that these real-time tasks must also be time deterministic.

Tasks may be invoked aperiodically (at an irregular, unpredictable rate), sporadically (at an irregular, upper-bounded rate) or periodically (at a regular rate) (Buttazzo et al. 1999; Buttazzo 2005a). For example, operator requests or displaying activities, responding to device interrupts and sensory acquisition or control loops respectively correspond to one of these types. Since many safety-critical activities are driven by periodic real-time tasks (Jeffay et al. 1991; Spuri et al. 1995), the discussion will be limited to periodic tasks (see also Section 2.6.2).

2.6 The task model

In this thesis, the task model consists of a task-set, π in which each task, $\tau \in \pi$ is distinguished by: an indication of its importance (the priority), a type (hard or soft real-time or non-real-time), an initial delay ($\phi(\tau)$ – the phase), the amount of time between consecutive releases ($P(\tau)$ – the period) and a list of tasks that should be executed before it (precedence constraints). A list of resources used by the task may also be included, sometimes annotated with the time and duration of usage of each resource.

Over the lifetime of the system, the task executes in a number, $K(\tau)$ of non-overlapping frames (Figure 2.3). The $k^{th} \in \mathbb{N}$ frame consumes $ET(\tau, k)$ units of processor time – between the best-case $BCET(\tau)$, and the worst-case $WCET(\tau)$ – but may be broken up into a number, $E(\tau, k)$ of executions due to higher priority tasks or to increase system responsiveness; the $e^{th} \in \mathbb{N}$ execution starts

at $S(\tau,k,e)$ and finishes at $F(\tau,k,e)$. The task's worst-case utilisation, $WCTU(\tau)$ of the processor may be used to decide how to execute a frame.

The k^{th} frame also has execution properties: it is released at $r(\tau,k)$ and it has a time by when it should have finished executing (a deadline specified absolutely $d(\tau,k)$ or relative to the release time $D(\tau,k)$). Contentions at run-time will result in the frame actually being released at $s(\tau,k)$, a finite time after $r(\tau,k)$, and finishing at $f(\tau,k)$, a finite time before $d(\tau,k)$.

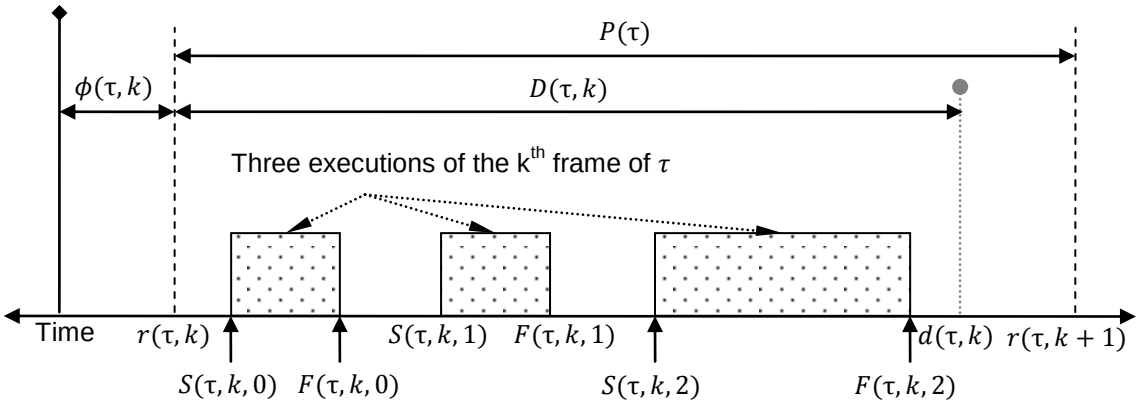

Figure 2.3: An example k^{th} frame of a task τ which has three executions

Using these definitions, some useful relationships can be drawn for a task $\tau \in \pi, 0 \leq k < K(\tau)$ as seen in Equations 2.1 to 2.9.

$$D(\tau,k) \leq P(\tau) \quad (2.1)$$

$$r(\tau,k) = \phi(\tau) + k * P(\tau) \quad (2.2)$$

$$D(\tau,k) = d(\tau,k) - r(\tau,k) \quad (2.3)$$

$$s(\tau,k) = S(\tau,k,0) \geq r(\tau,k) \quad (2.4)$$

$$f(\tau,k) = F(\tau,k,E(\tau,k)) \leq d(\tau,k) \quad (2.5)$$

$$ET(\tau,k) = \sum_{e}^{E(\tau,k)} \{F(\tau,k,e) - S(\tau,k,e)\} \quad 2.6$$

$$WCET(\tau) = \max_{k \in K(\tau)} \{ET(\tau,k)\} \quad 2.7$$

$$BCET(\tau) = \min_{k \in K(\tau)} \{ET(\tau,k)\} \quad 2.8$$

$$WCTU(\tau) = \frac{WCET(\tau)}{P(\tau)} \quad 2.9$$

At any time $t \geq \phi(\tau)$, a frame may have some computation time left $C(\tau,t)$ and available slack or laxity, $L(\tau,t)$ as seen in Equation 2.10; the maximum laxity (Buttazzo 2005a) is at the release time of the frame. The worst-case utilisation (WCU) for the task-set can be seen in Equation 2.11.

$$L(\tau,t) = d\left(\tau, \left\lceil \frac{t}{P(\tau)} \right\rceil\right) - t - C(\tau,t) \quad (2.10)$$

$$WCU = \sum_{\tau \in \pi} WCTU(\tau) \quad (2.11)$$

A task-set is complete or concrete if $\phi(\tau)$ is known *a priori* $\forall \tau \in \pi$. The least common multiple (LCM) of the periods of the tasks in a set is sometimes called the *major* cycle or *hyperperiod* $H(\pi)$. If a set of tasks can be executed in a particular system, then that set is *feasible* for that system. When checking for feasibility, it is necessary and sufficient (for $WCU < 1$) to analyse the interval from 0 to $\max_{\tau \in \pi}\{\phi(\tau)\} + 2 * H(\pi)$ (Pellizzoni et al. 2004). For brevity, the upper bound will be represented as $\phi_{\max} + 2H$ instead.

Sometimes, for ease of specification, the system is assumed to have implicit deadlines, i.e. $D(\tau,k) = P(\tau), \forall \tau \in \pi, 0 \leq k < K(\tau)$; and the task-set is assumed

to be synchronous, i.e. $\phi(\tau) = 0, \forall \tau \in \pi$. However, asynchronous task-sets can be advantageously feasible when synchronous ones aren't (Tindell 1994; Pellizzoni et al. 2004), for example, phases and priorities can be used to specify precedence constraints (Audsley 1991).

2.6.1 Harmonic dependencies between periods

Consider two tasks, $\tau_a$and τ_b with initial delays of $\phi_a$and ϕ_b respectively, such that $P(\tau_a) < P(\tau_b)$, then the two tasks will be released at the same time, at their k_ath and k_bth frame respectively, whenever Equation 2.12 holds.

$$k_a = \frac{k_b P(\tau_b)}{P(\tau_a)} + \phi \; where \; k_a, k_b \in \mathbb{N}, \phi = \frac{\phi_b - \phi_a}{P(\tau_a)}, \mathrm{k_b} \geq \frac{-\phi * \mathrm{P}(\tau_a)}{\mathrm{P}(\tau_b)} \qquad (2.12)$$

It is easy to see that if $P(\tau_b) = n * P(\tau_a), n \in \mathbb{N}^*$, then a release of τ_b can be expected after n releases of τ_a. In other words, if the periods are harmonic (one is a multiple of the other), the tasks will always be released with a constant relationship. Task-sets where all the tasks have the same or harmonically related periods are likely to have a low hyperperiod and they simplify feasibility analysis (Abdelzaher et al. 2000; Kuo et al. 2000).

On the other hand, if $\mathrm{P}(\tau_b) \neq \mathrm{n} * \mathrm{P}(\tau_a), \mathrm{n} \in \mathbb{N}^*$, then the hyperperiod may be large and the relationship between the tasks' release times will vary over the range $[0, \phi_{\max} + 2H]$. Moreover, these relationships need not be the same for every task combination, leading to changing execution patterns where the set of tasks executing before a particular task varies. Due to these variations, system utilisation may increase (Gill et al. 1999) and so certain scheduling techniques prefer to partition tasks on the basis of harmonic relations between their periods (Abdelzaher et al. 2000; Ekelin et al. 2001).

2.6.2 Other types of tasks

It is worth noting that this periodic task model can also be used for a task τ_s that is released sporadically (Equation 2.13) (Jeffay et al. 1991) and for a one-shot task τ_{os} that is only released once (Equations 2.14 to 2.17).

$$P(\tau_s) = \min\{r(\tau_s, k+1) - r(\tau_s, k) | 0 \leq k < K(\tau_s) - 1\} \quad (2.13)$$

$$\phi(\tau_{os}) \in \mathbb{N} \quad (2.14)$$

$$P(\tau_{os}) = 1 \quad (2.15)$$

$$D(\tau_{os}) \geq WCET(\tau_{os}) \quad (2.16)$$

$$|K(\tau_{os})| = 1 \quad (2.17)$$

2.6.3 Latency

The instantaneous latency or delay $Lt(\tau, k)$ of the k^{th} frame, $0 \leq k < K(\tau)$, of task τ, is the difference between the start time and the release time (Equation 2.18), though it can also be expressed as the maximum for the assessment of a scheduling algorithm (Equation 2.19).

$$Lt(\tau, k) = s(\tau, k) - r(\tau, k) \quad (2.18)$$

$$Lt_{max}(\tau) = \max\{s(\tau, k) - r(\tau, k) \mid 0 \leq k < K(\tau)\} \quad (2.19)$$

2.6.4 Jitter

The jitter $J(Q)$ in an ordered set Q of time measurements can be calculated as in Equation 2.20. In some work, the maximum difference between two consecutive time measurements is used as a relative measure of jitter

(Equation 2.21) and the range as the absolute measure (Equation 2.22) (Buttazzo 2005a).

$$J(Q) = \sqrt{\frac{\sum_{q \in Q}(q - \bar{Q})^2}{|Q|}} \tag{2.20}$$

$$J_A(Q) = \max_{1 \leq i < |Q|} \{|Q_{i+1} - Q_i|\} \tag{2.21}$$

$$J_R(Q) = \max(Q) - \min(Q) \tag{2.22}$$

Depending on the application, performance may be seriously impacted by jitter: it may cause instability due to a variable sampling period (Locke 1992; Törngren 1998; Marti et al. 2001); it may introduce errors significant enough to render a sampled signal meaningless (Cottet et al. 1999); it may upset the precise timing relationships between inputs and outputs for specialised I/O devices (Locke 1992); it may complicate and delay fault detection and recovery (Lin et al. 1996); and, it may generally degrade performance in control applications (Hong 1995). Since all observable behaviour in the systems under consideration is an outcome of task execution, the discussion changes to one of jitter in a task's various timing properties: a task can suffer from release jitter, execution jitter, completion jitter and finishing jitter (Buttazzo 2005a).

2.6.4.1 Execution jitter

Execution jitter, $J_E(\tau)$ for $\forall\tau \in \pi$, is a variance in the execution time of a task and is quantified as in Equation 2.23 and illustrated in Figure 2.4, where for $0 \leq k < K(\tau)$,$\{ET_1, ET_2 \ldots ET_n\} \in ET(\tau, k)$ and, $\{r_1, r_2 \ldots r_n\} \in r(\tau, k)$.

$$J_E(\tau) = J(\{ET(\tau, k) | 0 \leq k < K(\tau)\}) \tag{2.23}$$

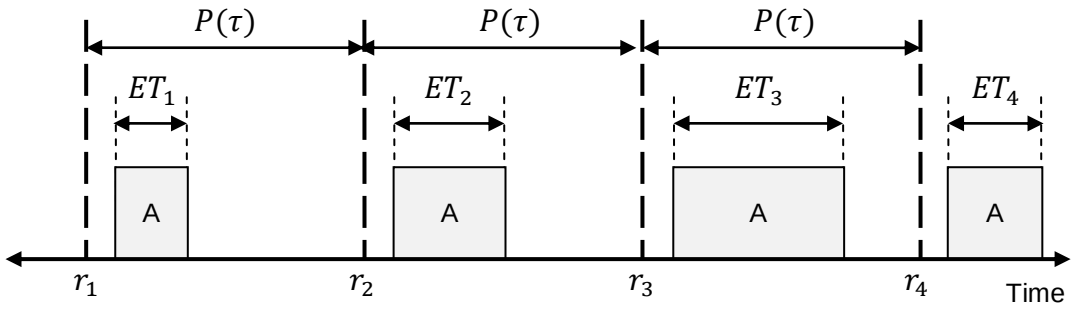

Figure 2.4: Execution jitter in periodic tasks

The primary reason for execution jitter is the presence of multiple execution paths in the program each of which may take different times to complete. The decision of which path to take is often based on unpredictable inputs and cannot be compensated for statically (Engblom 2002). The situation is exacerbated further by the trend to increase average performance with embedded processor developments (Berg et al. 2004).

When systems are designed statically, they are designed to worst-case performance (Engblom 2002) and a large execution jitter, e.g. larger than 10% of a task's period (Cottet et al. 1999), can be indicative of unusable computation time and may also cause schedule non-determinism (Gendy 2009).

2.6.4.2 Completion jitter

Completion jitter, $J_C(\tau)$ for $\forall\tau \in \pi$, is a variance in the execution time of a task in the presence of pre-emption and is quantified as in Equation 2.24, where for $0 \leq k < K(\tau)$,$\{ET_1, ET_2 \dots ET_n\} \in ET(\tau, k)$ and, $\{r_1, r_2 \dots r_n\} \in r(\tau, k)$.

$$J_C(\tau) = J(\{f(\tau, k) - s(\tau, k) | 0 \leq k < K(\tau)\}) \quad (2.24)$$

In the absence of pre-emption, completion jitter is the same as execution jitter.

2.6.4.3 Finishing jitter

Finishing jitter, $J_F(\tau)$ for $\forall\tau \in \pi$, is a variance in the finishing time of a task. It can be quantified as in Equation 2.25 and is illustrated in Figure 2.5, where for $0 \leq k < K(\tau)$, $\{j_{f1}, j_{f2} \dots j_{fn}\} \in J_F(\tau)$; $\{r_1, r_2 \dots r_n\} \in r(\tau, k)$; and, $\{f_1, f_2 \dots f_n\} \in f(\tau, k)$.

$$J_F(\tau) = \quad J(\{f(\tau,k) - r(\tau,k) \mid 0 \leq k < K(\tau)\}) \tag{2.25}$$

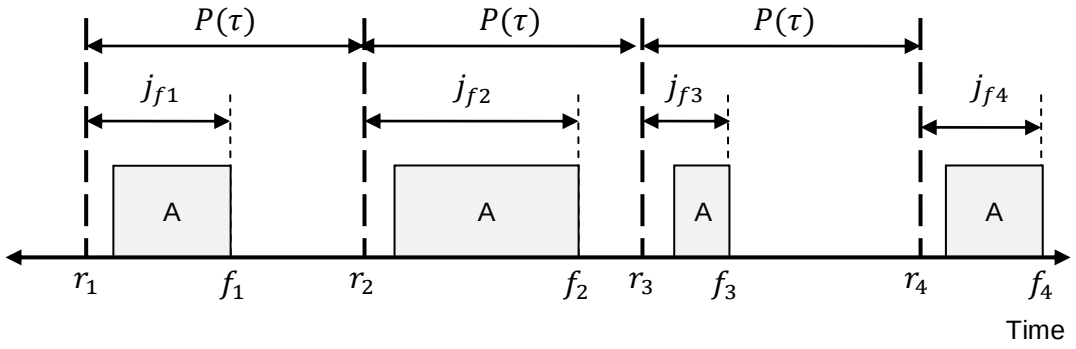

Figure 2.5: Finishing jitter in periodic tasks

This jitter is a function of execution jitter and release jitter.

2.6.4.4 Release jitter

Release jitter, $J_R(\tau)$ for $\forall\tau \in \pi$, is a variance in the actual release time, i.e. the start time, of a task. It can be quantified as in Equation 2.26 (see Section 2.6.3) and is illustrated in Figure 2.6, where for $0 \leq k < K(\tau)$, $\{j_{r1}, j_{r2} \dots j_{rn}\} \in J_R(\tau)$; $\{r_1, r_2 \dots r_n\} \in r(\tau, k)$; and, $\{s_1, s_2 \dots s_n\} \in s(\tau, k)$.

$$\begin{aligned} J_R(\tau) = \quad & J(\{s(\tau,k) - r(\tau,k) \mid 0 \leq k < K(\tau)\}) \\ & = J\{Lt(\tau,k) \mid 0 \leq k < K(\tau)\} \end{aligned} \tag{2.26}$$

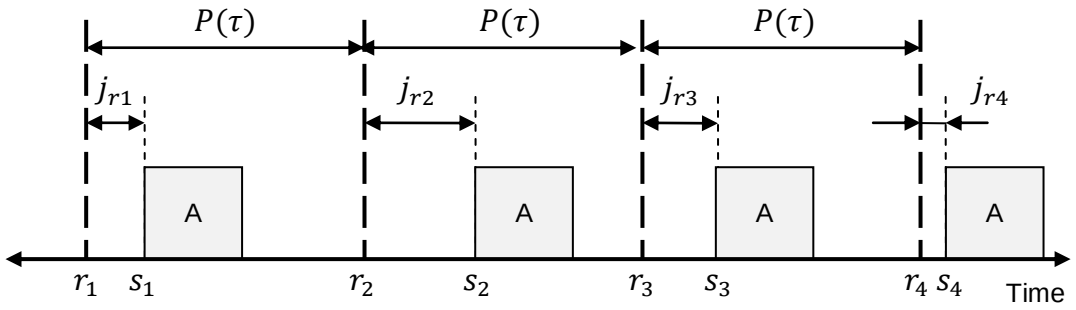

Figure 2.6: Release jitter in periodic tasks

2.7 Shared resource management

In a real-time system, resources include the peripherals, the memory banks, etc. Tasks will frequently share resources (Audsley et al. 1990), often for inter-task communication, and will, hence, possess *critical sections* that require mutually exclusive transactional access to the shared resources in order to maintain data coherence (Caccamo et al. 1999). In addition to temporal and precedence constraints, a scheduling algorithm must ensure that critical sections involving the same resource do not overlap (Pont 2001); this is why real-time scheduling has also been referred to as resource scheduling (Locke 1992; Ramamritham et al. 1994).

Most modern systems have concurrent execution paths, actual or simulated, and so *shared resource management* (the management of critical sections) has become an essential feature of modern operating systems. Resource contention is the primary reason for design complicatedness when dealing with concurrent software and care must be taken due to the implications that different management techniques may have on the real-time requirements. There are principally two predictable management techniques, the blocking and non-blocking techniques (Audsley 1991). In the blocking technique, a task is

stopped from doing all work while in the non-blocking technique, a task will always perform work; however, this work may or may not be pertinent to the task's purpose.

2.7.1 Blocking techniques

The most common way of preventing concurrent resource access is by one task claiming ownership of the resource. All other tasks needing that resource must wait until the owning task relinquishes ownership. Tasks typically use non-busy or pre-emptable waiting, i.e. they yield control of the CPU when blocked, allowing other tasks to go ahead (Mok 1983). When a task claims ownership of a resource, it is described as *locking* the resource and the tasks that must wait as a result are described as having their execution *blocked*.

One quick and simple locking method is to temporarily prevent tasks from being scheduled (Pont 2001; Wang et al. 2007) which, of course, runs the risk of missing events, adversely affecting system responsiveness and real-time performance. However, it is useful when the locking time is very small as it involves very little overhead. The other method is to employ a semaphore (Audsley 1991).

Locks can cause problems if not used carefully (Figure 2.7): some tasks may be always held up while others progress on (resource starvation); the entire system may be held up while each task cyclically waits on another to relinquish ownership (deadlock); tasks may continually choose to wait politely on detecting a contention resulting in a situation where no work is done other than alternated polite waiting (livelock); a task's input data may occasionally be unavailable or corrupted due to unconsidered flow and anti-dependencies (race conditions);

and, a high priority task may be pre-empted by lower priority ones when the former is blocked over resource contention with even lower-priority blocked tasks (bounded priority inversion) (Sha et al. 1990). Various resource protocols have been conceived to deal with these problems (Audsley 1991), but blocking can unpredictably affect a task's WCET, complicating software design (Audsley 1991; Chen et al. 1997a).

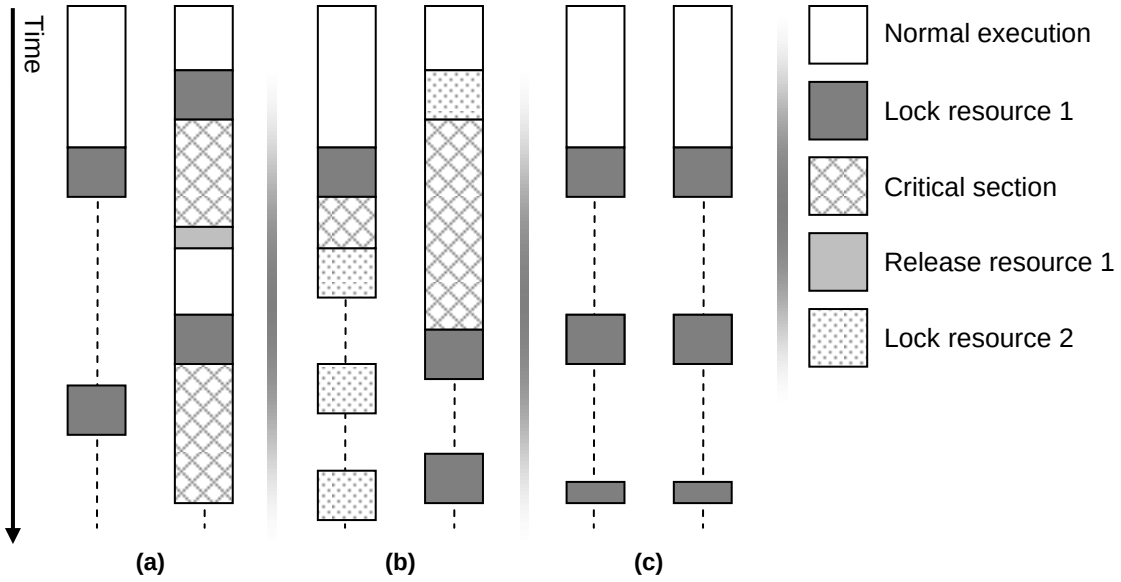

Figure 2.7: A few of the problems with resource sharing: (a) resource starvation; (b) deadlock; (c) livelock

To increase predictability, the blocking schemes may be implemented statically by carefully scheduling tasks at design time on the basis of their real-time properties and resource usage (Zhao et al. 1987; Xu et al. 1990), even for multi-processors (Xu 1993). This allows the blocking induced WCET changes to be known beforehand (Xu et al. 2000); however, regions in the task still need to be identified and marked as critical sections along with the resources used by those sections.

2.7.2 Non-blocking techniques

These techniques prevent priority inversion and deadlocks with varying levels of guarantees on the lack of starvation: at the basic level, *obstruction-free* techniques guarantee progress for a task that executes in isolation; the *lock-free* techniques ensure that at least one task will always make progress; and, the *wait-free* techniques ensure that every task will continue to make progress (Herlihy et al. 2003). The stronger guarantees are more difficult to achieve and require more complicated and expensive algorithms (Fich et al. 2005). These techniques have also been studied as the creation of *atomic registers*, an abstraction where overlapping reads and writes to an *n*-bit register always behave as if operating in a fixed sequential order (Lamport 1986a; Lamport 1986b; Tromp 1989; Anderson et al. 1992).

Non-blocking techniques alleviate the complexity introduced by resource-sharing and software can be written in a sequential manner, simplifying formal and informal reasoning (Herlihy 1993). They do not employ critical sections but allow concurrent resource access.

In some cases, the technique involves continually accessing a shared object, checking and retrying the access if coherence is lost (Cho et al. 2010), for example, the lock-free technique of “read-and-check” loops (Lamport 1977; Bershad 1993; Herlihy 1993; Kopetz et al. 1993; Anderson et al. 1997b); in one wait-free technique, tasks help other ones out (Anderson et al. 1997a). These two techniques are non-blocking in the sense that the task is performing work but may be blocking in the sense of the task’s real objective when the work being performed is not advancing towards the objective, i.e. the work being performed is a busy-wait, a repetition of work that was corrupted by a

concurrent task or the work of another task. There may also be effects on the real-timeliness depending on the properties of the task-sets (Anderson et al. 1997b; Holman et al. 2006).

To allow concurrent access without affecting the timing of the tasks, wait-free and *loop-free* techniques have been devised, such as the writer maintaining a buffer for each reader (Peterson 1983). However, this can be inefficient when the amount of data to be written is large; hence, an alternative of switching between buffers where writers and readers all access different buffers (Sorenson et al. 1975; Clark 1989) gained attention. The initial design showed that for *m*-writers and *n*-readers, ($n + m + 1$) buffers were sufficient, though subsequent work demonstrated that for a fully asynchronous single writer, single reader system four buffers were required (Simpson 1990). This was later generalised for *n* readers as requiring ($2n + 2$) buffers (Chen et al. 1998). Similar mechanisms were also devised in studies on the atomic register abstraction (Tromp 1989; Anderson et al. 1992; Anderson et al. 2000).

The optimal three buffer scheme for one-reader and one-writer was originally dismissed due to timing anomalies (Simpson 1990), but alterations since then have addressed the vulnerability (Chen et al. 1997b) and have also generalised the algorithm for *n* readers, requiring the optimal ($n + 2$) buffers (Chen et al. 1997b; Chen et al. 1998). The time and space costs of these algorithms have been further optimised by taking into account the timing properties of the readers and writers (Huang et al. 2002; Cho et al. 2005; Cho 2006; Cho et al. 2007).

The technique of non-blocking communication buffers also allows communication between tasks with non-harmonic period relationships (Buttazzo 2002b).

2.7.3 Multi-processor

The true concurrency available in multi-processor systems was part of the motivation for the non-blocking resource sharing schemes (Lamport 1977; Herlihy et al. 2008). In these systems, it is preferable to have wait-free rather than lock-free schemes and to avoid blocking (Brandenburg et al. 2008b). This is because blocking schemes effectively serialise resource access and impede any concurrency gain.

The non-blocking schemes can be applied directly (Kopetz et al. 1993) or through concurrent objects that employ non-blocking techniques (Herlihy et al. 2008) such as stacks, queues, lists etc. (Treiber 1986; Herlihy 1993; Valois 1994; Valois 1995; Michael et al. 1998; Tsigas et al. 1999; Åkesson 2001; Sundell 2004; Sundell et al. 2008) especially when the processors share memory. The processors may also communicate using protocols over message passing networks (Pont 2001) if available. Several application programming interfaces (APIs) have been built (sometimes as part of the RTOS) for inter-processor communications; but even these require changes to deal with the different constraints introduced by miniaturisation (Section 2.4) (Multicore Association 2008).

Another technique popularised in 1993 (Herlihy et al. 1993) was hardware transactional memory, later constrained as software-only transactional memory (Shavit et al. 1997), though work continues with more-efficient embedded (Ferri

et al. 2010) and real-time (Schoeberl et al. 2010) versions. Resource accesses are made optimistically in logical groups called transactions and are committed later upon successful validation. Transactions may be aborted at any time, at which point all modifications are undone automatically or retried. This technique bears similarities to the lock-free “read-and-check” scheme and may similarly suffer from the same timing uncertainties.

2.7.4 Peripheral management

Resources that interface with the environment, i.e. input/output (I/O) resources or *peripherals*, may be either partitioned amongst tasks or shared. Modifications made to registers in such resources may be non-transactional and may initiate immediate environmental reactions; hence, most of the resource management techniques above are applicable only to memory-type resources and need to be altered for peripherals.

In embedded systems, peripherals are generally accessed by reads and writes to memory (Berg 2009), but non-blocking management algorithms cannot be directly applied unless the peripheral has been specialised to participate. If the peripheral is to be treated as a resource, the only choice is to employ some form of blocking such as multiplexing between processors (Gary et al. 2004; Chen et al. 2009) or to perform some sort of specialisation such as pipelining (Fort et al. 2006). However, the multiplexing scheme still requires support for transactional accesses besides blocking one core from any sort of access. The pipelining method may also be limited depending on the peripheral or may be too costly based on the amount of duplication required.

Another solution is to use a gateway, i.e. a single task through which all communication with the peripheral is performed (Audsley et al. 1993). When done in this manner, the chore becomes coherent communication with the gateway – this can be performed using any of the techniques in the sections above.

In virtualisation, where multiple operating systems (OSes) are allowed to execute in isolation on the same hardware platform, an OS may have one software module that communicates with another software module controlling the peripheral, which may either be owned by another OS or exist as a separate entity (Heiser 2007). Other solutions offload some of this functionality into hardware for higher performance (Raj et al. 2007) while a safety-critical solution may strictly partition the peripherals (Crespo et al. 2010).

2.8 Real-time task scheduling

Real-time task scheduling sequences the execution of tasks so that their timing constraints are met (e.g. completion before the deadline), protects shared resources from simultaneous access and achieves predictable system behaviour (Ramamritham et al. 1994; Joseph 1996). The secondary goal is usually to achieve maximum processor and resource utilisation.

Scheduling decisions are initiated based on the occurrence of particular combinations of external or internal events. These events may be physical switch depressions, message arrivals, the elapse of one second, a resource being freed etc. Events can interrupt the system at any time, causing it to break the current execution flow and execute new instructions from the *interrupt service routine* (ISR) that handle the interrupt. ISRs steal CPU time and the

general convention is to keep them as short as possible and to reduce the number of resources they use. When not running instructions from real-time tasks or from the ISR, the system is said to be *slacking* (Davis et al. 1993). During this time, non-real-time or aperiodic tasks may be executed or the system may switch into a low-power mode.

One of the key components of an RTOS is the *scheduler.* A scheduler is identified by its trigger architecture (i.e. which events trigger the task execution) and by its execution architecture (i.e. the order in which various tasks are executed after being triggered) and enforces a task *schedule*, i.e. directives on when a task should be started by a task *dispatcher*. The schedule is created according to an algorithm and is considered *valid* if all tasks complete before their deadlines and if no task runs before its release time.

2.8.1 The scheduling algorithm

A scheduling algorithm is a set of rules derived from the scheduling goals and is realised by an implementation of a scheduler (Pont et al. 2007). The scheduling algorithm is considered *optimal* if it can find a schedule for any feasible task-set. Scheduling algorithms have associated feasibility tests: tests that must always succeed for feasibility are *necessary*; those that succeed for some task-sets but not for others even if they are feasible are *sufficient*. Sufficient tests can be overly pessimistic, missing several feasible schedules.

Under any algorithm, a task is started as soon as it is released only if it is a higher priority than the task currently executing or than all the tasks waiting to execute; and if all its precedence and resource constraints have been fulfilled. Priorities are assigned by the designer or by the algorithm on the basis of

certain task properties (Buttazzo 2005b). These properties may be fixed at run-time, such as the period or relative deadlines, or may vary, such as the laxity or absolute deadlines (Liu et al. 1973; Oh et al. 1998). This implies that the priority assignment can be made before the system is started (fixed priority) or has to be made while the system is running (dynamic priority).

Static or *offline* algorithms build a complete schedule prior to execution while *dynamic* or *online* algorithms build a part or the whole of the schedule at run-time (Stankovic et al. 1995). The algorithm will have to be dynamic if any of the properties depended upon change dynamically. The re-run of a dynamic algorithm may be triggered by events such as the time slice elapsing, a task finishing, a task being released, a task being blocked, a new task being added, etc.

Static schedules clearly provide schedule determinism but must assume the worst-case as the average behaviour and may, hence, never fully utilise the processor. Dynamic algorithms provide great adaptability and flexibility but have greater overhead and comparatively poorer schedule determinism. (Liu et al. 1973; Kopetz 1991a; Ramamritham et al. 1994; Fidge 2002; Scheler et al. 2006) Static scheduling algorithms, however, can be more complex and can search a larger state space as they run on development machines with more computational resources and can have a much greater amount of time in which to make a decision (Xu et al. 1993; Ramamritham et al. 1994; Xu et al. 2000; Goossens et al. 2004; Buttazzo 2005a; Lee 2009).

Hybrid algorithms have also been designed, for example, some systems have a “measurement” and/or “schedule creation” period immediately after start-up but

before regular operation where unknown quantities and the schedule are decided (Gendy et al. 2007a; Nahas 2008); others might have several static schedules that are non-deterministically switched in as the state of the system changes (Hanif et al. 2008); others may schedule some tasks statically and others dynamically.

2.8.1.1 Run-time complexity

Intuitively, it is better to check a task-set for feasibility (page 2-11) before expending resources to compute a schedule. Unfortunately, for most systems, the feasibility tests are computationally intractable except for special cases such as the implicit-deadline (Liu et al. 1973; Jeffay et al. 1991) and synchronous (Baruah et al. 1990) pre-emptive systems; and, the non-concrete and concrete sporadic (Jeffay et al. 1991) co-operative systems. The special cases may be used for all types but may prove to be sufficient but not necessary, i.e. overly pessimistic. The intractability only increases when tasks share resources by blocking schemes (Mok 1983; Audsley et al. 1990; Audsley 1991).

To keep costs low, suboptimal, tractable algorithms are employed based on branch and bound methods or on computationally simple heuristics (Burns et al. 1995). This has been tried on asynchronous pre-emptive task-sets (Pellizzoni et al. 2004), and for shared resources (Zhao et al. 1987). A similar algorithm for time-triggered co-operative schedulers was developed (Gendy et al. 2008a) and refined (Gendy et al. 2008b).

2.8.2 The trigger architecture

A trigger architecture where multiple, possibly aperiodic, events are allowed to trigger the task executions results in the *event-triggered architecture* (ETA)

(Kopetz 1991b). A specialisation of the ETA is the *time-triggered architecture* (TTA) which has *one* primary event, periodically originating from a highly accurate internal or external time-keeper. A TTA approaches ETA-level event-perception by polling the non-triggering events as required.

In the TTA, the time of arrival of at least the next event is known, whereas in the ETA, the time of arrival of the next event can at best be bounded. This makes the TTA more predictable (Albert 2004) with the potential to be dynamic by adjusting the occurrence of the next timing event (e.g. a flight control system about to initiate landing procedures).

The emphasis on time as a first-order quantity can simplify communication, establish state consistency, promptly perform error detection, and support the timeliness of real-time applications. TTA designs are recognised to have much more predictable behaviour and hence are widely recognised as providing benefits to both reliability and safety in some of the more safety-critical applications e.g. the main mission computer software in the Lockheed C130J (Amey 2002). (Kopetz et al. 1994; Maier et al. 2002)

However, the TTA may not be suitable for all systems: due to the polling of events, the TTA may not be as reactive as a full-blown ETA and may have a worse power profile; and, polling events with highly variable frequencies can also necessitate an unnecessarily high polling frequency (Albert 2004). The best-case performance of the ETA may always be better or the same as an equivalent TTA. However, the worst-case performance of the ETA might result in tasks missing deadlines, or worse, event occurrences being missed (this can be avoided, to an extent, with hardware enhancements (Siemers et al. 2005)).

Both display, more or less, the same average-case performance. (Scheler et al. 2006)

On the other hand, TTA designs have a very simple architecture, making them simple to understand and maintain (Liu et al. 1973); are more comfortably certified (Pont 2001); and, emphasise correctness-by-construction, a methodology which for the reason of predictable and highly reliable operation has inspired programming languages (Chapman 2006) and model-driven design approaches (Bordin et al. 2007).

2.8.3 The execution architecture

The execution architecture can take the form of executing tasks one after the other or the form of interleaving frames. This forms *co-operative* and *pre-emptive* architectures respectively; in the former, a task has to willingly relinquish resources and stop running while in the latter, a task can pause others, execute and then resume the paused tasks.

The sequential nature of the co-operative execution architecture attracts many software designers as there are no contentions over common resources and hence none of the complexity that comes with resource sharing interactions (Locke 1992; Kalinsky 2001). The architecture is very deterministic and is desirable particularly for use in safety-related systems (Allworth 1981; Ward 1991; Nissanke 1997; Bate 2000). Compared to a pre-emptive architecture, a co-operative one can be identified as being simpler, having lower overheads (as there are no context switches), being easier to test, having greater support from certification authorities (e.g. avionics standard DO-178B) and being supportive

of very tight jitter requirements (Jeffay et al. 1991; Locke 1992; Bate 2000; Fidge 2002).

2.8.4 Multi-processor scheduling

In scheduling a task-set on a homogeneous or heterogeneous multi-core, tasks can be allocated to the cores statically (partition scheduling) or dynamically (global scheduling) (Coffman Jr et al. 1972; Dhall et al. 1978; Burchard et al. 1995; Sha et al. 2004). Though global scheduling can achieve a more balanced workload and increased performance compared to partition scheduling (Kumar et al. 2004), it is not as good a match for hard real-time systems (Lauzac et al. 1998; Brandenburg et al. 2008a) and like other dynamic systems can affect simplicity and predictability (Burchard et al. 1995; Monot et al. 2010).

In partition scheduling, the choice of scheduling algorithm on a processor is made first and then the task-set is iteratively scanned and tasks allocated to processors. The allocation is constrained by the feasibility of the task-set under the scheduling algorithm of a processor if the task were assigned there and constraints on the properties of the task itself. However, the search for an optimal partitioning is computationally intractable and hence implementations of partitioning algorithms are heuristic in nature or use global optimisation, aiming to get as close as possible to the minimum number of cores required (Dhall et al. 1978; Monot et al. 2010).

Most of the existing partitioning algorithms have concentrated on pre-emptive fixed and dynamic priority scheduling algorithms where tasks are sorted in by specific property and then assigned to the next or first available processor

(Dhall et al. 1978) or according to the task utilisation (Oh et al. 1993; Burchard et al. 1995; Oh et al. 1995; Lauzac et al. 2003; Karrenbauer et al. 2009).

On the positive side, the heuristics used in these algorithms are general to the problem class and can be re-applied with a different scheduling algorithm. For example, when applied consecutively to an indexed list of processors and to tasks sorted by increasing periods, the current task is assigned by *first fit* (FF) to the processor with the smallest index, by *next fit* (NF) to the current processor or to the next processor and by *best fit* (BF) to the processor that will provide maximum utilisation. The FF heuristic has seen variations with better results such as sorting tasks by decreasing utilisation (FFDU) or by matching on a particular criterion (MFF) such as matching a period criterion (FFMP). Heuristics have also been devised for specialised task-sets, such as tasks with low load factors and for hybrid cases where low-load tasks occur with others.

More recently, an algorithm was designed that considers static cyclic scheduling and constrains tasks by their communication and by their requirements for particular peripherals (Monot et al. 2010).

2.8.5 A part of the system

The RTOS employed might be a commercial endeavour or might be built in-house and becomes a part of the developer's code (Pont 2001). This is manifested when it is used without a clear understanding of its costs or of its special mechanisms for predictability (Katcher et al. 1993) leading to applications exhibiting unpredictable behaviour (Reeves 1998). Scheduler behaviour may also change based on tasks' properties. This is why a

certification process may reassess the entire system even when a small part in a single task has been altered (RTCA SC-167 / EUROCAE WG- 12 1992).

2.9 The complexity of design

In this thesis, the concern is with application complexity rather than RTOS complexity. The RTOS may be simple or complex, but should attempt to provide an interface to the application that simplifies application development. Simplicity is lost if the interface is complex or if the interface doesn't include essential descriptions, such as temporal descriptions (Lee 2009).

Design complexity deals with the overall morphology of the system and, in this chapter's context, is affected by the number of jobs that another job depends on, the cohesiveness of each job and the degree of coupling among jobs (Herlihy 1993; Marco 1997; Sessions 2009). This suggests that tasks of the same job should: be working towards a common goal, have optimal dependence (Miller 1956) on the services of tasks from other jobs and be lightly coupled to tasks from other jobs. This accounts for the complexity of resource sharing which unintentionally couples tasks through a resource.

The tasks' data-flow then gives an early indication of the complexity by a simple count of the interactions. At a task's instruction level, objective metrics, each emphasising particular aspects, have been devised (Sneed 2008) to measure the complexity and can be weighted with subjective aspects such as the designer's experience and length of exposure to the system (Douce et al. 1999). One of these metric measures, the number of lines of code, can also be used as an indication of monetary cost, e.g. £13.6 - £27.3 per line for most

embedded projects, £68.2 per line for military projects and £682.0 per line to reach the level of IBM's space shuttle control software (Ganssle 2008).

The design complexity is important not only at design time but at maintenance, since the system maintainers are often different from the original designers and may have to bear the full weight of the complexity (Douce et al. 1999).

2.10 Conclusions

Real-time systems interact with their environment by sampling one or more signals and by driving one or more other signals. These chores, which build the functionality of the system, are usually performed by several software entities or tasks. This thesis considers only hard-real time systems which must be responsive and time deterministic, and hence requires the tasks to have low latency and low jitter during execution.

Task execution is the responsibility of the scheduler component of an RTOS and the decisions therein trade-off between performance and predictability as dictated by the system and resource sharing requirements; the large design space facilitates the creation of a variety of designs, each suitable for their own application. The pre-emptive variety of designs have gained acceptance due to issues with the responsiveness of the co-operative ones, but the former often lead to unpredictable run-time behaviour.

This thesis maintains that components at any level of abstraction should be made as predictable and repeatable as is technologically feasible with any remaining variability taken care of by a higher level (Lee 2009). For this reason, a scheduler architecture that is highly predictable is preferred for the simplicity

(and, hence, reliability) it lends to application development. One such architecture is the time-triggered co-operative architecture which is discussed further in the next chapter.

Chapter 3

The time-triggered co-operative architecture

3.1 Introduction

In real-time systems, predictability is the foremost goal (Stankovic 1988; Stankovic et al. 1990; Buttazzo 2005a) and a guarantee of achieving accurate real-time behaviour which is further aided by design simplicity (Dijkstra 1997). This is why integrating the predictable time-triggered architecture and the simple co-operative methodology of application design into a time-triggered co-operative architecture (TTCA) proves highly beneficial.

TTCA can simplify software design (Kalinsky 2001), can result in lower overheads, can be easier to test, has greater support from certification authorities, and can be supportive of very tight jitter requirements (Locke 1992; Bate 2000; Fidge 2002).

The next section examines the architecture in greater detail, followed by the feasibility constraints imposed by the architecture. The chapter concludes with a description of existing TTCA implementations on both software and hardware.

3.2 Architecture design

In TTCA, events from an accurate timing source, such as a timer peripheral, trigger the system to execute software sequentially (Figure 3.1) according to a schedule (Table 3.1) set up programmatically (Listing 3.1). In time-triggered jargon, an event from the timing source is referred to as a *tick* while the interval between ticks is the tick interval or *minor* cycle.

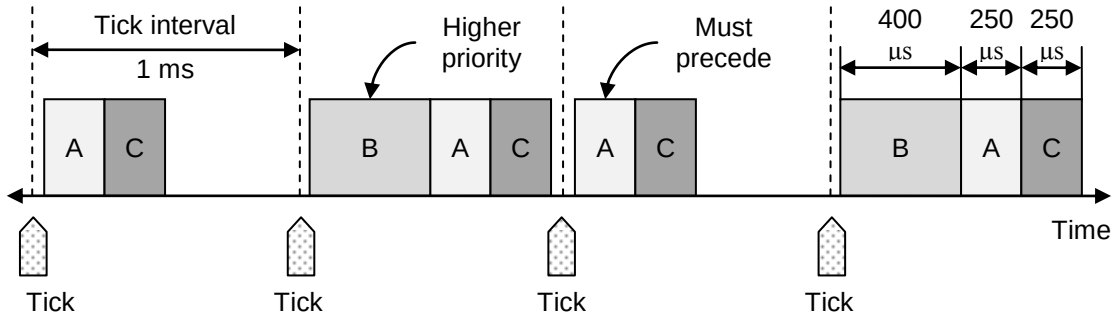

Figure 3.1: Operation of TTCA according to the schedule

Table 3.1: Task schedule

Task	Priority	Delay (ms)	Period (ms)	WCET (µs)	Must be preceded by
A	Normal	0	1	250	—
B	High	1	2	400	—
C	Normal	0	1	250	A

```
DEFINE Start:
   INIT timer WITH tick interval = 1 ms
   INIT scheduler
   INIT data FOR A,B,C

   ADD A, B, C TO scheduler
         WITH Priority    =     Low,   High,    Low,
              Delay       =      0x,     1x,     0x,
              Period      =      1x,     2x,     1x,
              WCET        =  250 µs, 400 µs, 250 µs

   CREATE schedule
   START scheduler

   DO FOREVER:
      SLEEP
      DISPATCH tasks
      CREATE schedule
```

Listing 3.1: Pseudo code for a TTCA implementation

In Listing 3.1, the task properties given in Table 3.1 are assigned to each task, except the resource list which is unnecessary baggage for a co-operative

system and the deadline which, in this case, is assumed to be the same as the period. The tick interval is set to one millisecond (see Equation 3.2 below) and, since tasks are always released at a tick, the release and delay times in the listing are in units of ticks. Precedence constraints are specified by adjusting release times or deadlines or by the order in which tasks are added. The resulting execution sequence is shown in Figure 3.1.

It is easy to see that when the designs avoid or make only pre-determined changes to the set of tasks, they make the system inherently schedule deterministic (Locke 1992; Fidge 2002), and hence execution timing is predictable (Baker et al. 1988).

3.2.1 The TTCA model

In TTCA, for a given task-set π, every θ^{th} ($\theta \in \mathbb{N}$) tick is followed by the θ^{th} tick interval and results in the release of the tasks in the ordered set $\rho(\theta) \subseteq \pi$ (Equation 3.1); this can be used to form the ordered set of tasks $\rho_{pre}(\theta, \tau) \subseteq \rho(\theta)$ that are to be released before a task τ as a result of the θ^{th} tick.

$$\rho(\theta) = \left\{\tau \middle| \frac{\theta - s(\tau)}{p(\tau)} \in \mathbb{N}, \forall \tau \in \pi\right\} \textit{ ordered by presence in the task table} \qquad (3.1)$$

Additionally, the schedule always cycles at the hyperperiod such that, if the tick interval is represented as $T \in \mathbb{N}^*$, then $H(\pi) = hT, h \in \mathbb{N}^*$ and $\forall \tau \in \pi: \phi(\tau) = s(\tau) * T$; $P(\tau) = p(\tau) * T$; $s(\tau) \in \mathbb{N}, p(\tau) \in \mathbb{N}^*$. The interval between ticks is hence upper bounded by the greatest common divisor (GCD) of the tasks' phases and periods as seen in Equation 3.2.

$$T \leq gcd\{\{\phi(\tau) | \tau \in \pi\} \cup \{P(\tau) | \tau \in \pi\}\} \qquad (3.2)$$

The overhead introduced by an implementation of this architecture (Katcher et al. 1993) can be accounted for by factoring in the overhead of the ISR σ_{isr} (Section 2.8), the scheduling dispatcher $\sigma_{disp}(\theta, \tau)$ for a task τ in the θ^{th} tick and the schedule creation algorithm $\sigma_{alg}(\theta)$ for the θ^{th} tick (Figure 3.2).

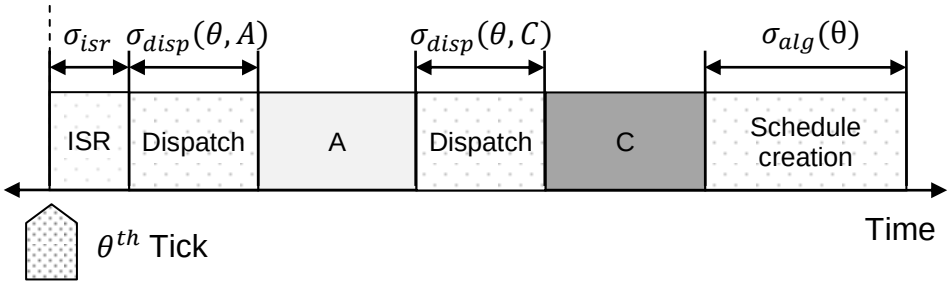

Figure 3.2: Overheads in a TTCA implementation

In an implementation with the schedule creation algorithm running concurrently in the θ^{th} tick, $\sigma_{alg}(\theta)$ is zero and it is useful to define a quantity $\gamma(\theta)$ as the amount of lag experienced by the scheduling dispatcher in sensing the θ^{th} tick compared to the schedule creation algorithm; for a schedule creation algorithm running non-concurrently, $\gamma(\theta)$ would be zero.

These overheads can be used to calculate the worst-case time to execute all the tasks released as a result of the θ^{th} tick ($\theta \in \mathbb{N}$) $WCET_{ti}(\theta)$ (Equation 3.3) and the excess time taken by that tick interval $OV_{ti}(\theta)$ (Equation 3.4).

$$WCET_{ti}(\theta) = \sigma_{isr} + \sigma_{alg}(\theta) + \sum_{\tau \in \rho(\theta)} \left(\sigma_{disp}(\theta, \tau) + WCET(\tau) \right) \tag{3.3}$$

$$OV_{ti}(\theta) = max(0, OV_{ti}(\theta - 1) + WCET_{ti}(\theta) - T) \tag{3.4}$$

The overheads can also be used in the calculation of the tasks which are to be released in the θ^{th} tick interval $\rho_r(\theta)$ and those actually started in the θ^{th} tick interval $\rho_s(\theta) \subseteq \rho_r(\theta)$ (Equation 3.5). If τ_{uf} is a task started in the θ^{th} tick

interval but finished in the $(\theta+1)^{th}$ tick interval, then $\rho_s(\theta)$ is the ordered set of highest cardinality that can satisfy Equation 3.6.

$$\rho_r(\theta) = \begin{cases} \rho(0) & \theta = 0 \\ \rho(\theta) \cup \left(\rho_r(\theta-1) - \rho_s(\theta-1)\right) & \theta > 0 \end{cases} \tag{3.5}$$

$$\sum_{\tau \in [\rho_s(\theta) - \{\tau_{uf}\}]} \left(\sigma_{disp}(\theta,\tau) + WCET(\tau)\right) \leq \left(T - OV_{ti}(\theta-1) - \sigma_{isr} - \sigma_{alg}(\theta) - \gamma(\theta)\right) \tag{3.6}$$

This thesis imposes no lower bound on the tick interval, though some designs do so to prevent any overflows, i.e. such that $OV_{ti}(\theta) = 0, 0 \leq \theta < \frac{\phi_{max} + 2H}{T}$ (Equation 3.7).

$$T \geq max\left(\left\{WCET_{ti}(\theta) \,\middle|\, 0 \leq \theta < \frac{\phi_{max} + 2H}{T}\right\}\right) \tag{3.7}$$

3.2.2 Timing event generator

In the case of single processor systems, the timing event generator is usually a timing circuit of required precision (Baker et al. 1988). In the case of multi-processor systems, a timing circuit is used either locally with network synchronisation (Kopetz et al. 1994; Kopetz et al. 2005) or globally through propagation on a common network (Pont 2001; Ayavoo et al. 2007).

3.2.3 Task design

Tasks are designed on the assumption that all their input data are available when they start and the output data are coherent when they complete. The tasks are also implemented as independent entities that never have to wait for

input/output operations and expect no resource contentions; they communicate by writing to shared memory unscrupulously.

3.2.4 Priority assignments

Figure 3.1, while demonstrating a TTCA implementation according to Table 3.1, is also an example of bad priority assignment: B has a regular period, while A and C experience oscillating ones. This can be improved by giving B a lower priority, as dictated by one of the well known scheduling algorithms for periodic pre-emptive systems (Liu et al. 1973), resulting in the new execution sequence seen in Figure 3.3 where all tasks have non-oscillating periods.

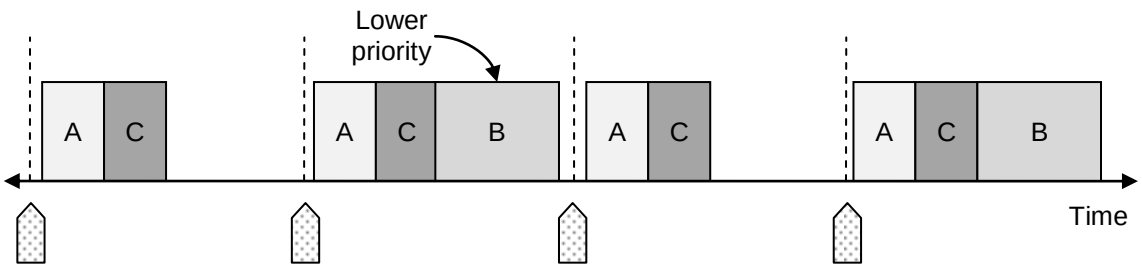

Figure 3.3: Effect of lowering the priority of task B

3.3 Feasibility

For a task under test $\tau_{ut} \in \pi$, the maximum schedule creation time σ_{alg_max} and the maximum task dispatch time σ_{disp_max}, the task-set is feasible (page 2-11) if Equations 3.2 and 3.8 hold; and if Equation 3.9 holds for $\theta = s(\tau_{ut}) + n * p(\tau_{ut})$, $n \in \mathbb{N}$, $s(\tau_{ut}) \leq \theta < \frac{\phi_{max} + 2H}{T}$.

$$\sigma_{isr} + \sigma_{alg\_max} + \sum_{\tau \in \pi} \left(\left(WCET(\tau) + \sigma_{disp\_max} \right) * \frac{1 - \frac{\phi(\tau)}{\phi_{max} + 2H}}{p(\tau)} \right) \leq T \qquad (3.8)$$

$$OV_{ti}(\theta - 1) + \theta * T + \sigma_{isr} + \sum_{\tau \in (\{\tau_{ut}\} \cup \rho_{pre}(t,\tau_{ut}))} \left(\sigma_{disp}(\theta, \tau) + WCET(\tau)\right) \leq d(\tau_{ut}) \quad (3.9)$$

Equations 3.7, 3.8 and 3.9 ensure respectively that the tick interval can meet the release times, that the processor is not overloaded and that a task's deadline is met.

3.4 Processor utilisation

Since TTCA uses the WCET in calculating the task schedule, the worst-case behaviour is made the average-case behaviour, resulting in less of the available processing power being utilised (Scheler et al. 2006). However, the availability of the hefty computational power of a development machine to calculate a schedule gives a higher chance (compared to an online algorithm) of finding a schedule that features better processor utilisation (Xu et al. 2000).

All the same, it is true that TTCA is not suitable for all applications and in those where signal frequencies vary frequently and unpredictably or where activities are mainly aperiodic or sporadic, it can result in wasting the available processor time due to over-sampling and excessive executions (Locke 1992; Davis et al. 2000). In such cases, it would be better to use alternatives (Pont 2001; Scheler et al. 2006).

3.5 Fragility

TTCA is fragile during overload situations, since a task exceeding its predicted execution time could generate a domino effect on subsequent tasks, causing schedule violations and more importantly, real-time violations (Buttazzo 2005b).

The problem is exacerbated by the increasing difficulty in calculating the WCET of a task (Puschner et al. 2002; Kirner et al. 2003). Also, since the violations occur only on overload, they can be hard to track down.

Techniques have been developed to deal with these such as the use of watchdog timers or task guardians to abort the task; however this adds greatly to scheduler complexity and can result in dangerous situations where the system could be in an unknown state (Locke 1992; Kalinsky 2001; Hughes et al. 2008). This condition is considered a fault in system design and is not explored further in this work.

3.6 Existing implementations

There are a wide range of implementations for TTCA, each emphasising a particular aspect – only a representative set is presented in the sections below.

3.6.1 The cyclic executive architecture

The cyclic executive architecture, also called the timeline scheduler (Buttazzo 2005b), was used in many applications before it was formally described (Baker et al. 1988). It has since been reused (Huang et al. 2003; Gangoiti et al. 2005), re-described (Kalinsky 2001) and criticised (Locke 1992). This architecture differs slightly from TTCA: the WCET of a sequence of tasks in a tick interval has to be smaller than the tick interval and the tick interval can be greater than the greatest common divisor. This is because it doesn’t require as strict an adherence to the release times as TTCA.

For example, for a set of tasks with WCET 1 ms, 2 ms and 3 ms and periods 14 ms, 20 ms and 22 ms respectively, a cyclic executive design can have a tick

interval of 4 ms, 5 ms or 7 ms (Baker et al. 1988) while in TTCA it would be a factor of 2 ms (Equation 3.7). A sample execution of these systems can be seen in Figure 3.4: TTCA has a tick of 2 ms and the cyclic executive is shown with the three tick intervals. The task frames are shown as shaded rectangles with a width proportional to their WCET; they run on the same execution unit but are shown at different heights for clarity. For TTCA and the cyclic executive, a task frame is released such that it is started at or after its release time in the ideal case. However, a cyclic executive requires tasks to finish execution within the tick interval and this release is sometimes delayed; this is also why the tick interval is shown as a closed rectangle for the cyclic executives.

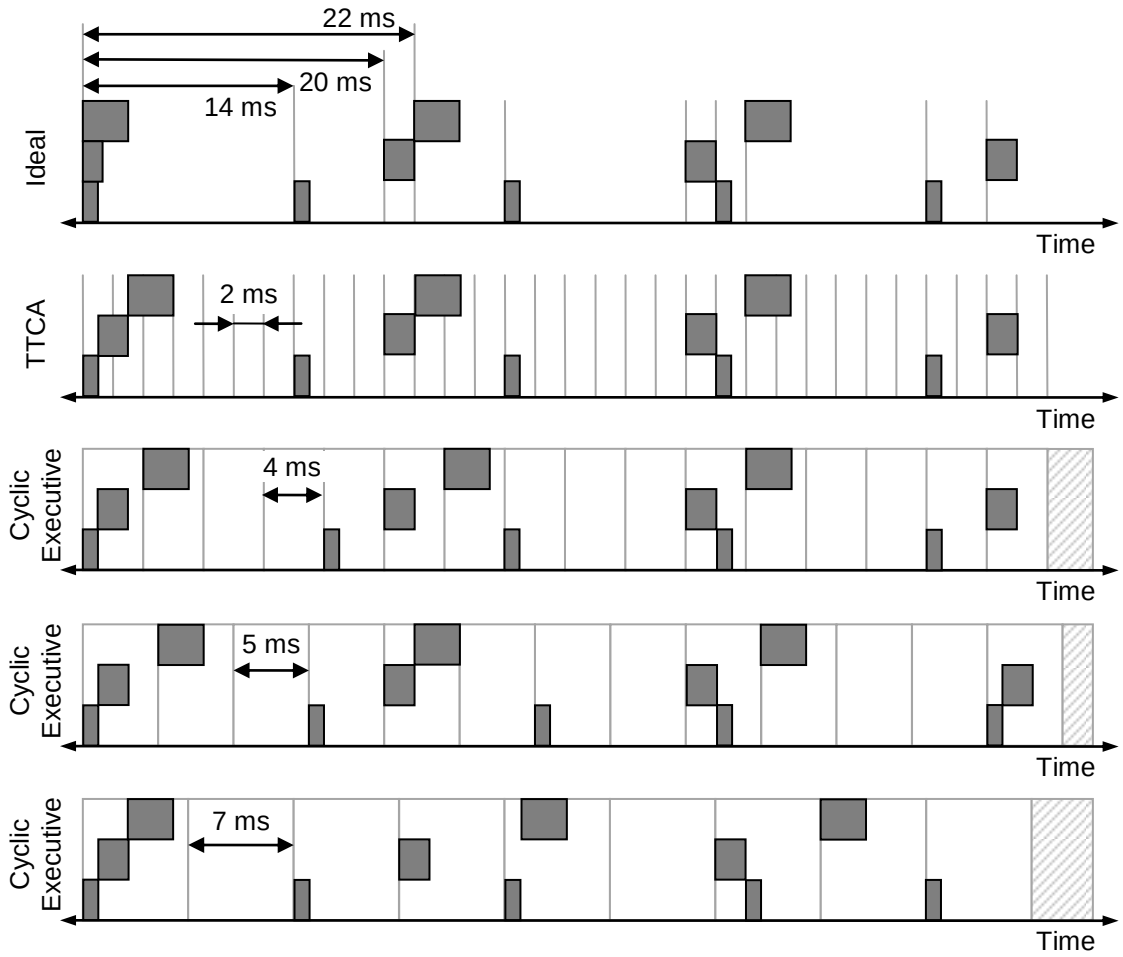

Figure 3.4: Execution of tasks with periods 14 ms, 20 ms, 22 ms, from top to bottom: the ideal case; TTCA with a tick of 2 ms; cyclic executives with tick intervals of 4 ms, 5 ms, 7 ms.

Versions of this scheduler have been written in Intel 8051 assembly (Key et al. 2003), ADA (Baker et al. 1988) and C (Pont 2001). In the single rate version of the cyclic executive architecture, every task is executed after a tick (Listing 3.2); and in the full-fledged, multi-rate version, some of the tasks might be executed at higher rates according to a statically created table indexed by the minor cycle (Listing 3.3); both versions have an ISR and schedule creation routine that do nothing useful. The extreme simplicity makes this design inherently static.

```
DEFINE dispatch OF scheduler:
   FOR EACH task:
      RUN task
```

Listing 3.2: Single-rate cycle executive dispatch

It is interesting to note that to be scheduled by the cyclic executive, the example above requires a large table size (a maximum of 257 entries for TTCA and between 220 to 385 entries for the cyclic executive).

```
DEFINE dispatch OF scheduler:
   SET minor_cycle TO (minor_cycle + 1) mod HYPERPERIOD

   CASE minor_cycle:
      WHEN 0 => RUN tasks FOR tick 0;
      WHEN 1 => RUN tasks FOR tick 1;
      …
```

Listing 3.3: Multi-rate cycle executive dispatch

When adapting this design to TTCA – which allows tasks to exceed the tick interval – the ticks that occur while tasks are still executing must be recorded (Listing 3.4).

```
DEFINE service OF interrupt:
   RAISE ticks BY 1
```

Listing 3.4: A TTCA implementation event service

The record can then be used to re-run the dispatch by guarding the call to SLEEP in Listing 3.1 as shown in Listing 3.5.

```
DEFINE Start:
   …
   DO FOREVER:
      IF ticks == 0:
         SLEEP

      LOWER ticks BY 1
      DISPATCH tasks
      CREATE schedule
   …
```

Listing 3.5: Allowing tasks to exceed the tick interval

In some literature, the cyclic executive architecture is also referred to as a "super loop" (Kurian et al. 2007; Nahas 2008), but due to implementation non-determinism, there is a significant difference: the former allows tasks to maintain a fixed *rate* (viz. the rate in the specifications) while the latter allows for a fixed *delay*. In a fixed-delay architecture, maintaining the task's frequency in the long-term is not as important as the accuracy in the short-term, for example, outputting a character to a screen as long as a key is held down. After an overload, a fixed-rate architecture might cause a task to be executed multiple times. The tasks under consideration are, by definition, fixed-rate (Section 2.6) and so the "super loop" is outside the scope of discussion.

3.6.2 Table-free multi-rate executive (TTC)

The table-driven multi-rate executive above has been criticised due to the large amount of memory required, especially as the hyperperiod increases. With a single-step simulation to create the table dynamically, a non-table driven multi-rate version (Listing 3.6) can be obtained (Kalinsky 2001; Pont 2001),

commonly referred to as the time-triggered co-operative (TTC). The tasks are still traversed in an order defined statically at compile time.

```
DEFINE dispatch OF scheduler:
   FOR EACH task:
      LOWER delay OF task BY 1

      IF delay OF task IS 0:
         RUN task
         SET delay OF task TO period OF task
```

Listing 3.6: Non-table-driven multi-rate executive dispatch

The schedule creation and dispatch which have been combined in the above routine could also be split up as in Listing 3.7 and Listing 3.8; the combination may be preferred in implementations due to the memory overheads of building another data structure for the run queue; the split-up version may be preferred since the schedule creation jitter is moved to the end of the dispatch cycle (where it can't introduce jitter into the task releases).

```
DEFINE dispatch OF scheduler:
   FOR EACH task IN run_queue:
      RUN task

   CLEAR run_queue
```

Listing 3.7: Run queue dispatch

```
DEFINE creation OF scheduler:
   FOR EACH task:
      LOWER delay OF task BY 1

      IF delay OF task IS 0:
         ADD task TO run_queue
         SET delay OF task TO period OF task
```

Listing 3.8: Table-free multi-rate executive schedule creation

3.6.3 Time-event queue

This design (Listing 3.9 and Listing 3.10) uses a time-event queue to remove the harmonic dependency in software frequencies (Hanif et al. 2008) present in the previous implementations.

```
DEFINE creation OF scheduler:
   SET minimum delay TO INF

   FOR EACH task:
      IF delay OF task = interval:
         ADD task TO run_queue
         SET delay TO period OF task
      ELSE
         LOWER delay BY interval

      IF delay OF task < minimum delay:
         SET minimum delay TO delay OF task

   IF ticks == 0:
      SET interval TO minimum delay
```

Listing 3.9: Time-event queue schedule creation

```
DEFINE dispatch OF scheduler:
   INIT timer TO generate event AFTER interval

   FOR EACH task IN run_queue:
      RUN task

   CLEAR run_queue
```

Listing 3.10: Time-event queue dispatch

Designs such as these that change the tick interval have to be given due care to prevent timing drift. The typical way is to factor the time to calculate the re-initialisation of the timer into the calculation of the interval. However this may be cumbersome or impossible due to implementation non-determinism.

3.6.4 Multiple timer interrupts (TTC-SHD)

This design achieves more precise release times when more than one task executes in a tick interval (Nahas 2008). Ticks are generated as before, but are

used to update the run queue and to initialise a second timing source (Listing 3.11) without using the modification in Listing 3.5. Contrary to what the name of the design may suggest, only two timer interrupts are enabled.

```
DEFINE service OF interrupt OF tick_timer:
   FOR EACH task:
      LOWER delay OF task BY 1

      IF delay OF task IS 0:
         ADD task TO run_queue
         SET delay OF task TO period OF task

   IF run_queue IS NOT EMPTY:
      INIT task_timer TO generate event
                      AFTER release_time OF
                      HEAD OF run_queue
```

Listing 3.11: ISR for the tick timer when using the MTI scheduler

The second source generates events within the tick interval that trigger task execution and the re-initialisation of the second timing source for the next task in the tick (Listing 3.12). This method avoids the long term timing drift in the event queue version by basing the task execution trigger events off of a source that is never reinitialised.

```
DEFINE service OF interrupt OF task_timer:
   SET task_to_dispatch TO HEAD OF run_queue
   SET run_queue TO TAIL OF run_queue

   IF run_queue IS NOT EMPTY:
      INIT task_timer TO generate event
                      AFTER release_time OF
                      HEAD OF run_queue
```

Listing 3.12: ISR for the task timer when using the MTI scheduler

The task release times are calculated as the maximum sum of the WCET of tasks that could execute before it in its tick interval, based on a simulation

performed at scheduler commencement. The schedule creation routine of this scheduler does nothing and the dispatch is minimal (Listing 3.13).

```
DEFINE dispatch OF scheduler:
   IF task_to_dispatch IS NOT NULL:
      RUN task_to_dispatch
      SET task_to_dispatch TO NULL
```

Listing 3.13: MTI scheduler dispatch

While this design violates the “one-interrupt” guideline of time-triggered design, it takes care to avoid any interrupt collisions by only generating the task interrupts within a tick interval.

This design employs a solution akin to the sandwich delay (Section 4.4.3.2). Like the sandwich delay, this design starts a timer at the beginning of a task-set to trigger at the WCET of the task. However, after the task execution, where the sandwich delays sits in a software loop polling the timer’s trigger state, this design sends the processor to sleep and uses an interrupt generated by the timer overflow to wake up the processor. Due to the delay being enforced by hardware, this scheduler is also referred to as TTC-SHD (for sandwich *hardware* delay) in this thesis to differentiate it from TTCA implementations that use the sandwich delay enforced by a software loop, e.g. TTC-SSD (for sandwich *software* delay) or TTH-SSD. As far as naming conventions are concerned, a pure hardware sandwich delay scheme for a TTCA implementation may be suffixed with “-HSD” (for hardware sandwich delay).

The design as presented also constrains all tasks from exceeding the tick interval in cumulative execution times; an interrupt from the tick timer resets the task timer (Listing 3.12). The constraint can be relaxed by reverting to the ISR in

Listing 3.4, the schedule creation in Listing 3.8 and by using the dispatch routine in Listing 3.14. However, the underlying processor still needs the ability to be triggered by at least two interrupts.

```
DEFINE dispatch OF scheduler:
   WHILE run_queue IS NOT EMPTY:
      SET task_to_dispatch TO HEAD OF run_queue
      SET run_queue TO TAIL OF run_queue

      IF run_queue IS NOT EMPTY:
         INIT task_timer TO generate event
                         AFTER release_time OF
                         HEAD OF run_queue
      RUN task_to_dispatch

      IF run_queue IS NOT EMPTY:
         DO:
            SET ticks_before TO ticks
            SLEEP
         WHILE ticks_before <> ticks
```

Listing 3.14: Adapting the MTI scheduler dispatch so that tasks can overrun ticks

3.6.5 Hardware multi-rate executive (HW-TTC)

The hardware multi-rate executive (Hughes 2009) has been modelled after the software table-free multi-rate executive (Section 3.6.2). Like the software version, its primary functionality has been split between an *update* and a *dispatch* component with a synchronous hardware FIFO as the run queue. A high level model can be seen in Figure 3.5.

The hardware scheduler is loaded with the task details in the same way as any other peripheral would receive data from the core. The update component is driven by ticks from the scheduler timer and works like the software version, decreasing a delay field for each task and then inserting a task's address into the run queue when the delay field goes to zero. The dispatch component is driven by a notification from the update component that the run queue is ready,

by the state of the run queue and by a notification from the core that a task has ended. The dispatch component stops (by requesting a switch into sleep mode) and starts (via the interrupt mechanism) code execution on the core as well as supplying the address of a task as the location at which to start code execution when the core is interrupted or when a task ends.

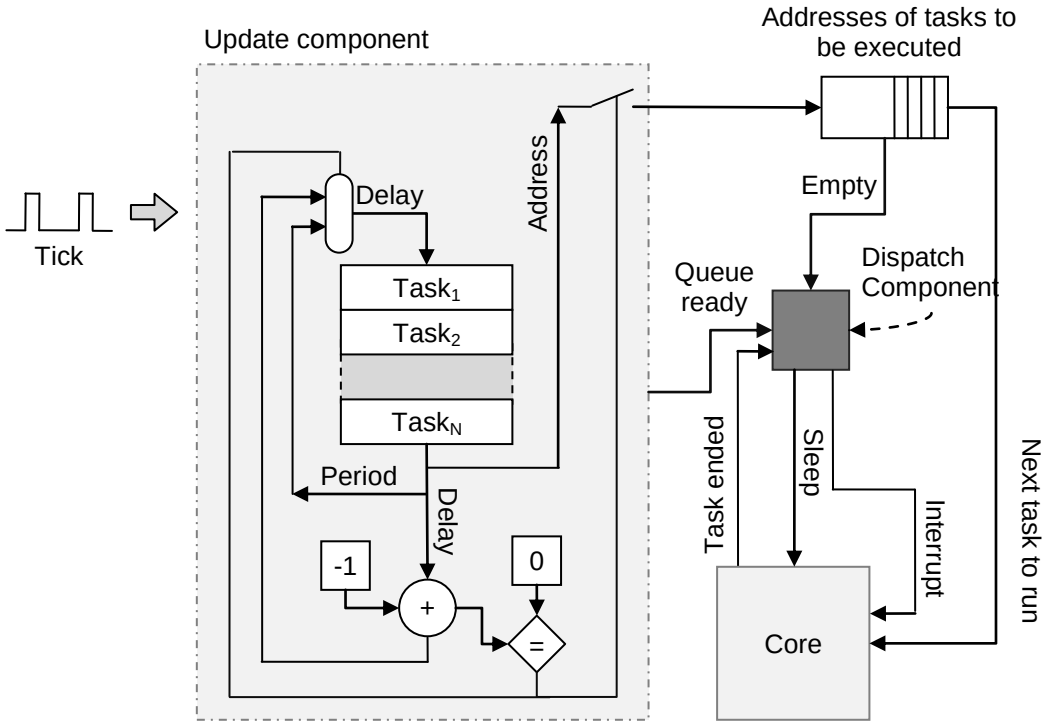

Figure 3.5: Functional overview of the hardware multi-rate executive

The address of the task that should be executed next is always maintained on the relevant signal. The interrupt signal is only used to notify a sleeping core that a task is waiting. When it finishes executing a task, the core asserts the “task ended” signal and automatically begins executing whatever instruction is at the “next task” address supplied by the dispatch component. The dispatch component uses the “task is ended” signal to dequeue a task and to put the processor to sleep when all tasks have been executed. This behaviour is illustrated in Figure 3.6.

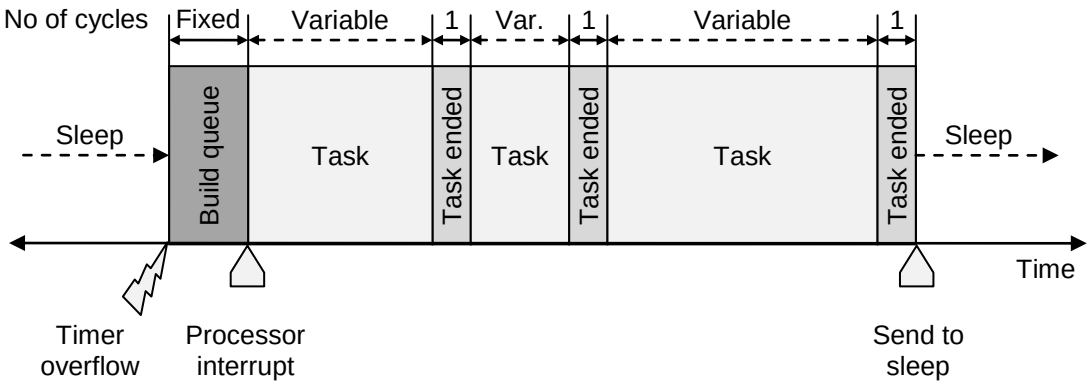

Figure 3.6: Timeline view of the HW-TTC operation

When executing the first task after a timer overflow, a fixed number of cycles are spent on iterating through the task list to find those tasks that are due to be released, on internal detection latency and on warming up the processor. Even though this initial build-up offsets the start of the first task from the timer overflow, there is *no* observable latency since it stays constant throughout the system uptime and since the timer overflows are precisely timed.

However, as the supported number of tasks increases, this initial build-up causes the notification from the update component to the dispatch component to be delayed, resulting in a larger $\gamma(\theta)$ (page 3-4) as seen in Equation 3.10; this quantity is also dependent on the number of cycles taken to signal the end of a task after fetching the last instruction of the task. Through $\rho_r(\theta)$ (Equation 3.5) and $\rho_s(\theta)$ (Equation 3.6), this increases the required length of the run queue (Equation 3.11). If the run queue is given a length less than this quantity, released tasks may never execute, resulting in deadline misses.

$$\gamma(\theta) = \frac{number\ of\ tasks + cycles\ to\ signal\ end\ of\ task}{clock\ rate\ of\ update\ component} \tag{3.10}$$

$$Length\ of\ run\ queue = max\big(\rho_r(\theta)\big), \theta \in \mathbb{N}, 0 \leq \theta < \frac{\phi_{max} + 2H}{T} \tag{3.11}$$

Special care has also been taken so that the overhead between task executions remains a constant one cycle; this one cycle is a result of the crucial “end task” instruction which notifies the core that a task has ended. However, when the task is written at a high level (Listing 3.15), this instruction is not generated by the compiler and so the task needs to be wrapped as in Listing 3.16 which adds two further cycles of overhead (Figure 3.7).

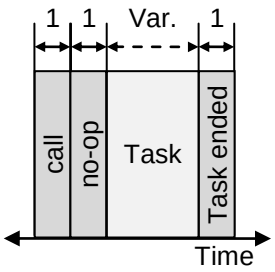

Figure 3.7: Scheduling overhead on one task

```
void Actual_Task()
   {
   // Do something
   }
```

Listing 3.15: Task definition at a high level in the C programming language

```
Called_Task:
      call Actual_Task
      no-op
      end task
```

Listing 3.16: Low level task wrapper in the hardware multi-rate executive

3.6.6 Other implementations

The TTCA designs above are table driven, the table being created either offline or online. Most often, the table entries are sorted statically based on the rate or deadline monotonic algorithms (Liu et al. 1973), however, the algorithms could be made dynamic by sorting their run queues (based on deadline, laxity, etc.) at run-time instead. The tasks could then non-pre-emptively be dispatched from the queue until it is empty. Others may also have run-time support to allow

tasks to be added (e.g. one-shot tasks) or removed (e.g. aborting scheduled releases).

Other implementations deal with orthogonal constraints: the multi-phase co-operative time-triggered design (Hanif et al. 2008) can be employed, at a loss of schedule determinism (but not, necessarily, output determinism), when dealing with a system with vastly different modes of operation (such as an airplane's taxiing, take-off, flying, etc. modes) or when sampling a signal with variable frequency; another design aims to reduce power consumption and the jitter that arises thence (Phatrapornnant et al. 2006); another reduces jitter in multi-processor systems where the timing event is propagated from another node (Nahas et al. 2004); another incorporates overrun protection mechanisms (Hughes et al. 2004; Hughes et al. 2008); another attempts to combine multiple designs to make a "perfect" implementation (Nahas 2008).

3.7 Conclusions

A TTCA implementation is fairly straightforward to compose due to the architectural simplicity and comes with very low overheads. The absence of pre-emption also means an RTOS with this architecture doesn't need any resource management interfaces. Despite this, industrial and academic research focus largely moved away from this architecture – a paradoxical shift that is explored further in the next chapter.

Chapter 4

Problems with the time-triggered co-operative architecture

4.1 Introduction

Chapter 2 mentioned the absolute imperative that real-time systems must be predictable and the way this translates to time determinism on the outputs. It went on to describe how time-triggered architectures and co-operative architectures have been used in a wide-range of safety-critical applications due to their tendency to facilitate high application reliability and predictability.

These two orthogonal architectures were then described as a combination in the time-triggered co-operative architecture (TTCA) in Chapter 3. It was further shown how some of the concerns against TTCA are based on a particularly rigid form of it, largely ignoring the alternatives available (Xu et al. 2000; Pont 2001). However, other concerns are indeed valid and legitimate problems have crippled its widespread adoption.

This chapter briefly reviews these problems, pointing out where existing solutions fail. The first section deals with the maintainability problem, followed by the problems of long-tasks, jitter and non-harmonic task-sets.

4.2 Maintainability

Maintenance carried out systematically is beneficial and adds value (Arnold 1989; Fowler et al. 1999) but can be exponentially costly if the required changes were unanticipated at design time (Griswold et al. 1993).

Under TTCA, great importance is placed on careful initial design and any change in system specifications (e.g. higher sampling rates) post-implementation requires either an exhaustive re-validation that the static schedule still holds or a recalculation of the schedule (Ramamritham et al. 1994; Liu et al. 1995; Sha et al. 2004). While an exhaustive validation is resource consuming, it reduces the probability of faults at run-time, exchanging software construction time for run-time reliability (Xu et al. 1993) and can be automated (Mwelwa et al. 2005; Kurian et al. 2007). Doing this validation at development-time also decreases the run-time overhead (Xu et al. 2000).

Any maintenance work, aided by a system's flexibility and extendibility, should either keep or increase the system reliability; for real-time systems, this means allowing resources or tasks to be added, removed or modified without causing other tasks to miss their deadlines. Under TTCA, three problems may arise from a maintenance activity: the long-task problem, an excessive increase in task jitter and the introduction of non-harmonic relationships in the task periods.

4.3 The long-task problem

Due to its non-pre-emptive nature, the long-task problem arises in TTCA when the worst-case execution time (WCET) of a group of tasks executing after a tick in the evaluation period $\phi_{max} + 2H$, exceeds the request period of one or more

tasks in the system. In the presence of the long-task problem, tasks may respond sluggishly to environmental stimuli and/or may miss their deadlines. An example is shown in the task-set in Table 4.1 and illustrated on a timeline in Figure 4.1. In the figure, task A, which has a tight deadline is prevented from starting due to task B which has a WCET greater than task A's period. As a result, every second frame of task A misses the deadline. The effect of long-task B cannot be mitigated by changing either the priorities or delays in this non-pre-emptive architecture.

Table 4.1: Task schedule with a long-task

Task	Priority	Delay (ms)	Period (ms)	WCET (μs)	Deadline (μs)
A	1	0	1	200	300
B	1	0	3	1500	20,000

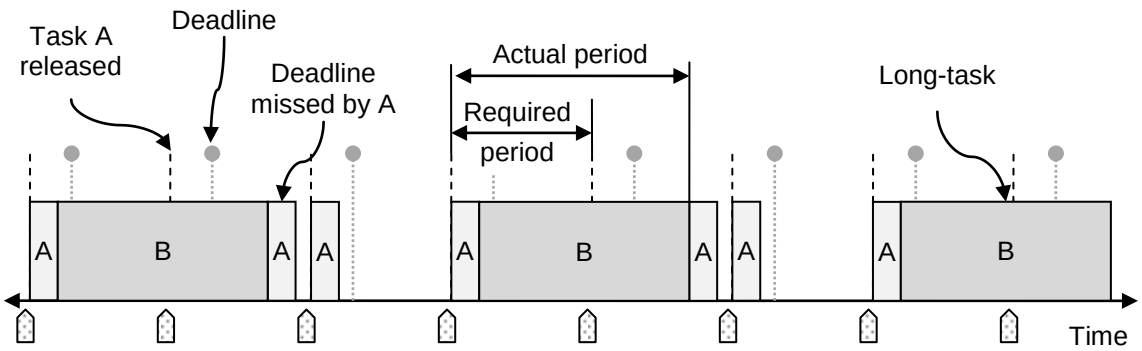

Figure 4.1: Long-task causing deadlines to be missed

This problem imposes a constraint, seen in Equation 3.9, that all tasks must have short execution times in order to improve system response times (Allworth 1981; Locke 1992) – a constraint that may be impossible to comply with.

It has been suggested that long-tasks have been over-emphasised as a disadvantage since many tasks have a small duration when compared to the smallest task period. For example, a proportional integral differential (PID)

controller can be carried out on an 8-bit 8051 processor in around 0.4 ms – fairly insignificant compared to the 10 ms sampling rate that is adequate for a flight control system (Pont 2001). However, the long-task problem can still arise if the execution time of a combined execution of tasks exceeds the period of any of the executing tasks. The next few sections examine solutions to handle long-tasks.

4.3.1 Improved hardware

Hardware improvements tend to be the first entertained solution since they are made at a very low abstraction level and well-established design, test and debug techniques can be left unchanged (Baruah 2006). This is why changes such as dynamically creating hardware (Memik et al. 2001), using better hardware (Wolf et al. 2002) or increasing the operating frequency are so favourable – software developers can continue with existing design techniques, all the while knowing that performance will scale.

Unfortunately, operating frequency increments may result in power consumption, heat generation or cooling costs averse to the requirements of an embedded design. Other improvements such as miniaturisation of silicon components and reductions in supply voltage are being impeded by increasing sub-threshold leakage currents; the laws of physics have proved to be the final, insurmountable barrier. This has spurred research into alternate materials like carbon nanotubes and graphene (Noorden 2006), but until such technologies mature, new or rehashed techniques of getting more performance from the existing silicon circuits must be pursued.

4.3.2 Improved algorithms

In addition to maintaining proper coding techniques such as avoiding potentially infinite loops (Pont 2001) and to optimising existing algorithms (Wolf et al. 2002), recent techniques have attempted to include the side-effects of scheduling into the development of the control algorithm with some success (Martí et al. 2001). Other algorithms have used feedback to change the execution times and periods of tasks dynamically (Cervin et al. 2003). This solution is effective if used from the initial design; when introduced during maintenance, however, the possibility of the task WCET changing is very high, requiring a recalculation of the WCET, schedule and control algorithm.

4.3.3 Breaking up long-tasks

A natural solution to the long-task problem is for the problematic group of tasks to relinquish control to the scheduler with sufficient regularity. It is possible to achieve this aim by interspersing the release of the tasks requiring frequent release with the release of tasks in the problematic group; however, the resultant jitter may be high enough to require another approach.

In TTCA, this may involve splitting up a task into several smaller chunks scheduled at the same rate as the original task, with incremental phases and with each chunk having a precedence constraint on the previous chunk (Pont 2001; Pont et al. 2007). In practical terms this involves moving some of the transient data (those hold information about the state) of the task from the stack into permanent storage space, increasing the amount of storage space required. Since each chunk forms a new task, the scheduler footprint also increases.

On the one hand, the split might be intuitive, especially when a timeout mechanism is being employed (e.g. waiting for a communication medium to be available between uses) (Pont 2001), but on the other, it might be subject to algorithmic constraints (Holden 2005). Other issues may also arise: an irreducible chunk might still take too long to execute, the restructuring might break the functionality, the new structure may not be suitably comprehendible or the schedulability of the system may decrease (Arnold 1989; Xu et al. 2000).

4.3.4 Pre-emptive designs

A priority-based pre-emptive design (Fidge 2002) automatically performs the split described in the section above at run-time and solves both the latency and maintainability issues.

However, these schedulers are not without their own problems as each pre-emption equates to a context switch which involves saving the state of the environment of the current task, running the new task and then restoring the saved state. This save-restore can constitute significant overhead (Locke 1992; Pont et al. 2007). Also, the start and completion times of tasks, especially low priority tasks, may be arbitrarily delayed due to pre-emption by higher priority tasks, resulting in higher jitter and unpredictability (Buttazzo 2005a). Resource contention may also add further complexity (Section 2.6.3) and latency or jitter (Buttazzo 2005b; Short et al. 2008).

To obtain the latency reduction benefits while limiting the other disadvantages, the time-triggered hybrid (TTH) pre-emptive scheduler was proposed (Pont 2001): it allows a single high priority task to be scheduled alongside one or more lower priority co-operative tasks with lengthy durations by calling the pre-

emptive task from the ISR , i.e. it is a multi-rate executive with interrupts (Kalinsky 2001). As an example, Figure 4.2 shows how the task schedule in Table 4.1 can be implemented validly with the TTH design with no deadline misses, compared to the TTCA implementation in Figure 4.1. In Figure 4.2, task A has been given the highest priority, which under TTH will result in task A starting as soon as it is released, breaking up any frames of B that are executing. As a result, every frame of task A meets its deadline even in the presence of long-task B.

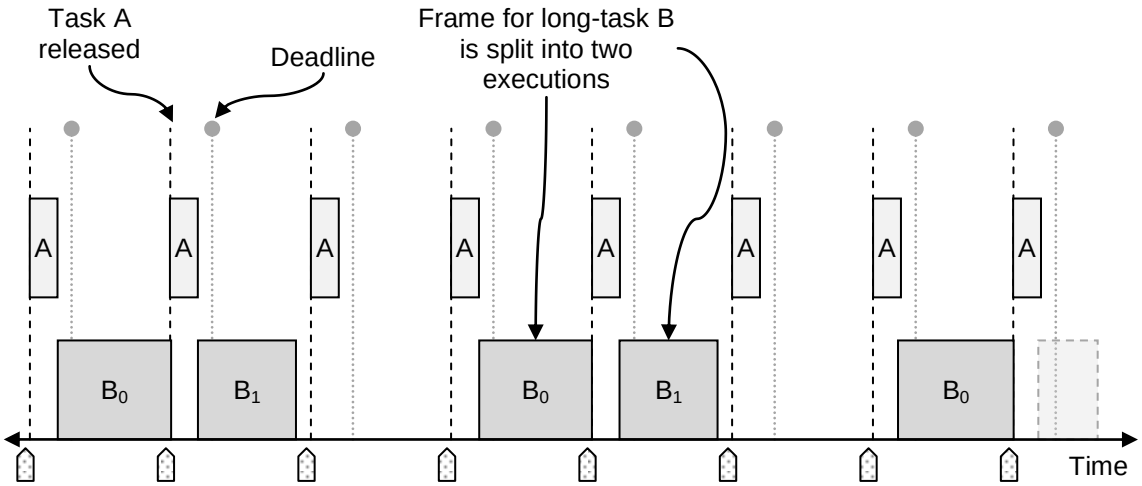

Figure 4.2: Handling a long-task with the TTH architecture

Other TTH versions exist: one reduces release jitter for the pre-emptive task by having the CPU placed in the same state (e.g. low power state) before each invocation of the pre-emptive task (Maaita et al. 2005). Another version allows more than one task to execute at the pre-emptive level (Hanif et al. 2008), using a second timer to reduce jitter for the subsequent tasks.

Another approach is to statically allocate pre-emption times (Puschner et al. 2006; Wang et al. 2008). This approach is a compile-time automation and is very similar to splitting up tasks into smaller ones (Section 4.3.3) and faces

many of the same problems such as having to maintain the pre-emption points (Locke 1992).

4.3.5 Increased concurrency

In such a design, at least one processing element will be available to respond with the required rapidity, negating the effect of the long-task. Existing architectures such as TTCA may be adapted to multiple processor designs (Pont 2001; Ayavoo et al. 2007) to maintain predictability. A long-standing complaint about these systems has been the high volume and weight of just the wiring required to connect processors (Leen et al. 2002); a complaint that is being taken care of by new networks such as the novel data network in the Airbus A380 (Brajou et al. 2004); or, where practical, by integrating processors onto a single-chip in the form of a multi-core (Obermaisser et al. 2009).

4.4 Task jitter

There are a variety of sources of jitter in tasks running under TTCA as outlined in Section 2.6.4. This jitter can also accumulate across the co-operative execution after a timing event turning otherwise miniscule variations into catastrophic fluctuations.

One of the factors affecting the release jitter is that for every set of tasks released at the same time, only the first one executed may have a precise hardware determined start time. The start time of the others will depend on the finishing jitter of all preceding tasks (Kalinsky 2001), including the ISR.

In Figure 4.3, task A and task C have the same period as the tick interval, with task A having a higher priority; as a result, task A is executed immediately after

the tick with task C following immediately. However, task A also has non-zero execution jitter (Section 2.6.4.1) due to varying execution time resulting in task C having a variable period and exhibiting release jitter (Section 2.6.4.4) due to varying release times.

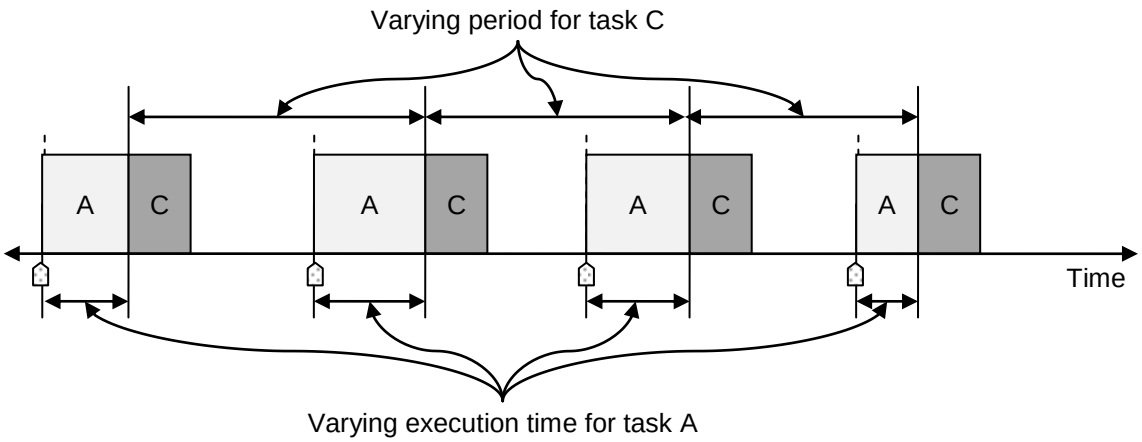

Figure 4.3: Release jitter caused by execution jitter in a preceding task

Release jitter may also be caused by a variation in the time the trigger event takes to interrupt the processor (Thiele et al. 2004), mainly due to the underlying instruction set architecture of the processor where instructions may have different execution times.

It is also possible that a task may be released with very low jitter, but due to execution jitter, the actual jitter sensitive portion of the task may experience a higher jitter. For example, Figure 4.4 shows a task A with a portion of execution in a darker shade; this portion requires that it be started at the same time relative to the same portion in the previous frame. However, this requirement cannot be satisfied due to the preceding portions of task A, even though task A has a low release jitter.

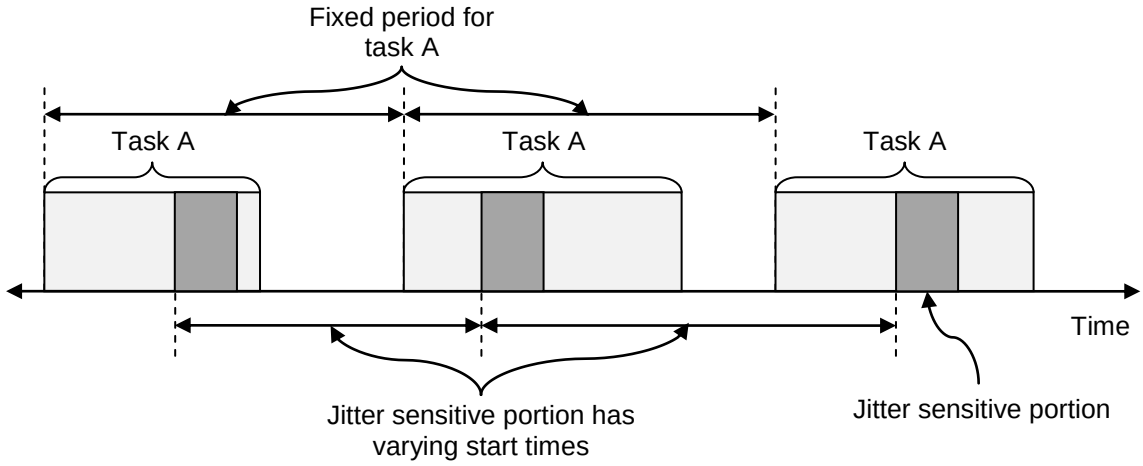

Figure 4.4: High execution jitter may cause high release jitter in a portion of a task which otherwise has low release jitter

Jitter may also arise within the TTCA implementation from the interrupt service routine, the scheduling algorithm and the dispatcher. All of these can be considered to be implicit tasks, out of which, the scheduling algorithm alone can be moved to the end of the schedule to minimise its effect. On the other hand, in a purely static system, the scheduling algorithm doesn't run at all, and the dispatcher can be designed to execute the same number of instructions before running one task, though not necessarily the same number for all tasks. Since these functions are implicit tasks, they exhibit the same types of jitter enumerated in Section 2.6.4. The next few sections examine solutions for tackling jitter.

4.4.1 Improved algorithms

The improved control techniques mentioned in Section 4.3.2 aimed at reducing latency, do the same to jitter. Some other work has explored finding upper bounds on the output jitter of a task; and reducing jitter by adjusting task phases using simulated annealing and by adjusting relative deadlines (Baruah et al. 1999; Buttazzo et al. 2007). Algorithms have also been devised to tackle jitter

caused as a side-effect of error-protection schemes in networks (Nahas et al. 2004).

4.4.2 Task properties

Section 3.2.4 demonstrated the effect a priority assignment can have on maintaining a steady period (Figure 3.1 vs. Figure 3.3), choosing to assign higher priorities to tasks with higher rates of execution. In a similar manner, the issue of execution jitter for tasks following the first one after a tick can be solved by increasing the tick interval and changing the phases (Tindell 1994). For example, the problem in Figure 4.3 where a varying execution time in task A caused release jitter in task C can be solved by halving the tick interval and giving task C a phase of one tick as shown in Figure 4.5.

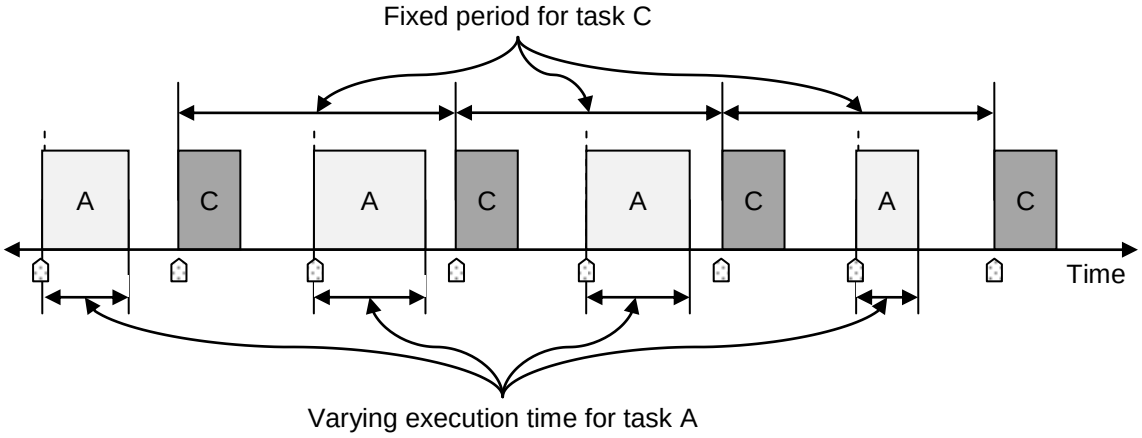

Figure 4.5: Effect of phases and an increase in tick rates on release jitter caused by execution jitter in a previous task

However, this technique results in increased power consumption due to more frequent ticks and imposes a harsher limit on the worst-case execution time of the tasks by shortening the tick interval. For example, in Figure 4.3, task A and task C could have a combined worst-case execution time less than the tick interval, whereas the changes in Figure 4.5 require each to finish in half the time to avoid increasing the release jitter in the other. A similar technique of

changing the phases and priorities has been used in pre-emptive periodic systems (Cottet et al. 1999).

In data acquisition applications, where jitter can lead to signal distortion, the task that samples the signal can be released at a higher rate. However this is often by a factor of five or ten and leads to an unfavourable increase in processor utilisation (Cottet et al. 1999).

4.4.3 Utilising spare computational capacity

This class of methods makes use of any computational capacity that cannot be used by the scheduling algorithm. With TTCA, two methods have been tried: single path programming and balancing with delays.

4.4.3.1 Single path programming

The single-path programming paradigm aims to produce software with a constant execution time, for an execution time jitter that is purely dependent on the underlying hardware and not on the software design (Puschner et al. 2002). However the technique, which has spawned a processor (Schoeberl et al. 2009), requires predicated instruction support and can increase power consumption (Gendy et al. 2007b).

4.4.3.2 Code balancing with delays

Inserted idle time or delays or timeouts can be used to establish certain guarantees about the execution time of a code segment. Accurate delays may be created by starting a dedicated hardware timer and looping or idling until the timer overflows. The dedicated timer may also be used and accessed via special instructions. Software loops generate more jitter than hardware loops since they cannot react as quickly to timer overflows.

This mechanism has been used by the precision timed architecture (Lickly et al. 2008) to guarantee a lower bound on the execution time, giving inputs time to settle (Figure 4.6 (a)). The sandwich delay algorithm uses a software-controlled timer to create delays that make the average execution time of a task the same as the worst-case (Phatrapornnant et al. 2006; Gendy et al. 2007b; Das et al. 2009), as in Figure 4.6 (b), so that succeeding code always starts at the same time. A version of this algorithm that provides more accurate delays and that decreases power consumption has also been devised (Section 3.6.4). This version ensures constant interrupt overheads, reducing release jitter, but still necessitates a second mechanism to generate accurate timing events and is susceptible to overheads and jitter in the ISR and in the scheduler dispatch. There is also a need to factor the time required to setup the timer into the calculation which can be error prone.

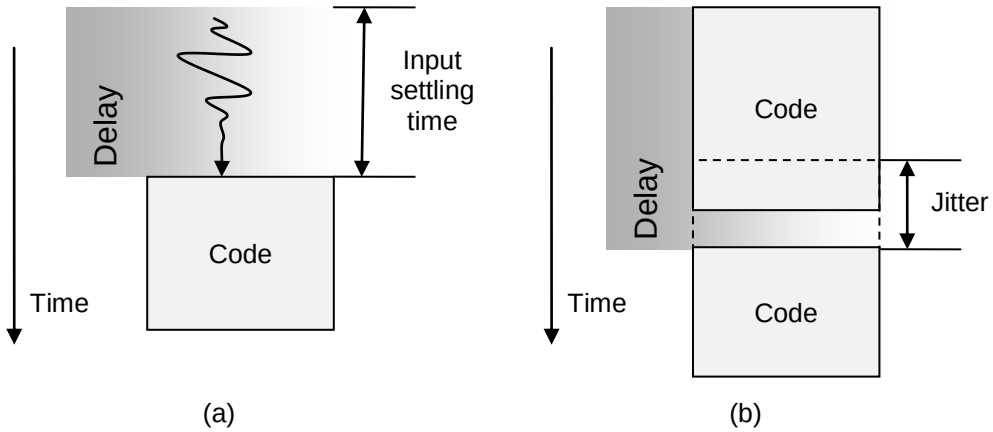

Figure 4.6: Using delays to place guarantees on the execution time

For pre-emptive, on-line systems, the delay method was criticised as the delay is wasted idle time (Baruah et al. 1999). The solution there used processor utilisation and deadlines to increase the priorities of tasks needing low output jitter. But the effectiveness of the solution degraded as the number of tasks requiring low jitter was increased. In comparison, static TTCA implementations

are designed for worst-case scenarios, and hence, the extra inserted time is already accounted for in the scheduling algorithm – even without delays the extra time would be wasted, but at the end of the execution sequence instead.

4.4.4 Jitter sensitive code inside a task

Jitter inside a task can be straightforwardly tackled by breaking up the task (bearing in mind the concerns of Section 4.3.3) so that the jitter sensitive portion constitutes a new task and changing the task properties as required (Buttazzo et al. 2007). For example, the problem in Figure 4.4 where the sensitive portion of a task A experienced high jitter even though the task itself had low release jitter, can be solved by: splitting task A up into tasks A_0 and A_1 with the same period such that the sensitive portion is at the beginning of task A_1, halving the tick interval and giving task A_1 a phase of one tick (Figure 4.7). As can be seen, this is a trade-off between reducing jitter and increasing latency.

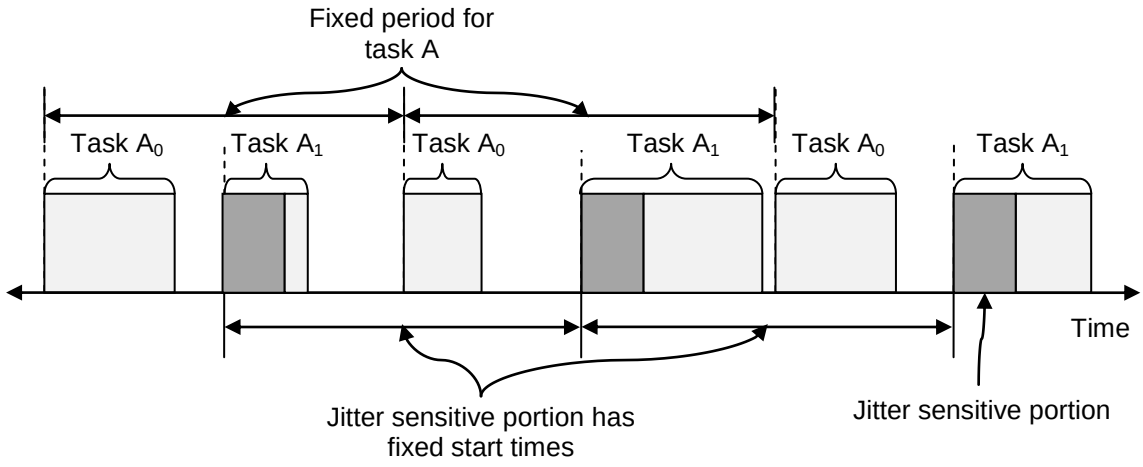

Figure 4.7: Creating new tasks to handle a jitter-sensitive portion inside a task

Alternatives can be used either to prevent the problem from occurring in the first place (Section 4.4.3.1) or to balance the first portion of the task (Phatrapornnant et al. 2006) (Section 4.4.3.2).

4.5 Non-harmonic task-sets

Non-harmonic task-sets are those in which the greatest common factor of all task periods is less than the period of the fastest executing task in the set. Such task-sets can cause wide swings in release jitter as tasks will be released in varying combinations over the hyperperiod – an example can be seen in Figure 4.8 which has two tasks, A and C, with periods two and three tick units respectively. While TTCA is able to schedule these two tasks (over time, the average period of C will converge to the ideal), the jitter in C would be largely dependent on tasks with non-harmonically related periods.

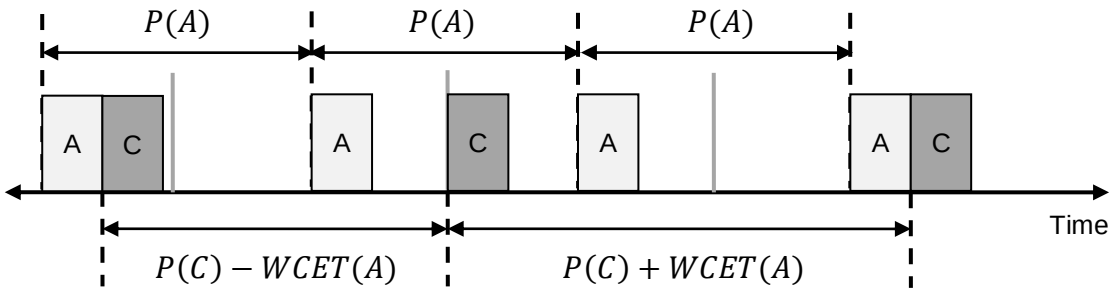

Figure 4.8: Release jitter caused by non-harmonic periods

The sandwich delay or the TTC-SHD scheduler could be used to handle the release jitter caused by non-harmonic periods (Figure 4.8), as seen in Figure 4.9 where C has been delayed. However, the WCET of C which could previously tend to T, has now to be limited to $T - WCET(A)$ to avoid causing jitter in A. This constraint arises because the delay is no longer making use of slack time, but is inserting a new task in the form of a delay into the system. Not only does this cause a decrease in the available computational power, as shown, but it may interfere with the schedulability of the system.

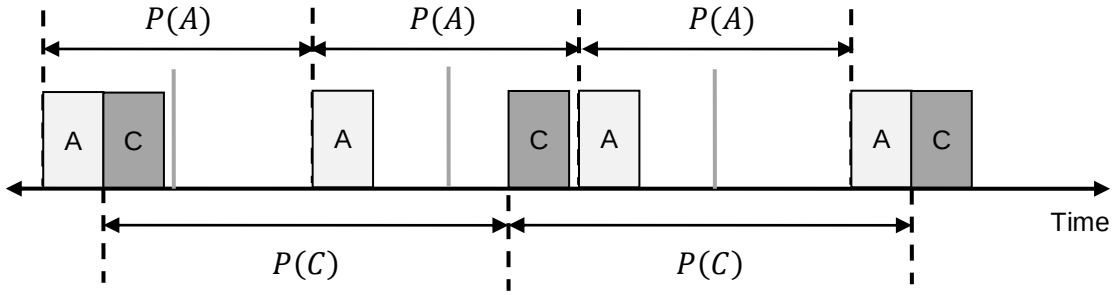

Figure 4.9: Attempting to reduce release jitter caused by non-harmonic periods by inserting delays

Attempts to reduce jitter in non-harmonic tasks sets generally employ some form of multiprocessing and the clustering of tasks on the basis of harmonic relationships between their periods (Abdelzaher et al. 2000; Ekelin et al. 2001).

4.6 Conclusions

Long-tasks introduce unnecessary latency into high frequency tasks, degrading the responsiveness of a time-triggered co-operative system. Existing solutions of improved hardware or algorithms, of breaking up tasks and of pre-emptive designs are either not feasible or introduce complexity into the systems. The solution of using multi-processors is the most attractive, particularly the option of a CMP or multi-core (Section 2.4). However, with this option, software design may still be unduly complicated by inter-task communication requirements.

Likewise, task jitter degrades the output determinism, increasing the unreliability of the system. The solutions of using improved algorithms and changing task properties again introduce the possibility of producing unfeasible or complex designs. Single path programming is also not very attractive due to the requirement for certain types of processors. On the other hand, code balancing techniques like the sandwich delay have been used in TTCA implementations before and make use of pre-allocated slack time and are, hence, attractive.

However, they impose additional scheduling overhead and are unable to completely reduce the jitter due to the presence of a software element.

The third problem of non-harmonic task-sets is more serious and can be seen from the mathematical model (Section 2.6.1) as unsolvable on single-processor systems. For this reason, the sole solution has been to use multiple processors. Fortunately, this is already the most attractive solution for tackling long-tasks and so it may be possible to solve both the long-task and non-harmonic task-set problems by pursuing a multi-core system.

Chapter 5

Increasing the concurrency in single-processor TTCA designs

5.1 Introduction

TTCA provides a highly predictable method of designing embedded systems. It is highly beneficial for safety-critical systems to use this architecture to schedule a system. Once deployed, however, maintenance efforts might require certain modifications or additions that can have detrimental effects on the real-timeliness of the system.

One of these modifications is the introduction of the long-task problem i.e. a group of tasks with an execution time larger than the period of the task that executes with the highest frequency. The long-task problem decreases the system responsiveness and increases the amount of output jitter.

Another modification is the creation of non-harmonic task-sets, i.e. sets wherein tasks have periods that are not exact multiples of each other. This leads to greater release jitter in all tasks of the system, affecting the system behaviour, and possibly even corrupting sampled signals.

This chapter explores the alleviation of the long-task and non-harmonic task problems by increasing the concurrency of the execution path with hardware extensions. The first section explores the areas where such extensions may be made before settling on a multi-core design and the requirements from such a

design. The subsequent section examines suitable multi-cores, ultimately choosing the PH core (Hughes 2009). The next few sections examine the design of a multi-core PH system for TTCA including inter-task communication, scheduler design and a predictable initialisation sequence.

5.2 Design choices

A high level view of the execution path of a single-core processor is shown in Figure 5.1: instructions are read from memory, decoded and then the operation indicated by the instruction is performed. The operation is usually performed with the help of computational units like arithmetic-logic-units, multiplier-dividers, etc. The operands for the operation are fetched from the register bank or from memory. The memory may also defer access to peripherals which may connect to the environment.

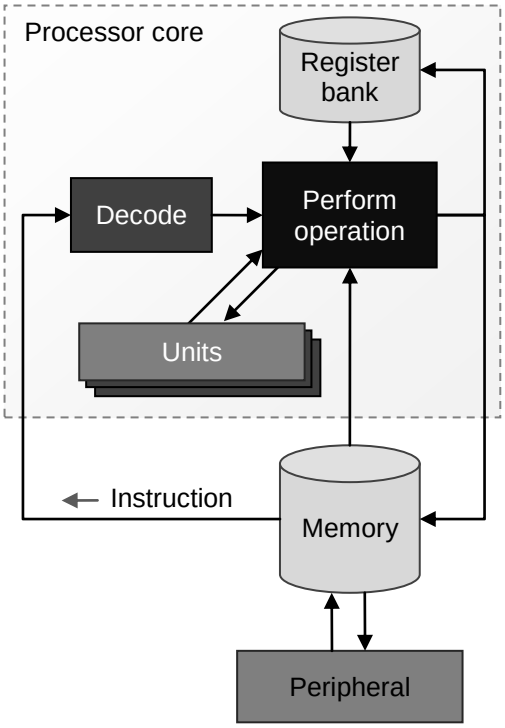

Figure 5.1: A generic single processor design

5.2.1 Increasing concurrency

Due to the single execution path and the non-pre-emptive nature of the TTCA, tasks with high execution times decrease the rate at which other software can be executed, decreasing response times. This interference can be removed by allowing tasks to execute concurrently with other tasks. Since the software architecture is considered fixed in this thesis (Section 1.5), hardware techniques

must be employed either by sharing the execution path (pseudo-concurrency) or by duplicating it. Sharing the execution path via hardware (temporal multitasking) is functionally equivalent to sharing it via software (pre-emptive architectures), may similarly give rise to execution jitter and is not considered further.

The alternative of duplicating the execution path can be carried out with a complete or with a partial duplication of the coprocessors, leading to chip level multi-processors (CMPs) (Section 2.4) and simultaneous multithreading (SMT) designs respectively.

SMT designs are superscalar in nature, i.e. they execute more than one instruction at the same time; their speciality is that the concurrently executing instructions come from different tasks. These designs are aimed at ensuring that a core is never needlessly idling because of a co-processor that might be held up. Nevertheless, due to only a partial duplication of co-processors, they cannot run all operations in parallel, causing execution jitter and, hence, are not considered further.

This leaves the CMP as the last option, though not without an abundance of design choices: the processors in a CMP may be identical (homogeneous) or may differ in the functional units or operating frequency (heterogeneous); they may also share, through caches, the memory used for instructions and data (symmetric) or maintain separate memory banks (asymmetric); and inter processor communication may either be done through shared memory, if present, or through a dedicated on-chip network taking various topologies such as meshes, hypercubes, etc.

5.2.2 Inter core communication

With an increase in the number of cores in the system, a mechanism is necessitated for tasks on different cores to communicate. Moreover, since this work aims to maintain the simplicity that TTCA lends to application design, such a mechanism must be transparent to the application designer; it may however be spread across the RTOS and the hardware. It must also avoid interfering with the timing of applications, i.e. it must run asynchronously to the task.

In a single-processor TTCA implementation, tasks communicate by reading and writing to common memory locations (Section 3.2.3). In the discussion below, the core that is executing a task writing to a common memory location is termed a *writer* and the core executing a task reading from this location is termed a *reader.*

The assumption made of the use of TTCA allows for a specialisation: since the periodicity of the applications is a part of the application design, it can be assumed that the writer will buffer data appropriately at the application level if it runs faster than the reader. For example, in Figure 5.2 (a), the application task performing the write will only use one buffer while in Figure 5.2 (b), the writer will create and use two buffers at the application level.

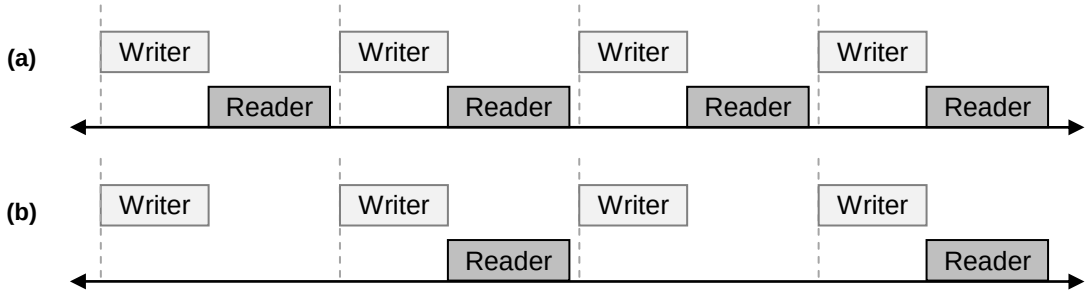

Figure 5.2: Reader and writer tasks running at different rates

However, when executed on a multi-core a scenario unanticipated by the co-operative application may occur: the writer may execute concurrently with the reader and so while the reader is reading the shared memory area, a concurrent execution of the writer may modify the memory area leading to incoherent data. This is the result of the cases in Figure 5.3 (a) & (b) where task frames occur within each other, Figure 5.3 (c) where the writer is started while the reader is executing and Figure 5.3 (d) where the reader is started while the writer is executing. For a multi-core TTCA, such overlap is permissible but requires special measures to maintain coherence.

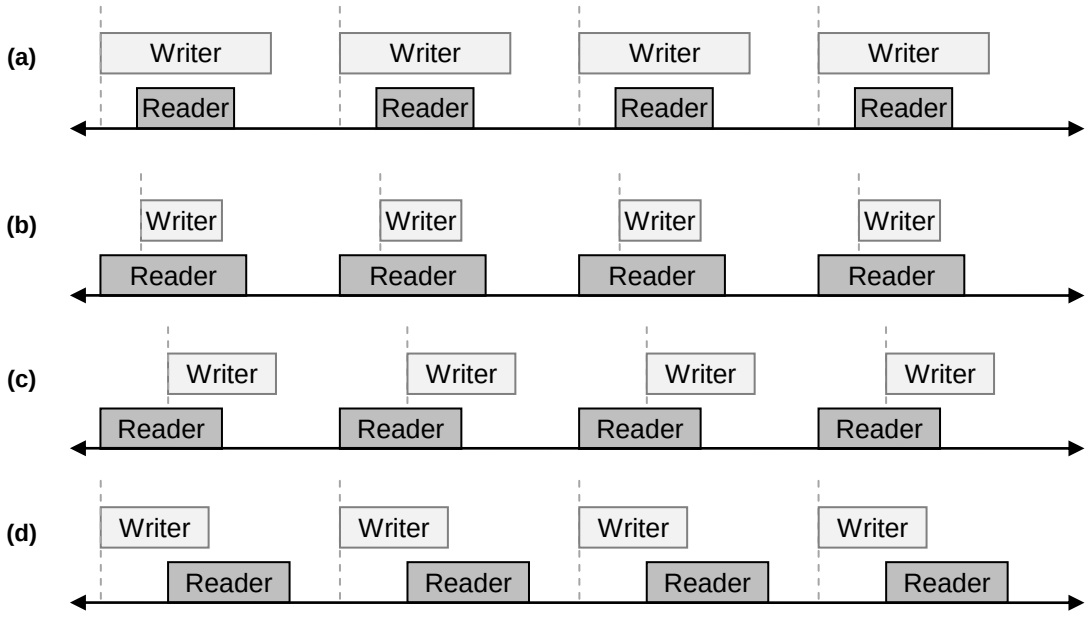

Figure 5.3: Possible overlaps between a writer and a reader (Kopetz et al. 1993)

5.2.3 Constraints

The work described in this thesis considers the application software to be non-modifiable and changes to be permissible only in the RTOS or in the hardware. This restriction imposes a couple of constraints:

- *The hardware for the study must be easily modifiable*: To this end, a programmable logic device, specifically, a field programmable gate array (FGPA) is employed. FPGA technology, which is used to prototype the production-costly application specific integrated circuits (ASICs), has matured to the extent that FPGAs can be deployed in the field as cost-effective replacements to ASICs (Rodriguez-Andina et al. 2007). An FPGA can be developed with a schematic or textually with a hardware description language (HDL) such as Verilog or VHDL.
- *The complete hardware must be modifiable*: To avoid having the application software change it is necessary to do a lot of snooping on existing control lines and to be able to control the execution paths of the software processors. For these reasons, it was imperative that the whole of the source of the design be readily modifiable. This subsequently implies that a soft-core (i.e. not tied to a particular FPGA) must be used, over both a commercial off-the-shelf (COTS) and a hard-core processor (i.e. exactly fitted to a particular FPGA).

Additionally, the desired application area encompasses real-time systems for which predictability is the foremost goal. Thus, any unpredictability or non-determinism and latency introduced into the system by the techniques developed must be easily identifiable. To this end, external contributors of unnecessary complexity, non-determinism and latency must be minimised as much as possible. Though highly restrictive, such a system can be highly beneficial for predictable real-time systems. These requirements led to the following desirable features for the soft multi-core:

- *The cores must process a single instruction at a time*: This is to remove the complexity and non-determinism inherent in the superscalar approach. The very long instruction word architecture is an exception to this since it is a static approach; but, it is aimed at concurrency *within* the task and is highly unsuitable for concurrency amongst tasks.
- *Cores must not compete for any resources*: To match TTCA, this can be accomplished by not sharing peripherals amongst the cores.
- *Memory latency must be constant and guaranteed*: This is accomplished by each core having its own memory banks, by avoiding any caching techniques and by using on-chip memory clocked at the same rate as the core. Also, the memory architecture must help avoid the non-determinism of structural hazards, e.g. the Harvard architecture where separate instruction and data memories are employed.
- *Instruction execution times must be constant and guaranteed*: While instructions may take different amounts of time to complete, the same instruction must always take the same amount of time to complete regardless of the value of its operands.
- *The inter-core communication network must be point-to-point*: This is to avoid the latency and variations in packet or circuit switching and in message routing. It should be noted that point-to-point network topologies are costly and hence generally avoided; however, their use does not preclude a future move to a more cost-effective topology.
- *Tasks will be statically allocated to cores*: This technique is also referred to as asymmetric multiprocessing (AMP) and is used to avoid the non-determinism involved with task migration. Asymmetric processing is an

ideal fit for embedded applications where although symmetric multiprocessing (SMP) allows all the required operations to be performed, they may not be done as efficiently or in as few cycles as under the former (Leibson 2007; Guerin et al. 2009).

5.3 Selecting a soft multi-core processor

Because of the ubiquitous nature of embedded computing, there are a wide variety of application classes and a multi-core to match nearly every one. In keeping with the design constraints and desirable features, only soft multi-cores that are asymmetric and available with completely free-to-use source which can be synthesised for an FPGA will be examined. Due to the relative simplicity that the desirable features put on the multi-core (AMP & point-to-point links), soft cores without multi-core designs and soft multi-cores with unsuitable designs will also be examined with the view that a suitable multi-core design may be easily created.

5.3.1 Existing soft multi-cores

There are two soft multi-cores families: LEON (Gaisler Research) and OpenSPARC (Sun Microsystems). The LEON family has seen four versions, the latest having been released earlier this year; the multi-core version uses a shared memory architecture, but can have caches disabled. The SMT multi-core OpenSPARC family similarly also uses a shared memory architecture and is an open-sourced version of the existing UltraSPARC family; it has been released with designs fixed at eight-cores with caches enabled. An independent initiative has used the source for the OpenSPARC T1 to extract what is called

the S1 core and it has been released in versions with and without multithreading and caches.

As mentioned, the multi-core versions of these families use a shared memory architecture and as such are unsuitable. On the other hand, it is possible to extract the soft-cores and thus treat them like the other soft-cores that do not have multi-core platforms.

5.3.2 Soft-cores with no multi-core platforms

This group consists of the OpenFire, AEMB family (Aeste Works), PacoBlaze, ZPU (Zylin), OpenRisc (OpenCores community), JOP and PH (TTE Systems). There are others like the Freedom CPU (F-CPU), a high performance microprocessor which has not seen any development since 2004; its more active spin-off, YASEP (Yet Another Small Embedded Processor) which sees active development but is incomplete; and Lattice's Mico32 which is unsuitable due to its varying instruction execution times.

Among the aforementioned, OpenFire and PacoBlaze are open-source clones of the commercial Xilinx MicroBlaze (closed-source) and PicoBlaze (source available on purchase) soft-cores respectively, both seeing their last updates in 2007; OpenFire is still not fully feature-compatible. The recently updated AEMB family started out with sharing the MicroBlaze instruction set but the latest soft-core, AEMB2 is SMT and only has a subset of the original instruction set. Then again, the original scalar AEMB1 fits the requirements as does the OpenRISC 1200 which implements the OpenRISC 1000 architectural description and Zylin's soft CPU (ZPU) which is touted for its small size and is marketed as a co-processor to FPGA operations rather than as a COTS replacement.

As seen above, there is a wide range of suitable soft-cores available, some catering to the general application space, some catering to a specific need. Alongside these, the Java Optimised Processor (JOP) and PH are soft-cores that also fit the requirements, but have the advantage of being built from scratch to be highly predictable. The Precision Time (PRET) Machine is also designed for timing predictability and repeatability; however, it has only seen a cycle-accurate simulator with several examples and a soft-core implementation is in progress (Lickly et al. 2008).

The JOP has been placed into a multi-core design where a cyclic executive software design, single-path programming and synchronisation with a time-sliced access mechanism to shared memory allow for execution determinism (Schoeberl et al. 2009). This system appears to offer the ideal platform; however, it is not clear how transactional memory access is ensured without keeping within the extremely small allotted time slice or by using Java's blocking synchronisation mechanism.

The PH soft-core doesn't offer facilities for single-path programming but has the advantage of being designed specifically for time-triggered applications. It has been built for timing-determinism and has some desirable extensions that deal with time-triggered issues. For these reasons, it is the soft-core of choice for this work.

5.3.3 The PH core

The PH core is a 32-bit "research" version of the commercial TTE®32 core present in the TTE32-SM3 microcontroller (TTE Systems 2010). It was first described in (Hughes et al. 2005) with improvements presented in (Athaide et

al. 2007) and (Hughes 2009). It is a cut-down version of a R2000 core (Kane 1987) which is compatible with the MIPS® I instruction set. It possesses a 32-bit Harvard-architecture with 32 registers, a five-stage pipeline and support for precise exceptions.

The core has been designed specifically for time-triggered applications: it bakes the general time-triggered design guideline, “only one interrupt”, into its design and guarantees memory latency and instruction execution times. It has been extended to ensure constant interrupt overhead (especially for multi-cycle operations) and to incorporate a TTCA hardware implementation and a task guardian (Hughes 2009). The task guardian extension is irrelevant for this work and will not be described in detail.

Various platform designs for the PH soft-core have been implemented in VHDL and were originally targeted at the Xilinx Spartan 3 FPGA on the Digilent Spartan 3 starter kit (Digilent Inc. 2004). They have since been ported to the Altera Cyclone® II on the Altera DE2-70 development board and, in the course of this work, to the Xilinx Virtex 5 LXT on the Xilinx ML505 development board.

5.3.3.1 Microcontroller block diagram

Being a soft core, the PH processor core can be incorporated with any number of custom hardware components to create an FPGA-based microcontroller or system-on-chip or platform. A block diagram of one such platform can be seen in Figure 5.4.

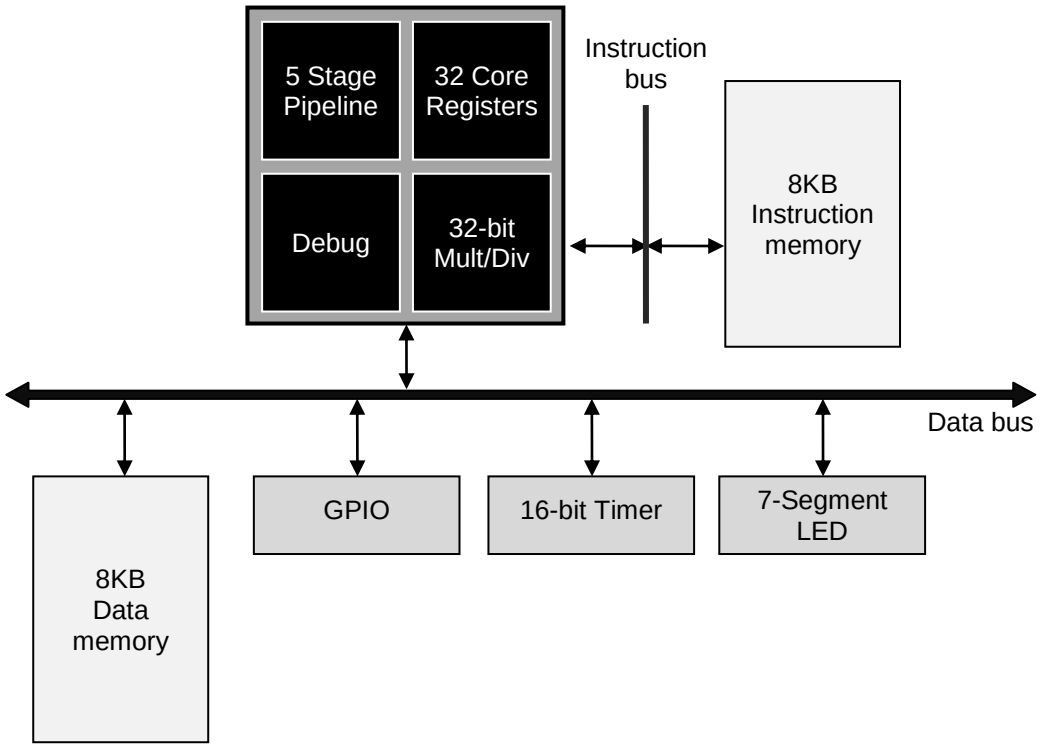

Figure 5.4: PH processor implementation (Hughes 2009)

5.3.3.2 A single interrupt

Many commercial off-the-shelf (COTS) processors support a wide range of event sources, all of which might interrupt the processor execution. However, the use of a (pure) time-triggered software architecture generally requires that only a single interrupt be enabled (Pont 2001). While this might be enforceable through conventions at the design stage, it is possible that a subsequent maintenance or upgrade might fail to check against the conventions, introducing unreliable behaviour.

The PH core is designed against this, so that out of the many event sources, it is impossible for software to enable more than one as capable of interrupting the processor. The events may still set flags that can be checked and cleared by polling.

5.3.3.3 Guaranteed instruction execution times

The PH core processes an instruction in five stages: read from memory, decode, perform requested calculations, access data memory and modify registers. To avoid resource wastage and to increase computational speed, instructions are processed in parallel with all five stages kept occupied (Figure 5.5).

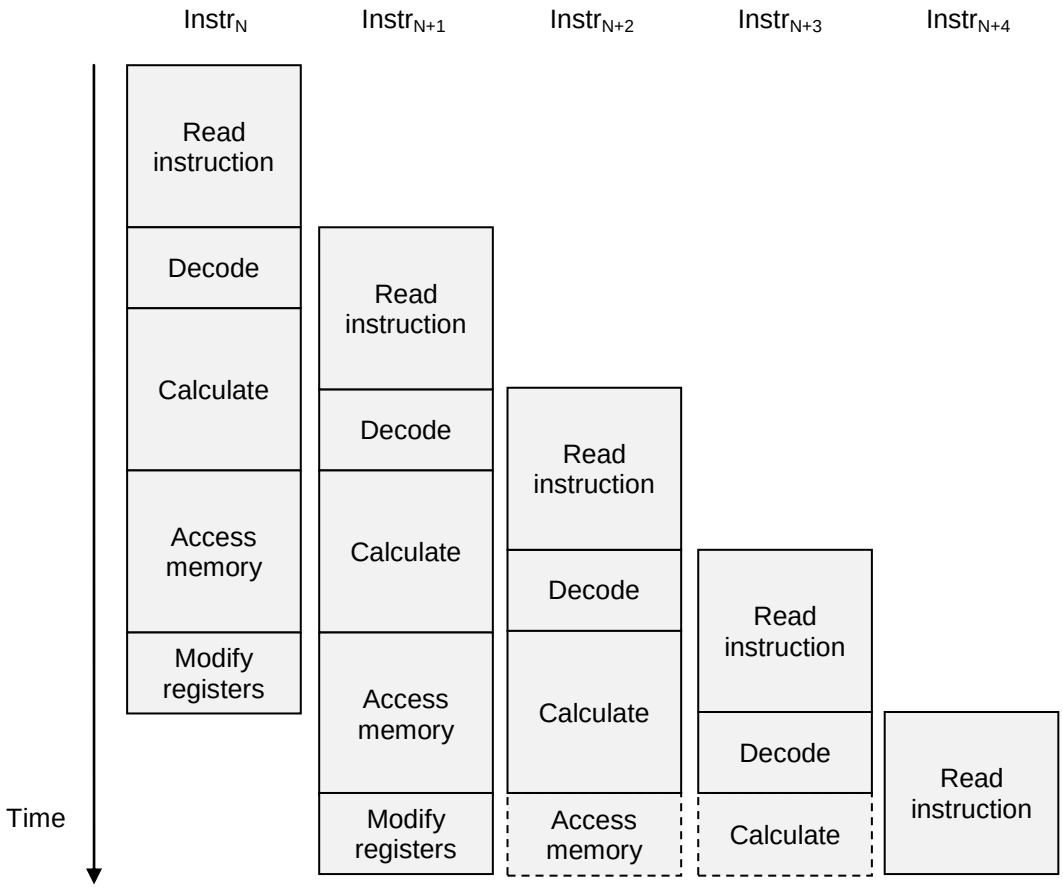

Figure 5.5: Five stage pipeline in the PH core

An instruction goes through every one of the stages and requires a fixed number of cycles to be processed, even if the operands allow for optimisations. The core also relies on the compiler to insert suitable instructions after a branch instruction to avoid unpredictable delays due to branches being taken or not taken.

Additionally, instructions may raise exceptions (calculation overflow, invalid address, etc.) in the fourth stage upon which the instruction flow is broken. These exceptions cause the core to abort the instructions that are in the first three stages of processing and to start reading instructions from an appropriate handler. The first three stages can be aborted because instructions do not change the state of the core until the fourth and fifth stage.

5.3.3.4 Guaranteed memory latency

In the Harvard architecture, separate memory buses are used for the instruction and data memories. By implementing this architecture, the PH core is able to avoid conflicts over multiple pipeline stages fighting for memory access (structural hazards), preventing stalls. In addition, caches are omitted and the memories and peripherals (memory-mapped access) are clocked at the same speed as the processor allowing for single cycle data access.

5.3.3.5 Constant interrupt overhead (PH-MT)

In the PH core, multi-cycle instructions, even if guaranteed to be a fixed number of cycles, can generate unwanted interrupt servicing jitter since they may be aborted any number of cycles into their execution and always have to be restarted from the beginning after the ISR has executed.

To avoid this jitter, PH-MT (Hughes 2009) duplicates the program instruction counter, the register file, the registers for the first three of its five pipeline stages and the co-processor registers. The result is a multithreaded core that has the effect of halting (instead of aborting) an instruction when an interrupt is raised and resuming it after the interrupt service routine code has been executed in the

duplicated pipeline. If the ISR execution time is kept jitter free, then this technique ensures that the interrupt overhead remains constant.

5.4 A processor with multiple PH cores

A high level view of the microcontroller design developed in this work can be seen in Figure 5.6. The resultant multi-core is a heterogeneous multi-core since each core can have varying internal organisations and peripherals attached. The cores may also be set at different operating frequencies. While not shown in the figure, peripherals (including memories) are connected via a bus.

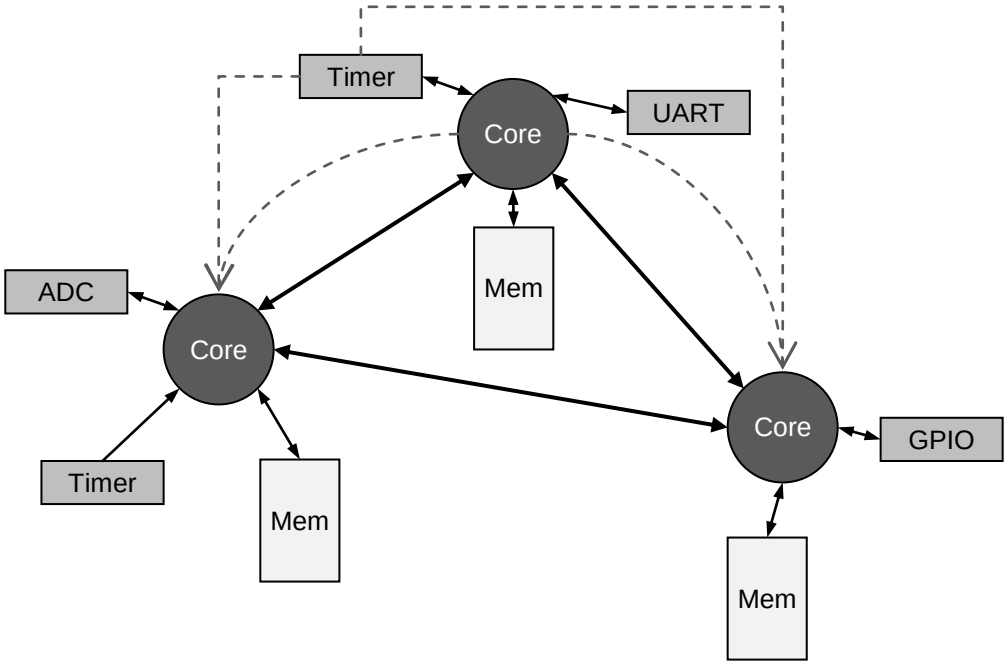

Figure 5.6: Hardware organisation

Each core is connected to every other core by a direct asynchronous point-to-point link implemented as two asynchronous FIFOs (Nebhrajani 2007) for inter-core communication. In addition to these links, the timer on one core (the timing master) and the core itself are connected as external event sources for all the other cores (the timing slaves), allowing for timing events to be propagated to all

cores either at the same time or after one core has done some application specific processing.

5.4.1 Delayed sleep extension to the PH core (PH-DS)

Many of the TTCA implementations (Listing 3.6, Listing 3.8, Listing 3.9 and Listing 3.14) use the interrupt mechanism only to keep track of the number of times the schedule must be simulated so as to build the run queue. Such a simple ISR hardly justifies the hardware overhead of PH-MT and the temporal overhead of the invoked ISR. PH-DS is a hardware simplification that uses the interrupt to indicate how many software sleep requests can be ignored; i.e. it delays the sleep requests.

The mechanism can be seen in Figure 5.7 where an 8-bit counter (initialised to zero) is decremented every time an interrupt occurs and is incremented when there is a request to sleep; only when this counter equals *one* does the core actually sleep. Referring back to Listing 3.1 and the extension in Listing 3.5, it is clear that the execute “tick”-number-of-times behaviour is preserved with this scheme.

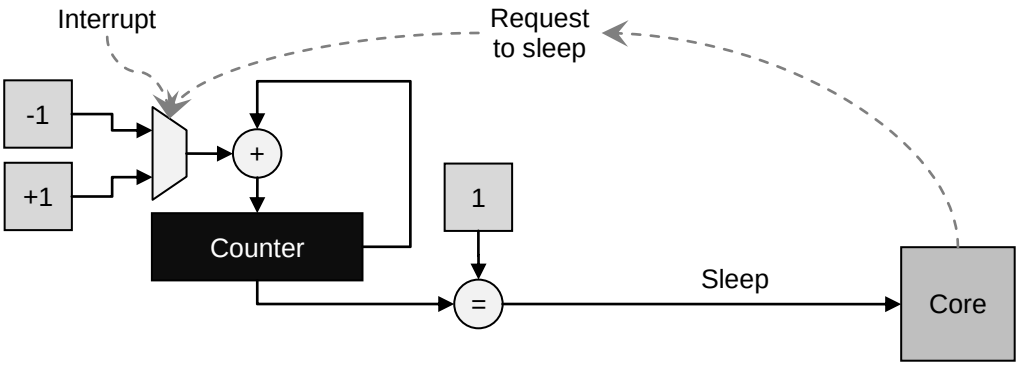

Figure 5.7: The PH-DS mechanism

The PH-DS calls no ISR and hence, the multi-threaded logic can be omitted

without affecting the predictability while retaining the flexibility of designing the schedule creation algorithm in software.

5.5 Inter-task communication scheme

5.5.1 Overview

As required by Section 5.2.2, the communication mechanism developed is completely transparent to the application software: there is no effect on the timing of the tasks and it is implemented in a hardware communication controller with the RTOS' scheduler (running on the core) synchronising the functionality with task execution.

The communication controller is attached to the same bus as the data memory and the rest of the peripherals, allowing it to be directly controlled by software. It receives messages from other cores via the link mentioned in Section 5.4 and writes them directly into data memory (Figure 5.8). This is similar to the architecture in (Kopetz et al. 1993) which receives messages from other cores on a bus instead of on individual connections.

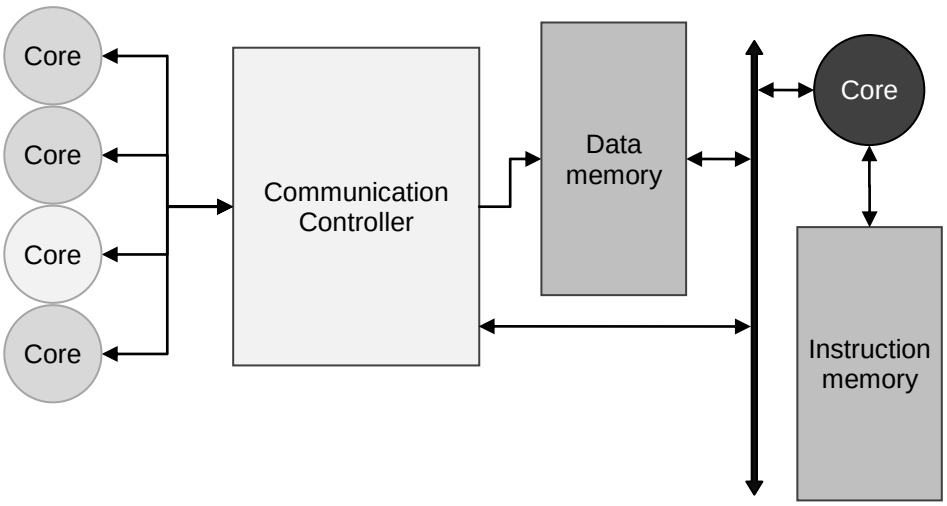

Figure 5.8: Overview of the communication hardware

In this architecture, data memory has one writer (the communication controller or the core) and one reader (the core) and the overlaps mentioned in Section 5.2.2 need to be safeguarded. The communication controller never has to read from data memory since it uses (local) bus snooping to decide when to send data to the other cores.

When safeguarding the overlaps, the lock-free solution in (Kopetz et al. 1993) is unsuitable for the reasons mentioned in Section 2.7.2. From conclusions drawn in that and the following section, it was decided to use a hardware implementation of the 3-buffer single-writer, single-reader mechanism described in Appendix B, with the entire data memory being buffered. Interestingly, if the multi-cores were only running tasks capable of running on a single-core and considering that the cores are synchronised by the periodicity of the tasks, a double-buffer scheme would be sufficient. However, a three-buffer scheme allows arbitrarily overlapping tasks, provides more flexibility and can be easily downgraded to a two-buffer scheme should the resource usage become a concern.

It should be noted that the buffers referred to in Section 5.2.2, are *application buffers* while the context of this discussion refers to *communication buffers*. There may be several application buffers, pertaining even to different tasks, in one communication buffer. In this discussion, the application buffers from one task form one or more *shared memory areas* (SMAs) in the three communication buffers.

The communication controller maintains separate registers (the description) for each SMA: a globally unique identifier, the address and size of the area, an

indication of whether the SMA has been read since the last write and the state of each buffer (latest data, being written, being read). The controller also maintains a lookup table that allows for half-cycle conversions from a memory address to a SMA identifier.

The scheduler component of the RTOS associates SMAs with tasks, requests the controller to switch to the latest buffer for those areas when the task is about to execute and releases the area when the task is finished, i.e. the whole task is considered a critical section (Section 3.2.3). This managerial role of the scheduler is depicted in the overview in Figure 5.9.

In the case of a write, for example, when $core_1$ writes a value to its local data memory, it is noticed (indicated by the rightmost eye in Figure 5.9) by that core's communication controller which then uses the address of the data that was modified to locate a SMA from a local list of descriptions. If a SMA description is found, a message is sent to the communication controller connected to $core_0$ which then uses the identifier in the message to extract a SMA description from its own list of descriptions. If this second SMA description is found, it is used to select the right buffer and the address in that buffer at which to write the data in the message from the other communication controller.

In the case of a read, for example, when $core_0$ reads a value from its local data memory, it is noticed (indicated by the leftmost eye in Figure 5.9) by that core's communication controller which then uses the address of the attempted read to locate a SMA from a local list of descriptions. If a SMA description is found, it is used to select the correct buffer from which to fetch the required data.

These mechanisms are explained further in the subsequent sub-sections.

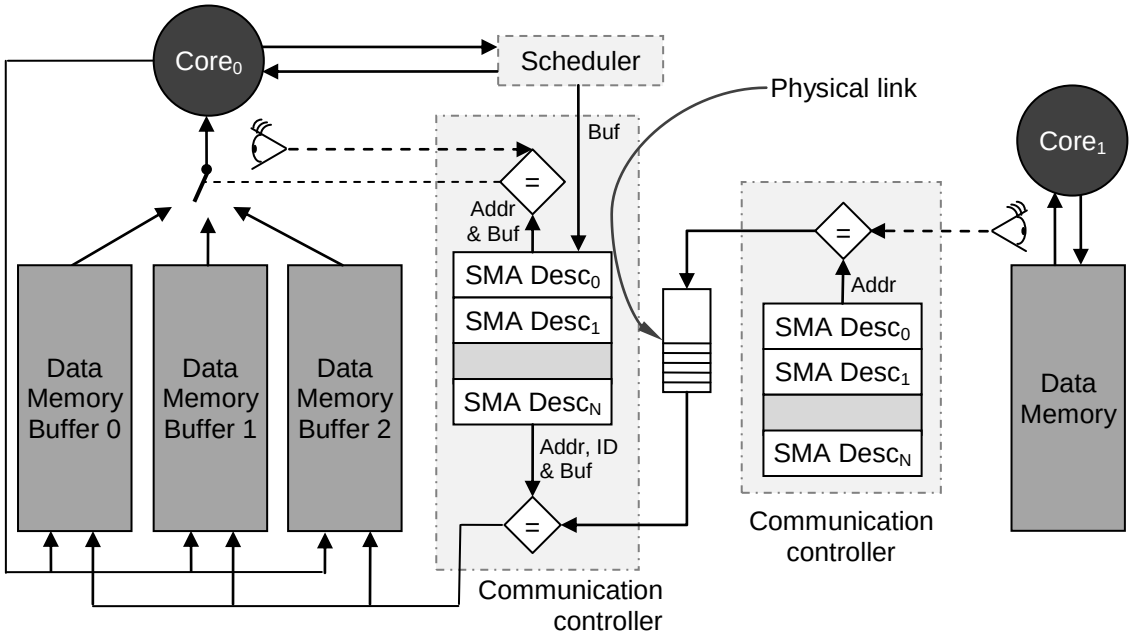

Figure 5.9: An overview of communication between two cores

5.5.2 Creating the descriptions

The SMA descriptions are created upon request by the RTOS and are associated with an identifier decided at compile-time. On receiving the request, the communication controller spends one or more cycles updating the lookup table that converts addresses to SMA identifiers.

5.5.3 Writing

Writes from a core are applied to all three buffers. This is necessary since the half-cycle required to fetch the correct buffer number combined with the additional half-cycle to actually write the data might cause data hazards in the processor pipeline. Unfortunately, this prevents a task from using a SMA for both reading and writing.

If the address being written to is part of a SMA, then after half a cycle when the address has yielded valid SMA information, a notification message is sent to all cores. The message contains the identifier of the area, the offset of the write address from the area's origin and the data that was written. This content is

sufficient for the other cores to write the data into their own buffers at the proper location.

Since shared memory areas may be of different sizes even if associated with the same identifier, the hardware ignores write requests from other cores that cross defined boundaries.

5.5.4 Reading

In the PH processors, memory is clocked at the same rate as the processor, with no caches; hence, after the core places an address on the bus, valid data are expected in the next clock cycle. Translating from a memory address to a shared memory identifier (to fetch the number of the buffer with the latest data) takes half a cycle; and so, all buffers fetch data concurrently from the same address and the data are multiplexed when the right buffer is known.

5.5.5 Switching between buffers

As mentioned in the overview, the scheduler switches the buffers for the SMAs used by a task before it executes. This includes the local buffers (local switch) and the buffers in other cores (external switch) sharing these memory areas. A switch also locks the buffers and so they must be released by the scheduler when the task is finished. A local switch sets the local buffer to the latest written buffer; an external switch reserves a buffer that is not the latest and which is not being read. An external switch uses the last written buffer if a local switch has not occurred since the last external switch; this allows tasks working at different rates to function properly. A switch may also be performed locally only (read switch) if a SMA has multiple readers since multiple readers attempting external switches can disrupt each other.

As seen in Figure 5.10, a buffer may then be in one of several states: available (a), being used locally (l), being used externally (e) and being the last used externally (u); and several guards: an external switch (ES) and release (ER); a local switch (LS) and release (LR); and a local switch having happened since the last external switch (LSSLE). The transitions from the *available* state are the least preferred; transitions by a buffer from another state are always performed instead, if possible.

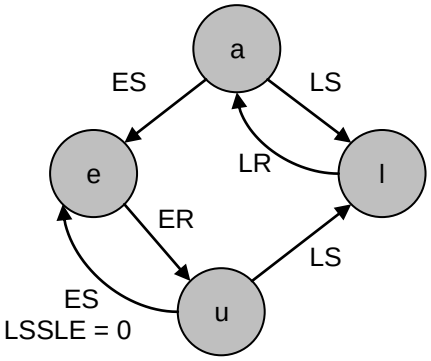

Figure 5.10: States of a buffer

The switching behaviour is examined in more detail in Figure 5.11 and Figure 5.12 where the states of the buffers (Figure 5.10) are shown from the point of view of a task “B”. The condition of a local switch having happened since the last external switch is also shown. The start times of a task “A” are shown as when the external switch request reaches the hardware of the core on which B executes, and likewise task A ends when the external release request is received.

In Figure 5.11, both tasks run at the same rate and overlaps from Figure 5.3 are chosen. The tasks in (a), (b) and (c) can all be scheduled on single processors; (b) is the sort of timeline that can occur on a single processor. It is interesting to note, that (b) only ever uses one buffer, and only (d) uses all three buffers. If the precedence constraint of task B needing to run after task A was added to (d) & (e), then they would resemble (c) and would also use only two buffers. A great disadvantage in this system is that initial data are lost if the tasks are given non-

zero phases due to the extra condition imposed by LSSLE. This behaviour is clearer in Figure 5.12.

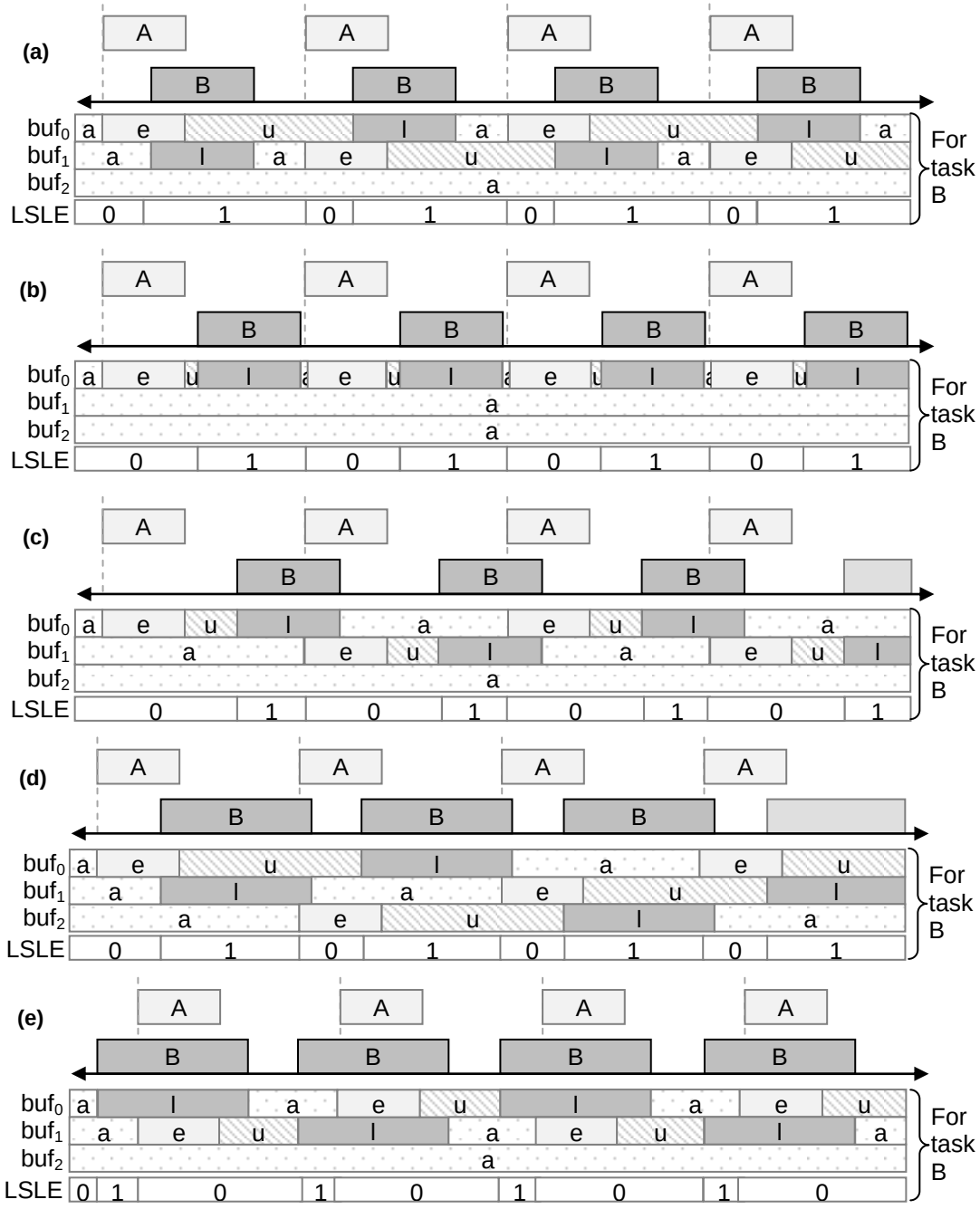

Figure 5.11: Buffer switches from the view of task B when it overlaps with a task A running at the same rate with a combined utilisation less than one

In Figure 5.12, task A runs at twice the rate of B, with the first execution of B taking place after two executions of A, so that data are valid. As before, various combinations are taken: either B runs before the next execution of A or not,

either B starts before the next plus one execution of A or not, either B finishes before the next execution of A or not and either B finishes before the next plus one execution of A or not. In all cases, executions of A which haven't seen an execution of B after a prior execution of A cause no switches.

Figure 5.12 (a) is another example of a single processor type system and accordingly, one buffer is sufficient. (b), (c) and (d) exhibit the long-task problem. However, depending on the data structure used by the application buffers, data losses may occur in (c), (d) and (e) and may be sustained. (c), (d) and (e) could avoid data losses with proper scheduling but will recover in the next tick (not shown).

The example in Figure 5.12 can be expanded to higher frequency rate mismatches as well. As long as the application buffers data appropriately and the reading task executes (on another core) after the last execution of a writer in a batch but before the first execution of a writer in the next batch, then the reader can execute concurrently until the start of the next plus one batch without any data losses or any incoherence.

Multiple buffers can be toggled by a single register write to the communication controller and there is no variability introduced by tasks using variable numbers of shared memory areas. This prevents the communication controller from increasing a task's release jitter.

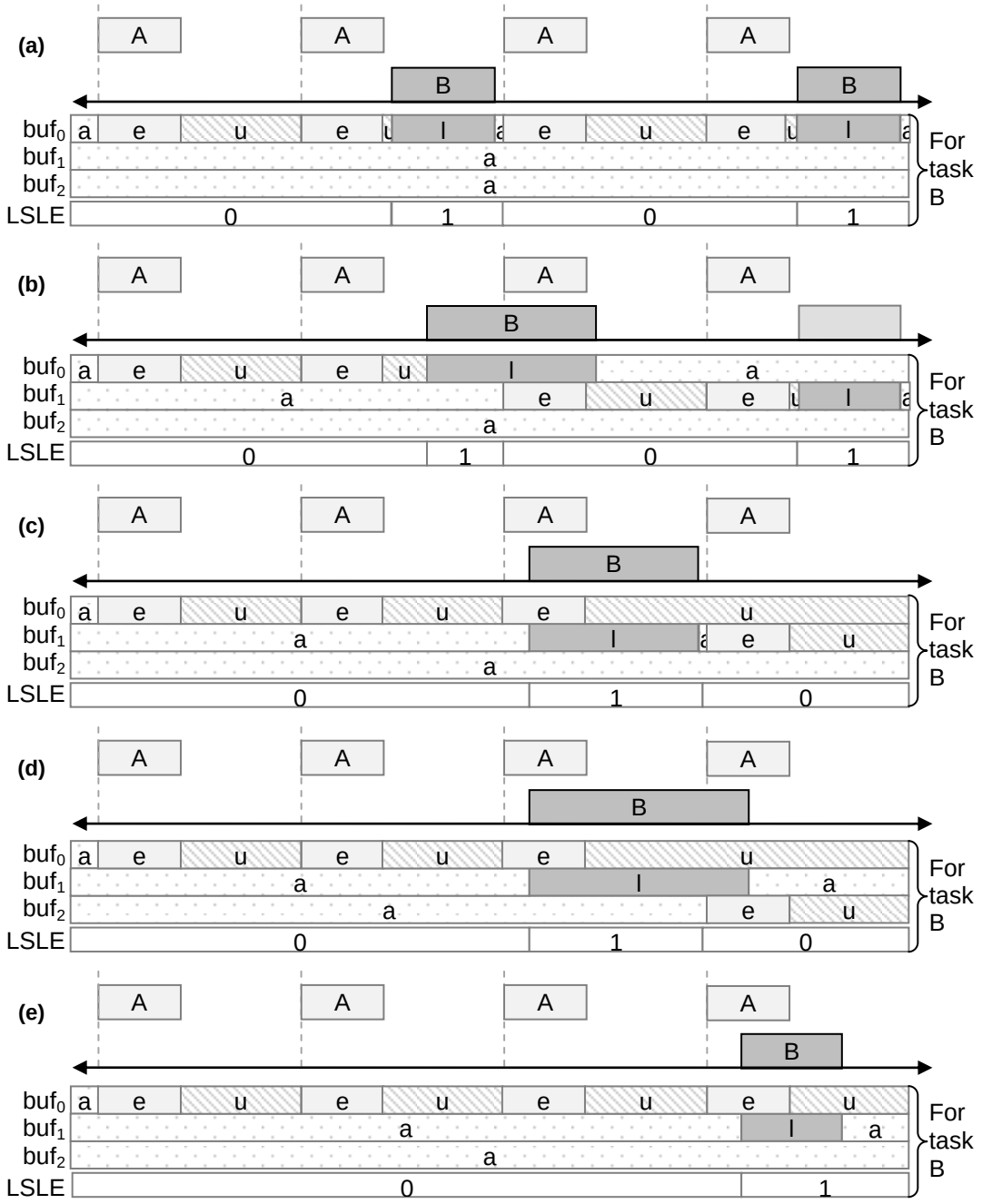

Figure 5.12: Buffer switches from the view of task B when it overlaps with a task A running at twice the rate

5.6 The scheduler design

5.6.1 Overview

In this design, each core stores only the code and data of the tasks that it will be running. Two scheduler implementations were examined:

- Multiple schedule builders: each core is triggered by the timing master, stores the properties of only those tasks that it will be running and creates and dispatches the schedule for these tasks.
- Single schedule builder: the scheduling core stores all the properties of the tasks, creates the schedule for all other cores and triggers them at the start of its ISR. The individual cores still dispatch their own tasks.

5.6.2 Precedence constraints

In a single-processor TTCA design, tasks may have precedence constraints, i.e. a frame of one task may be required to precede a matching frame of another task. Figure 5.13 explores scenarios where these constraints may not be honoured in a multi-core TTCA design. In Figure 5.13, the tick is shown as a dotted vertical line on a horizontal timeline, two tasks are shown as shaded rectangles and the task represented as a rectangle with the smaller width must precede the other one. Figure 5.13 (a) shows the single-processor TTCA design where the constraint is implicitly defined by the order in which tasks are added to the task list (Section 3.2). When the tasks execute concurrently, the task ordering is no longer feasible for this purpose (Figure 5.13 (b)).

Changing the task phase may be used as a solution (Figure 5.13 (c)), but the granularity of one tick is too large for phase changes and increases the latency of the dependent task. The natural follow-up is to decrease the tick interval so as to decrease the granularity (Figure 5.13 (d)) but this increases the scheduler overhead. This design tackles the problem by inserted idle time using code balancing techniques (Section 4.4.3.2) on the appropriate cores, shown as deeply shaded rectangles in Figure 5.13 (e). A schedule creation algorithm may choose to instead execute other tasks in the time slot, to make use of the

available concurrency, since inserted delay time effectively looses the available parallelism.

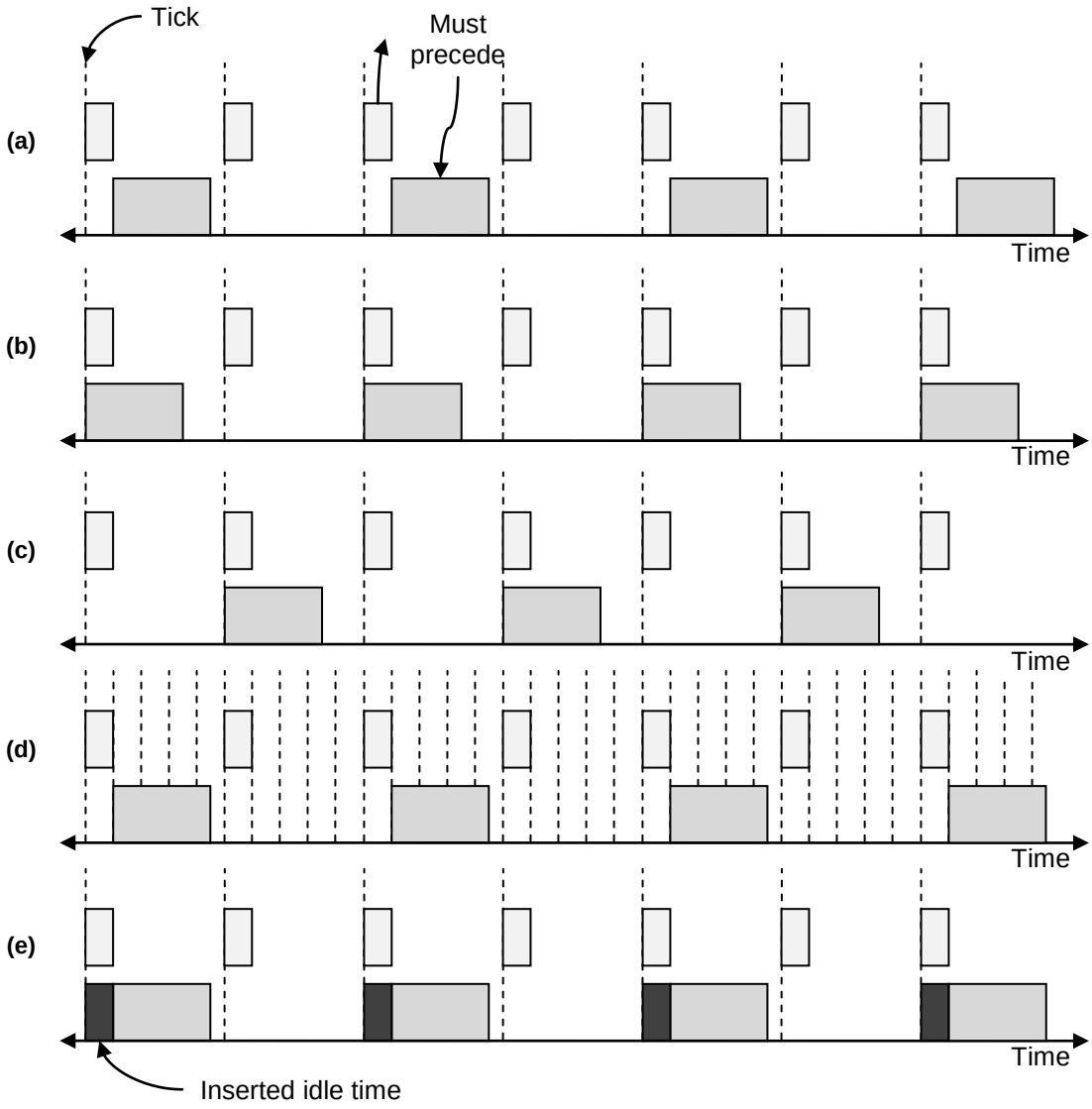

Figure 5.13: (a) Using task order to enforce precedence constraints in a sequential system, (b) has no effect in concurrent execution which must be handled (c) by changing phases, (d) by increasing the tick interval or (e) by inserting idle time

5.6.3 Deterministic initialisation sequence

To ensure all cores are at the required state before the scheduler starts, an initialisation mechanism must be well defined. This has been done as follows: after a reset, the timing master is the only core executing instructions, the other cores are held in a low power mode at their reset addresses and the event

source for the timing slaves is set to the timing master with interrupts enabled. When ready, the timing master interrupts the other cores, causing them to begin executing instructions. In this way, the timing master can create two SMAs and initialise them before waking up the other cores. These SMAs are used to synchronise the cores after initialisation (Listing 5.1 and Listing 5.2).

```
GLOBAL status IS INTEGER
GLOBAL acks   IS INTEGER

DEFINE initialisation OF scheduler:
   SET status TO 0
   SET acks   TO 0
   SHARE status WITH ID = NN
   SHARE acks   WITH ID = NN + 1
   INTERRUPT cores

   ... Perform initialisation   ...

   SET alive TO 1

   WHILE alive IS NOT ((2 EXP number OF cores) - 1)
      SET rack TO 0

      DO
         HOLD status
         READ HOLD acks
         SET rack TO acks
         SET status TO alive
         FREE status, acks
      WHILE rack /= (alive + 1)

      SET alive TO alive + rack

      HOLD status
      SET status TO alive
      FREE status
   ...
```

Listing 5.1: Initialisation on the timing master

```
GLOBAL status IS INTEGER
GLOBAL ack    IS INTEGER

DEFINE initialisation OF scheduler:
   ...
   SET status TO 0
   SET ack    TO 0
   SHARE status WITH ID = NN
   SHARE ack    WITH ID = NN + 1
   ...
   Perform initialisation
   ...
   SET mask TO 2 EXP number OF core
   SET stat TO 0

   DO
      READ HOLD status
      SET stat TO status
      FREE status
   WHILE stat /= (mask – 1)

   HOLD ack
   SET ack TO mask
   FREE ack
   ...
```

Listing 5.2: Initialisation on the timing slaves

This algorithm implements a barrier synchronisation, where the timing master progresses only when all slaves have sent acknowledgements and the slaves progress by an identifier-based ticket mechanism to prevent contention. Once the acknowledgement is sent, a slave core immediately goes to sleep awaiting the first tick. With this algorithm, the master core, which runs the scheduler, is always the last core to go to sleep and so the scheduler is guaranteed to start when all the cores are ready.

5.6.4 The multiple schedule builders implementation (TTC-MC-MSB)

In this implementation, the timer on the timing master is used as a global interrupt generator. Such a method can accommodate reducing die sizes as the wire propagation delay in timing events to each core is negligible compared to an expected tick interval. For example, tick intervals may rarely drop below 0.01

ms whereas propagation delays are expected to be approximately 2.859 ns per μm in 2015 for a 1mm diameter copper wire (ITRS 2007).

Even though a core schedules its own tasks, it is still coupled to the timing master via the interrupt mechanism; this coupling keeps the cores synchronised. TTC-MC-MSB is based off of the table-free multi-rate executive with the dispatch and schedule creation left almost unchanged except for ensuring that the shared buffers for a task are switched before a task executes and released after it finishes.

5.6.5 The single schedule builder implementation (TTC-MC-1SB)

In this implementation, the timing master is called the schedul*ing* core and the other cores are the schedul*ed* cores. The scheduling core handles tasks running on its own core separately from those running on other cores. For tasks on the same core, the schedule is created when the scheduler starts and at the end of the dispatch. For tasks running on other cores, the schedule is created every tick in addition to the initial creation when the scheduler starts.

The run queues for the scheduled cores are implemented as circular buffers in a SMA (Listing 5.3) in co-operation with the scheduling core (Listing 5.4). The memory areas have their buffers switched at the start of the dispatch routines in the scheduled cores (Listing 5.6).

```
GLOBAL run_queue  IS <NNNN> ARRAY OF INTEGERS
GLOBAL index IS INTEGER

DEFINE initialisation OF scheduler:
   ...
   SET ALL OF run_queue TO -1
   SHARE run_queue WITH ID = number OF core
   SET index TO 0
   SET fifo_index TO 0
   ...

DEFINE read_from_queue WITH OUTPUT data:
   IF MSB_16 OF run_queue[fifo_index] IS EQUAL TO index:
      SET data TO LSB_16 OF run_queue[fifo_index]
      SET index TO (index + 1) MOD number OF tasks
      SET fifo_index TO (fifo_index + 1) MOD SIZE OF run_queue
   ELSE:
      SET data TO NULL
```

Listing 5.3: Managing the run queue in the scheduled queue

```
GLOBAL run_queue IS <number OF cores> ARRAY OF
                      (<NNNN> ARRAY OF INTEGERS)
GLOBAL index     IS <number OF cores> ARRAY OF INTEGERS

DEFINE initialisation OF scheduler:
   ...
   FOR EACH core:
      SET ALL OF run_queue[number OF core] TO -1
      SHARE run_queue[number OF core] WITH ID = number OF core
      SET index[number OF core] TO 0
      SET fifo_index[number of core] TO 0
   ...

DEFINE write_to_queue WITH INPUT number AND INPUT data:
   SET run_queue[number][fifo_index[number]] TO
               MSB_16 = index[number]
               LSB_16 = data
   SET index[number] TO (index[number] + 1) MOD number OF tasks
   SET fifo_index[number of core] TO
           (fifo_index[number of core] + 1) MOD SIZE OF run_queue
```

Listing 5.4: Managing the run queues in the scheduling core

The scheduler is built as a modification to the table-free multi-rate executive (Section 3.6.2), adding a schedule creation stage for the scheduled cores to the ISR (Listing 5.5). The scheduling core is triggered by its own timer and triggers the scheduled cores in the ISR with an interrupt. The scheduling core must

delay for a little while before building the run queues, so that the scheduled cores have the opportunity to switch their buffers (Listing 5.6).

```
DEFINE service OF interrupt:
   INTERRUPT CORES
   RAISE ticks BY 1
   DELAY
   HOLD run_queue
   FOR EACH task IN tasks WHERE core_number IS NOT 0:
      LOWER delay OF task BY 1

      IF delay OF task IS 0:
         write_to_queue WITH number = core_number OF task
                             data   = identifier  OF task

         SET delay OF task TO period OF task
   FREE run_queue
```

Listing 5.5: Event service of the scheduling core

To allow the scheduling core to run tasks, the co-operative dispatch and schedule creation are left almost unchanged – the only changes required are checking that the new `core_number` property is zero and that the buffers for all shared memory areas used by the task are switched before the task executes.

For scheduled cores, there is no schedule creation to be done and the ISR stays the same as the general TTCA implementation ISR (Listing 3.4). The dispatch reads from the queue in shared memory and executes tasks appropriately (Listing 5.6).

```
DEFINE dispatch OF scheduler:
   WHILE ticks > 0:
      LOWER ticks BY 1
      READ HOLD run_queue

      LOOP:
         SET task_id TO data FROM read_from_queue
         IF task_id IS NULL:
            EXIT LOOP

         HOLD EACH shared memory area OF task
         RUN task WITH identifier = task_id
         FREE EACH shared memory area OF task

      FREE run_queue
```

Listing 5.6: Task dispatch in a scheduled core

5.7 Evaluation

The developments in this chapter were aimed at eliminating the timing anomalies introduced by long-tasks and non-harmonic tasks into a TTCA implementation. This was performed by switching to a multi-core processor with an application-transparent communication scheme for tasks on different cores. This section will examine the effects of these alterations compared to the situation under a single-core scalar processor. The two multi-core TTCA implementations that were created to utilise these changes will be evaluated as part of the case study in Chapter 6.

For brevity, the original PH core will be referred to as PH and the various extensions with hyphenations: multi-threaded as PH-MT and an *n* core as MC-PH*n*. In the same vein, the original table-free multi-rate TTCA implementation will be referred to as TTC and the single and multiple schedule builder multi-core versions as TTC-MC-1SB and TTC-MC-MSB respectively. Finally, all code will either be in MIPS I assembly or in C.

For timing measurements, special instructions were inserted at appropriate points in the software code. These instructions triggered the transmission of the value of a 16-bit or 28-bit (depending on the state of a physical switch) hardware counter down a serial link to a development machine; the transmissions have no effect on the operation of the cores. The hardware counter is reset by hardware logic when an interrupt is generated by timer 0 on core 0; is incremented at the rate of 25 MHz; and can count to approximately either 2.6 ms or 10,737 ms before it overflows.

5.7.1 Hardware utilised

All experiments were run with the soft microcontrollers compiled for the Xilinx Virtex-5 LX50T FPGA on the ML505 development board. The FPGA was developed in VHDL, compiled with Xilinx ISE WebPACK 12.2 and simulated with ModelSim XE III 6.5c. A MIPS-I port of GCC 3.3.3 was used to compile the software source code.

TTC-MC-MSB was implemented on all PH-DS cores while TTC-MC-1SB had a PH-MT as the scheduling core and PH-DS as the scheduled cores. The cores were driven by asynchronous clock signals at similar 25 MHz frequencies; with all interaction between them protected by circuits such as asynchronous FIFOs.

5.7.1.1 Results

Figure 5.14 shows the hardware utilisation, as the number of slices occupied for solely one function, for the PH, the PH-MT and the PH-DS, each in a configuration with 32 Kb code memory, 32 Kb data memory, three timers and one GPIO. When compared to the unpredictable PH core, the predictable PH-MT core results in approximately 14% more hardware being used while the

predictable and co-operative PH-DS core adds less than 1% to the hardware cost. The savings from employing PH-DS cores increase as the number of cores in a MC-PH*n* increase.

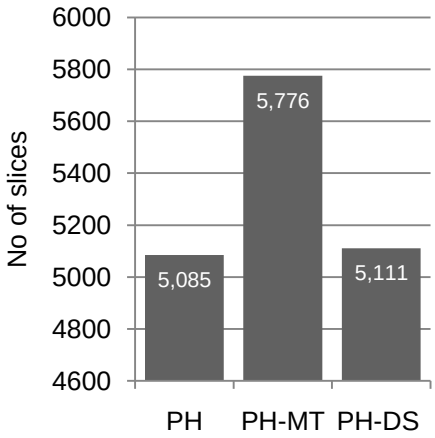

Figure 5.14: Hardware utilisation on changing the core type

The hardware utilisation when the communications mechanism is enabled and disabled is shown in Figure 5.15 for three MC-PH*n* implementations where each core is configured with 32 Kb code memory, 8 Kb data memory, one GPIO and at least one timer – core 0 has three timers. The relative increase in hardware due to the communication capability scaled by the number of cores in all three cases is approximately 45% of the implementation containing the PH core (Figure 5.14).

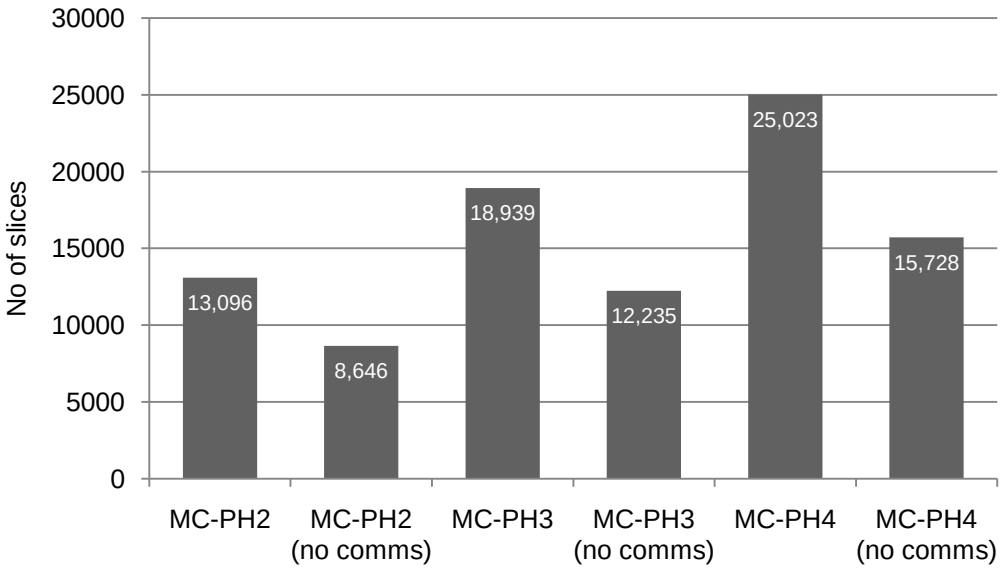

Figure 5.15: Hardware utilisation on removing the communication mechanism from MC-PH*n* implementations

5.7.2 Inter core communication

To evaluate inter-task communication, two tasks were implemented and run on different cores (τ_0 and τ_1 on cores 0 and 1 respectively) with TTC-MC-1SB and two SMAs of a 100 words each. Both tasks check one of the SMAs against a consecutive range of 100 numbers and then write the next set of consecutive 100 numbers to the other SMA (Listing 5.7 and Figure 5.16). To avoid release jitter, the scheduler was carefully code-balanced with software techniques.

```
static uint32_t Shared_I_G[NUM];
static uint32_t Shared_O_G[NUM];
static uint32_t Errors_G[MAX_RUNS];
static uint32_t Runs_G;
static uint32_t Base_G; // Initialised to 0 or NUM

void Latency_Check_Update()
   {
   if (Runs_G < MAX_RUNS)
      {
      uint32_t index;
      uint32_t upper = Base_G + NUM;

      for (index = Base_G; index < upper; ++index)
       {
       if (Shared_I_G[index - Base_G] != index)
          ++Errors_G[Runs_G];
       }

      Base_G += NUM;
      upper = Base_G + NUM;

      for (index = Base_G; index < upper; ++index)
       {
       Shared_O_G[index - Base_G] = index;
       }

      Base_G += NUM;
      ++Runs_G;
      }
   }
```

Listing 5.7: C code for one of the identical tasks in the evaluation of inter-task communication

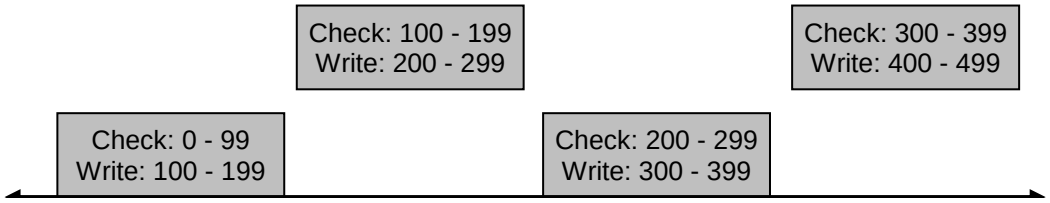

Figure 5.16: Task functionality for inter core communication evaluation

Each task was run a fixed number of times, recording the number of errors in the sequence for that run. The two tasks were executed at the same frequency and in the same tick, but τ_1 had a delay task, τ_d (Listing 5.8) inserted before it so that its start time relative to τ_0 could be evaluated. The value of *NN* in the delay task was varied along with the insertion of up to 3 NOP instructions, in order to find the precise number of cycles (Equation 5.1) at which the errors disappear.

$$Number\ of\ Cycles = \omega = 3 + 4 * NN + number\ of\ extra\ NOPs \qquad (5.1)$$

```
delay_task:
   li $8, NN
loop:
   addiu $8, $8, -1
   nop
   bnez  $8, loop
   nop
   nop x [0...3]
   jr $31
   nop
```

Listing 5.8: Assembly code for a delay task

5.7.2.1 Hardware results

The execution time of each task was consistently measured to be 1448 cycles or 57.92 µs, except in the case of errors, when the time increased to 1948 cycles or 77.92 µs. τ_0 encountered zero errors in all the trials, whereas the number of errors encountered by τ_1 jumped from 100 to 0 as the delay task (Listing 5.8) was lengthened (Figure 5.17). τ_0 and τ_d were found to consistently

start at the same cycle count on each run and a sample execution in the absence of errors can be seen in Figure 5.18.

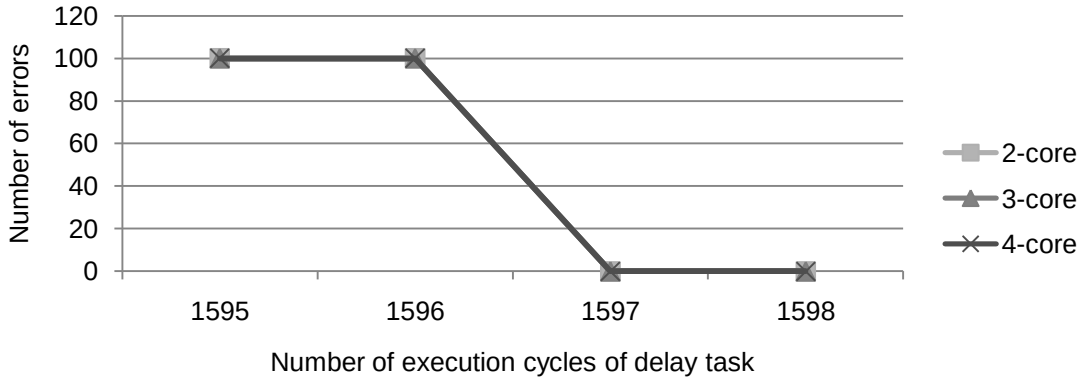

Figure 5.17: Number of errors encountered by τ_1

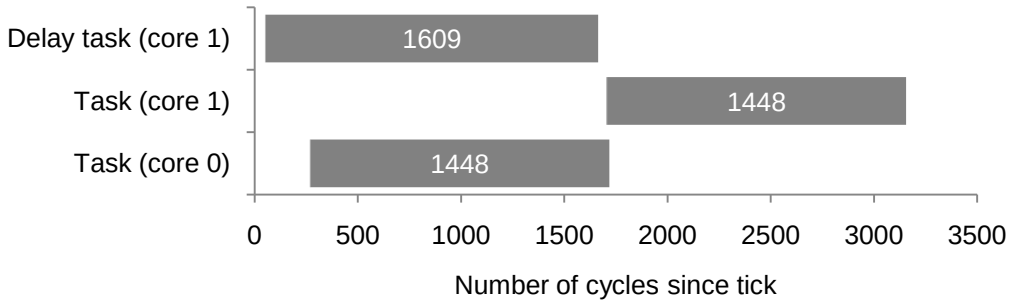

Figure 5.18: Snapshot of task execution on a dual-core with no errors (NN = 399, 0 NOPs, ω = 1599 cycles)

Since τ_1 always completes before the next execution of τ_0, the latter experiences no errors. The errors experienced by τ_1 are due to it beginning execution while τ_0 is still executing; in particular, when τ_1 tries to switch the buffers while $\tau_0$is still writing. Interestingly, in this particular case, the errors disappear while the tasks are still overlapping in execution by 10 cycles.

It is also worth pointing out that the number of errors is either 0 or 100, signifying that data are either wholly corrupt or wholly accurate. Figure 5.17 indicates that the hardware synthesis tool output can be trusted in terms of timing accuracy down to the cycle level even when consuming large portions of

the FPGA. Also, the communication controllers, which have to multiplex between cores, demonstrate determinism in that there is no change in the required number of delay cycles even when the number of cores is increased.

5.7.2.2 Simulation results

Figure 5.19 and Figure 5.20 respectively show simulations of the system when the number of errors in τ_1 are about to decrease to zero, as τ_d is lengthened and when the number of errors is zero a cycle later.

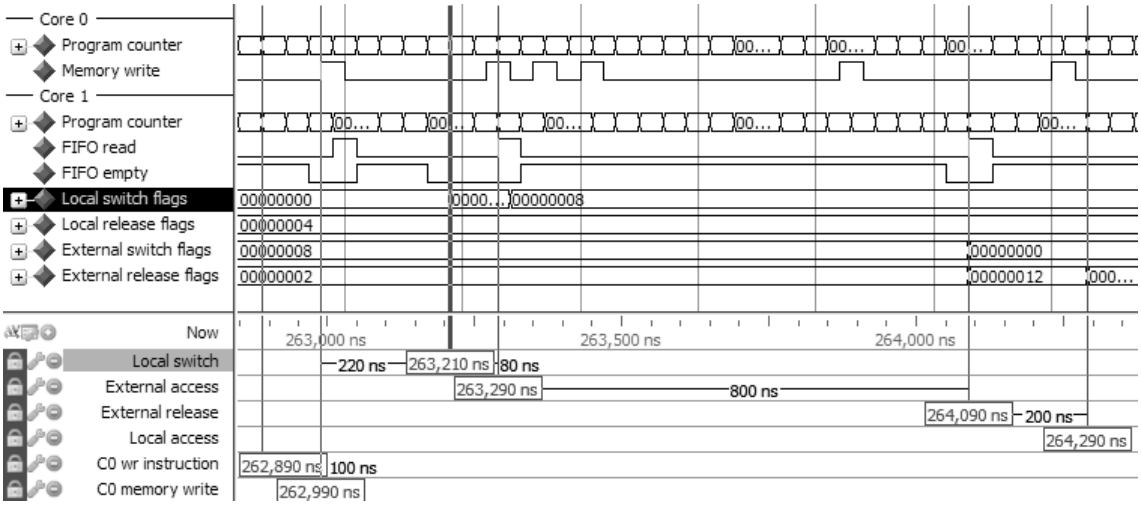

Figure 5.19: Simulation of buffer switches with errors at ω = 1596 cycles

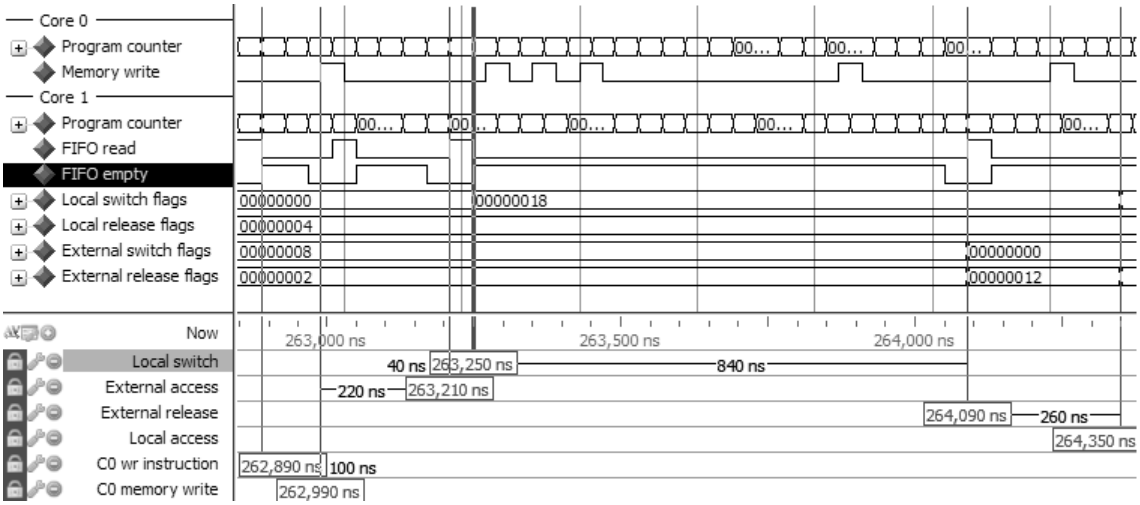

Figure 5.20: Simulation of buffer switches just after errors stop at ω = 1597 cycles

These figures show that the errors begin when the local switch on τ_1 overlaps with the data access of τ_0, and matches the hardware observations. The implementation only toggles buffers on data access and so the external access causes a premature toggle. The overlap amount is highly dependent on the task design. If the task were designed so, the last access of a SMA might even be inserted in the delay slot for the task return instruction, making it the last statement executed. Similarly, the load instruction could be made the first statement of the other task executed. In such a scenario, the communication latency might manifest itself to the application developer. However, most often, the compiler inserts broiler plate stack setup and take-down code in the function preludes and postludes which are sufficient to absorb the communication latencies.

Figure 5.20 also shows the communication delay, from when the SMA write instruction is first loaded into core 0: it is executed 2½ cycles later and then propagated by the communication controller to the FIFO one cycle later. The FIFO spends 2 cycles on metastability protection and 1½ cycles on internal propagation. Finally, core 1 takes one cycle to notice that the FIFO contains a value.

5.7.3 Initialisation

To verify that the initialisation sequence is predictable, the initialisation time for each core was measured after a thousand resets: on hardware for up to four cores and in simulation for up to eight cores. Additionally, the hardware test was executed on dual, triple and quad core platforms. A single-core platform was also examined for the sake of comparison.

5.7.3.1 Results

The average number of cycles taken to initialise each core on four hardware platforms, for a thousand trials, can be seen in Figure 5.21 with the standard deviation presented in Figure 5.22; each line represents a particular hardware platform with a data point for each core in the platform. Thus, the quad core system has four data points, while the single-core platform has only one.

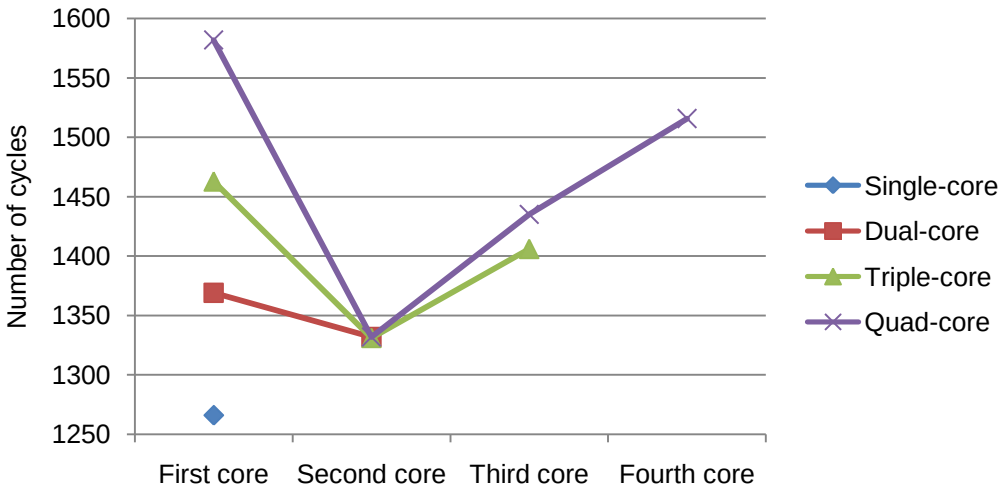

Figure 5.21: Average number of cycles taken for a core to initialise on one- to four-core devices

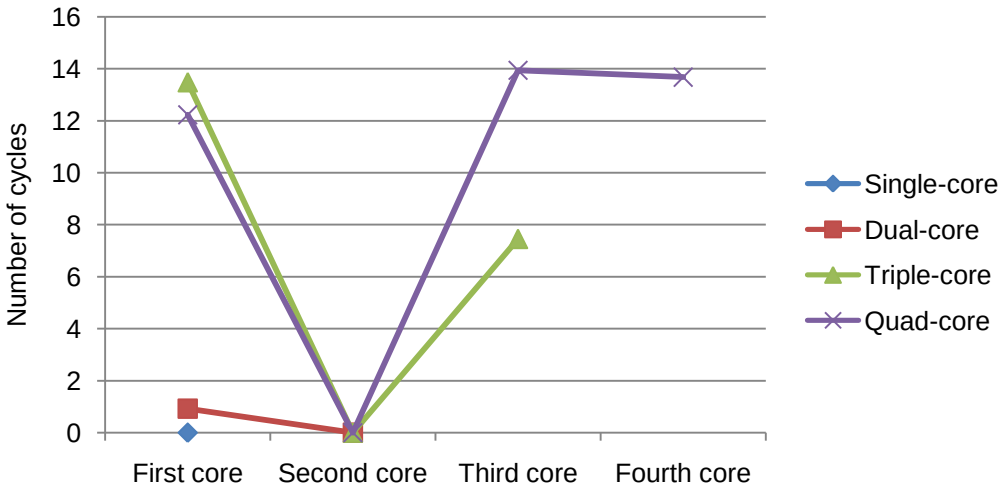

Figure 5.22: Standard deviation in initialisation times on one- to four-core devices

The reference single-core datum in Figure 5.21 shows the amount of time taken by the application software to initialise and Figure 5.22 shows that this

initialisation has zero variations. Thus, the increase in the number of cycles and the variation thereof for the other hardware platforms are purely due to the barrier synchronisation (Section 5.6.3) that ensures that cores finish initialisation in the order of their numbering except for the first core which always finishes last. In the dual, triple and quad core platforms, the second core has a zero variance because in this particular evaluation, the second core always finishes its application initialisation after the first core has begun the barrier synchronisation; when the second core reaches its synchronisation point, it can carry through immediately. For the third and the fourth cores, the number of cycles taken to initialise increases linearly, though no such trend can be observed about the standard deviation which is dependent on the software implementation of the barrier synchronisation (Listing 5.2).

Figure 5.23 shows the average number of cycles taken on eight simulated platforms, and Figure 5.24 shows a simulation of eight cores. As might be expected, the simulations have no variation between runs for the same number of simulated cores, and no simulation equivalent for Figure 5.22 is presented.

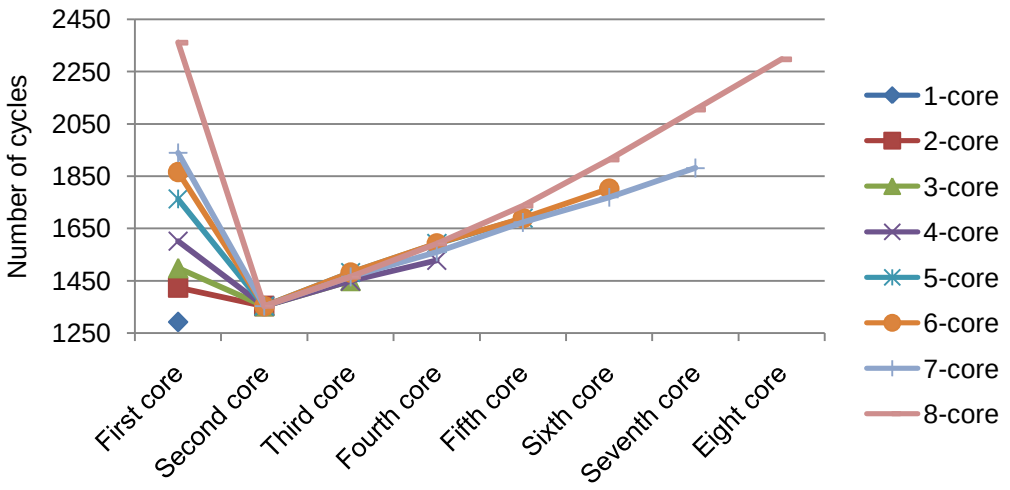

Figure 5.23: Number of cycles taken for a core to initialise on simulated one- to eight-core devices

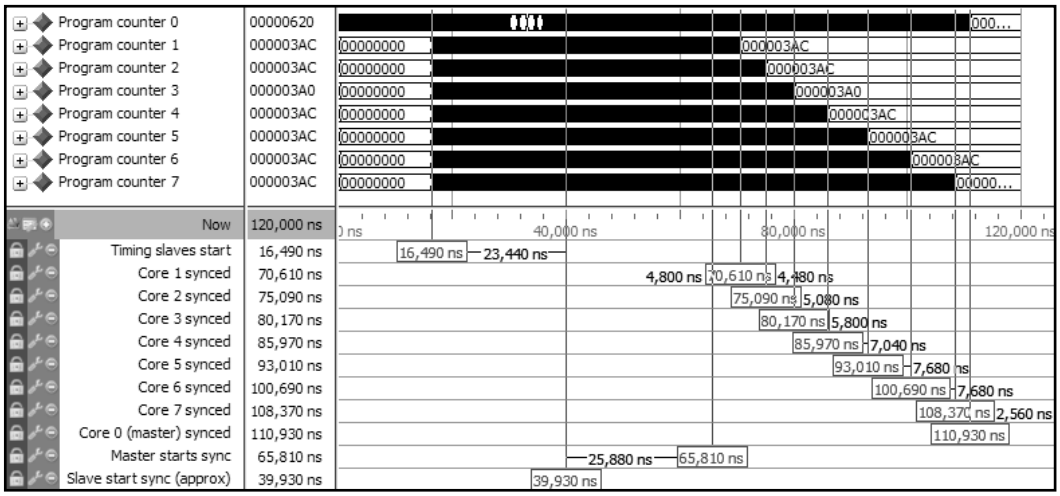

Figure 5.24: Simulation of the initialisation sequence for 8-cores

Figure 5.23 and Figure 5.24 exhibit the same linear trend in initialisation time as that in Figure 5.21. Figure 5.17 shows that the hardware synthesis tool is able to maintain the timing relationships when synthesising different numbers of cores and so the variations in Figure 5.22 and between the equivalently numbered cores in Figure 5.21 are the cause of jitter in the software algorithm employed, an observation reinforced by the simulation result in Figure 5.23. However, the amount and reduction of this jitter is of little importance, the only requirement of the initialisation algorithm was to start the cores in a pre-defined order, which the results demonstrate is always maintained, with the slaves starting sequentially and the master always starting last.

5.8 Conclusions

This chapter has examined a solution to the problem posed to TTCA by the introduction of long and non-harmonic tasks, especially during maintenance when the work is most often performed by individuals who were not the original developers and who are not familiar with the system. The proposed solution is to replace the single-core that is proving inadequate for TTCA with other,

possibly simpler, cores, without breaking the interface with the environment. Such an action may also be done at design infancy with an eye to future development, potentially deploying devices with unused cores.

To increase the applicability of the solution, the inter-task communication scheme was examined in detail with the aim to allow tasks to be written as in a single-core system, maintaining the design simplicity. The evaluation in this chapter demonstrated that this has been successful to the extent of allowing even a tiny overlap in execution.

The next chapter uses a case study to evaluate the schedulers described in this chapter and their effectiveness in coping with the long-task problem and the introduction of non-harmonic task periods.

Chapter 6

Case study: F-16 flight system

6.1 Introduction

The previous chapter described two TTCA implementations aimed at enabling or improving the scheduling of single-processor TTCA designs that possess the long-task problem or that contain tasks with non-harmonic periods; the two implementations were made possible by increasing the concurrency at the hardware level with a multi-core design. The previous chapter also described an inter-core communication scheme that allows tasks' designs to remain unchanged and a predictable initialisation sequence so that all the cores are guaranteed to be at a ready state when the TTCA implementations take over.

To evaluate the implementations and their ability to grapple with the demands of maintenance and future development, this chapter studies a simulated F-16 flight system, as specified in (Abdelzaher et al. 1997).

The next section delves into the technical details of the system, followed by a detailed description of the evaluation setup. Finally, the results of the evaluation are presented with relevant discussions.

6.2 Technical details

In the published study, the period for each task in the system was varied depending on the required quality of service (the higher the reward, the better the quality) (Table 6.1). The flight system followed a pattern of first taking-off

and climbing; then holding a constant altitude around a rectangular path; ending with descending and the final approach to landing. The military operation of destroying possible enemy targets using an onboard radar and missiles was also simulated.

Table 6.1: Evaluation task-set without a long-task

Name	WCET (ms)	Period (s)	QoS Level	Reward
Guidance	100	10	0	10
	100	5	1	15
	100	1	2	20
Controller	80	5	0	1
	80	1	1	100
	80	0.2	2	120
Slow Navigation	100	10	0	10
	100	5	1	20
	100	1	2	25
Fast Navigation	60	5	0	1
	60	1	1	100
	60	0.2	2	120
Missile Control	500	10	0	1
	500	1	1	30

Four basic flight control tasks were utilised: “Guidance” was responsible for setting the reference trajectory of the aircraft in terms of altitude and heading with sensor values supplied by “Slow Navigation”; “Control” was responsible for executing closed-loop control of the actuator with sensor values supplied by “Fast Navigation”. “Missile Control” reads a radar and, if necessary, launches a missile; execution at a higher rate allows faster-moving targets to be destroyed.

The original study aimed to gracefully downgrade the tasks in quality according to the run-time capabilities of the system. When running on a single-core, the periods shown in Table 6.2, TS-1, were utilised: the flight control was stable but sluggish and the aircraft was unable to destroy fast moving targets. When trying to hit fast moving targets, two processors were required to prevent destabilisation in the flight control.

Table 6.2: Task-set yielding sluggish control (TS-1)

Name	WCET (ms)	Period (s)	QoS Level	Utilisation
Guidance	100	10	0	0.01
Controller	80	1	1	0.08
Slow Navigation	100	10	0	0.01
Fast Navigation	60	1	1	0.06
Missile Control	500	10	0	0.05

The flight control can be made slightly better with the task-set, TS-2, in Table 6.3. On a co-operative system, however, TS-2 can be expected to have large release jitter due to the long-task problem: the combined WCET of the tasks *Guidance* and *Slow Navigation* and the WCET of the task *Missile Control* are greater than or equal to the periods of the tasks *Controller* and *Fast Navigation*.

Table 6.3: Task-set with better control but unable to hit fast moving targets (TS-2)

Name	WCET (ms)	Period (s)	QoS Level	Utilisation
Guidance	100	10	0	0.01
Controller	80	0.2	2	0.4
Slow Navigation	100	10	0	0.01
Fast Navigation	60	0.2	2	0.3
Missile Control	500	10	0	0.05

Likewise, the responsiveness of the targeting system can be improved by increasing the period while still staying within the utilisation bounds of a single processor system. This is seen in the task-set, TS-3 in Table 6.4, where the period of *Missile Control* is increased so that the utilisation is approximately 90%; the extra 10% is to accommodate scheduling overheads. However, with this change, it may be noted that the tasks of TS-3 have non-harmonic relationships in their periods.

Table 6.4: Task-set to improve the ability to hit fast moving targets (TS-3)

Name	WCET (ms)	Period (s)	QoS Level	Utilisation
Guidance	100	10	0	0.01
Controller	80	0.2	2	0.4
Slow Navigation	100	10	0	0.01
Fast Navigation	60	0.2	2	0.3
Missile Control	500	2.78	-	0.18

6.3 Setup

In this evaluation, it is considered that the F-16 flight system has been deployed in the field on a suitable hardware platform with TS-1. Further, it is considered that at a point in the future, it becomes desirable to have snappier control performance, resulting in the specifications of TS-2 being applied. Similarly, at another future point, it is considered necessary to hit faster moving targets and so, TS-3 becomes the new system specification. The evaluation details the amount of effort required to make each upgrade and the implications therein, grouped by the scheduler implementation.

The scheduler implementations chosen depended on the choice of the initial hardware platform:

- A PH-MT platform: This was a platform with a single predictable PH core supporting interrupts. Various scheduler implementations were trialled:
    - TTC-MT: A TTCA implementation that executes tasks co-operatively with fixed priorities.
    - DPC: An implementation that executes tasks co-operatively with the highest priority dynamically assigned to the task with the earliest deadline.
    - TTP: A fixed-priority pre-emptive implementation that allocates stack space as a task executes.
    - TTP-MJ: Functionally identical to TTP, but with code-balancing techniques applied to key scheduler areas to minimise jitter.
    - TTH: A type of TTCA implementation that permits a single pre-emptive task in order to tackle the long-task problem (page 4-7).
    - FPP and DPP: Fixed and dynamic priority pre-emptive implementations that reserve space on the stack for each task on creation. In either case, the task with the earliest deadline is given the highest priority.
- A PH-DS platform: This was a platform with a single predictable PH core supporting only co-operative software running a TTCA implementation, TTC-DS.
- A MC-PH3 platform: This was a triple-core processor where core 0 was either PH-MT or PH-DS while the remaining cores were PH-DS. Either TTC-MC-1SB or TTC-MC-MSB was deployed depending on whether

core 0 was PH-MT or PH-DS. *Fast Navigation* and *Controller* were always executed on core 1 and the rest of the tasks on core 0 except in TS-3 where *Missile Control* was executed on core 2.

The scheduler implementations (excepting the multi-core ones) were adapted from those found in the RapidiTTy® toolset from TTE® Systems; the tasks were implemented as dummy tasks to meet the stated WCET with a small amount of execution jitter introduced by an online pseudo-random number generator. The comparison between schedulers was made on the amount of release and completion jitter on the five tasks and on the software and run-time overheads of the scheduler. The release jitter is indicative of stable sampling and the completion jitter of stable actuation.

For all three task-sets, the tick interval was set as the GCD of all the task periods: TS-1 had a tick interval of one second; TS-2 had a tick interval of 200 milliseconds and TS-3 had a tick interval of 20 milliseconds. A heartbeat LED task (Pont 2001) was also always scheduled on core 0, at the lowest priority, at the rate of one Hertz. For clarity, this task has been omitted from the results.

In the fixed-priority scheduler implementations, the highest priority was assigned to the task with the earliest deadline, while honouring precedence constraints (i.e. a navigation task is to be executed before the corresponding control task). In the dynamic-priority implementations, this was maintained by giving the navigation tasks deadlines slightly earlier than the corresponding control tasks.

In the TTH implementation, the *Fast Navigation* and *Control* tasks were combined since only a single pre-emptive task was supported, and both these

tasks were of the highest priority. This resulted in TS-3 being unfeasible under TTH since the length of the pre-emptive task exceeded the tick interval.

6.4 Measured task timing

The tasks' execution times under TTC-MT can be seen in Table 6.5 – the times match the task specifications and have a controlled amount of jitter (measured according to Equation 2.23) inserted into them.

Table 6.5: The run-time timing properties of the tasks under TTC-MT

Name	WCET (ms)	Execution jitter (us)	Execution jitter %
Guidance	100.0	288.9	0.29
Controller	80.0	290.7	0.36
Slow Navigation	100.0	301.7	0.30
Fast Navigation	60.0	293.8	0.49
Missile Control	500.0	307.4	0.06

The tasks all use the same pseudo-random number generation algorithm for jitter generation. They maintain their own copies of the registers for the algorithm and always use the same initial seed value between runs, allowing them to maintain their run-time properties even when pre-empted or executed in different sequences.

6.5 Release and completion jitter

The release and completion jitter for the five tasks across the different schedulers and task-sets can be seen in Figure 6.1, Figure 6.2 and Figure 6.3. All the scheduler implementations perform well with TS-1 (Figure 6.1) since this task-set is suitable for co-operative systems, and the pre-emptive

implementations have little opportunity to employ their advanced features which are the primary cause of jitter. As would be expected, the release jitter increases as the priority decreases, equalling the cumulative completion jitter of the higher priority tasks. The co-operative execution is distinctive by the fairly uniform completion jitter displayed by the different implementations. TTH displays a peculiarly high release jitter in the three lowest priority tasks due to a large completion jitter in the two combined high priority tasks.

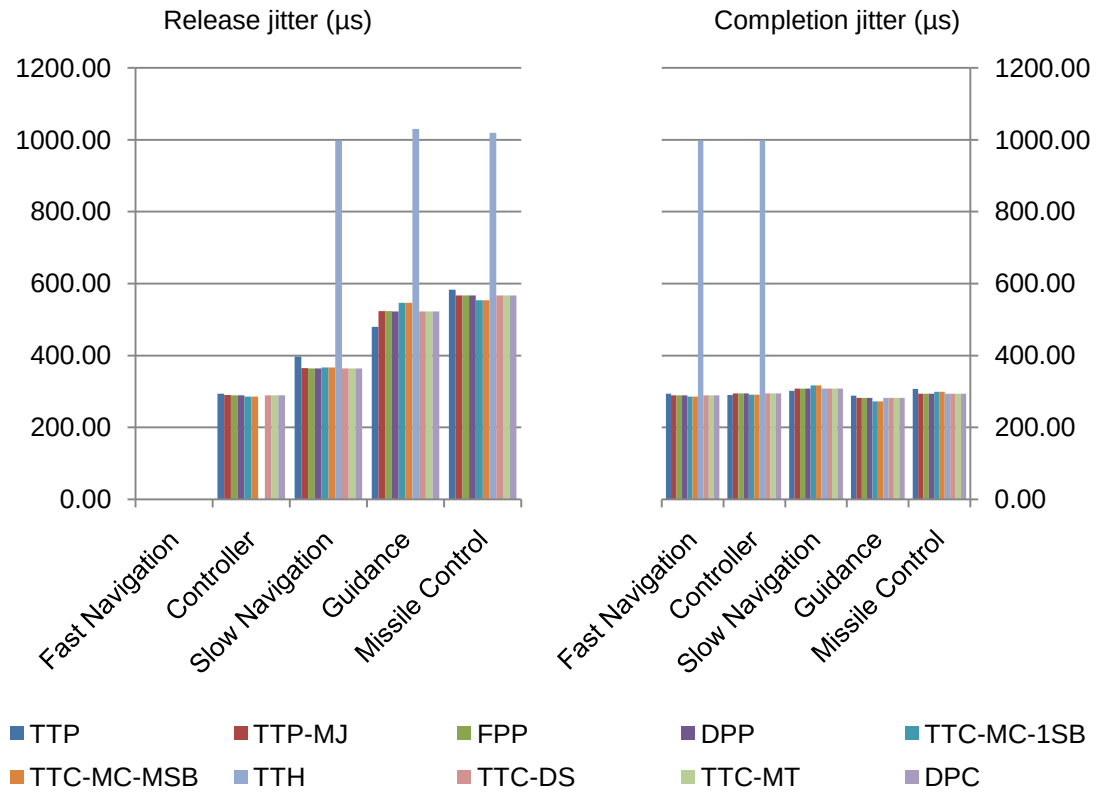

Figure 6.1: Jitter for TS-1

The single-core co-operative schedulers have been omitted from Figure 6.2 as they are unfeasible with TS-2: both TTC-DS and TTC-MT displayed approximately 165 ms release jitter; though DPC was able to halve this to around 85 ms, the figure is unacceptable. With TS-2 (Figure 6.2), the pre-emptive schedulers exhibit higher jitter for the lower priority tasks than TS-1

(Figure 6.1), due to the increased amount of pre-emption. The multi-core schedulers, due to co-operative execution, are able to keep the levels of release jitter more or less the same as with TS-1; for some tasks the release jitter has even decreased since they are preceded by fewer tasks.

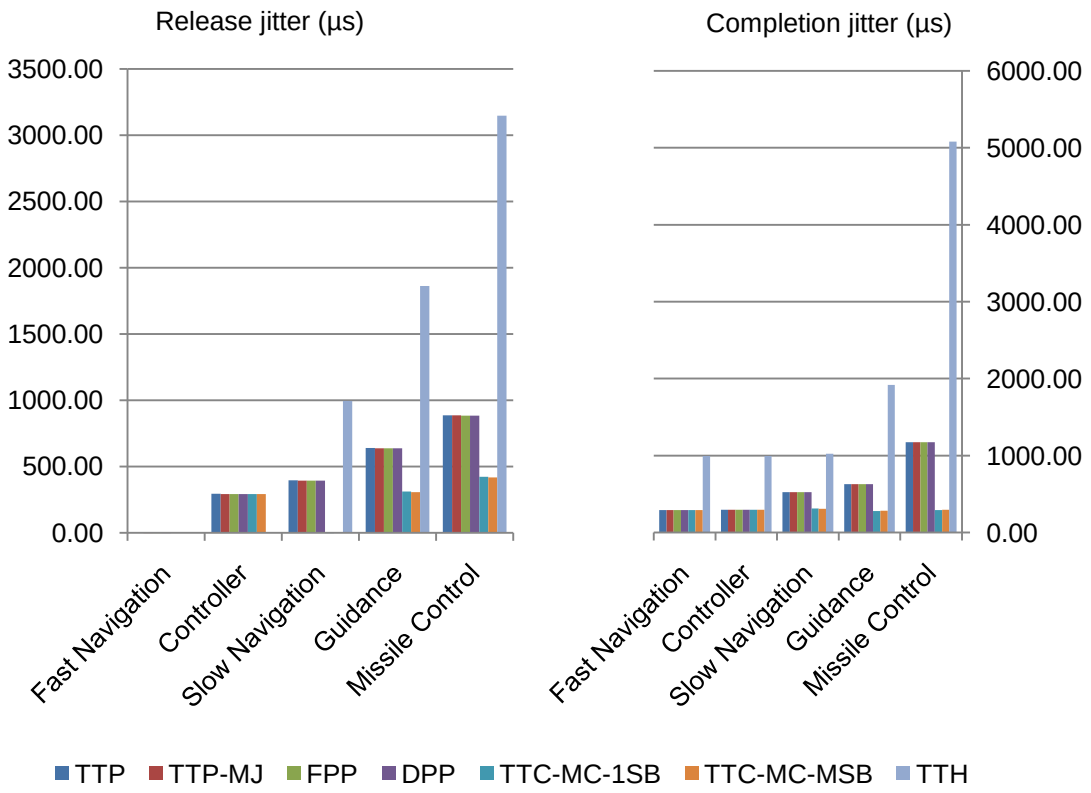

Figure 6.2: Jitter for TS-2

The highest release jitter is seen with TS-3 (Figure 6.3) due to the presence of non-harmonic relationships in the task periods; the high priority tasks exhibit approximately 18 ms of jitter – a significant increase over TS-1 (Figure 6.1) and TS-2 (Figure 6.2) which had jitter less than 0.5 ms; the lower priority tasks exhibit even higher release jitter, exceeding 750 ms in the case of the dynamic-priority implementation. The completion jitter is also quite high (Figure 6.3), particularly in the case of the fully pre-emptive implementations. In contrast, the multi-core schedulers are able to maintain the jitter levels at the previous levels or better since tasks executing on one core maintain harmonic relationships in

their periods and have fewer tasks with high completion jitter executing ahead of them.

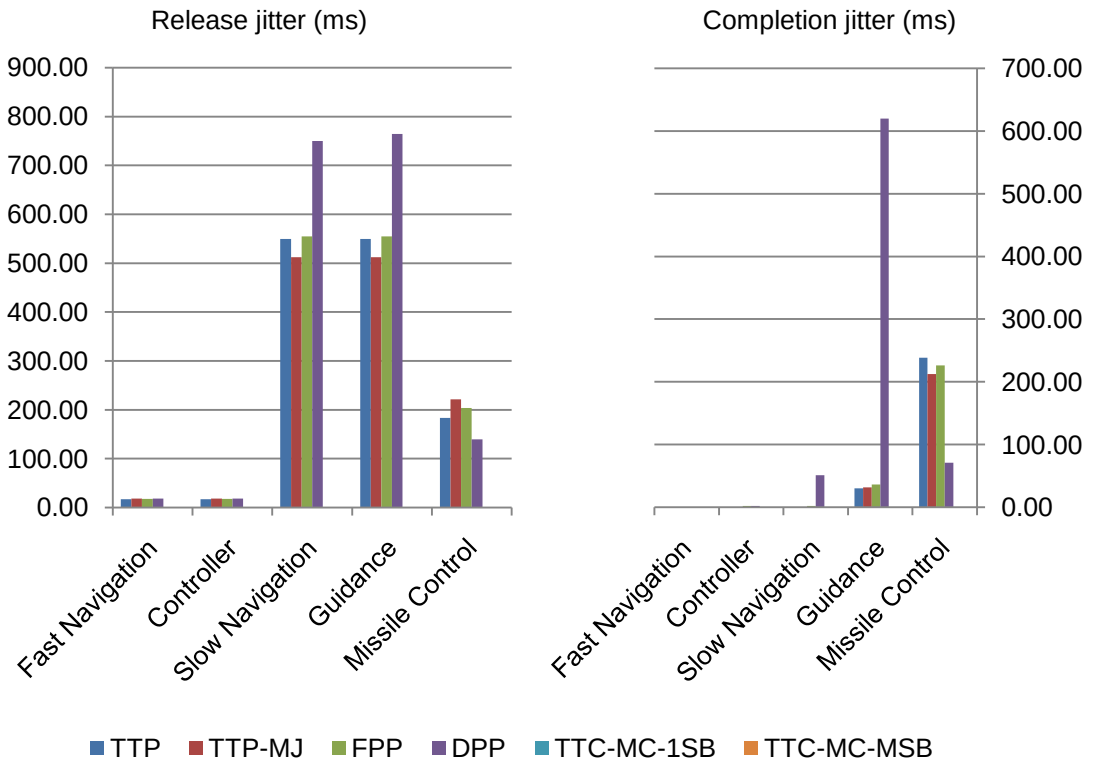

Figure 6.3: Jitter for TS-3

6.6 Overheads

The software overheads for each of the scheduler implementations can be seen in Figure 6.4; the overhead of the communications API has been added to the multi-core implementations. The overheads presented are the static usages from the application binary; depending on the scheduler, there may be greater demands on the stack. In the case of the pre-emptive systems, where tasks utilise the stack for context storage, the memory requirement increases quite severely, though this is not shown in the figure. In the case of FPP and DPP, context storage has to be reserved at the time of creation, whereas TTP allocates the storage as required. In the worst-case (where each priority level is successively pre-empted), TTP tends to FPP and DPP stack requirements. In

that respect, the latter two may be better due to the upfront exposure of memory requirements.

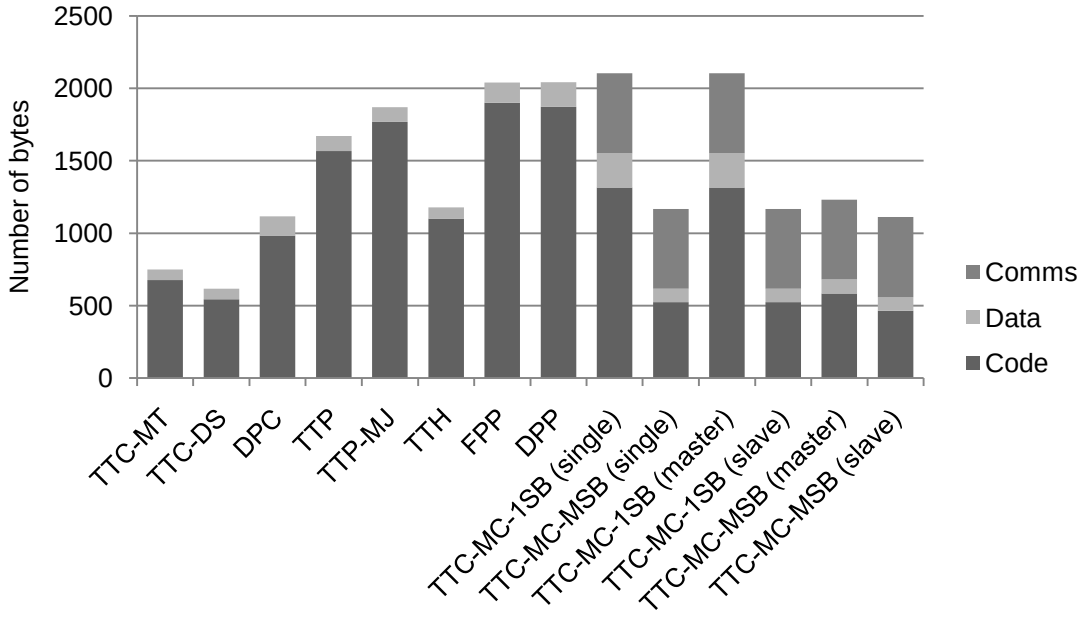

Figure 6.4: Software overhead of the scheduler implementations

The software overhead for the multi-core single builder schedulers includes the run queue overheads, particularly evident on the master which has to store all the run queues as shared areas. The size of the run queues is dependent on both the number of processors and the number of tasks – in this case study, a maximum of four processors were supported with a run queue size of eight each.

The run-time overhead of the scheduler implementations is similarly dependent on the task-set and is shown relative to the tick interval for TS-1 and TS-2 in Figure 6.6 and for TS-3 in Figure 6.6. In this case study, the task-sets only vary in: the necessity to pre-empt, which shows a minor increase for TS-2; and in the tick interval, which shows a large increase with the non-harmonic task-set, TS-3. These increases are not seen in the multi-core scheduler

implementations because of the simplicity of TTCA, because the tasks always execute co-operatively with harmonic relationships in their periods and because the cores have fewer tasks to execute.

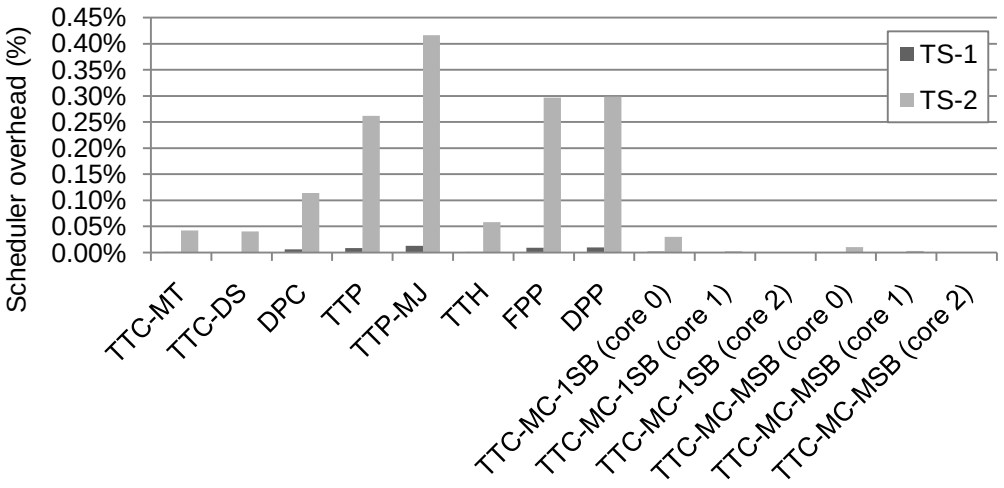

Figure 6.5: Run-time overhead of the scheduler implementations relative to the tick interval for TS-1 and TS-2

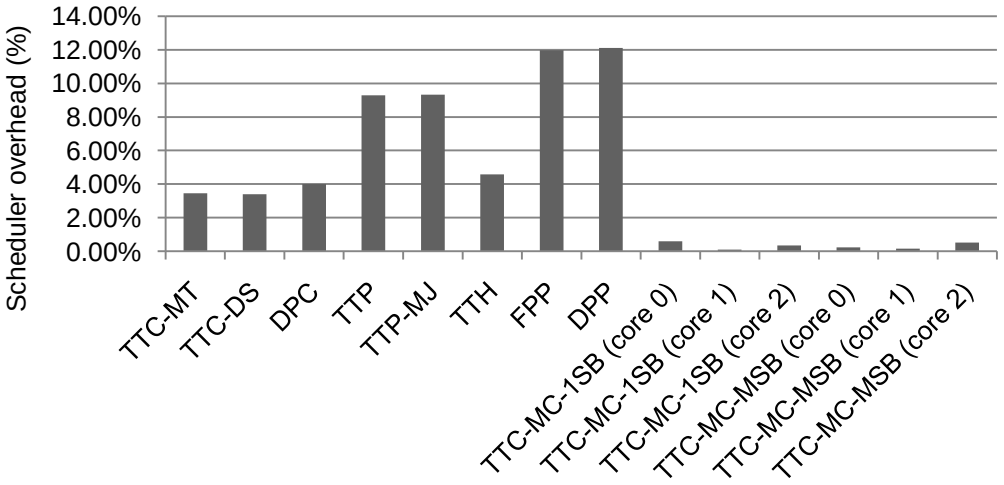

Figure 6.6: Run-time overhead of the scheduler implementations relative to the tick interval for TS-1

When the overheads in the multi-core scheduler implementations are viewed separately for all three task-sets (Figure 6.7), it can be seen that the scheduler takes up less overhead on the slave cores of the single builder implementation compared to the multiple builder one because the latter is running the scheduler

creation algorithm on all the cores. The master core on the single builder implementation, however, sees a greater overhead because it builds a schedule for all other cores.

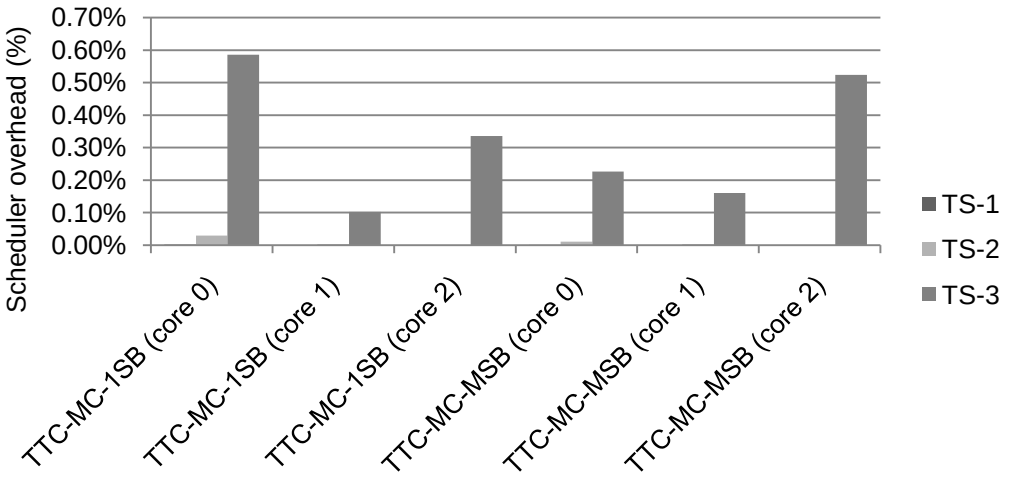

Figure 6.7: Run-time overhead of the multi-core scheduler implementations relative to the tick interval

It should be noted that in the case of the pre-emptive schedulers, the locking mechanisms were left unimplemented since they were not being used. These mechanisms, when in use, add to the jitter and overheads and the figures shown above are the baseline for the pre-emptive implementations.

6.7 Discussion

The PH-DS hardware platform provided moderate silicon and overhead reduction compared to the PH-MT, but was the worst platform for moving between the task-sets, incurring massive amounts of jitter with TS-2.

PH-MT fared better due to the provision of interrupt capabilities to scheduler implementations; of these, the co-operative implementations fared similarly to TTC-DS. The pre-emptive implementations were able to schedule TS-2, albeit with 36% and 75% increases in release and completion jitter respectively in the

lowest priority task; the highest priority tasks maintained their original run-time characteristics. The fixed priority schedulers imposed an additional complexity of assigning priorities, while the dynamic priority schedulers required verification that a correct priority decision would be made at run-time when the decision criteria proved ambiguous. Under the TTH implementation, not only did the highest priority tasks have to be identified, but the tasks had to be merged as well since only one high priority task was supported.

Under the non-harmonic task-set, TS-3, however, all the pre-emptive implementations fared poorly, resulting in jitter at least a couple of magnitudes greater than that in TS-1, even for the highest priority tasks. The TTH implementation fails completely with this task-set since the WCET of the pre-emptive task exceeds the tick interval.

Against these, the TTCA multi-core implementations were able to cope with the increased demand and the disruption in task period harmony, resulting in jitter no more than in the TS-1 case. Moving between task-sets was similar to the pre-emptive implementations in having to make a decision of assigning tasks to cores; however, the decision is conceptually simpler since blocking time does not have to be considered.

On the other hand, the multi-core implementations introduced a compile-time management complexity since each core required a different binary. In this case study, this requirement was handled via extensive use of GNU C pre-processor directives to ensure the right C header files were used, to selectively compile task code for the right core and to generate common code, such as scheduler

setup code. Fortunately, the process is repetitious and is a suitable target for simplification by automation.

6.8 Conclusions

This case study served to show the ease with which the multi-core TTCA implementations may be able to cope with changes in system requirements – changes that may introduce the long-task problem or that may introduce non-harmonic relationships between tasks' periods. A system deployed with dormant cores can allay the demands introduced by maintenance; with a multi-core TTCA, this can be performed with simplicity and reliability.

Chapter 7

A TTCA multi-core hardware implementation

7.1 Introduction

Chapter 3 introduced the time-triggered co-operative architecture (TTCA) and a hardware implementation of the same (HW-TTC) with the advantage of a massive reduction in scheduler overhead and a constant overhead between task dispatches. Chapter 5 described the scheduler implementations, TTC-MC-1SB and TTC-MC-MSB, based off of the table-free multi-rate TTCA implementation. These allowed for increased concurrency in the system so as to achieve a reduction in the latency of high frequency tasks when scheduled alongside task sequences with large execution times and for the separation of tasks with non-harmonic period relationships. However, both implementations display overheads dependent on the number of tasks in the system and a variable inter-task dispatch overhead.

This chapter, therefore, looks at further reducing the latency caused by the HW-TTC overheads to produce an implementation with zero overheads, HW-TTC-ZSO and at incorporating the techniques of TTC-MC-1SB and TTC-MC-MSB into HW-TTC-ZSO, hoping to achieve implementations, HW-TTC-ZSO-MC-1SB and HW-TTC-ZSO-MC-MSB, with zero overheads, low latency and high responsiveness despite the presence of long-tasks.

This chapter also examines the incorporation of release jitter reduction mechanisms into the hardware schedulers to ultimately yield an implementation

that not only possesses zero scheduler overhead but also zero release jitter for harmonic task-sets.

The next section will examine existing multi-core hardware schedulers and existing techniques aimed to reduce overheads and jitter in general scheduler implementations. Next, the changes made to HW-TTC to reduce overheads are detailed, followed by descriptions of the multi-core schedulers incorporated into hardware. Finally the jitter reduction mechanism is introduced before the different schedulers are evaluated on the F16 flight system simulation from Chapter 6.

7.2 Related work

Hardware techniques are gaining popularity particularly on multi-cores particularly due to the observation that smaller tasks ease the partitioning and allocation effort and improve performance; but, coincidently lead to an increase in the number of tasks which, in turn, increases the overheads in software scheduler implementations (Kumar et al. 2007; Själander et al. 2008). These techniques are also preferred to dedicating a core to scheduling in order to achieve lower power consumption and better silicon area usage (Gupta et al. 2007).

Hardware scheduling techniques immediately eliminate a portion or all of the overhead introduced by a software scheduler since the extra hardware acts as another processor running concurrently with the application processor. Simultaneously, the jitter in a scheduler can be greatly reduced by employing constant time algorithms to sort or search the run queues; and by servicing interrupts in separate processors so that jitter from context switching and ISR

processing are eliminated (Agron et al. 2004; Andrews et al. 2005). The reduction of jitter is also made easier in hardware design due to the exposure of the underlying timing model (Lee 2009).

Hardware kernel techniques have been pursued for dynamic task scheduling, for example, there have been attempts to decrease the overheads of software run queues by implementing key functionality in hardware (Lai et al. 2005; Kumar et al. 2007). However, while these approaches sped up dynamic load balancing, the overhead of checking for precedence fulfilments remained in software. This led to other studies that employed distributed hardware task management units to perform these checks in the background, with the aim of decreasing the start time of tasks with dependencies (Själander et al. 2008; Al-Kadi et al. 2009). Other approaches have aspired to lower power consumption (Gupta et al. 2007). Yet another, HW-RTOS, in the spirit of this thesis, improves the efficiency of the OS and the API support transparently to the application, but like the other aforementioned approaches, suffers in that a global scheduling approach is used (Nácul et al. 2007) which is unsuitable for hard real-time systems (Section 2.8.4).

Under a partitioned approach, any of a number of single-core hardware schedulers (Stärner et al. 1996; Kohout et al. 2003; Kuacharoen et al. 2003; Andrews et al. 2004) could be employed, however these are aimed at achieving performance rather than predictability and present a programming model to the developer that is considerably more complex and less predictable than the intended co-operative approach. Hence, it is desirable to expand existing TTCA implementations in order to achieve the required reduction in overheads.

One such implementation was presented in Section 3.6.5 – the hardware table-free multi-rate TTCA implementation, HW-TTC, that reduces almost all scheduler overhead. HW-TTC forms an ideal base for the exploration of the hardware designs of the multi-core TTCA implementations since all three have originated from the same basic implementation. However, a couple of deficiencies remain in HW-TTC:

- The requirement for the “endtask” instruction to indicate the end of the task results in unnecessary overhead between task dispatches which gets compounded with each dispatch. The lack of compiler support to verify the presence of this instruction may also create unexpected bugs.
- The dispatcher is designed to run tasks as close together as possible so as to reduce overheads. This makes tasks susceptible to increased release jitter due to the execution jitter in the prior tasks (Section 2.6.4.1).

Section 7.4 will address the first criticism, while Section 7.7 will outline a method of attaining zero release jitter.

7.3 HW-TTC support for precise exceptions

Prior to discussing the changes made to HW-TTC, it is necessary to briefly mention that HW-TTC maintains support for precise exceptions (Section 5.3.3.3) by dequeuing a task from the run queue only when the “end task” instruction is in the third processing stage – that is, only when an error cannot be generated by instructions in a previous task. This is done even though the address is already used in the first stage as an indication of where to fetch instructions from (Figure 7.1).

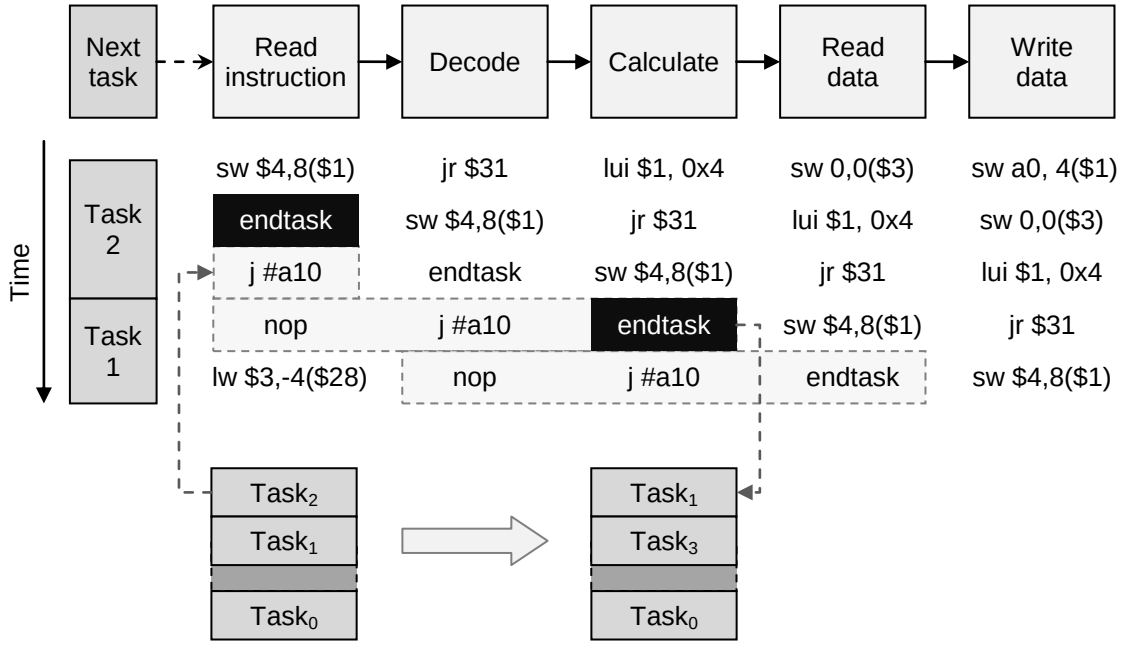

Figure 7.1: The effect of the endtask instruction on the run queues and instruction execution

Figure 7.1 also shows the inter-task dispatch overhead. The first instruction of Task 2 is "`lw $3,-4($28)`" and the last instruction of the previous task is "`sw $4,8($1)`", yet there are three instructions in between: `endtask`, a jump to the actual function and the no-operation in the delay slot for this jump. As mentioned above, this overhead is due to the `endtask` instruction which only serves as a marker and necessitates an assembly language wrapping of tasks written in a higher level language.

7.4 A hardware TTCA implementation with zero overheads

The solution proposed to eliminate this overhead is to overload an instruction that is always inserted by the compiler, with the duty of marking the end of a task and in doing so, eliminate the need for "endtask" and the wrapper. A key observation about tasks under the TTCA implementations is that they are always written as run-to-completion routines and so, the compiler can be guaranteed to always use a *return-to-caller* instruction at the end of the task.

In MIPS I, the *return-to-caller* instruction has the mnemonic "`jr`", takes a register number as the operand and causes an unconditional jump to the address in the register with that number. Under MIPS conventions, the return address for a function call is stored in register 31 and compilers generate the "return to caller" instruction as "`jr $31`". Since the register number is fixed, the contents of this register can be used to distinguish between different "return to caller" requests; that is, whether the function making the request is a task or not, and so the *return-to-caller* instruction can be used to indicate the end of a task to hardware.

The revisions to produce HW-TTC-ZSO were made as follows: general purpose register 31 is reset to the value zero when the processor is interrupted; the processor sets the program counter to whatever value a "`jr`" instruction has read from its register as normal, unless that value is zero (a "task-jr"), in which case the program counter is instead set to the address of the next task; and, the "task-jr" sends the "end task" signal to the dispatch component in the WB stage. This is illustrated in Figure 7.2.

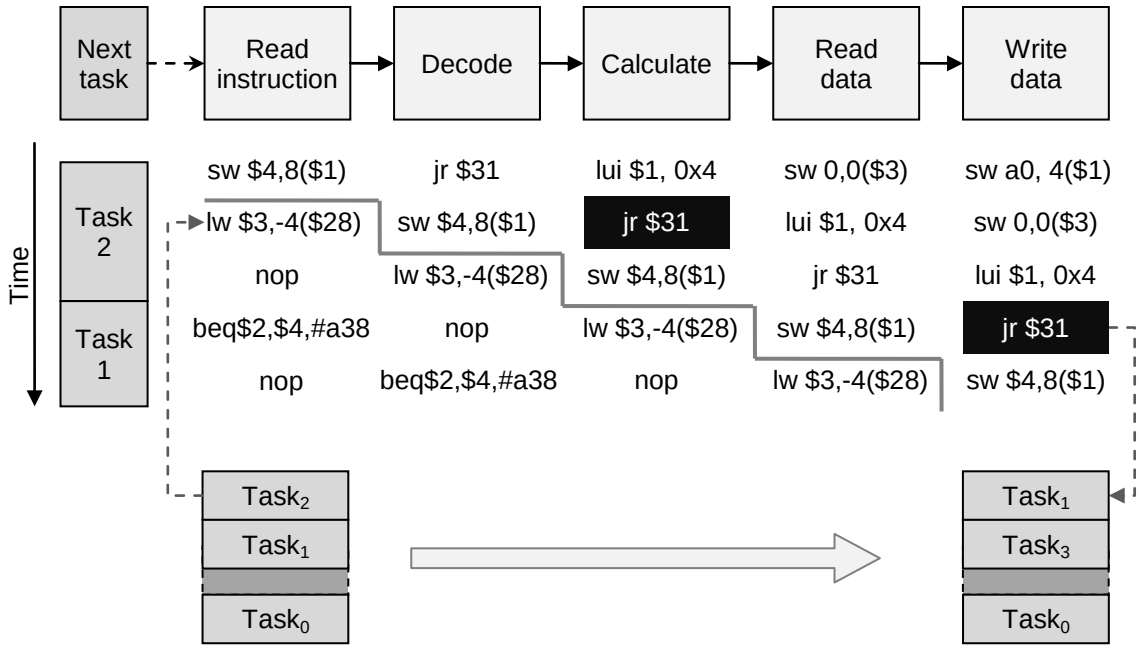

Figure 7.2: The effect of overloading jr with the work of endtask

The `jr` instruction can generate an “end task” only at the fifth stage, since the instruction in its delay slot may generate an exception in the fourth stage (“`sw $4,8($1)`” in this case). The distinction between tasks is shown with gray borders in Figure 7.2 and true back-to-back execution can be observed when compared with Figure 7.1.

A sample execution sequence is as follows: as before, the processor is woken from sleep using an interrupt and is provided the address of the first task. When the task requests to return to caller, the processor starts reading instructions from the next task. When the last task requests to return to the caller, the processor starts to read from an undefined next task address. However, the scheduler instructs the core to insert NOPs into the pipeline until the last instruction of the last task has passed through the last pipeline stage whereupon the processor is requested to sleep.

Another change was the switch of task storage from registers to SRAM, allowing for a maximum of 128 tasks compared to the original 8. This necessitates two cycles for each task during the build cycle, increasing the schedule build time for 8 tasks from 8 cycles to 16 cycles. As in the original design (Section 3.6.5), this latency remains invisible to an application, but increases the length requirement on the run queue (page 3-18) as $\gamma(\theta)$ is doubled (Equation 7.1). Equation 7.1 takes into account the five cycles that elapse from fetching the `jr` instruction to actually sending the “end task” signal to the dispatch component.

$$\gamma(\theta) = \frac{2 * number\ of\ tasks + 5}{clock\ rate\ of\ update\ component} \tag{7.1}$$

7.5 The hardware multiple schedule builders implementation

This scheduler implementation is a straightforward extension of HW-TTC-ZSO. Similar to the pure software version, it consists of duplicating HW-TTC-ZSO for each core with one exception: the update component in each HW-TTC-ZSO is triggered by the timer from the timing master (Figure 7.3).

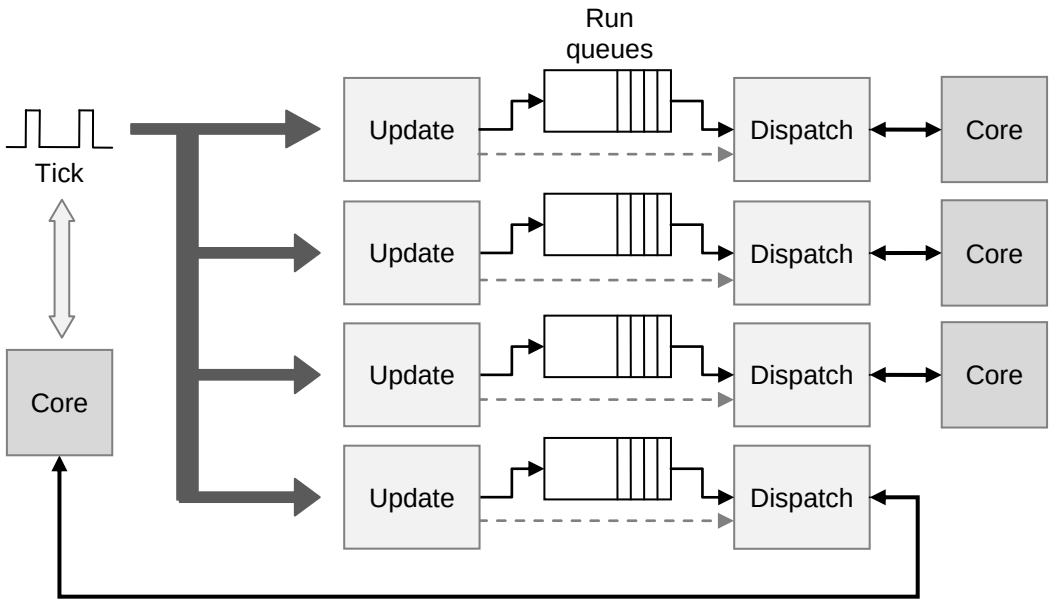

Figure 7.3: Functional overview of the hardware multi-core multiple schedule builder scheduler

To maintain predictability, the update components all spin for the same number of cycles to look for tasks to insert into their queues, even if the maximum number of tasks supported by each one is different. In order to ensure that all dispatches start at the same time on a N-core device, an additional user-specified number of cycles $\delta(n)$ are also added after the schedule has been built from task-set $\pi(n)$ for the n^{th} core ($0 \leq n < N$) to allow for communication latency, increasing $\gamma(\theta)$ (Equation 7.2) and the required run queue length (page 3-18) for that core. As before (Equation 7.1), the five cycles spent from fetching the `jr` instruction to signalling “end task” are factored in.

$$\gamma(\theta) = \frac{2 * max\{|\pi(x)|, 0 \leq x < N\} + \delta(n) + 5}{clock\ rate\ of\ update\ component_n} \quad for\ the\ n^{th} core \qquad (7.2)$$

7.6 The hardware single schedule builder implementation

The hardware implementation of TTC-ZSO-MC-1SB deviates in its storage of the run queues, opting for hardware versions instead of simulated versions in shared memory in order to avoid contention between the core and the communication manager. Each core has its own run queue as an asynchronous FIFO and a dispatch component that reads from this queue; the scheduling core alone has an update component (Figure 7.4).

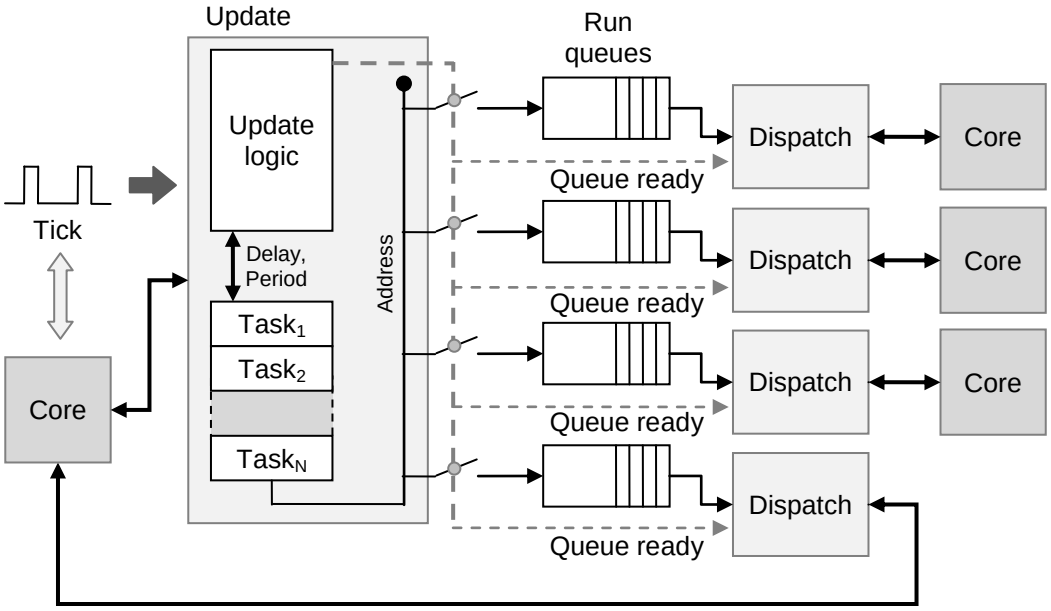

Figure 7.4: Functional overview of the hardware multi-core single schedule builder scheduler

The single update component creates the entire schedule as before (Section 3.6.5) and pushes task addresses into the proper queues. Once the schedule has been created, it then sends the "queue ready" signal on an asynchronous line to all the dispatch components at the same time. To allow for latency in the FIFOs on an N-core, this signal is sent a user-specified number of cycles δ after the queues have been built for each task-set $\pi(n)$ of the n^{th} core ($0 \leq n < N$),

resulting in a uniform $\gamma(\theta)$ (Equation 7.3) and required run queue length (page 3-18) for each core. As before (Equation 7.1), the five cycles spent from fetching the `jr` instruction to signalling “end task” are factored in. To be effective, δ must be at least as large as the number of cycles of asynchronous delay introduced by the FIFO.

$$\gamma(\theta) = \frac{2 * max\{|\pi(x)|, 0 \leq x < N\} + \delta + 5}{clock\ rate\ of\ update\ component}\ \ for\ all\ cores \tag{7.3}$$

7.7 A pure hardware sandwich delay mechanism (-HSD)

As a solution to release jitter in TTCA, Section 4.4.3.2 has already touched upon the sandwich delay mechanism which for its precise timing requires a hardware timer to be *set up* and a *delay* until this timer overflows. Section 3.6.4 explored a hardware encapsulation of the *delay* under the TTC-SHD scheduler. This scheduler provides a solution with low power consumption but at the cost of extra overhead and the need to maintain an accurate measurement of the execution time of the *set up* so as to compensate for it.

The sandwich delay mechanism can be fully incorporated into the hardware schedulers without many fundamental changes since the WCET (in cycles) can be provided with the other task parameters, the hardware is already in control of when a task starts and the required run queue lengths already assume the worst-case (page 3-18 and Equations 7.1, 7.2 and 7.3).

To obtain this functionality, the update component must supply a tuple of the address and WCET cycles in the ready queue instead of just the address: the general principle is to initialise a counter with the WCET every time a task

executes and to decrement the counter until it reaches zero (Figure 7.5). If the task finishes before the counter reaches zero, the core is put to sleep instead and woken up again when the counter *does* reach zero. This action is unnecessary and so, omitted, for the last task in a run.

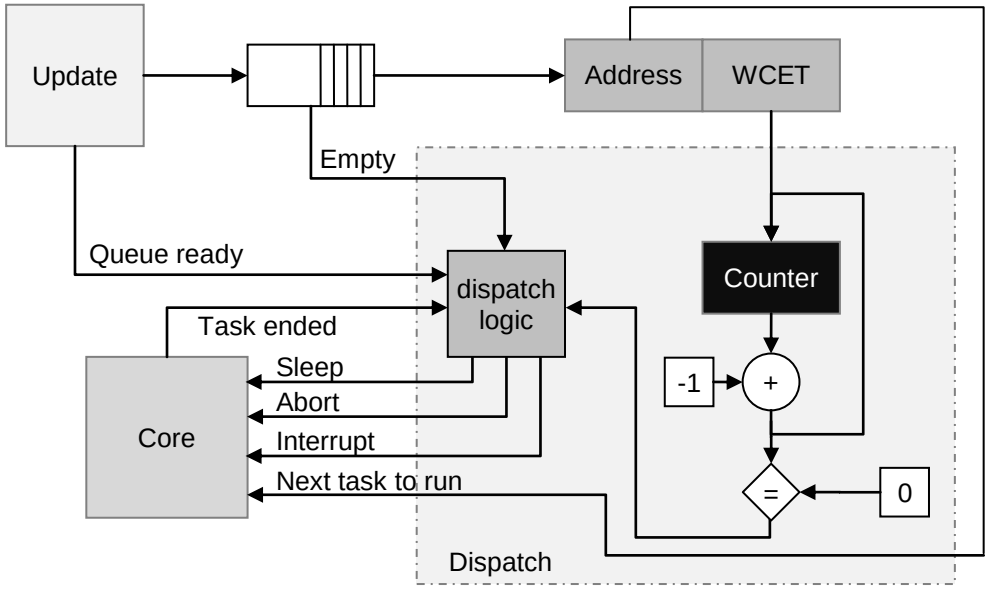

Figure 7.5: Changes made to the dispatch component to support sandwich delays

Waking the core up from sleep requires the use of the interrupt mechanism which introduces a little delay as the processor warms up. This can be countered by factoring the delay into the counter value, i.e. subtracting the number of warm-up cycles from the counter's initialisation value. This technique is suitable for the first task as it will always experience the delay. However, the subsequent tasks will experience the delay only if their execution times are sufficiently smaller than their WCETs. The very use, then, of the sandwich mechanism might introduce jitter depending on whether a warm-up is necessary.

To avoid this, it was observed that the warm-up is required since the processor aborts the instructions in the first three pipeline stages (Section 5.3.3.3). This is

necessary for the first task since those instructions are undefined and cannot be allowed to change the processor state; however in the case of subsequent tasks, those instructions are strongly defined as belonging to the next task and need not be aborted. Thus for the subsequent tasks, another signal was added between the dispatch component and the core to indicate the necessity of clearing the pipeline on an exception.

As an aside, the sandwich delay mechanism also has the ability to easily create precisely timed tasks for the purposes of maintaining precedence constraints (Section 5.6.2), with the tasks using a minimum of code space (only two instructions are required: a “return to caller” instruction and a NOP for its branch delay slot).

7.8 Evaluation

The developments in this chapter were aimed at completely eliminating the overheads of a TTCA implementation and the release jitter exhibited by such an implementation. To demonstrate the effectiveness of the solution, the various schedulers were evaluated with the F16 flight system case study in Chapter 5: TS-1 on the single-core hardware schedulers, TS-2 on dual-core schedulers and TS-3 on triple-core schedulers. The results are shown in the subsequent sections alongside those from the co-operative schedulers in Chapter 5 and the co-operative schedulers with sandwich delays enabled (-SSD), for comparison.

7.8.1 Release and completion jitter

The release and completion jitter for the five tasks across the different schedulers and task-sets can be seen in Figure 7.6, Figure 7.7 and Figure 7.8. The jitter measurements for TS-1 on the single-core schedulers in Figure 7.6

immediately indicate that the translation of the schedulers into hardware provides no noticeable advantage in reducing jitter. However, a striking difference appears with the –HSD schedulers, where the release jitter drops to zero (indicated by gaps in the charts) for all the tasks and the completion jitter drops to zero for all tasks that do not execute last in a tick interval; in this case, all the tasks have zero completion jitter because a heartbeat LED task executes last on the first core. As mentioned in Section 7.7, this behaviour is by design as there is no task executing after the last task that requires zero release jitter.

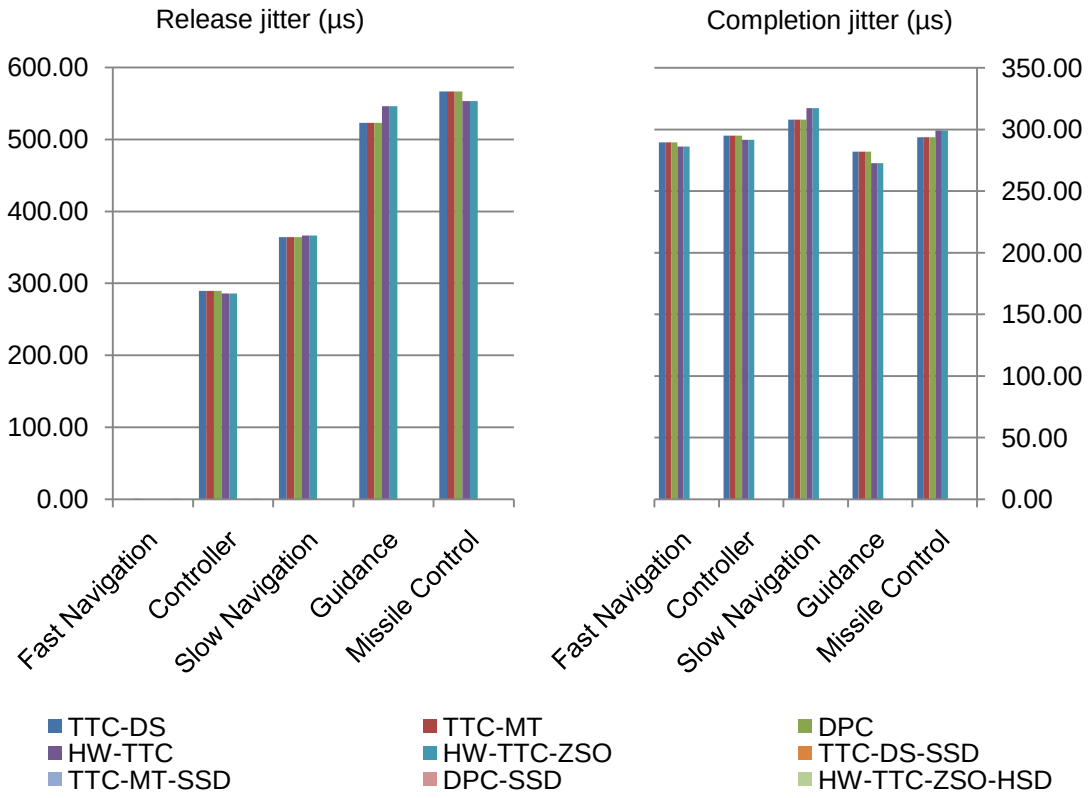

Figure 7.6: Jitter for TS-1

Figure 7.7 shows TS-2 on various dual-core schedulers: *Fast Navigation* and *Controller* on the second core and all other tasks on the first core. As expected, the three tasks that execute after other tasks show a non-zero release jitter due to the completion jitter in the previously executing task; and, as before, the –HSD schedulers are able to remove release jitter completely by removing the

completion jitter in the previous tasks. The completion jitter remains almost the same as in Figure 7.6 except for *Controller* which is the last task executing on the second core; the heartbeat LED task still executes last on the first core.

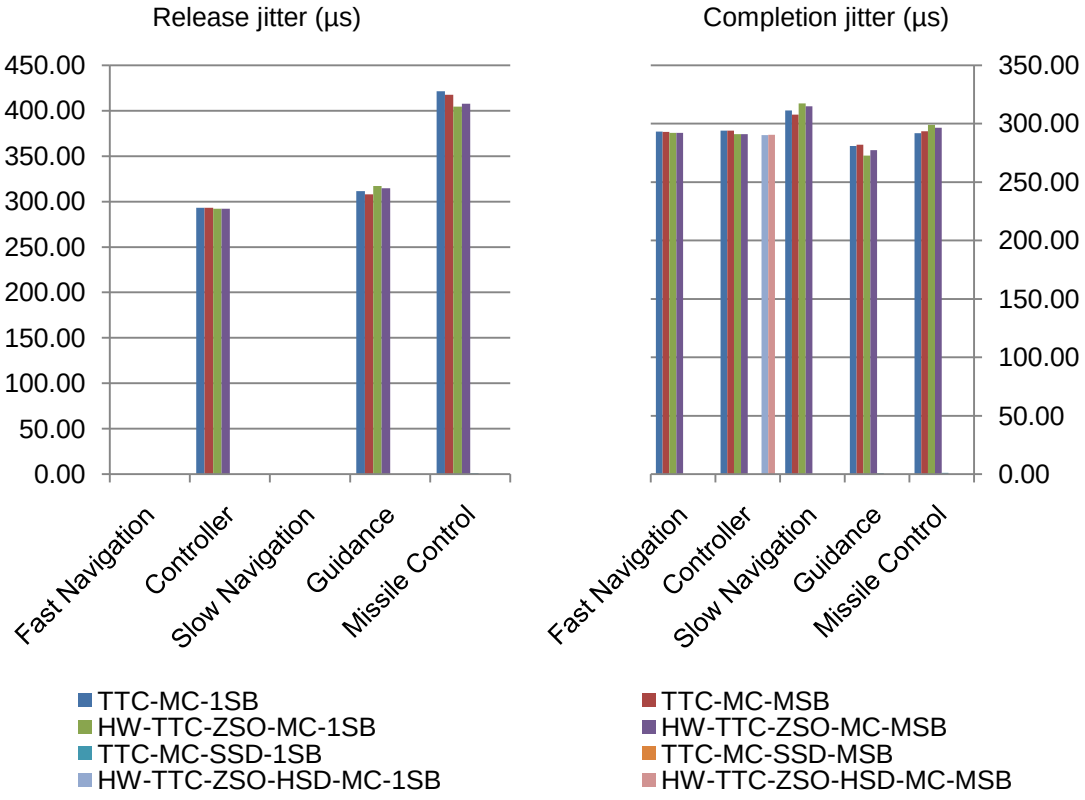

Figure 7.7: Jitter for TS-2

Despite their advantageous jitter reducing effects on TS-1 and TS-2, the –HSD schedulers are ineffective in doing the same for the non-harmonic (Section 4.5) task-set TS-3, as in that case, the jitter is not caused by execution jitter in the previous tasks, but by variation in the task execution sequence within the hyperperiod. Hence, TS-3 was executed on three cores as seen in Figure 7.8: *Missile Control* on the third core, *Fast Navigation* and *Controller* on the second core and all other tasks on the first core. As before, tasks executing after other tasks exhibit non-zero release jitter, the –HSD schedulers are able to remove release jitter completely and completion jitter remains for the tasks that execute last on a core.

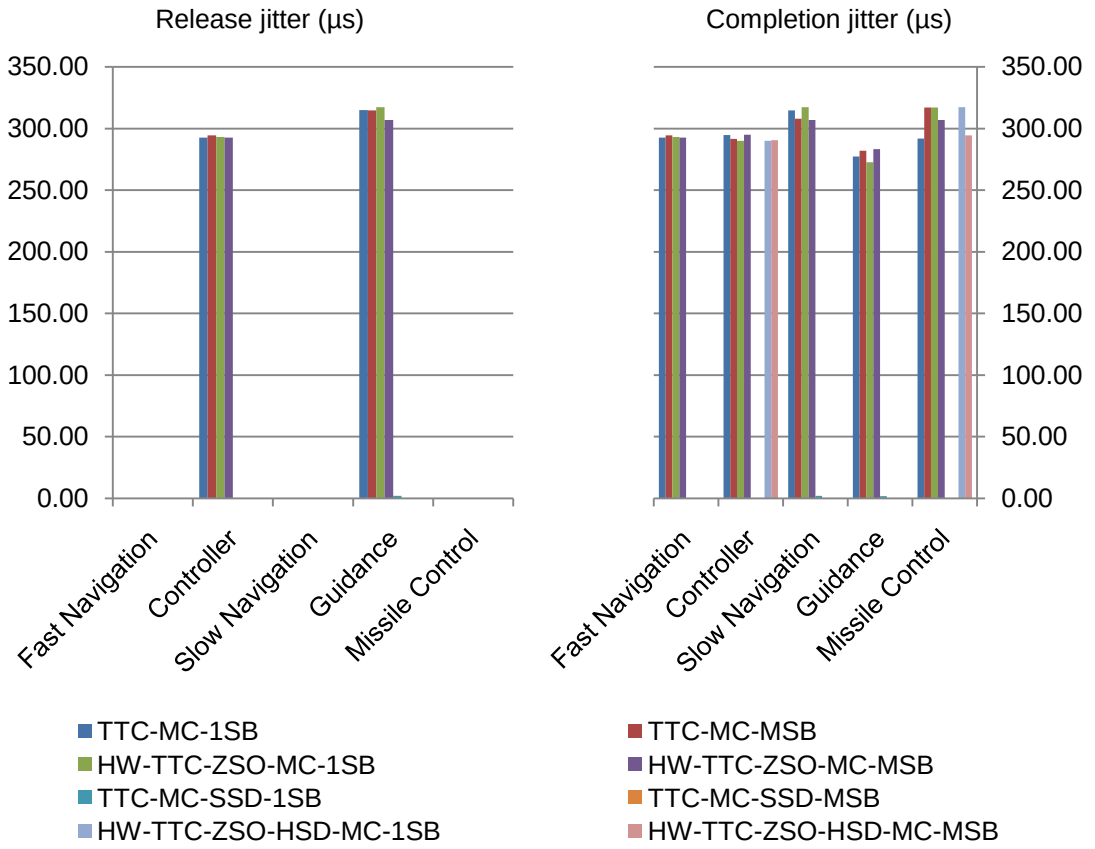

Figure 7.8: Jitter for TS-3

7.8.2 Overheads

The software overheads for the schedulers can be seen in Figure 7.9. Unlike Chapter 5, the overhead of the communications API has been omitted from the multi-core implementations since the schedulers make no use of the API, though it is still present for the initialisation procedure. The run-time overheads have not been presented, since the schedulers developed in this chapter display none. For the case study, this has the benefit of being able to make use of all processing power in order to improve the missile control performance. For example: the missile control task could be assigned a period of 1.79 seconds, bringing the total system utilisation to 0.999 without any adverse effects.

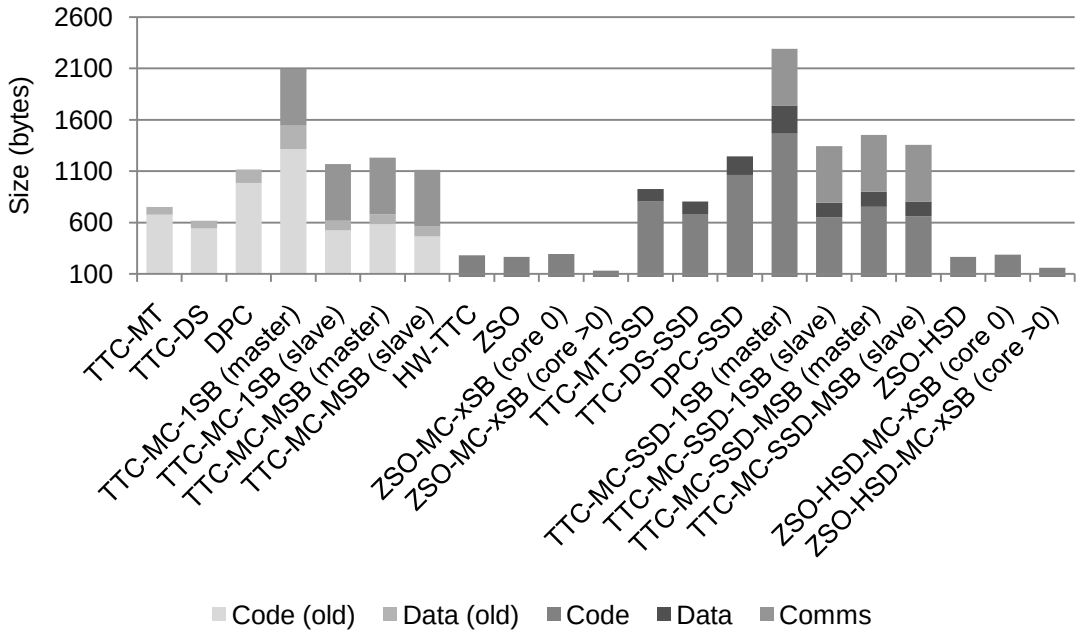

Figure 7.9: Software overhead of the scheduler implementations

The reduction in code size is dramatic, though not unexpected since the scheduler functionality has been moved from software into the hardware. Figure 7.10 shows the hardware utilised, as the number of slices occupied for solely one function, when the hardware scheduler is implemented in a configuration with one core possessing 32 Kb code memory, 32 Kb data memory, three timers and one GPIO. The figure shows the utilisation for each combination of the inclusion of the overhead reduction and the jitter reduction mechanisms alongside results from Figure 5.14. The figure illustrates that a hardware scheduler results in at least 14% increased hardware consumption over an unpredictable PH core and is very similar to that consumed by hardware including a predictable PH-MT (Section 5.7.1.1). The overhead reduction mechanism adds negligible hardware cost to the base hardware scheduler implementation, while the jitter reduction mechanism adds approximately 5%, close to the combined cost of approximately 6%.

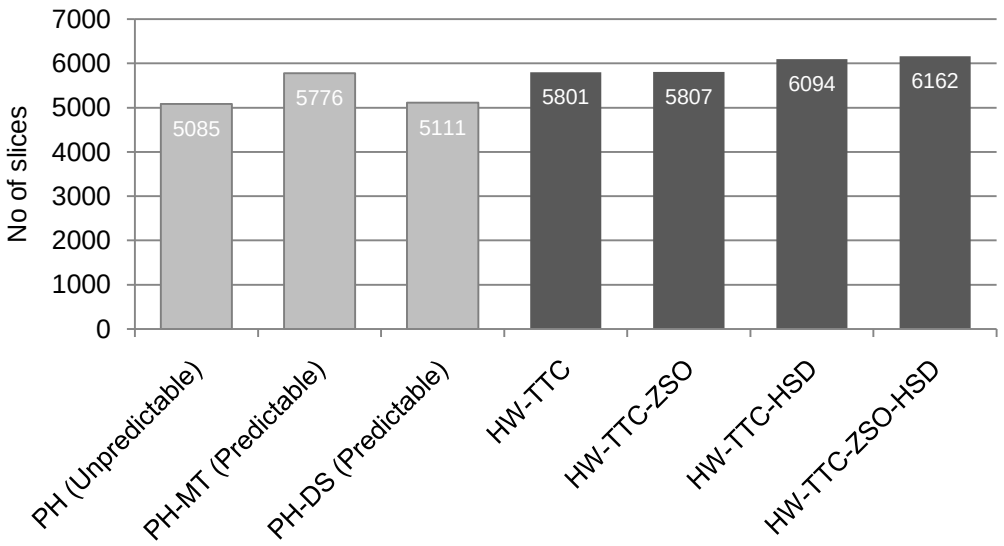

Figure 7.10: Hardware utilisation when using a hardware scheduler with and without the overhead and jitter reduction mechanisms

It is worth noting that the increase in the hardware to support a scheduler is much less than the result of adding another core dedicated to scheduling (Section 5.7.1.1), demonstrating the silicon area advantage of a hardware scheduler component.

The hardware utilisation for a dual-core system with the multi-core versions of HW-TTC-ZSO with varying combinations of single and multiple builders, the jitter reduction mechanism and the capability of inter-core communication is shown in Figure 7.11. Each core has been configured with 32 Kb code memory, 8 Kb data memory, one GPIO and at least one timer – core 0 has three timers. The scaled relative cost for communication has increased from approximately 45% (Figure 5.15) to approximately 54% due to the need to add a second channel to the communication controller on which local requests to switch buffers may be handled: one from the core (required by the initialisation routine) and one from the scheduler. On average, the multiple scheduler builders increase the hardware cost by less than 1% despite the savings from moving

from an asynchronous to a synchronous run queue; this is largely due to the schedule building mechanism and the increased data path for task data (i.e. periods, phases, etc. in addition to the address in memory) for a core building its own schedule.

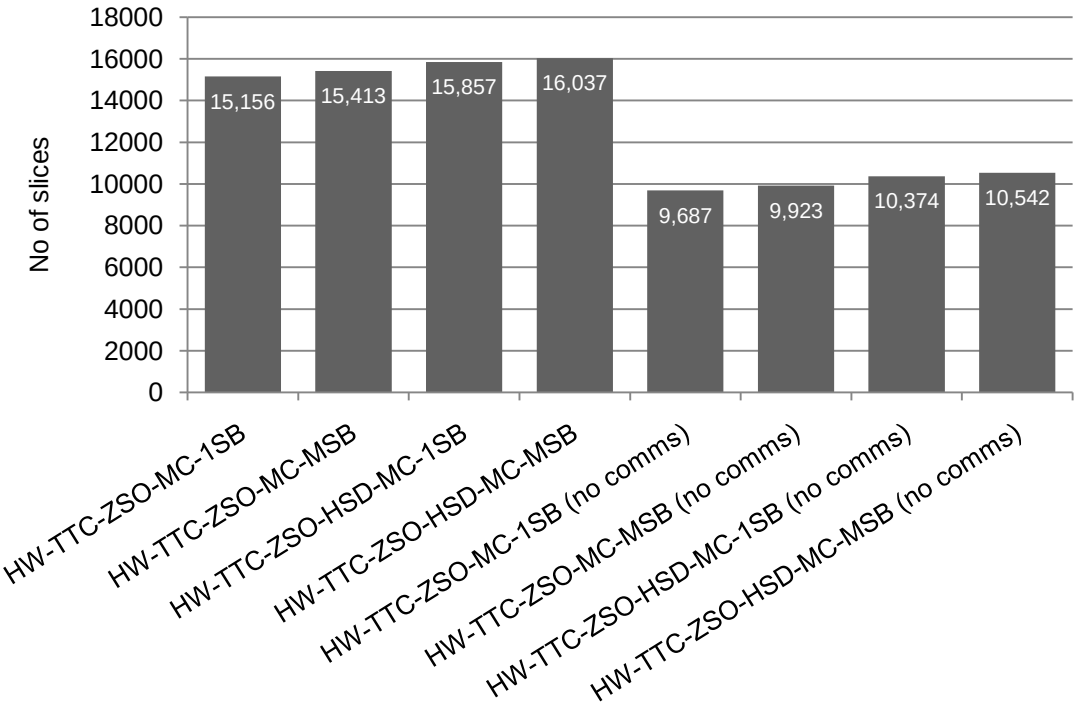

Figure 7.11: Hardware utilisation when using a multi-core hardware scheduler with and without the jitter reduction mechanism and inter-core communication

7.8.3 Simulation

To illustrate the operation of the scheduler when simulated, a very simple task-set was created with two empty tasks (tasks A and B) and a small heartbeat LED task (task C). The operation of the scheduler under HW-TTC and HW-TTC-ZSO can be seen in Figure 7.12 and Figure 7.13 respectively.

Being empty, tasks A and B have only two instructions, the return-to-caller instruction and its delay slot instruction (a NOP). As might be expected under HW-TTC, Figure 7.12 shows a delay of 200 ns execution time for either task, a delay that equates to 5 cycles at the operating frequency of 25 MHz. Since the

PH cores execute the tasks' instructions in a single-cycle, the extra three cycles are the overhead of the tasks wrappers. In the case of the last task, the "Insert NOPs" command removes the "endtask" instruction from the pipeline and hence it is not visible in the last three stages.

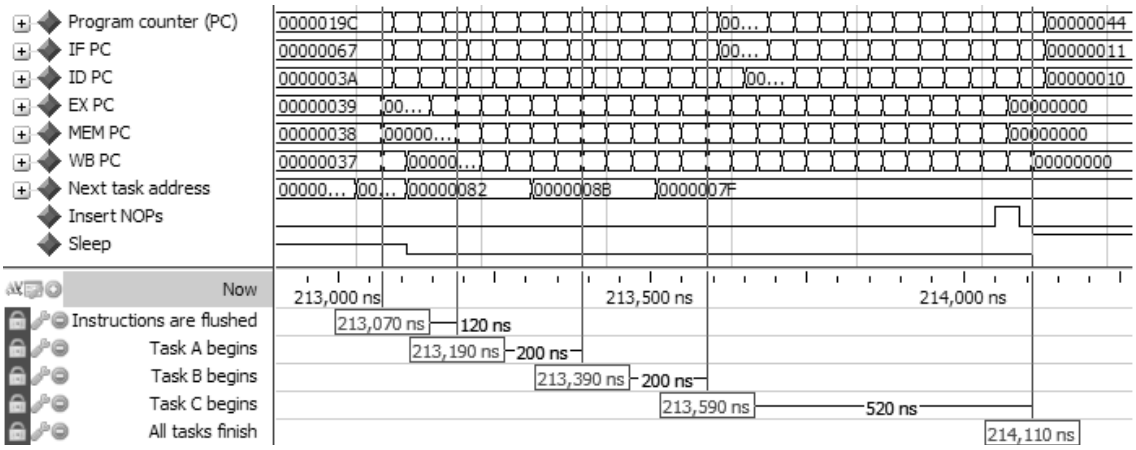

Figure 7.12: Sample execution of three tasks under HW-TTC

On the other hand, Figure 7.13 gets rid of the wrappers completely with the ZSO technique, such that tasks A and B execute for exactly their duration of two cycles or 80 ns. It should be noted that in both cases, the processor is put to sleep only when the last instruction of the last task has completed the last pipeline stage.

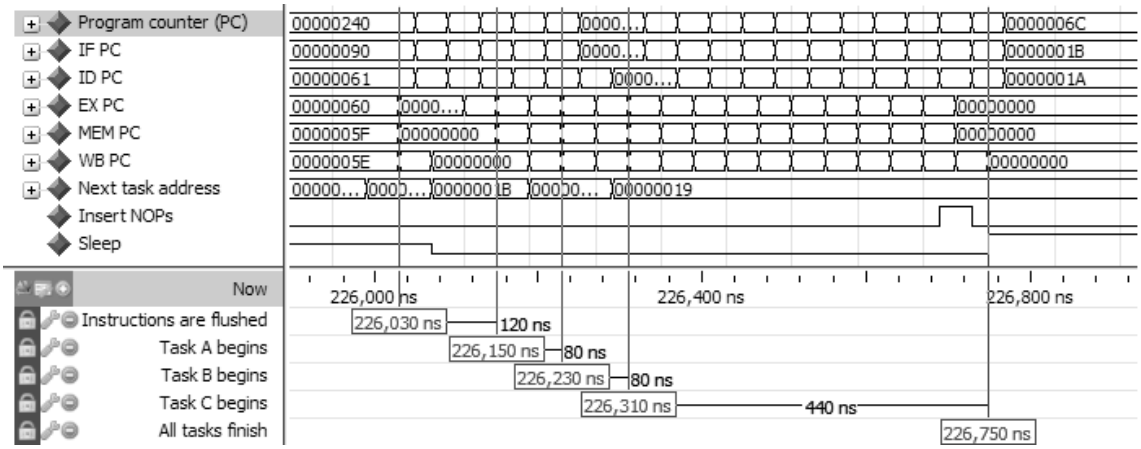

Figure 7.13: Sample execution under HW-TTC-ZSO

Figure 7.14 shows the schedule creation for a dual-core system under HW-TTC-ZSO-MC-1SB. From the figure, a total of 19½ cycles can be observed to elapse before the scheduler begins to dispatch tasks: ½ a cycle is spent to recognise the timer interrupt, 16 to update the run queues and 3 to give sufficient clearance for communication latency. This result was corroborated by the measurement of a consistent 760 ns interval from timer overflow to core wakeup on the hardware. This delay is greater than that for HW-TTC, but imposes no additional overhead or jitter as it stays constant over the uptime of the system, provided the delay for communication latency is only modified at initialisation.

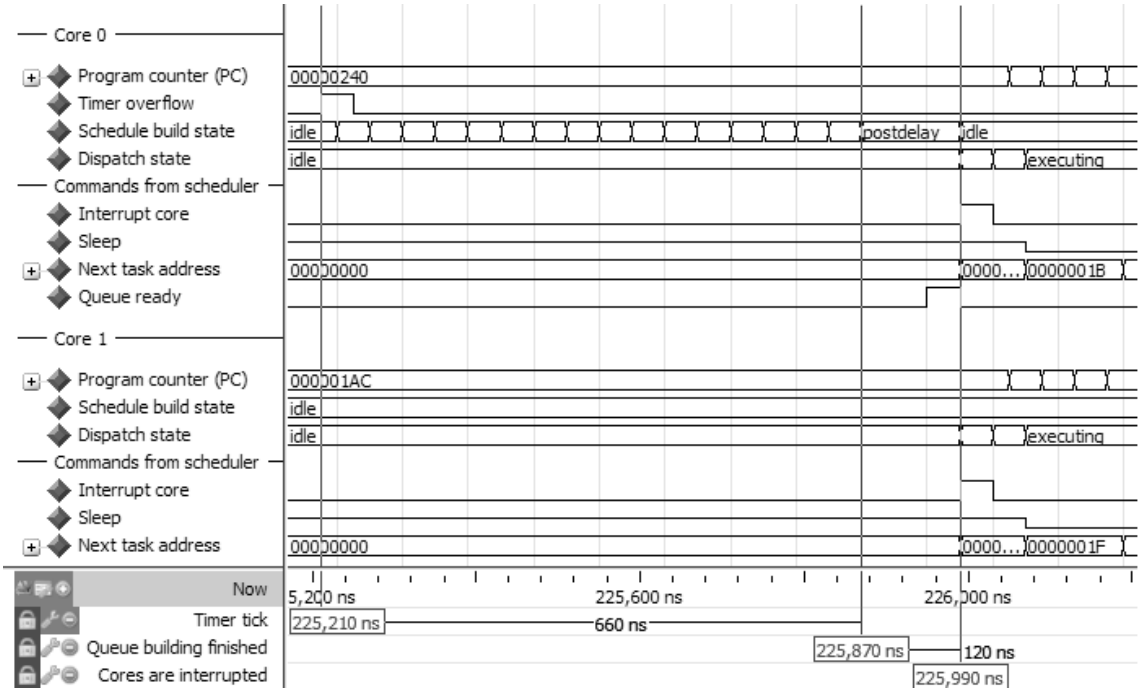

Figure 7.14: Schedule creation for a dual-core HW-TTC-ZSO-MC-1SB

7.9 Conclusions

This chapter has extended the multi-core schedulers developed in the previous chapter to yield implementations that exhibit zero scheduler run-time overhead

and zero release jitter for harmonic task-sets. The zero run-time overhead has been achieved by tapping into the concurrency provided by a hardware scheduler without affecting the run-time properties of the tasks such as release or completion jitter. The latter two were reduced by incorporating the sandwich delay mechanism (Section 4.4.3.2) completely into hardware. This mechanism utilises the spare capacity that has already been reserved in the TTCA implementation during design time and requires no special consideration by a scheduling algorithm.

Chapter 8

Case study: The BR715 Engine Controller

8.1 Introduction

The case study described in Chapter 6 investigated the deployment of a single-processor design on a multi-core platform as a means of aiding future development and maintenance. However, this route is only available to systems in development that are yet to be deployed; therefore, this chapter uses the case of an existing electronic engine controller system to study the migration of an existing system to the multi-core platform.

The system under study has been borrowed from the case study in (Bate 1998) which similarly studied a migration from one scheduler architecture to another. The work in the original study was eventually adopted by Rolls-Royce for use on an actual engine in an aircraft. For confidentiality, the original study changed or omitted some of the task properties, such as names and purpose; however, important timing requirements were left as-is. Some of the content of the original study is reproduced in Appendix C.

In subsequent sections, the reasons for moving the system to a multi-core design are first outlined, followed by a description of previous work done in formulating algorithms capable of building schedules for the multi-core design. Next, an algorithm for partitioning and scheduling the tasks on a multi-core system is presented, followed by the results of using the algorithm on a system running HW-TTC-ZSO-MC-MSB.

8.2 Technical details

For this case study, the tasks shown in Table C.1 (Appendix C) were implemented as execution-jitter-free dummy tasks using a hardware timer, with the execution times either at or a maximum of 120 µs below the WCET.

The large number of tasks in Table C.1 and the complex transactions requirements in Figure C.4 are indicative that the system is non-trivial. With a total processor utilisation of 84.3% for the tasks alone, there is a tight timing margin which complicates the creation of a system schedule and the maintenance of the system in the long run. Such requirements may then necessitate costly system redesigns and/or alterations of timing requirements.

In the face of these, it may prove more cost-effective to move to a multi-core design instead. As seen in earlier chapters, such a move is able to accommodate maintenance issues, however the problem of creating the system schedule for the multi-core remains.

8.3 Previous work

As outlined in Section 2.8.4, no partitioning algorithms have been specifically defined for TTCA, though many of the heuristic algorithms can be extracted from existing partitioning algorithms and applied to the TTCA by simply switching the feasibility checks performed (Section 3.3) to validate a task assignment to a core. While the work in (Monot et al. 2010) deals with the cyclic executive, it is not directly applicable since it was designed for integrating a multi-processor design into a multi-core design and expects many non-communicating groups of communicating tasks; the technique under

consideration, on the other hand, targets a single processor design and non-communicating groups of tasks are expected to be scarce.

Another assignment criterion is to assign tasks using the same peripheral to the same core. A soft multi-core gives some manoeuvring ground in this aspect in that peripherals can be connected to cores as dictated by the assigned task-set. However, the matter gets more complicated if a peripheral needed by a task has already been assigned to a core but the task itself cannot be feasibly assigned to that core. One solution is to perform an initial clustering of tasks so that clusters do not share resources – the heuristics then work on clusters instead of tasks (Monot et al. 2010). Another solution is to create gateway tasks that have sole access to peripherals and that receive instructions on how to manipulate the peripherals from tasks on other cores (Audsley et al. 1993). Chapter 9 will explore a third alternative.

8.4 A static schedule creation algorithm

A static schedule creation algorithm must first assign a scheduling algorithm per core, then partition tasks amongst cores and finally create a schedule for each core, honouring all constraints. For the multi-core system developed in this thesis, the scheduling algorithm is irrevocably an implementation of TTCA.

As explained in Sections 2.8.1 and 2.8.4, both the creation of a static schedule and the static partitioning of tasks are computationally intractable, and for practical usage, computationally simple heuristics must be employed.

8.4.1 Schedule creation

For this case study, an implementation of the time-triggered scheduling algorithm 1 (TTSA1) (Gendy 2009) was used. TTSA1 aims to create a schedule that meets all task constraints and that keeps power consumption as low as possible. The algorithm takes tasks' attributes such as the period and the deadline as input and produces a tick interval and new phases for each task as output.

The algorithm works by sorting tasks first according to precedence constraints and next by some other user defined strategy:

- Shortest deadline first: In a task-set with implicit deadlines, this provides the same results as a shortest period first strategy.
- Least laxity first
- Shortest WCET first

For ease of reference, the scheduling algorithm using these strategies will be referred to as TTSA1-SDF, TTSA1-LLF and TTSA1-SWF respectively.

After the tasks are sorted, the algorithm simulates execution under TTCA with the largest possible tick interval, including implementation overheads, by considering one task at a time; if a simulation fails as a result of a constraint violation, the phase of the task under consideration is increased and the simulation is re-run. The phase is increased until it is evident that further increase will provide no new feasible or unfeasible solution, whereupon the entire process is repeated with the next largest tick interval. The algorithm continues until all tasks' constraints have been met.

The algorithm does not perform an exhaustive search, merely stopping at the first schedule that meets all constraints. It assumes that because the search was begun with the “best” tick interval, the first feasible schedule will represent a good (but not necessarily optimal) solution.

The implementation of the algorithm used in this work was restricted to produce only TTCA task-sets, though the complete algorithm can also produce schedules for the time-triggered hybrid implementation (Section 4.3.4). The algorithm was also supplemented with an optional capability to increase the phase of tasks that do not take part in transactions, reducing the number of task group executions that exceed the tick interval and, hence, reducing the overall release jitter. When this capability is enabled, the algorithm will be referred to as TTSA1-JR.

8.4.2 Task partitioning

The task partitioning scheme employed a first fit (FF) heuristic (Section 2.8.4) since this provides a more efficient solution compared to the next fit heuristic. The best fit heuristic may have provided better results (Burchard et al. 1995), but also requires an algorithm capable of calculating the optimal schedule for a set of tasks, a job that TTSA1 avoids due to the large computation time. The FF heuristic allocates tasks purely on the basis of their utilisation and is, therefore, much more computationally tractable. The output of the heuristic was varied by the specification of a per-core utilisation cut-off, from 0.5 to 1.0 in 0.1 intervals; and, by the task sorting strategy: by increasing periods or by decreasing utilisation. For ease of reference, task partitioning using either of these strategies will be referred to as TP-IP and TP-DU respectively.

The partition algorithm used the same initial step as in (Monot et al. 2010), by placing all tasks involved in a transaction on the same core. This allowed the single processor TTSA1 algorithm to be used as-is, without having to be made aware of the presence of multiple cores.

8.5 Evaluation platform

This case study made exclusive use of the HW-TTC-ZSO-MC-MSB scheduler (Section 7.5) on a MC-PH2 (Section 5.4). The HSD mechanism (Section 7.7) was not used since the tasks were implemented with zero executive jitter. In cases where all the tasks were scheduled on one core, the other core was left dormant. Since a HW-TTC-ZSO- scheduler was used, TTSA1 was executed under the expectation of zero scheduling overheads.

The following sections present the system configurations obtained by executing the partitioning and schedule creation algorithms and their effect on the task release jitter.

8.6 Task distribution

As the utilisation threshold for the partition algorithm was varied from 1.0 to 0.5, the required number of cores rose to a maximum of two. Figure 8.1 shows the distribution of the 71 tasks across the cores.

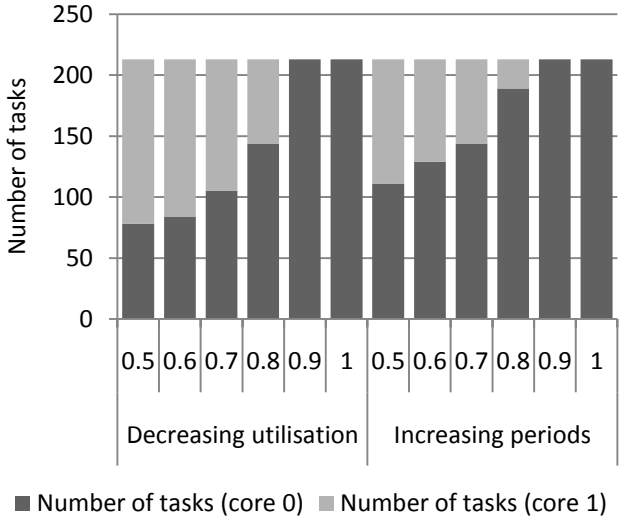

Figure 8.1: Task distribution after partitioning based on different task sorting strategies

When the tasks were sorted by decreasing utilisation, the partitioning algorithm was able to fill up the first core with the high utilisation tasks, leaving the lower utilisation tasks to the second core. Hence, under TP-DU, more tasks were placed on the second core compared to TP-IP, especially with a lower utilisation threshold. A similarly strong correlation cannot be made for TP-IP since there is no general correlation between a task period and the utilisation. In this particular case, TP-IP resulted in partitions where more tasks were placed on the first core.

It can also be observed from Figure 8.1 that an utilisation cut-off of 0.9 and 1 resulted in the same single-processor partitioning of tasks under both strategies. This is because the overall utilisation is 0.84. The systems can also be expected to perform similarly, since the scheduling algorithm applies its own sorting.

8.7 Release jitter

The task-set used in this study consists of 71 tasks, and it would be unwieldy to present the release jitter for each task individually for each partition and scheduling strategy used. For this reason, in this and subsequent sections, the release jitter is shown as the average of the release jitter of the 71 tasks. This measure is sufficient to assess the overall effect of the scheduler (“a lower value is better”), though it should be mentioned that some tasks do indeed show zero release jitter.

Figure 8.2 and Figure 8.3 show the release jitter for TTSA1 and TTSA1-JR respectively. The blank spaces in Figure 8.3 do *not* represent zero jitter, but cases where the algorithm was unable to generate a schedule. This can be

observed whenever the task-set utilisation is high. In all cases, the release jitter was within requirements (Section 8.2).

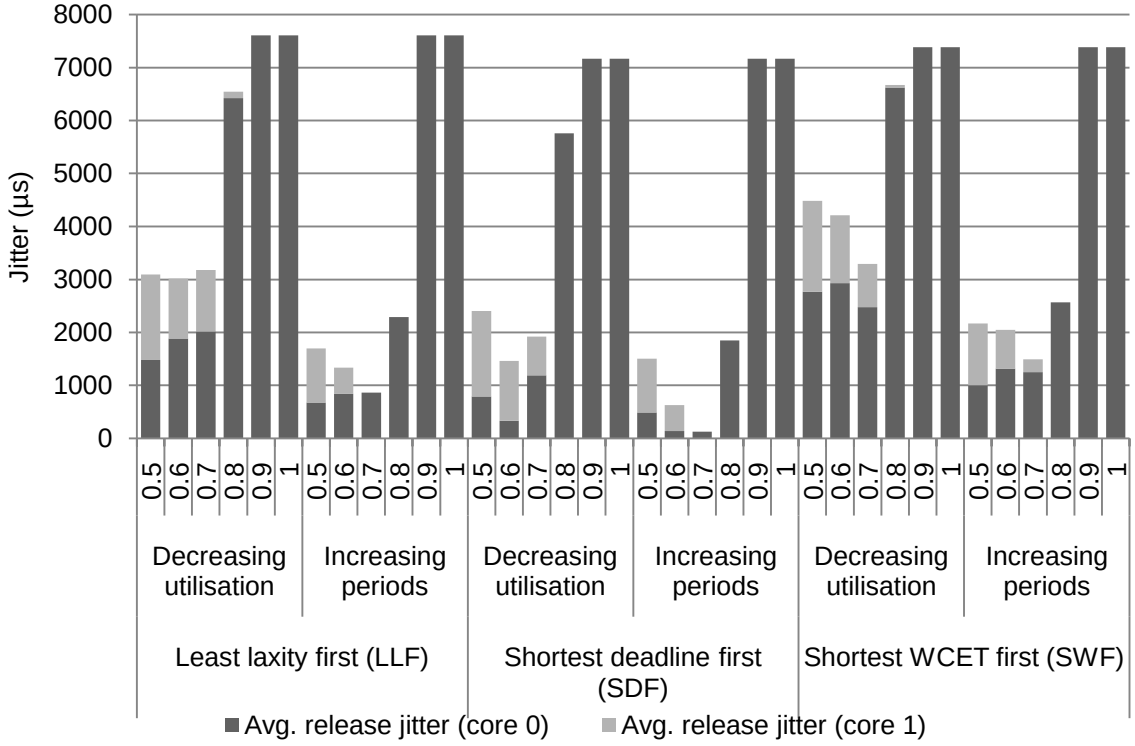

Figure 8.2: Release jitter when using TTSA1 with different strategies

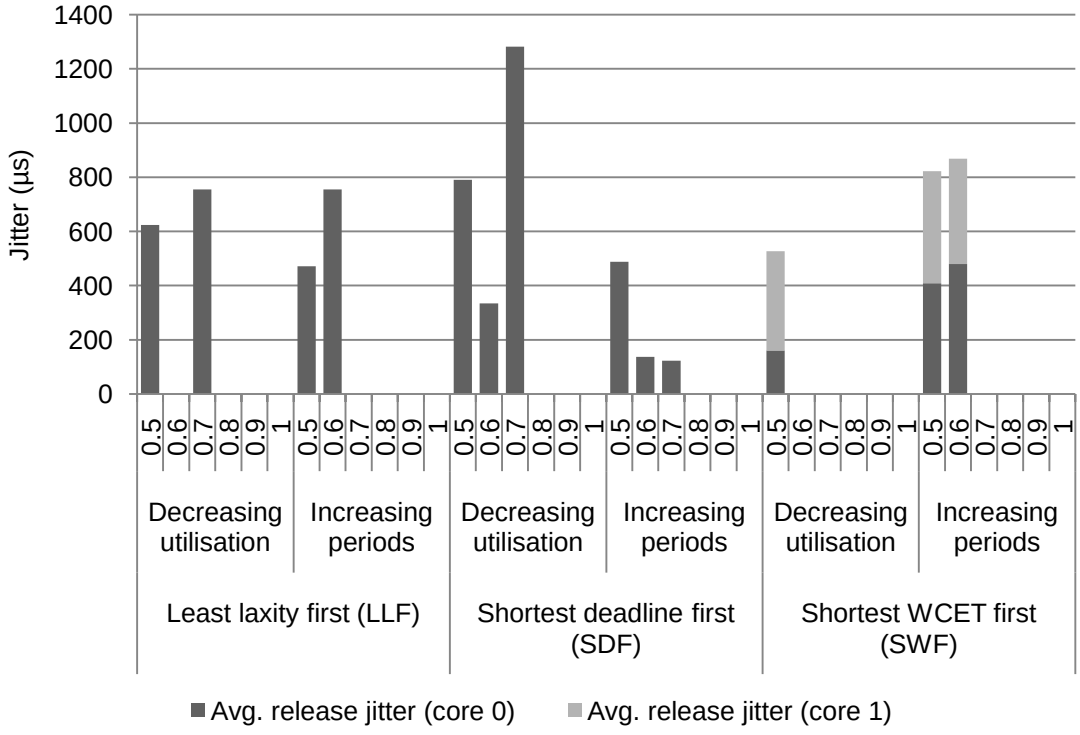

Figure 8.3: Release jitter when using TTSA1-JR under different strategies

The release jitter arises from occasional tick overruns by the execution of task sequences; these overruns are occasional but regular because tasks in the sequence have different request rates. This formed the motivation behind TTSA1-JR and also explains its effectiveness: in the case of TTSA1-JR-SWF with TP-DU and a cut-off of 0.5, a reduction of 88% is observed. However, in other cases, the effect is minute, for example a reduction of just 3% is observed for TTSA1-JR-SDF with TP-IP and a cut-off of 0.7. This latter case is because the original schedule already exhibits low amounts of jitter.

The release jitter is not eliminated completely because TTSA1-JR ignores tasks involved in transactions – these are still allowed to overrun the tick interval. Under TTSA1-LLF and TTSA1-SDF, the second core – which has no tasks involved in transactions – shows zero release jitter. This is because the sorting method nullifies the effect of different request rates, an effect that is very evident in TTSA1-SWF. This trend can also be observed in Figure 8.2 where TTSA1-SWF has the highest amount of jitter in all but the single-core cases.

Figure 8.2 also shows that TP-IP always results in lower release jitter compared to TP-DU. It also appears that an utilisation cut-off of 0.7 is a sweet spot for this algorithm when applied to this task-set, resulting in very low jitter under TTSA1 that is only marginally improved (if at all) by TTSA1-JR. It also performs best with TTSA1-LLF and TTSA1-SDF because of the strong correlation between the criteria used in these two sorting strategies and that used in TP-IP itself.

8.8 Tick interval

TTSA1 realises its goal of minimum scheduler overhead by creating a schedule that uses as large a tick interval as possible. The tick intervals for the different schedules in this case study can be seen in Figure 8.4.

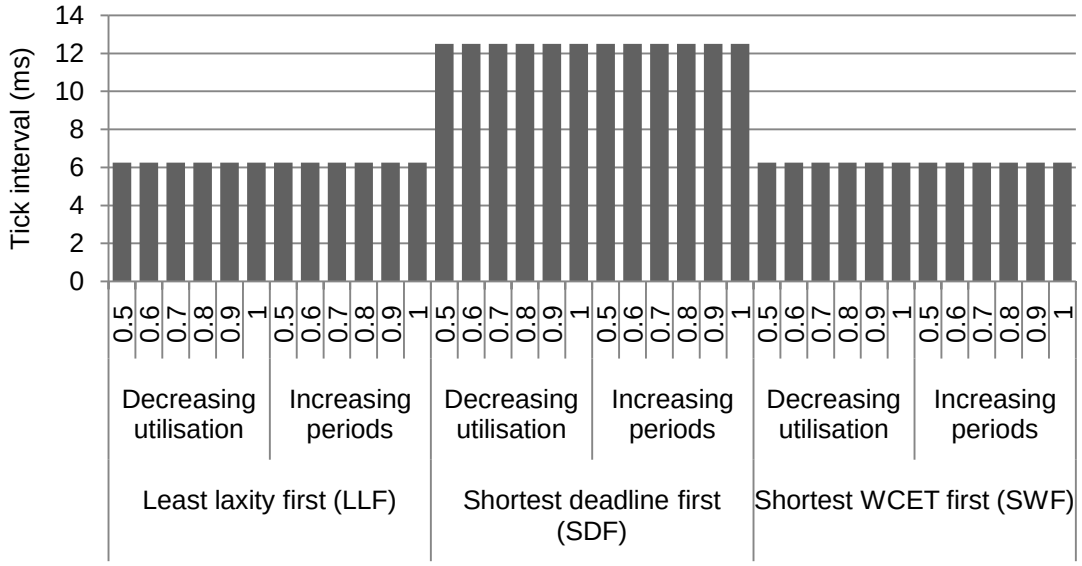

Figure 8.4: Task intervals under the different strategies

Both TTSA1-LLF and TTSA1-SWF generate a tick interval of 6.25 ms while TTSA1-SDF generates a tick interval of 12.5 ms.

8.9 Computation time

The case study was originally undertaken to study the move to a multi-core TTCA as an alternative to a costly redesign or re-evaluation of timing properties. The amount of time taken to generate a schedule for the former is then a valuable measure. This time taken for the task-set in this case study can be seen in Figure 8.5 and Figure 8.6 for TTSA1 and TTSA1-JR respectively.

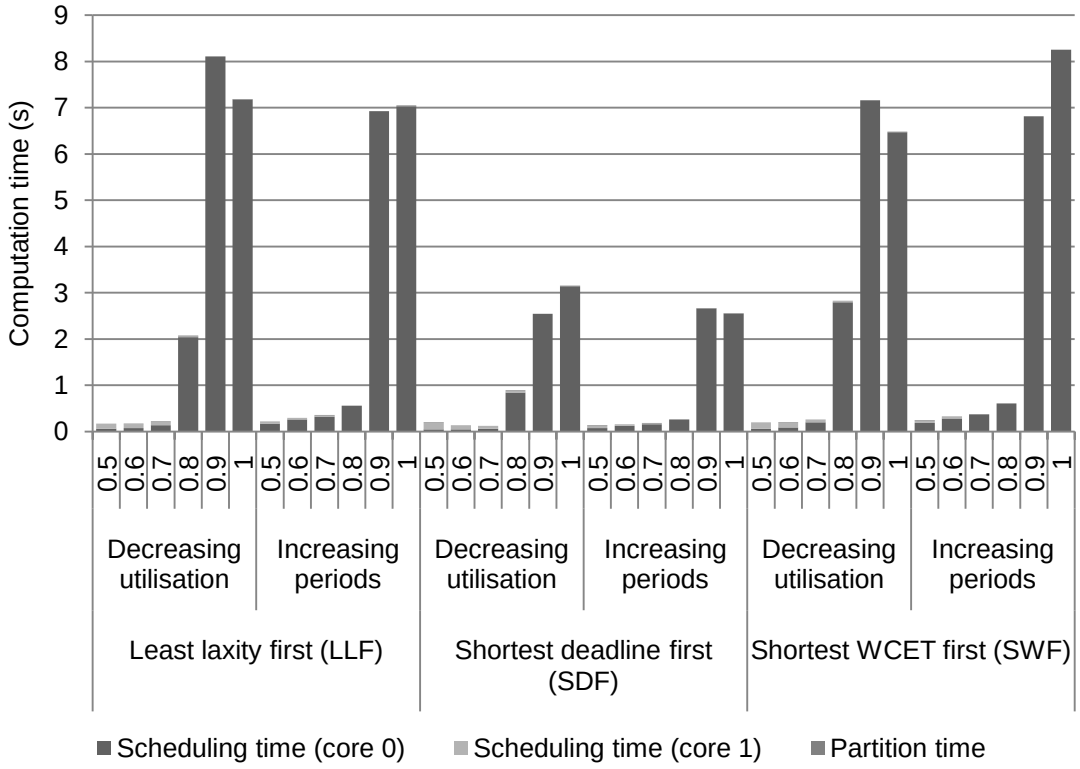

Figure 8.5: Time taken to compute the schedules with TTSA1

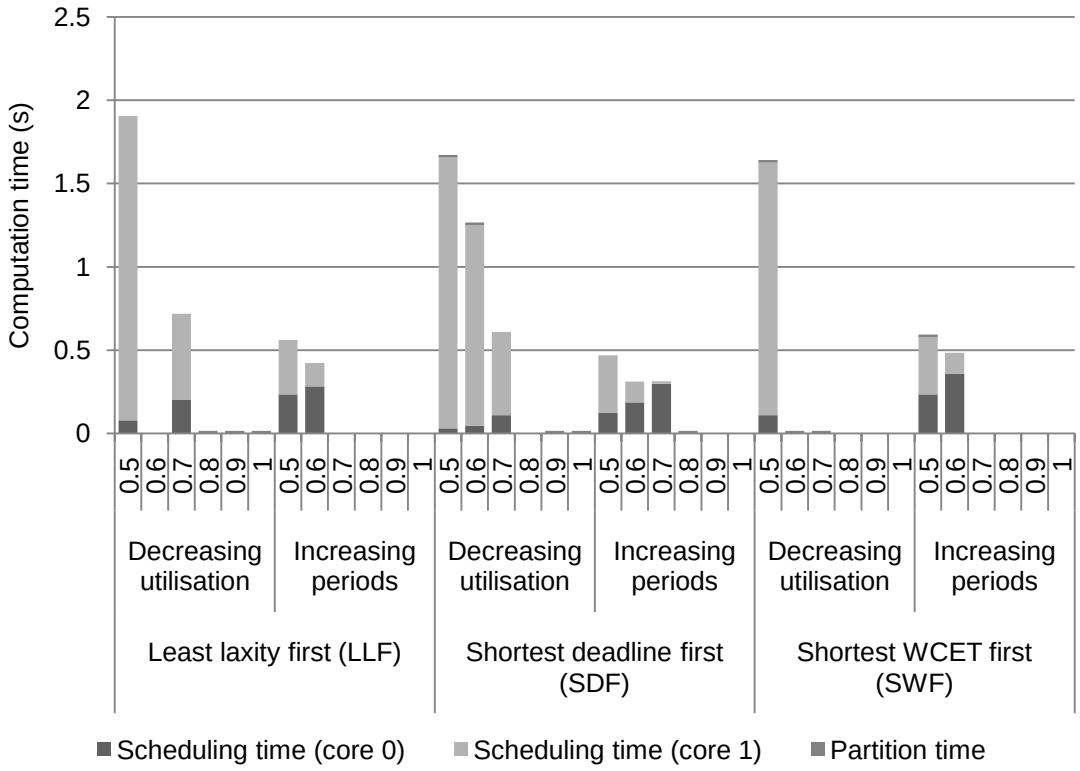

Figure 8.6: Time taken to compute the schedules with TTSA1-JR

The algorithms were executed on the Java VM v1.6.0 executing on a 32-bit version of Windows XP, on one core of an Intel Pentium D CPU running at 3.40 GHz with 3.24 GB of RAM. The run-times were gathered from the millisecond accurate operating system timer.

The partitioning algorithm takes very little time in all cases due to the extremely low computational complexity. On the other hand, the scheduling algorithm has a higher complexity and takes a much longer time, especially when the processor utilisation is high. Despite this, at a maximum, the time reaches a very reasonable eight seconds. It should be noted that TTSA1-SDF displays the lowest run-times overall.

As would be expected, TTSA1-JR increases the computation time, particularly on the core with fewer tasks involved in transactions since there are more opportunities to apply the algorithm.

8.10 Conclusion

This chapter used an electronic engine controller as a case study for the migration of an existing single-processor design to a multi-core TTCA design. The major deterrent to such a migration is the allocation of tasks to different cores, for which reason, a common allocation heuristic was applied to the electronic engine controller task-set combined with a TTCA schedule creation heuristic.

A number of heuristic variations were studied, out of which a partitioning based on increasing periods and a schedule creation based on the sorting of tasks according to their deadlines proved most effective particularly when the

maximum utilisation of all cores was capped at 70%. This system demonstrated lower release jitter, lower power consumption as indicated by a higher tick interval and took a shorter time for the computation of a task schedule.

The case study demonstrated that the allocation of tasks to cores and the generation of an appropriate TTCA schedule is not time consuming and can be sufficiently automated with good results.

Chapter 9

Non-blocking transparent resource sharing

9.1 Introduction

Chapter 5 first introduced the concept of increasing the applicability of single-processor TTCA designs by making use of a multi-core platform. To facilitate this move, a predictable, wait-free communication scheme was introduced that allowed the programmer to move tasks that share common data memory on to different cores without the bother of introducing coherence or consistency schemes.

Unfortunately, this scheme is not applicable to input-output (I/O) resources and, as yet, it has been left implicit that a secondary scheme in software is to be built on top of the communication scheme to allow these other resources to be shared. However, this extra effort of implementing software schemes impedes and complicates the move to TTCA on a multi-core, against the aims of this thesis. This chapter will examine ways of automating the management of shared I/O resources in the designed multi-core TTCA, counting on the guarantees provided in Chapter 7 of zero scheduler overhead and of a jitter-free start time for tasks.

The first section will briefly re-visit I/O resources, first introduced in Section 2.7.4 and then move on to the requirements imposed by the TTCA multi-core design. Next, a scheme of sharing the resources without blocking the software-core is examined together with methods of supporting atomic non-cancellable

transactional access to the resources. The last section presents an evaluation based on two shared peripherals: a general-purpose input-output (GPIO) peripheral and an analogue-to-digital converter (ADC) peripheral.

9.2 Input/output resources

As indicated by the name, input/output (I/O) resources provide I/O capabilities to a software processing core by providing a direct or indirect interface to the environment. Since these resources often lie on the periphery of a device (as opposed to co-processors), they are also referred to as *peripherals*.

The peripheral interfaces to the environment via analogue or digital signalling and is generally accessed by embedded software via reads or writes to the global memory space (Figure 9.1) (Berg 2009), which are translated by hardware into reads and writes to internal peripheral registers. In addition to these software-controlled writes, registers in a peripheral may also change as a result of an environmental action (e.g. a button being pressed) or due to another register being written (e.g. a character LCD incrementing the cursor position when the character register is written). Some register writes may trigger processor notifications in the form of interrupts or through the assertion of polled-flags.

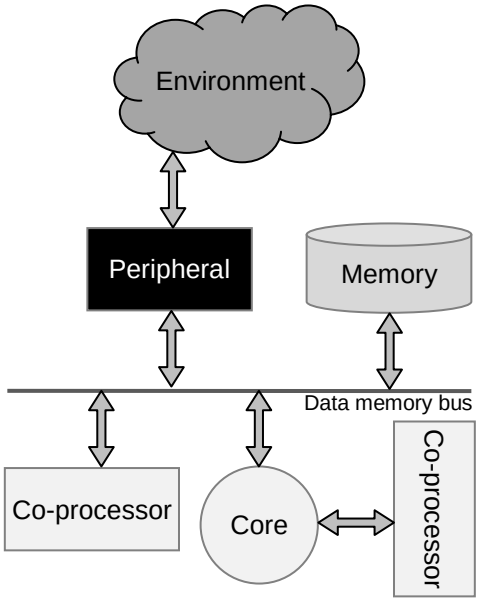

Figure 9.1: The I/O resource or peripheral

Register writes from the processor may be commands to start an operation, the initialisation of data to be used by a future command or the scan of data

calculated by previous software- or environmentally-initiated operations. In some cases, the operations are perceived as instantaneous by software (e.g. changing the voltage level of a pin) and in others, they may non-deterministically take several processor cycles to complete (e.g. converting an analogue value to a digital one).

Peripherals may also be used by embedded software as-is or as the base for another interface, for example, software serial protocols on top of general-purpose input-output peripherals, or an Internet protocol stack on top of an Ethernet peripheral. This wrapping of another peripheral may also be performed at the hardware level, introducing new registers and/or eclipsing existing ones.

9.3 Design constraints

To facilitate the move from a single-core TTCA implementation to a multi-core one as proposed in Chapter 5, it is essential that the tasks that share resources and that have been placed on different cores (hence capable of overlapping executions) do not interfere with each other. The aforementioned chapter has explored this requirement and provided a solution for memory resources; a similarly beneficial solution is required for peripherals.

9.3.1 Non-blocking scheme

To prevent altering the WCET of a task, which may have repercussions on the system schedule, it is necessary to implement a non-blocking scheme. Ideally the scheme would be wait-free, similar to that designed for memory-type resources. Even though peripherals are accessed via the global memory space, the developed buffer scheme (Section 5.5) cannot be used since tasks may make requests and await responses in the same I/O resource transaction.

9.3.2 Low jitter

For predictability, it is necessary to avoid affecting the time determinism of the system, that is, the relationship between a timed output stream and a timed input stream must not be broken and must be reasonably independent of the number and periods of tasks using a resource (Thiele et al. 2004).

9.4 Related work

9.4.1 The gateway scheme

In this scheme, all communication with the peripheral is performed via a single task, i.e. a gateway (Audsley et al. 1993). Communication with the task is performed via memory resources and, hence, the I/O resource sharing problem is transformed into a memory resource sharing problem – a problem that can be tackled by the non-blocking communication. Gateways may commonly be implemented as part of the driver for a peripheral – a driver being a software module that aids portability by creating an abstraction between the task and the peripheral.

Gateways have the flexibility of being software implementations and may hence be changed as the system requirements change. However, being software implementations, they suffer a performance hit (Raj et al. 2007). Additionally, the gateway introduces latency into the system since it has to be scheduled as a task and may hence be affected by jitter in the system. This jitter will be added on to the jitter of the client tasks.

The scheduling constraints on a gateway may also increase as the number of client tasks of the peripheral protected by the gateway increases, for example, each task may require the gateway to run at different frequencies. These

constraints may then interfere with other tasks on the gateway core, leading to the need for a costly arrangement where an entire core is dedicated to the gateway task. In such a situation, a hardware solution may be more cost-effective.

9.4.2 Partitioning resources

In a partitioning scheme, tasks are allocated to the cores to which the required peripherals are attached (Crespo et al. 2010). There is usually no provision for the peripheral to be shared with the other cores and memory caches, if present, may also be partitioned (Bui et al. 2008). Partitioning is a static procedure and may be done at the software or hardware level depending on the requirements of the system. It is the principle behind the Real-Time Virtual Multi-processor (RVMP) architecture that uses static partitioning to divide a single cache-less processor with fully-pipelined functional units into separate virtual processors (El-Haj-Mahmoud et al. 2005).

Partitioning is sometimes matched with safety-critical systems (Crespo et al. 2010) and is a part of the AUTOSAR automotive standard (Monot et al. 2010). This method is particularly suited for systems that are designed off a soft-core, since the peripherals can be moved around freely until a suitable allocation is found.

9.4.3 Time-division multiple access

Time-division multiple access (TDMA) schemes have been used to confine access to a shared memory bus to a statically decided slot (Rosén et al. 2007; Schoeberl et al. 2009). Under such a scheme, the system workload needs to be known *a priori*, so that the slot length and frequency can be fixed. However, the

size of a slot has to be greater than the worst-case transaction time, the identification and calculation of which may be non-trivial.

This solution is also susceptible to failures due to the execution non-determinism within the task execution: if a transaction starts too late compared to the timeslot, it might not complete before the end of the timeslot (an early access can be delayed until the slot begins). This could be handled by widening the timeslot by a margin equal to the highest release jitter of all tasks using the peripheral. However, when this same consideration is applied to all the resources that may be used by a task and to all the tasks running on the core, the allocation of timeslots becomes non-trivial. On the other hand, execution non-determinism within a task could be eliminated completely as in one study that uses the single-path programming paradigm and a TDMA scheme to access shared memory in a multi-core (Schoeberl et al. 2009). This, however, increases the power consumption of the system.

A common timeline also re-introduces the requirement for harmonic period relationships – re-linking tasks that may have been moved to separate cores for the sole purpose of breaking up non-harmonic relationships.

9.4.4 Other approaches

Another technique attempts to bound the amount of a time a task may wait on a shared resource to be available (Paolieri et al. 2009). This same work explores a processor mode capable of calculating the task WCET using the maximum bound and the grouping of tasks according to the frequency of resource demands, allowing for high processor utilisation; groups are scanned for requests in a round-robin fashion.

9.5 Global and proxy peripherals

In the designed resource sharing scheme, the peripherals that were connected directly to a core, i.e. *local* peripherals, were placed outside the control of the core, forming *global* peripherals. However, to permit the core to still have cycle time access to a peripheral, virtual, or *proxy* peripherals were left behind that give the core the impression that the peripheral is still directly connected (Figure 9.2). This is similar to the software driver implementations in embedded virtualisation (Heiser 2007) but differs from other hardware implementations where the processor is blocked (Gary et al. 2004; Chen et al. 2009).

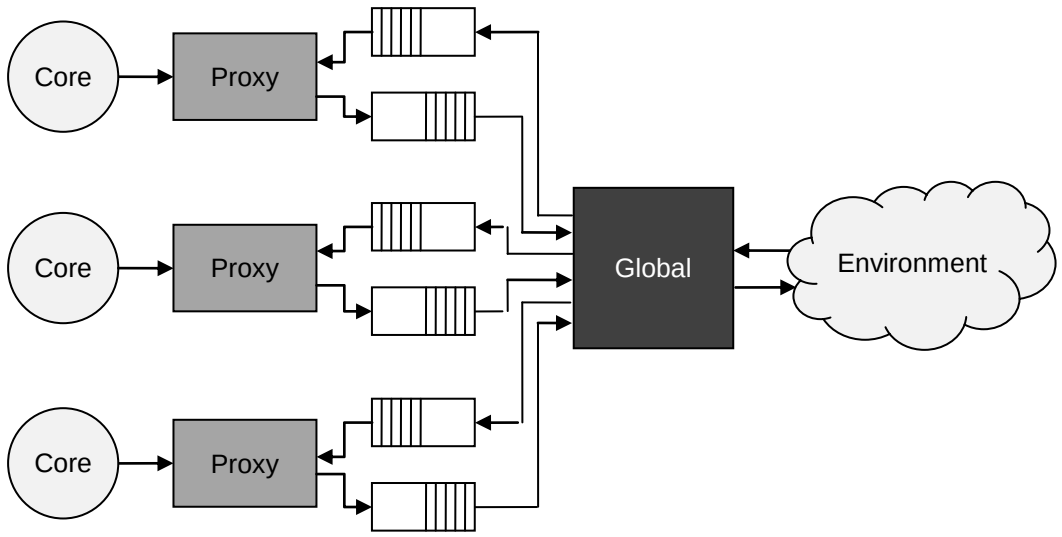

Figure 9.2: Global peripherals and proxies to allow resource sharing

Each global peripheral is connected to the outside world but has no state registers of its own – these are in the proxies instead. When the cores write data to the peripheral, they communicate with the proxy in the same way as with a local peripheral. The proxy wraps data and transmits it to the global peripheral via a FIFO while the global peripheral scans each incoming FIFO in a fixed order, performing the first requested operation, if any, before moving to the next FIFO, even if the current FIFO contains more requests.

When the global peripheral senses or is notified of a change in the environment, e.g. a button is pressed by the user, it broadcasts the event to all the proxies which then modify their own copies of the registers. This pushback mechanism eliminates the bandwidth that would otherwise be required if each proxy were to poll for register changes.

For this work, the communication mechanism between the global peripherals and the proxies has been chosen to be FIFOs, but another network may also be employed. The FIFOs are at one extreme, being a point to point topology and allow for other quantities to be measured deterministically. On the other extreme is a bus topology where all the global peripherals are connected together with all the proxies. A slightly more moderate arrangement may be one bus for each type of peripheral. Ultimately, the topology *must support asynchronous operation* since cores may be clocked at different frequencies.

As information from a global peripheral is broadcast, an upper bound can be established on the length of a proxy's incoming FIFO, taking into the account the number of cycles of asynchronous delay in the FIFO α, clock rate mismatches between the global peripheral and the proxy β (Equation 9.1) and the maximum number of cycles required by the proxy to process the broadcast data (Equation 9.2).

$$\beta = \frac{Clock\ rate\ of\ global\ peripheral}{Clock\ rate\ of\ proxy} \tag{9.1}$$

$$Length\ of\ a\ proxy's\ incoming\ FIFO = \alpha + \lceil\beta\rceil * (processing\ cycles_{max}) \tag{9.2}$$

In a similar manner, a finite upper bound can be established on the size of the outgoing FIFO, taking into account the number of cores n, the number of cycles

of asynchronous delay in the FIFO α, clock rate mismatches between the global peripheral and the proxy β (Equation 9.1) and the maximum amount of time required to perform a peripheral operation (Equation 9.3).

$$Length\ of\ a\ proxy's\ outgoing\ FIFO = 1 + \alpha + n * \left\lceil \frac{(Clock\ rate\ of\ proxy) * (operation\ time_{max})}{\beta} \right\rceil \quad (9.3)$$

9.5.1 A time-triggered approach

As explained above, a global peripheral stops scanning its incoming FIFOs when performing an operation, using them both as a buffering mechanism and as a communication medium. Thus, any non-determinism in the peripheral usage pattern or the operation execution time is transferred to the scanning pattern, implying that a global peripheral could be scanning a different FIFO each time a task starts. A task would then experience jitter in accessing the resource even if it has zero release jitter. Using a hardware sandwich delay mechanism (Section 4.4.3.2) on the resource operation is one solution, but would be impractical since the operations carried out by a peripheral are varied and may be requested in any order. They may then be given the same WCET, but this would lead to a waste of resources.

Even if the FIFO scanning were kept continuous, with new data being stored in an internal buffer, the newly buffered data would still be applied at times dependent on the completion of the previous operation and will have the same implications as the previous case.

To increase the predictability in the developed system, the scanning mechanism is restarted whenever the timer on core 0 generates an interrupt. Thus, the

mechanism is always in a known state after a tick has occurred. While it may still gather non-deterministic behaviour as execution progresses for the tick, the cause for the non-determinism is localised to the previous tasks in the tick and is, hence, more controllable.

The determinism could be increased further by restarting the scan at the start of every task. However, since different cores may be running different tasks at different times, the scanning may be continuously restarted, giving the global peripheral no opportunity to service cores further down the scan sequence, resulting in resource starvation instead.

This reset of the scanning mechanism can analogously be applied to other network topologies, for example, a bus arbitration mechanism where tables used to influence arbitration could be cleared or reset to a known state on the generation of a tick.

9.5.2 Transaction capable

While the above hierarchy allows for resource sharing, the requirement for transactions is still not fulfilled: by cycling rapidly between the FIFOs for the cores, critical sections may be interleaved – a violation of their atomicity.

One way to avoid this is to use the blocking method of locking a global peripheral onto a particular core until a transaction is complete. In such a state, the peripheral will only scan the FIFO of the core that has locked the peripheral and will only send data to this particular core. A locked peripheral will appear “busy” to the cores that do not own the lock and these other cores will have to operate on stale data in the proxies for the duration of the lock. This method, then, requires a technique to mark the critical sections.

9.5.2.1 Marking the critical sections

Critical sections could be demarcated with explicit software instructions; however, this complicates the design with the need to assess the locations of potential resource conflicts. This could be alleviated by doing the identification and code insertion automatically. However, the application software may be written with a co-operative mindset, i.e. it may assume full control of the processor and may unnecessarily interleave accesses to different resources (Section 3.2.3).

Under this assumption, the critical section for a resource used by a task can be considered to start either at the start of the task or at the time of first access of the resource; the end of the critical section can be considered to be either at the end of the task or at the time of the last access of the resource. These demarcations can be easily handled on behalf of the application software: resource access can be automatically tracked by hardware and locking on task start or end can be performed in the same manner as for the switching of the communication buffers.

Unfortunately, race conditions, arising from the lack of execution determinism inside a task, may introduce jitter when locking or unlocking is performed on the basis of accesses. At the same time, making the entire task a critical section for each resource used by it can be overly pessimistic, may completely serialise code execution and may reintroduce the long-task problem.

9.5.2.2 Timed access

An alternative approach may be to create a TDMA schedule for each global peripheral where any core requiring access to that peripheral is statically

granted a timeslot. This is similar to the scheme discussed in Section 9.4.3 and bears the same disadvantages.

9.5.2.3 Intelligent peripherals

The key problem with the approaches above is the use of a locking mechanism to protect the resources and the protection of the entire task as a critical section. In concurrent systems, locking mechanisms are best employed to protect small (in execution time) areas of shared resources, even if the result is a large number of locks.

The remaining alternative is to wrap the peripherals with additional logic that can recognise software-access patterns and take appropriate action. Two particular peripherals are considered in the following sections: a general-purpose input-output (GPIO) peripheral and an analogue-to-digital converter (ADC) peripheral.

9.6 Globalising the GPIO peripheral

The GPIO peripheral has a number of uses; from very simple ones like flashing an LED or sensing a button press to more complex ones like the implementation of a serial protocol. It provides a digital interface to the environment via a set of pins on the chip. The direction (input or output) of a pin and the voltage level on the pin are controlled by different registers in the peripheral. GPIO pins are normally connected to various different devices (perhaps through optical or electrical isolators), and it is a very common usage to control pins individually without modifying the state of the others.

The single-core GPIO implementation used a single register to maintain the state of the pins. Asserting a pin required an OR operation between this register and an appropriate mask, while an unassert operation required an AND operation. When globalising the peripheral, this behaviour was inappropriate since a single bit operation required a write to the entire register. A different mechanism of SET and CLEAR commands was used, similar to the ARM microcontroller.

As a local peripheral, the GPIO peripheral is able to read the state of the pins whenever requested to do so, since this is an instantaneous operation. However, a proxy peripheral is unable to do this and must maintain the state of the pins in a register. This register is updated by the global peripheral via a pushback mechanism; the global peripheral itself polls the pins at regular intervals to be aware of any state changes.

Neither the proxy nor the global peripheral in this implementation implement any extra flow control, hence requiring the size of the FIFOs between them to be configured according to Equation 9.2 and Equation 9.3.

9.7 Globalising the ADC peripheral

The ADC peripheral is very popular in embedded control systems for sampling analogue signals. Most often, the peripheral will support multiple analogue signals or channels, with only one conversion logic tree, requiring tasks sampling on different channels to wait for the previous conversions to complete. To aid this mutual co-operation, the ADC peripheral includes a register to indicate its “busy” status. Tasks wait (by polling or by interrupts) on changes to this register before starting a conversion or reading the conversion result.

Typical ADC usage includes a write to a configuration register to pick the channel, a command to start conversion, a wait for the conversion to be completed and a read of the converted value. When globalising the peripheral, this sequence was treated as a critical section: it should not be interrupted by any other core. For example, the configuration must not be changed after it has been written, but before the conversion starts.

The proxy honours this by writing to the configuration register locally. When the conversion is actually requested, the conversion request is sent along with the configuration value to the global peripheral. The global peripheral then broadcasts the busy status to all the proxies, configures itself and starts the conversion; the converted value is sent back to the proxy that requested the conversion.

The global and proxy peripherals have to be designed with the same care as concurrent software. For example, two proxy peripherals may send a conversion request at the same time, whereupon the global peripheral must clearly indicate which request is being handled. In this particular case, this is performed by sending the converted value back only to the proxy whose request was handled; this proxy alone clears the “busy” status.

While Equation 9.2 supplies an upper bound for the proxy’s incoming FIFO, the proxy’s included flow control requires only one entry in the outgoing FIFO, contrary to Equation 9.3.

9.8 Evaluation

The evaluation of the globalised ADC peripheral consisted of the execution of a simple task that asks for the value on channel 0 of the ADC peripheral and then waits until the conversion is performed. The evaluation utilised four task-sets built from one to four replicas of this basic task. The task-sets were executed on four systems with varying number of cores such that one task executed on each core. In cases where the number of tasks exceeded the number of cores, the excess tasks were left unscheduled. The use of a HW-TTC-ZSO-MC-MSB scheduler allowed all the tasks to be started at the same time, so as to guarantee resource conflicts on the ADC.

The evaluation of the globalised GPIO peripheral was performed similarly, except that the job performed by the task was to assert a single pin. The times for this peripheral were measured from the nearest tick to when the global GPIO actually handled the assertion request, since the tasks do not wait on the peripheral.

9.9 Operational jitter

The tasks utilising the ADC peripheral had zero release jitter as expected, but the execution jitter varied as seen in Figure 9.3. However, as soon as the FIFO scanning was reset on a tick, the execution jitter dropped to zero (not shown).

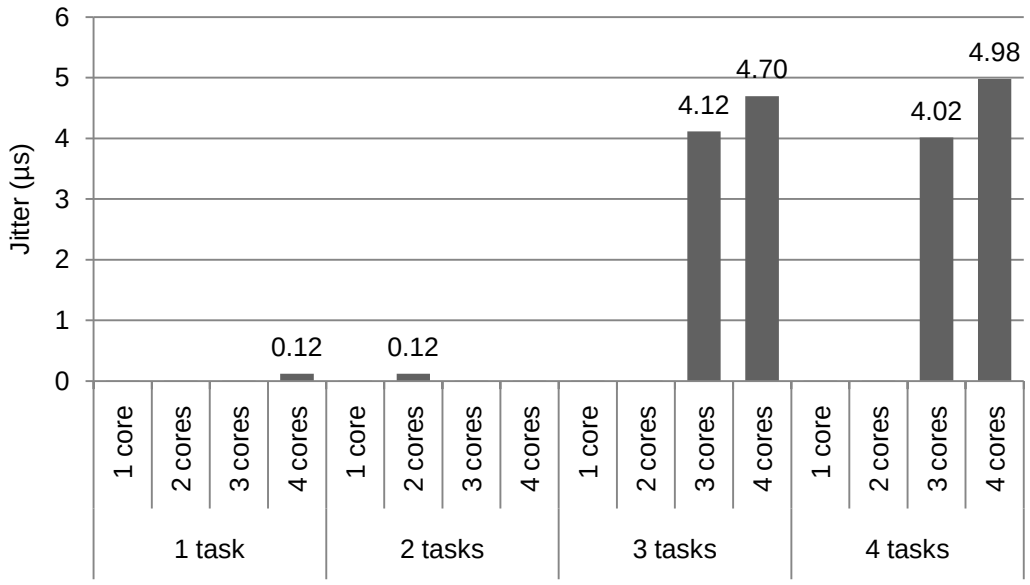

Figure 9.3: Execution jitter when waiting for the completion of an ADC conversion

The execution jitter in waiting for an ADC conversion arises from contention between the different tasks attempting to utilise the same peripheral and from the FIFO scan being at a different position at each task start. Resetting this scan at the tick helps to reduce execution jitter if the task execution and its use of the peripheral is deterministic. The execution time of the other tasks is lengthened depending on the core on which they execute (Figure 9.4).

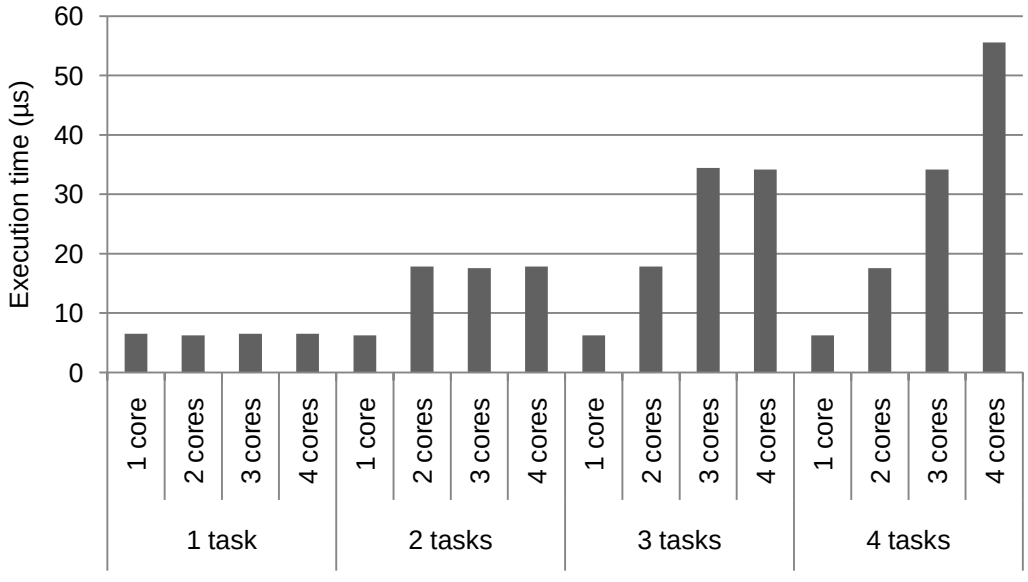

Figure 9.4: Execution time of the ADC sampling tasks with scan reset enabled

The amount of jitter displayed is directly proportional to the operation time. For the GPIO peripheral, where the tasks do not have to wait for the peripheral and where the operation takes one cycle, miniscule jitter is observed when the scan reset mechanism is employed (Figure 9.5).

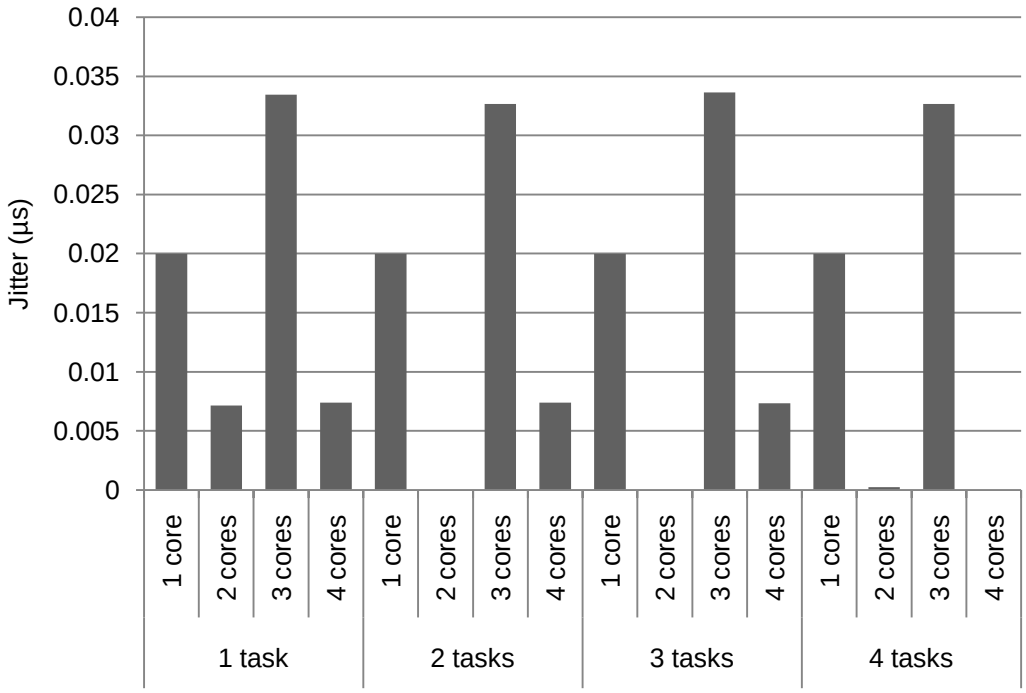

Figure 9.5: Jitter in servicing a pin assertion request on a global GPIO peripheral

It is conceivable that larger operations, for example, sending a large string of characters down a serial communication medium will cause larger amounts of jitter without the scan mechanism being reset. The jitter will also cascade as execution progresses across the tick interval and as the number of resources used by a task increases.

The mechanism is unable, however, to insulate from changes in other cores (Figure 9.4). For example, an ADC sampling task running alone on core 3 may have its execution time lengthened if another task is added to core 0 that attempts to sample from a different channel on the same ADC even if the

completion jitter remains unaffected. The mechanism improves on the situation compared to the original situation, but is still not capable of complete protection.

9.10 Hardware utilisation

The hardware utilised, as number of slices occupied for solely one function, for various configurations can be seen in Figure 9.6. The basic configuration was a dual-core with each core having 32 kb code memory, 8 kb data memory, the ability to communicate and a hardware scheduler with multiple schedule builders, overhead reduction and jitter reduction; the first core had two timers and the second core had one timer. This configuration was then used to produce four variations: two where the first core has either a local GPIO or a local ADC peripheral; and two where both cores have access to either a global GPIO or to a global ADC peripheral.

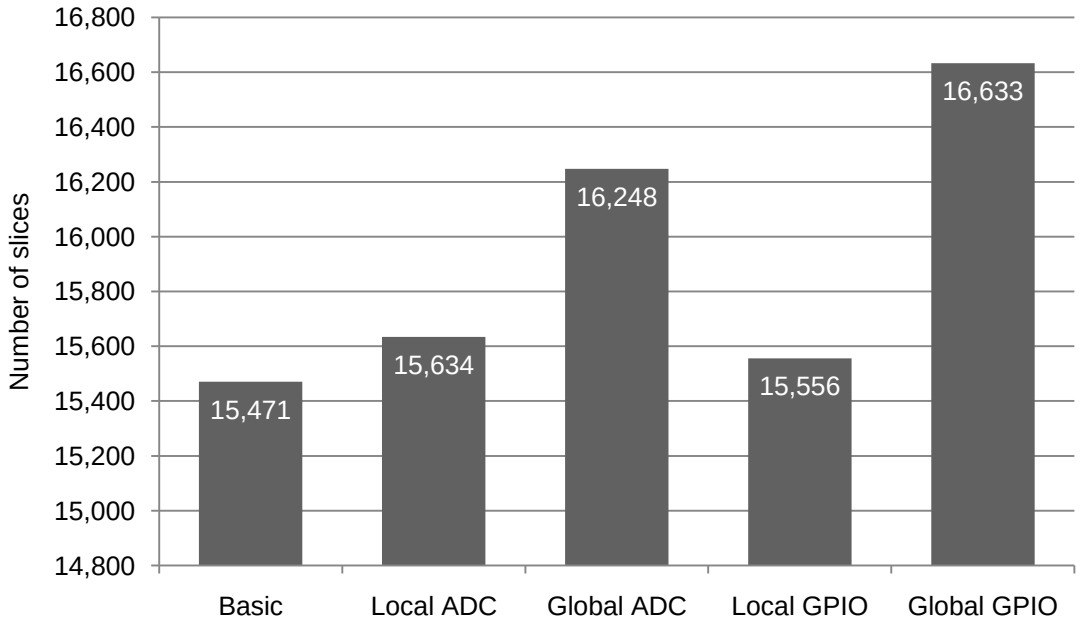

Figure 9.6: Hardware utilisation when moving to the resource sharing scheme

As expected, adding a local peripheral to the first core increases the hardware usage by approximately 1%, with the GPIO peripheral introducing half the

utilisation compared to that of the ADC peripheral. Given this, it is unusual that when globalised, the cost for the global peripheral and two proxies is approximately 2% more for the GPIO than for the ADC. This is both because the data sent from the global to the proxy for the ADC is half the size of that for the GPIO and because the GPIO proxy has to maintain more registers than the ADC proxy. As explained in Section 9.6, some of these registers do not exist in the local GPIO peripheral because they can be read directly from the hardware when required. Accordingly, the cost of globalising the GPIO is approximately 4% of the basic configuration per core compared to an approximate 2% per core for the ADC and this trend can be expected to continue as the number of cores is increased as illustrated by the trends in Figure 5.15 and Figure 7.11.

9.11 Conclusions

For safety-critical systems, it is a good idea to restrict tasks sharing a peripheral to one core to avoid the nuances of resource sharing. However, for cases where this is not possible, this chapter has discussed the use of a global and proxy peripheral scheme that allows resources to be shared without blocking the processing cores.

This scheme was also combined with a technique to reset the communication controllers between the global and proxy peripherals when a tick occurs. If the task sequence and resource usage are deterministic, this technique reduces the jitter in performing operations on a shared peripheral.

However, the technique is unable to completely isolate tasks, and the probability exists that a task added in the future may increase the run-time of all tasks on the cores that it shares peripherals with.

Chapter 10

Discussion and conclusions

10.1 Introduction

Simplicity is an outstanding methodology for achieving a reliable design, primarily because it makes systems easier to reason about, leading to easier and faster design which has a higher probability of being not only correct but both efficient and reliable. While complexity has the appearance of adding to a product's value, it does not necessarily aid the construction of complex functionality; and, simplicity in design does not imply simplicity in functionality. With this motivation, this thesis has proposed the continued utilisation of the simple and highly predictable time-triggered co-operative architecture (TTCA).

Nevertheless, this architecture does not see much support, primarily due to the long-task problem. While this may be taken care of at the design stage through appropriate techniques, it has a high probability of being introduced during maintenance when the system is modified by developers who have little to no experience with the system. Another similar issue is the creation of non-harmonic relationships in the task periods which can cause wide swings in all tasks' release and finishing jitter which are indicative of a system's sampling and actuating jitter respectively.

Another drawback is that TTCA can guarantee low release jitter for only the first task in the sequence of task executions after a tick, leaving remaining tasks to the mercy of whatever execution jitter exists in the preceding tasks.

This chapter reviews the work presented in this thesis that tackles these problems and presents a solution so as to enlarge the set of feasible single-processor TTCA designs. The chapter concludes with a discussion on the limitations of the work and scope for future work.

10.2 Multi-core TTCA implementations

As a solution to the long-task problem, this thesis has proposed increasing the concurrency in the system through a multi-core TTCA design. In doing so, it was deemed important to begin the multi-core design from a predictable processor design. One such processor was found to be the PH-MT soft core that has already been designed from the ground-up to be as predictable as possible and that supports TTCA, for e.g. by only supporting a single enabled interrupt.

In order to maintain the simple application design of TTCA, the memory system on the multi-core was specially structured so that tasks running on different cores could share data while still written sequentially and still running concurrently. This prevented the need for code to be sprinkled with special markers designating areas of contention, greatly simplifying the design of concurrent tasks. The memory system was based around a wait-free loop-free three-buffer scheme that ensures that the reading task always reads the latest data and that the writing tasks never overwrite the most recent data. The system also accommodates tasks that execute at different frequencies to allow for the common “*n* sample task executions for one control execution”.

Under evaluation, the latency in the communication scheme was found to be absorbed in the function preludes and postludes normally inserted by a C

compiler to the extent that the evaluation program was even able to overlap the execution of the tasks sharing resources by about 10 cycles. Also noteworthy is that the communication controller is approximately half the size of the PH-MT core and that it produces a deterministic output even when the number of cores is increased.

After the communication controller, two multi-core TTCA implementations were built for the multi-core: one that filled the run queues for all other cores (TTC-MC-1SB) and one where each core filled its own run queue (TTC-MC-MSB). The former provides scope for a global management scheme whereas the latter provides security through isolation of the schedule building mechanism. TTC-MC-MSB is also flexible in that the schedule creation algorithm for a core may be changed independently of the other cores. Such a change can also be made in TTC-MC-1SB, but requires a modification of the master core and may have further ramifications. Under TTC-MC-1SB, each core is triggered by a virtual interrupt from the “master” core when the run queue is ready, whereas under TTC-MC-MSB, each core is triggered by the timer on the “master” core. Under evaluation, these mechanisms were found sufficient to start the task execution on the slave cores at the same time.

Due to the dependence on the interrupts from the master core, a proper initialisation sequence was also formulated in software that guaranteed that the master core always finished the sequence last. This was found to hold true on hardware when synthesised for up to four cores and in simulation for eight cores.

With the entire multi-core design implemented as discussed above and along with the multi-core TTCA implementations, it was then proposed through a case study to solve the long-task and non-harmonic task problem by partitioning the tasks amongst cores so that no core in the system has a long-task and so that each task-set on a core has harmonic periods.

10.3 Hardware multi-core TTCA implementations

The multi-core TTCA implementations still had the disadvantage of imposing the overhead and jitter of schedule creation and task dispatch on software running on the multi-core. When considering the number of tasks that can potentially be scheduled by a multi-core implementation, it was desirable to provide an alternative implementation that could reduce this overhead and jitter.

In doing so, an existing TTCA hardware implementation, HW-TTC, was extended by first making a modification that would allow for truly zero scheduler overheads without destroying support for precise exceptions, resulting in HW-TTC-ZSO. This same scheduler was again extended with a hardware version of the software sandwich delay scheme for reducing jitter, producing a hardware implementation with zero overheads and zero release-jitter (HW-TTC-ZSO-HSD). An evaluation of the system proved these expectations and also demonstrated that the scheduler core is only 18% of PH-MT, while the processor core decreases in size by 23%.

It was also possible to add a user configurable delay to task execution that ensured that the latency of communicating the task information did not delay task execution on any of the cores. The evaluation proved this valid when the

tasks on all cores, even on the master core in HW-TTC-ZSO-MC-1SB, started at the same number of cycles after the timer interrupt.

10.4 An I/O resource sharing scheme

I/O resources are different from the memory resources considered in the multi-core communication scheme because writes to their registers may cause immediate interactions with the environment. However, since this thesis considers exploding a single-processor design across a multi-core, it became necessary to consider ways of sharing I/O resources.

In this endeavour, the peripherals were split up into a proxy component that stayed attached to a core and a global component that was attached to the environment. Proxies in different cores communicated with the global component via a network, with the system employing a relaxed memory consistency scheme where the proxies stored all the peripheral data and the global peripherals pushed back new data when available. A novel technique of resetting the network structures on the system timer overflow was also utilised to add determinism into the system, with the maximum effect being produced on a 4-core system with a jitter reduction from 4.98 µs to zero. This jitter was found to be dependent on the length of transaction with the resource.

Unfortunately, the techniques are unable to completely eradicate the effects of resource sharing and it is still possible for a later change in the system to unduly elongate the execution time of an existing task on a different core related only by the shared peripheral.

10.5 A simpler, but predictable processor

One observation made in almost all TTCA implementations is that the interrupt service routine is merely used to count the number of ticks. Yet, this simple routine may interfere with the execution of a task, increasing latency and jitter. To avoid this, the mechanism of counting ticks was moved into hardware by matching sleep requests against core interrupts; the core is put to sleep only when every interrupt is matched by a sleep request. The rest of the scheduler remained in software and retained flexibility in the formulation of the scheduling algorithm.

This core, dubbed the PH-DS, is as predictable as the PH-MT, but not as flexible. It is however more flexible than the core that utilises a complete hardware scheduler. The evaluation found PH-DS to be 24% smaller than PH-MT, a savings that sees greater benefit when the core has to be replicated in a multi-core. For example, under TTC-MC-1SB, it is sufficient to employ PH-DS as the core for the slave cores; TTC-MC-MSB is simple enough to allow this for all the cores.

10.6 Multi-core schedule creation algorithm

While not considered a novel contribution, the schedule creation algorithm used in the engine controller case study merits a mention. This case study examined the migration of an existing system to the multi-core TTCA. This system was interesting in that the tasks could be run co-operatively but there were a great many tasks, all of small execution length and the transactions between tasks were complex.

To avoid the error prone method of allocating and scheduling tasks manually, the case study implemented an algorithm by tying together a popular bin-packing heuristic and a TTCA scheduling algorithm heuristic (TTSA1). Among the different variations that were evaluated, it was discovered that a partitioning based on increasing periods with an utilisation threshold of 0.7 per core and a schedule creation based on earliest deadline first, proved most effective. With such parameters, the algorithm also took a shorter time to compute its result and produced a result possessing lower release jitter than all the other variations. It is also expected that the result will consume the least amount of power owing to the generation of scheduling parameters with the largest tick interval.

10.7 Limitations

This thesis has demonstrated how the major problems in TTCA can be solved by moving to multi-core TTCA implementations, without any impact on the application software. However, there are a number of limitations in the presented work.

The first limitation is that the communication scheme is very closely tied to the memory structure. One of the implications is that a particular shared memory area cannot be used for both read and writing. The processor also only uses scratchpad memory and, so, each buffer is also implemented similarly on the SRAM present in the FPGA. If the multi-core were to be used for general-purpose TTCA applications, then the memory might be changed to a larger external memory. In such a case, it becomes more expensive to duplicate the entire data memory.

Another limitation is that the predictable initialisation scheme requires the communication mechanism and that this initialisation scheme is required by all the multi-core schedulers. Thus, even if the choice were made to run the cores completely independently, with no communication between tasks on different cores, no hardware savings can be made by eliminating the communication mechanism.

The hardware scheduler suffers from the same inflexibility limitation that affects any other hardware component, but is also limited in its portability. Crucial features of the scheduler, such as the zero scheduler overhead, required in-depth knowledge of the functioning of the PH core and also unrestricted access to modify the code for the core. The scheduler is also closely tied to the MIPS instruction set and the 5-stage pipeline of the PH core.

10.8 Novelty contributions

The thesis has made five key contributions to the field of safety-critical embedded systems. The first contribution is a novel processor that widens the applicability of the time-triggered co-operative architecture (TTCA) by enabling long-tasks and non-harmonic tasks to be scheduled alongside high frequency tasks without changing the application software. This contribution will allow for more responsive and reliable systems.

The second contribution is the incorporation of these techniques into hardware, achieving a system that is able to expose the whole of the processor's computing resource to the application software. This contribution will allow for more surety of the available processing power during system development.

The third contribution extends the second contribution to protect the execution of a task so that it experiences zero release jitter. This contribution will reduce the chance that a sampled signal will be wrongly reconstructed, ensuring stable system behaviour.

The fourth contribution allows resources to be shared amongst cores without the cores blocking when trying to access the same resource concurrently and the re-initialisation of the communication infrastructure with the shared resource so that resource access is deterministic. This contribution allows for a relaxation in the constraint that tasks on different cores must not share resources, hence allowing for a computationally simpler allocation and scheduling algorithm that is faster to run, reducing development time.

The fifth contribution is a predictable processor core for purely co-operative software that does not introduce the latency and jitter from the interrupt service routine into the executing co-operative task. This contribution reduces the silicon cost of a multi-core TTCA implementation. Additionally, like the second contribution, it allows for greater surety of the available processing power during development.

10.9 Recommendations for future work

The resolution of the limitations described above can constitute a start to future work. For example, to allow the communication scheme to work with a larger amount of memory, only a portion of the data memory needs to be buffered. Under this scheme, compiler support will then be required to place all shared memory variables into the shared area. The limitation with the initialisation

scheme can be resolved quite straightforwardly by a hardware implementation with single-bit asynchronous signalling.

A bulk of future work can also examine the automation of many of the development-side management issues such as the allocation of tasks to cores. Such a tool could also be made intelligent enough to extract tasks' timing properties and present a range of possible allocations along with the tradeoffs made by each one.

Such a tool would require an improved multi-core TTCA allocation and scheduling algorithm in the first instance that can handle communicating tasks across cores as well as tasks sharing resources. Since the schemes presented in this thesis are deterministic, the analysis can be done off-line and so utilise a much greater range of computing resources to arrive at a result.

Further work is also required to improve the scalability of the multi-core implementations, primarily in the communication and the resource sharing schemes. As a first step, this will involve a move to a more scalable network topology, away from the point-to-point topology currently being used. This move can be expected to increase hardware costs as well as communication latency, perhaps necessitating a hybrid application-specific topology, rather than a generalised, regular one.

Appendix A

Glossary

A.1 Abbreviations

ALU – Arithmetic Logic Unit

AMP – Asymmetric multiprocessing

API – Application Programming Interface

ASIC – Application specific integrated circuit

BCET – Best-case execution time

CMP – Chip-level multi-processors

COTS – Commercial off-the-shelf

CPU – Central Processing Unit

DPC – A table-free multi-rate TTCA implementation that executes tasks co-operatively with the highest priority dynamically assigned to the task with the earliest deadline

DPC-SSD – A modified DPC implementation with software-blocked sandwich delays

DPP – A pre-emptive implementation that reserves space on the stack for each task on creation and that dynamically associates a higher priority to the task with the earliest deadline

ETA – Event Triggered Architecture

FIFO – First-in first-out data structure

FPGA – Field programmable gate arrays

FPP – A fixed priority pre-emptive implementation that reserves space on the stack for each task on creation

GCD – Greatest Common Divisor

HDL – Hardware description language

HW-TTC – A hardware table-free multi-rate TTCA implementation that executes tasks co-operatively with fixed priorities

HW-TTC-ZSO – A modification of the HW-TTC implementation that exhibits zero scheduling overheads

HW-TTC-ZSO-HSD – A modified HW-TTC-ZSO implementation with purely hardware implemented sandwich delays

HW-TTC-ZSO-HSD-MC-1SB – A modified HW-TTC-ZSO-MC-1SB implementation with purely hardware implemented sandwich delays

HW-TTC-ZSO-HSD-MC-MSB – A modified HW-TTC-ZSO-MC-MSB implementation with purely hardware implemented sandwich delays

HW-TTC-ZSO-MC-1SB – A hardware table-free multi-rate TTCA implementation that executes on a MC-PHn with the schedule builder running on one core

HW-TTC-ZSO-MC-MSB – A hardware table-free multi-rate TTCA implementation that executes on a MC-PHn with schedule builders running on each core

I/O – Input/output

ISR – Interrupt Service Routine

Kb – 1 Kilobyte or 1024 bytes

LCM – Least Common Multiple

MC-PHn – A microcontroller with n PH-DS cores or 1 PH-MT core and n 1 PH-DS cores

MPSoC – Multi-processor system-on-chip

PH – The PH soft-core implementation

PH-DS – A modified PH implementation with the delayed sleep mechanism

PH-MT – A modified PH implementation with multithreading

RTOS – Real time Operating System

SMA – Shared memory area

SMP – Symmetric multiprocessing

SMT – Simultaneous multithreading

SoC – System-on-chip

TP-IP – An algorithm that sorts a set of tasks in ascending order of their periods before partitioning it

TP-DU – An algorithm that sorts a set of tasks in descending order of their deadlines before partitioning it

TTA – Time Triggered Architecture

TTC – A table-free multi-rate TTCA implementation that executes tasks co-operatively with fixed priorities

TTCA – Time Triggered Co-operative Architecture

TTC-DS – A TTC implementation that executes on the PH-DS

TTC-DS-SSD – A modified TTC implementation with software-blocked sandwich delays that executes on the PH-DS

TTC-MC-1SB – A table-free multi-rate TTCA implementation that executes on a MC-PHn with the schedule builder running on one core

TTC-MC-MSB – A table-free multi-rate TTCA implementation that executes on a MC-PHn with schedule builders running on each core

TTC-MC-SSD-1SB – A modified TTC-MC-1SB implementation with software-blocked sandwich delays

TTC-MC-SSD-MSB – A modified TTC-MC-MSB implementation with software-blocked sandwich delays

TTC-MT – A TTC implementation that executes on the PH-MT

TTC-MT-SSD – A modified TTC implementation with software-blocked sandwich delays that executes on the PH-MT

TTC-SHD – A modified TTC implementation with hardware-blocked sandwich delays

TTH – Time Triggered Hybrid

TTH – A type of TTCA implementation that permits a single pre-emptive task in order to tackle the long-task problem

TTP – A fixed-priority pre-emptive implementation that allocates stack space as a task executes

TTP-MJ – Functionally identical to TTP, but with code-balancing techniques applied to key scheduler areas to minimise jitter

TTSA1 – A heuristic algorithm that builds a schedule for time-triggered co-operative architectures

TTSA1-LLF – A version of TTSA1 that sorts the tasks in ascending order of their laxities before building the schedule

TTSA1-SDF – A version of TTSA1 that sorts the tasks in ascending order of their deadlines before building the schedule

TTSA1-SWF – A version of TTSA1 that sorts the tasks in ascending order of their WCETs before building the schedule

VHDL – VHSIC hardware description language

VHSIC – Very-high-speed integrated circuit

WCET – Worst-case execution time

A.2 Definitions used by the task model

absolute deadline – The deadline of a frame when measured from when the system starts.

complete – See *concrete*.

concrete – A set of tasks in which the phase of all tasks is known a priori.

Also known as *complete*.

deadline – The time at which a frame should have finished its work.

See also *absolute deadline*, *implicit deadlines* and *relative deadlines*.

execution – A portion of a frame usually produced as a result of interruptions by higher priority tasks.

execution time – Amount of time consumed by a frame, usually demarcated by the best-case and the worst-case amongst all frames.

feasible – An indication that a set of tasks can be executed on a particular system.

finishing time – The time at which a frame or execution finishes its work.

frame – A particular release of the task.

harmonic – A task-set where the period of each task is an integer multiple of the task with the smallest period.

hyperperiod – The amount of time after which the sequence of executions of tasks' frames repeats.

Also known as *major cycl*e.

implicit deadline – When the deadline of the task is assumed to equal its period.

laxity – The amount of computation time left at any particular time.

Also known as *slack*.

major cycle – See *hyperperiod*.

minor cycle – See *tick interval*.

period – The amount of time between consecutive releases.

phase – The time from when the system starts to the release time of the first frame of a task.

precedence constraints – A constraint on a particular task that lists the tasks that should have executed before it.

priority – An indication of its importance.

relative deadline – The deadline of a frame measured relative to the release time of the frame.

release time – The time at which a frame should begin its work.

slack – See *laxity*.

start time – The time at which a frame or execution actually begins its work.

synchronous – A task-set in which all tasks have a zero phase.

tick – The interruption caused by a time event in a time-triggered architecture.

tick interval – The amount of time between two consecutive ticks in a time-triggered architecture.

Also known as *minor cycle.*

utilisation – The percentage of computational time consumed by a task or task-set, upper bounded by the worst-case.

See also Section 2.6, “The task model” on page 2-9, particularly Figure 2.3.

A.3 Units

ms	Milliseconds (10^{-3} seconds)
μs	Micro-seconds (10^{-6} seconds)

A.4 Notations

$\mathbb{N}$	Set of non-negative integers $= \{0,1,2,3,4,5, \ldots\}$	
$\mathbb{N}^*$	Set of positive integers $= \{1,2,3,4,5, \ldots\}$	
X	Ordered set X of size = $\lvert X \rvert$ $= \{X_0, X_1 \ldots X_{n-1}\}$	
$\lfloor X \rfloor$	Largest integer that is smaller than the real number X.	
$\lceil X \rceil$	Smallest integer that is greater than the real number X.	
WCU	The worst-case utilisation of a task-set	Eq. 2.11
$H(\pi)$	The hyperperiod of a task-set π.	Pg. 2-11
$\phi(\tau)$	The phase of a task τ.	Pg. 2-9
$P(\tau)$	The period of a task τ.	Pg. 2-9
$K(\tau)$	The number of frames of a task τ.	Pg. 2-9
$BCET(\tau)$	The best-case execution time of a task τ.	Eq. 2.8

$WCET(\tau)$	The worst-case execution time of a task τ.	Eq. 2.7
$WCTU(\tau)$	The worst-case utilisation of a task τ.	Eq. 2.9
$ET(\tau, k)$	Execution time of the k^{th} frame of a task τ.	Eq. 2.6
$r(\tau, k)$	The release time of the k^{th} frame of a task τ.	Eq. 2.2
$s(\tau, k)$	The start time of the k^{th} frame of a task τ.	Eq. 2.4
$d(\tau, k)$	The relative deadline of the k^{th} frame of a task τ.	Pg. 2-9
$D(\tau, k)$	The absolute deadline of the k^{th} frame of a task τ.	Eq. 2.3
$f(\tau, k)$	The finishing time of the k^{th} frame of a task τ.	Eq. 2.5
$C(\tau, t)$	The computation time left for a task τ at a time t.	Pg. 2-11
$L(\tau, t)$	The slack or laxity for a task τ at a time t.	Eq. 2.10
$E(\tau, k)$	The number of executions of the k^{th} frame of a task τ.	Pg. 2-9
$S(\tau, k, e)$	The start time of the e^{th} execution of the k^{th} frame of a task τ.	Pg. 2-9
$F(\tau, k, e)$	The finishing time of the e^{th} execution of the k^{th} frame of a task τ.	Pg. 2-9
$\phi_{max} + 2H$	The upper bound of the interval that must be evaluated for feasibility analysis; a more concise form of $\max_{\tau \in \pi}\{\phi(\tau)\} + 2 * H(\pi)$ for a task-set π.	Pg. 2-11
T	The tick interval.	Pg. 3-3
σ_{isr}	Overhead introduced by the interrupt servicing routine.	Pg. 3-3
σ_{disp}	Overhead introduced by dispatching a task.	Pg. 3-3

σ_{alg}	Overhead introduced by a runtime schedule creation algorithm.	Pg. 3-3
$\rho(\theta)$	The set of tasks, ordered by presence in the task table, released as a result of the θ^{th} tick.	Eq. 3.1
$\rho_{pre}(\theta, \tau)$	The set of tasks, ordered by presence in the task table, released before a task τ as a result of the θ^{th} tick.	Pg. 3-3
$\rho_r(\theta)$	The set of tasks, ordered by release time and by presence in the task table, to be released in the θ^{th} tick interval.	Eq. 3.5
$\rho_s(\theta)$	The set of tasks, ordered by release time and by presence in the task table, started in the θ^{th} tick interval.	Eq. 3.6
$WCET_{ti}(\theta)$	The worst-case execution time of the θ^{th} tick interval.	Eq. 3.3
$OV_{ti}(\theta)$	The amount of time by which the execution time of the θ^{th} tick interval exceeds the tick interval.	Eq. 3.4
$\gamma(\theta)$	The amount of lag experienced by the scheduler dispatch component in sensing the θ^{th} tick compared to the schedule creation component.	Pg. 3-4

See also Section 2.6, “The task model” on page 2-9, particularly Figure 2.3.

Section 3.2.1, “The TTCA model” on page 3-3.

Appendix B

The three buffer single-writer, single-reader mechanism

This chapter describes the functioning of the asynchronous three buffer single-writer, single-reader mechanism, as detailed in (Chen et al. 1997b). This scheme was first named in Section 2.7.2 and was used as the basis of the hardware communication mechanism described in Section 5.5.

B.1 Introduction

Data sharing is a basic approach to achieving inter-task communication within a variety of applications. At a basic level, this approach can be described as a *writer* task copying data into a common location (a *buffer*) which is later scanned by a *reader* task (Figure B.1).

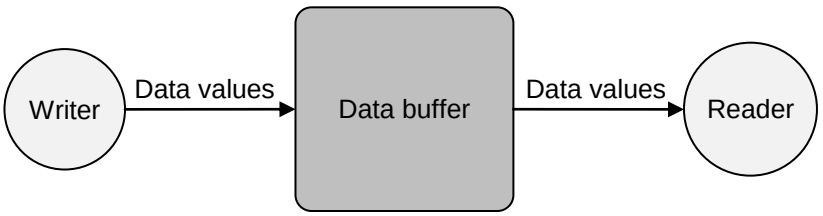

Figure B.1: Data sharing between one writer and one reader

For successful data sharing, the *coherence* as well as the *integrity* of the shared data values must be guaranteed, i.e. the shared data must arrive at the reader both wholly uncorrupted and in a totally ordered manner. It is also necessary that the data be kept fresh by making the latest complete version of the shared data produced by the writer always available to the reader.

The three-buffer mechanism described here facilitates successful data sharing in real-time, multi-processor systems where: the data transferred from the writer to the reader must both be accurate and arrive in a timely manner; the data transfer mechanism must not reduce the available parallelism; the data to be transferred is too large to be a candidate for any atomic data transfer techniques built into the microprocessor hardware; and, there is no relationship between the times taken for the write operations and the read operations. The mechanism does not protect against data either being overwritten when the writer is faster than the reader or being skewed when the reader is faster than the writer; such protection is expected to be implicit in the real-time properties of the writer and reader.

B.2 The design of the mechanism

The decision on the number of buffers in the mechanism was driven by the need to use as few resources as possible and the freshness requirement mentioned above: one buffer to allow the reader to run concurrently with the writer; and at least two buffers for the writer to switch between so that the reader can still get the data from the last completely written buffer when the writer is busy. To preserve coherence and integrity, the buffer which is to be read by the reader and the reader's state are always made known to the writer, significantly in the event that a writer begins in between the reader's decision of which buffer to read and the announcement of this decision.

In the design illustrated in Figure B.2 and in Listing B.1, the data accessed by the reader and the writer include the three-element buffer array, *buffer* and the two control variables, *reading* and *latest*. The data transferred from writer to

reader can be held in any buffer. The control variable *latest* always indicates the buffer holding the latest version of the completely written shared data whilst *reading* is used to indicate both whether the reader is in the process of deciding which buffer to read and the buffer which was read by the reader. A constant table *next* is constructed to help the writer to efficiently decide which buffer to write into. The writer helps the reader to update the *reading* control variable so that it always indicates the buffer that is going to be read.

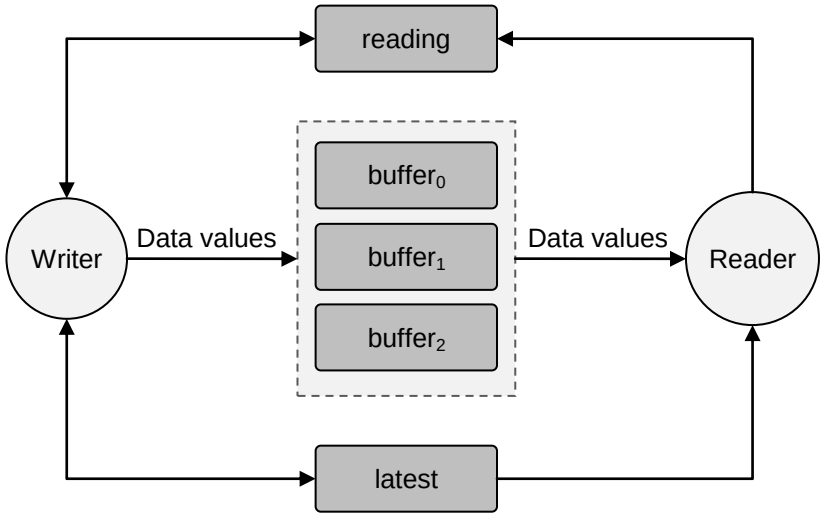

Figure B.2: A representation of the three-buffer single-writer single-reader mechanism

When the writer starts its execution, it delivers a new version of the shared data into a buffer which has the index value different from the current values of *latest* and *reading*. The value of *latest* is then updated. The writer then attempts to update *reading*, if it hasn't already been updated by the reader. The reader, before accessing a buffer, updates *reading* with the value of *latest* to indicate it is reading that buffer.

Since no guarantees can be made that the update to *reading* by the reader will not be interleaved with executions of the writer, the update is actually done in two steps: first setting *reading* to a value that indicates that it is about to be

updated and then updating it with the value of *latest*. Additionally, both the updates to *reading* by the writer and reader are done under atomically evaluated conditions, i.e. the value of *reading* is guaranteed not to change from when the condition is evaluated to when it is finally changed/left unchanged.

```
SHARED buffer IS 3 ARRAY OF DATA
SHARED reading IS INTEGER FROM 0 to 3
SHARED latest IS INTEGER FROM 0 to 2

DEFINE Write:
   INIT next AS 4 ARRAY OF (3 ARRAY OF INTEGERS) = [ [ 1, 2, 1 ],
                                                     [ 2, 2, 0 ],
                                                     [ 1, 0, 0 ],
                                                     [ 1, 2, 0 ] ]
   SET widx1 TO reading
   SET widx2 TO latest
   SET windex TO widx2 OF (widx1 OF next)
   WRITE DATA INTO windex OF buffer
   SET latest TO windex
   ATOMIC SET reading TO windex IF reading IS 3

DEFINE Read:
   SET reading TO 3
   SET rindex TO latest
   ATOMIC SET reading TO rindex IF reading IS 3
   SET rindex TO reading
   READ DATA FROM rindex OF buffer
```

Listing B.1: Pseudo code for the three-buffer single-writer single-reader mechanism

A correctness proof for this mechanism can be found in (Chen et al. 1997b).

B.3 Conclusions

This appendix described a three-buffer mechanism supporting asynchronous data sharing between a single writer and a single reader. The mechanism is intended for real-time concurrent systems where the available parallelism must not be reduced, the timing behaviour of tasks must remain predictable and analysable and the shared data must be coherent and properly ordered.

Appendix C

The BR715 Engine Controller

Some of the content from the electronic engine controller system case study in (Bate 1998) is reproduced here as background material for Chapter 8.

C.1 Purpose of the Electronic Engine Controller System

An electronic engine controller is essentially a safety-critical embedded system in charge of maintaining correct and safe operation of an aircraft engine. As seen in Figure C.3, the electronic engine controller uses sensors to monitor the engine condition (e.g. fuel flow) and other components to monitor the aircraft operation (e.g. thrust request). The electronic engine controller controls the engine via the operation of actuators such as valves, ignitors and pumps. It also accepts pilot commands, and provides status information about the engine back to the cockpit. Components of the controller are normally replicated to provide fault tolerance.

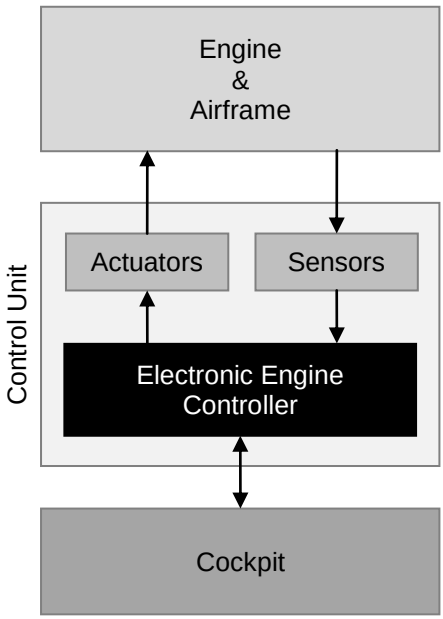

Figure C.3: Overview of an Electronic Engine Control Unit

A particular characteristic of an electronic engine controller system is that transactions between tasks are common and fundamental to safe operation. Transactions consist of reading data from a number of sensors, performing

calculations based on the available data and of conveying the results to the appropriate actuators; the system also has tasks to handle aspects such as health monitoring and maintenance. Also of particular note, is that the system is normally implemented from a single processor design and that the processor utilisation is fairly high.

C.2 Task details

For confidentiality, the original study changed or omitted some of the task properties, such as names and purpose; however, important timing requirements were left as-is (Table C.1).

Table C.1: Task timing properties

Name	Period	WCET
P11	25,000	671
P21	25,000	684
P35	50,000	173
P3	25,000	461
P1	25,000	300
P2	25,000	2088
P4	25,000	340
P5	25,000	7
P6	25,000	85
P7	25,000	1910
P8	25,000	1971
P9	25,000	640
P10	25,000	17
P12	25,000	103
P13	25,000	203
P14	25,000	26
P15	25,000	14
P16	25,000	408
P17	25,000	278
P18	25,000	190
P19	25,000	32
P20	25,000	228
P22	25,000	273
P23	25,000	1265

Name	Period	WCET
P24	50,000	318
P25	100,000	1334
P26	50,000	52
P27	200,000	796
P28	50,000	336
P29	50,000	408
P30	50,000	798
P31	100,000	457
P32	50,000	351
P33	50,000	390
P34	50,000	201
P36	50,000	925
P37	50,000	321
P38	50,000	1801
P39	50,000	522
P40	50,000	256
P41	100,000	196
P42	50,000	900
P43	50,000	1945
P44	100,000	528
P45	100,000	551
P46	100,000	272
P47	100,000	271
P48	100,000	378

Name	Period	WCET
P49	100,000	107
P50	100,000	217
P51	100,000	4698
P52	100,000	232
P53	100,000	30
P54	100,000	763
P55	100,000	62
P56	200,000	304
P57	200,000	336
P58	200,000	100
P59	200,000	8
P60	200,000	378
P61	200,000	38
P62	200,000	428
P63	200,000	2258
P64	200,000	328
P65	1000,000	5040
P66	1000,000	5040
P67	1000,000	5040
P68	1000,000	5040
P69	1000,000	5040
P70	1000,000	5040
P71	1000,000	5040

In the original study, the system was implemented using a scheduler with the cyclic executive architecture (Section 3.6.1) which was configured with a major cycle of 25,000 units and a minor cycle of 3,125 units; the scale of the unit was not mentioned. For this work, it was assumed that one unit is equal to one microsecond.

C.3 Transaction deadlines

The transactions between the tasks and their deadlines are presented in Figure C.4, with arrows designating the transaction flow.

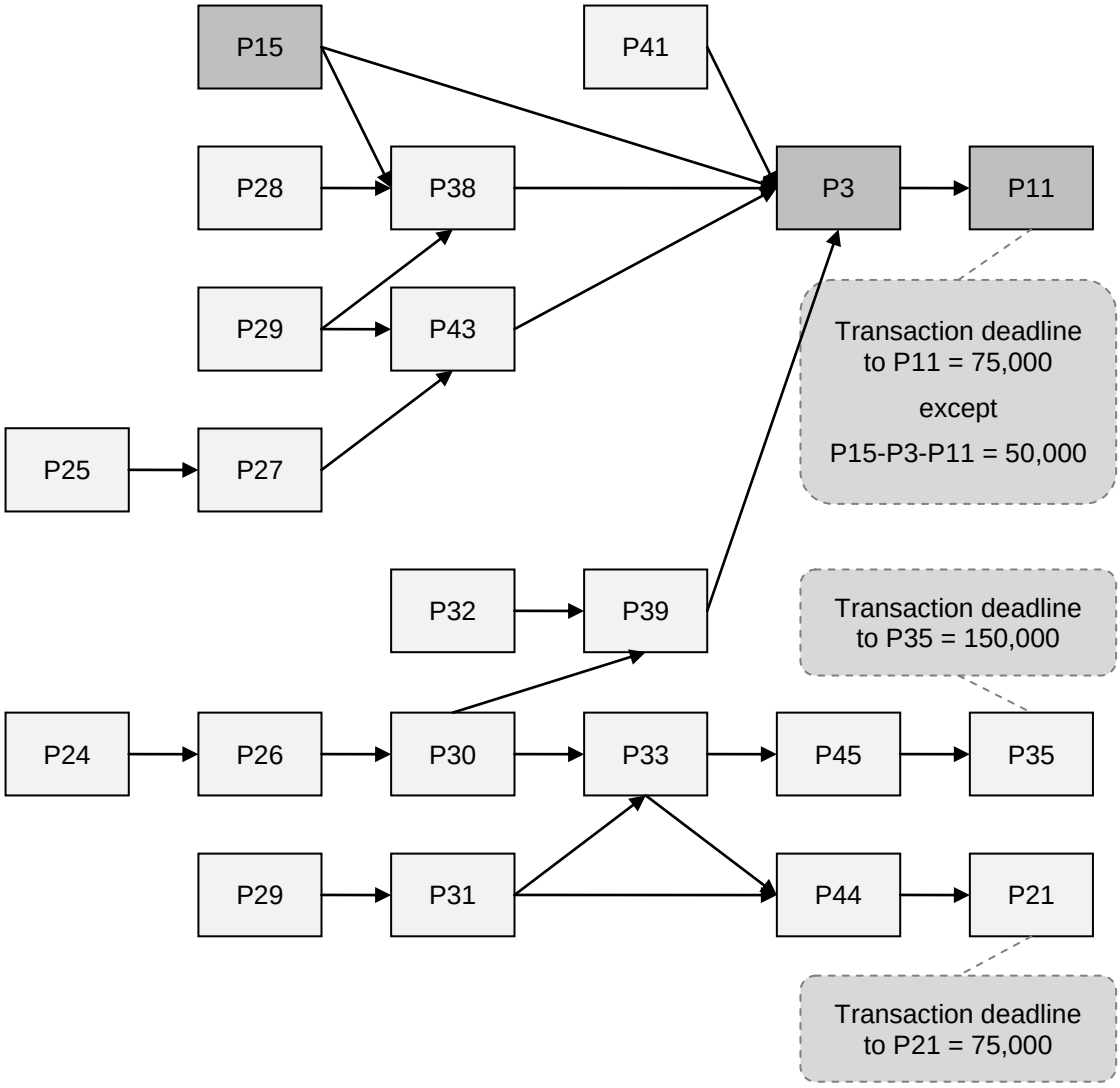

Figure C.4: Task transaction requirements

The task-set had implicit deadlines with minimum release jitter requirements of at most 12,500 units each for P11, P21, P35 and P3; there were no such requirements for the other tasks.

C.4 Conclusions

This appendix presented the task-set for an electronic engine controller system. This task-set is a good match for case studies due to the large number of tasks, the complex transaction requirements between the tasks and a high processor utilisation of 84.3%.

Bibliography

Abdelzaher, T. F., Atkins, E. M. and Shin, K. G. (1997). QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. *The 3rd IEEE Real-Time Technology and Applications Symposium*. Montreal, Quebec , Canada**:** 228 - 238.

Abdelzaher, T. F. and Shin, K. G. (2000). "Period-Based Load Partitioning and Assignment for Large Real-Time Applications." *IEEE Transactions on Computers* **49**(1): 81 - 87.

Agron, J., Andrews, D., Finley, M., Komp, E. and Peck, W. (2004). *FPGA Implementation of a Priority Scheduler Module*. The 25th IEEE International Real-Time Systems Symposium, Works In Progress Session (RTSS, WIP 2004), Lisbon, Portugal.

Åkesson, J. F. (2001). Interprocess Communication Utilising Special Purpose Hardware. *Department of Computer Systems*. Uppsala, Sweden, Uppsala University. **Licentiate of Philosophy in Computer Systems**.

Al-Kadi, G. and Terechko, A. (2009). A Hardware Task Scheduler for Embedded Video Processing. *High Performance Embedded Architectures and Compilers - Lecture Notes in Computer Science*, Springer Berlin / Heidelberg. **5409:** 140 - 152.

Albert, A. (2004). Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. *Embedded World 2004*. Nürnberg, WEKA Verlag, Germany**:** 235 - 252.

Allworth, S. T. (1981). *Introduction to Real-Time Software Design*, Macmillan.

Alves, I. G. (2007). Artemis, Advanced Research and Technology in Embedded Intelligence and Systems. *Speeding up innovation*.

Amey, P. (2002). Correctness By Construction: Better Can Also Be Cheaper. *CrossTalk Magazine - The Journal of Defense Software Engineering*.

Anderson, J. H. and Gouda, M. G. (1992). "A Criterion for Atomicity." *Formal Aspects of Computing* **4**(3): 273 - 298.

Anderson, J. H. and Holman, P. (2000). Efficient Pure-buffer Algorithms for Real-time Systems. *The 7th International Conference on Real-Time Computing Systems and Applications*. Cheju Island, South Korea**:** 57 - 64.

Anderson, J. H., Jain, R. and Ramamurthy, S. (1997a). *Wait-free Object-sharing Schemes for Real-time Uniprocessors and Multiprocessors*. The 18th IEEE Real-Time Systems Symposium.

Anderson, J. H., Ramamurthy, S. and Jeffay, K. (1997b). "Real-Time Computing with Lock-Free Shared Objects." *ACM Transactions on Computer Systems (TOCS)* **15**(2): 134 - 165.

Andrews, D., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbie, M., Ortiz, J., Komp, E. and Ashenden, P. (2004). "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link." *IEEE Micro* **24**(4): 42-53.

Andrews, D., Peck, W., Agron, J., Preston, K., Komp, E., Finley, M. and Sass, R. (2005). hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. *The 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA '05)*. Catania, Italy**:** 338 - 445.

Arnold, R. S. (1989). "Software Restructuring." *IEEE* **77**(4): 607 - 617.

ARTEMIS SRAWG (2006). Strategic Research Agenda.

Ascia, G., Catania, V. and Palesi, M. (2005). "Mapping Cores on Network-on-Chip." *International Journal of Computational Intelligence Research (IJCIR)* **1**(2): 109-126.

Athaide, K. F., Hughes, Z. M. and Pont, M. J. (2007). Towards a Time-Triggered Processor. *The 3rd UK Embedded Forum*. Durham, UK.

Audsley, N. and Burns, A. (1990). Real-time System Scheduling, University of York.

Audsley, N. C. (1991). Resource control for hard real time systems: A review. York, Real-Time Systems Research Group, Department of Computer Science, University of York.

Audsley, N. C., Burns, A., Richardson, M. F. and Wellings, A. J. (1993). Data Consistency In Hard Real-Time Systems, Department of Computer Science, University of York.

Avižienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing* **1**(1): 11 - 33.

Ayavoo, D., Pont, M. J. and Parker, S. (2004). *Using simulation to support the design of distributed embedded control systems: a case study*. The 1st UK Embedded Forum, Birmingham, UK, Published by University of Newcastle upon Tyne.

Ayavoo, D., Pont, M. J., Short, M. and Parker, S. (2007). "Two novel shared-clock scheduling algorithms for use with 'Controller Area Network' and related protocols." *Microprocessors & Microsystems* **31**(5): 326-334.

Baker, T. P. and Shaw, A. (1988). *The Cyclic Executive Model and Ada*. Real-Time Systems Symposium Huntsville, AL, USA.

Baruah, S., Buttazzo, G., Gorinsky, S. and Lipari, G. (1999). Scheduling periodic task systems to minimize output jitter. *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*. Hong Kong, China**:** 62 - 69.

Baruah, S. K. (2006). "The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors." *Real-Time Systems* **32**(1 - 2): 9 - 20.

Baruah, S. K., Howell, R. R. and Rosier, L. E. (1990). "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor." *Real-Time Systems* **2**: 301 - 324.

Basumallick, S. and Nilsen, K. (1994). *Cache Issues in Real-Time Systems*. ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems.

Bate, I. J. (1998). Scheduling and Timing Analysis for Safety Critical Real-Time Systems. *Department of Computer Science*. York, University of York. **Doctor of Philosophy**.

Bate, I. J. (2000). *Introduction to Scheduling and Timing Analysis*. The Use of Ada in Real-Time Systems, IEE Conference Publication 00/034.

Bautista, R., Pont, M. J. and Edwards, T. (2005). *Comparing the performance and resource requirements of 'PID' and 'LQR' algorithms when used in a practical embedded control system: A pilot study*. The 2nd UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne.

Berg, C., Engblom, J. and Wilhelm, R. (2004). *Requirements for and Design of a Processor with Predictable Timing*. Design of Systems with Predictable Behaviour, Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss.

Berg, T. B. (2009). "Maintaining I/O Data Coherence in Embedded Multicore Systems." *IEEE Micro* **29**(3): 10 - 19.

Bergenhem, C. (2007). Time-triggered technology for Automation and Machinery Control Communities. N.-N. I. Centre, SP Technical Research Institute of Sweden.

Bershad, B. N. (1993). Practical Considerations for Non-Blocking Concurrent Objects. *The 13th International Conference on Distributed Computing Systems*. Pittsburgh, PA, USA**:** 264 - 273.

Bjerregaard, T. and Mahadevan, S. (2006). "A survey of research and practices of Network-on-chip." *ACM Computing Surveys* **38**(1): 1.

Bober, T. and Shih, F. Y. (2009). "Image Processing-Based Methodology for Optimizing Automotive Ignition Timing." *IEEE Transactions on Vehicular Technology* **58**(1): 85 - 92.

Bordin, M. and Vardanega, T. (2007). Correctness by Construction for High-Integrity Real-Time Systems: A Metamodel-Driven Approach. *Reliable Software Technologies - Ada Europe 2007*, Springer Berlin / Heidelberg. **4498:** 114 - 127.

Borkar, S. (2007). Thousand Core Chips - A Technology Perspective. *44th IEEE Design Automation Conference*. San Diego, California**:** 746 - 749.

Borriello, G. and Want, R. (2000). "Embedded Computation meets the World Wide Web." *Communications of the ACM* **43**(5): 59 - 66.

Brajou, F. and Ricco, P. (2004). The Airbus A380 - an AFDX-based flight test computer concept. *Systems Readiness Technology Conference (AUTOTESTCON)*. Geneva, Switzerland**:** 460 - 463.

Brandenburg, B. B., Calandrino, J. M. and Anderson, J. H. (2008a). On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. *Real-Time Systems Symposium*. Barcelona, Spain**:** 157 - 169.

Brandenburg, B. B., Calandrino, J. M., Block, A., Leontyev, H. and Anderson, J. H. (2008b). Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*. St. Louis, MO**:** 342 - 353.

Bui, B. D., Caccamo, M., Sha, L. and Martinez, J. (2008). Impact of Cache Partitioning on Multi-Tasking Real Time Embedded Systems. *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*. Kaohsiung, Taiwan**:** 101 - 110.

Burchard, A., Liebeherr, J., Oh, Y. and Son, S. H. (1995). "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems." *IEEE Transactions on Computers* **44**(12): 1429 - 1442.

Burns, A., Hayes, N. and Richardson, M. F. (1995). "Generating Feasible Cyclic Schedules." *Control Engineering Practice* **3**(2): 151 - 162.

Buttazzo, G. and Cervin, A. (2007). Comparative Assessment and Evaluation of Jitter Control Methods. *The 15th International Conference on Real-Time and Network Systems (RTNS '07)*. Loria, Nancy, France**:** 163 - 172.

Buttazzo, G. C. (2002a). Real-Time Operating Systems: Problems and Novel Solutions. *The 7th International Synopsium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '02)*. Oldenburg, Germany, Springer Berlin / Heidelberg. **2469:** 37 - 51.

Buttazzo, G. C. (2002b). Scalable Applications for Energy-Aware Processors. *The 2nd International Conference on Embedded Software (EMSOFT '02)*. A. Sangiovanni-Vincentelli and J. Sifakis, Lecture Notes In Computer Science, Springer-Verlag Berlin Heidelberg. **2491:** 153 - 165.

Buttazzo, G. C. (2005a). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Springer.

Buttazzo, G. C. (2005b). "Rate Monotonic vs. EDF: Judgment Day." *Real-Time Systems* **29**(1): 5 - 26.

Buttazzo, G. C. and Caccamo, M. (1999). "Minimizing Aperiodic Response Times in a Firm Real-Time Environment." *IEEE Transactions on Software Engineering* **25**(1): 22 - 32.

Caccamo, M., Lipari, G. and Buttazzo, G. C. (1999). *Sharing Resources among Periodic and Aperiodic Tasks with Dynamic Deadlines*. IEEE Real-Time Systems Symposium.

Cervin, A., Henriksson, D., Lincoln, B., Eker, J. a. and Arzen, K.-E. (2003). How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*. **23:** 16 - 30.

Chapman, R. (2006). Correctness by Construction: A Manifesto for High Integrity Software. *The 10th Australian workshop on Safety Critical Systems and Software*. Sydney, Australia. **55:** 43 - 46.

Charette, R. N. (2009). This Car Runs on Code. *IEEE Spectrum*.

Chen, J. and Burns, A. (1997a). A Fully Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems. York, UK, Real-Time System Group, Department of Computer Science, University of York.

Chen, J. and Burns, A. (1997b). A Three-Slot Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems. York, UK, Real-Time System Group, Department of Computer Science, University of York.

Chen, J. and Burns, A. (1998). Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Consensus. *The 10th Euromicro Workshop on Real-Time Systems*. Berlin, Germany**:** 2 - 9.

Chen, Z., Pittman, R. N. and Forin, A. (2009). MultiCore eMIPS. Redmond, WA, Microsoft Research, Microsoft Corporation.

Cho, H. (2006). Utility Accrual Real-Time Scheduling and Synchronization on Single and Multiprocessors: Models, Algorithms, and Tradeoffs. *Computer Engineering*. Blacksburg, Virginia, Virginia Polytechnic Institute and State University. **Doctor of Philosophy**.

Cho, H., Ravindran, B. and Jensen, E. D. (2005). A Space-Optimal Wait-Free Real-Time Synchronization Protocol. *The 17th Euromicro Conference on Real-Time Systems (ECRTS'05)***:** 79 - 88.

Cho, H., Ravindran, B. and Jensen, E. D. (2007). "Space-Optimal, Wait-Free Real-Time Synchronization." *IEEE Transactions on Computers* **56**(3): 373 - 384.

Cho, H., Ravindran, B. and Jensen, E. D. (2010). "Lock-Free Synchronization for Dynamic Embedded Real-Time Systems." *ACM Transactions on Embedded Computing Systems (TECS)* **9**(3).

Clark, D. (1989). HIC: An Operating System for Hierarchies of Servo Loops. *IEEE International Conference on Robotics and Automation*. Scottsdale, AZ , USA**:** 1004 - 1009.

Coffman Jr, E. G. and Graham, R. L. (1972). "Optimal Scheduling for Two-Processor Systems." *Acta Informatica* **1**: 200 - 213.

Cottet, F. and David, L. (1999). *A Solution to the Time Jitter Removal in Deadline Based Scheduling of Real-time Applications*. The 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada.

Crespo, A., Ripoll, I. and Masmano, M. (2010). Partitioned Embedded Architecture based on Hypervisor: the XtratuM approach. *European Dependable Computing Conference*. Valencia, Spain**:** 67 - 72.

Das, A., Lakhani, F. N., Gendy, A. K. and Pont, M. J. (2009). Two simple patterns to support the development of reliable, realtime embedded systems. *European Conference on Pattern Languages of Programs (EuroPLoP)*.

Davis, R., Merriam, N. and Tracey, N. (2000). How Embedded Applications using an RTOS can stay within On-chip Memory Limits. *Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on RealTime Systems***:** 43 - 50.

Davis, R. I., Tindell, K. W. and Burns, A. (1993). *Scheduling Slack Time in Fixed Priority Pre-emptive Systems*. Real-Time Systems Symposium.

Dhall, S. K. and Liu, C. L. (1978). "On a Real-Time Scheduling Problem." *Operations Research* **26**(1): 127 - 140.

Digilent Inc. (2004). "Spartan 3 Board." from http://www.digilentinc.com/.

Dijkstra, E. W. (1997). The tide, not the waves. *Beyond Calculation: The Next Fifty Years of Computing*. P. Denning and R. Metcalfe, Copernicus (Springer-Verlag).

Dimond, R., Madhvani, N. and Mathai, J. (2002). Software for NASA in 2050: an impossible mission? *SURPRISE*.

Douce, C. R., Layzell, P. J. and Buckley, J. (1999). *Spatial Measures of Software Complexity*. 11th Annual Workshop of the Psychology of Programming Interest Group, Leeds, UK.

Duller, A., Towner, D., Panesar, G., Gray, A. and Robbins, W. (2005). *picoArray technology: the tool's story*. Design, Automation and Test in Europe (DATE '05), IEEE Computer Society.

Ebert, C. and Jones, C. (2009). "Embedded Software: Facts, Figures and Future." *IEEE Computer* **42**(4): 42 - 52.

Edwards, T., Pont, M. J., Scotson, P. and Crumpler, S. (2004). *A test-bed for evaluating and comparing designs for embedded control systems*. The 1st UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne.

Einstein, A. (1933). On the Method of Theoretical Physics. The Herbert Spencer Lecture. Oxford.

Ekelin, C. and Jonsson, J. (2001). Evaluation of Search Heuristics for Embedded System Scheduling Problems. *The 7th International Conference on Principles and Practice of Constraint Programming***:** 640 - 654.

El-Haj-Mahmoud, A., AL-Zawawi, A. S., Anantaraman, A. and Rotenberg, E. (2005). Virtual Multiprocessor: An Analyzable, HighPerformance Microarchitecture for RealTime Computing. *The International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '05)*. San Francisco, CA, USA**:** 213 - 224.

Engblom, J. (2002). Processor Pipelines and Static Worst-Case Execution Time Analysis. *Dept. of Information Technology*. Acta Universitatis Upsaliensis, Uppsala University. **Doctor of Philosophy**.

Engel, F., Kuz, I., Petters, S. M. and Ruocco, S. (2004). *Operating Systems on SoCs: A good idea?* Embedded Real-Time Systems Implimentation (ERTSI 2004) Workshop, Lisbon, Portugal.

Ergen, S. C., Sangiovanni-Vincentelli, A., Sun, X., Tebano, R., Alalusi, S., Audisio, G. and Sabatini, M. (2009). "The Tire as an Intelligent Sensor." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **28**(7): 941 - 955.

Ferri, C., Wood, S., Moreshet, T., Bahar, R. I. and Herlihy, M. (2010). "Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems." *Journal of Parallel and Distributed Computing*.

Fich, F. E., Luchangco, V., Moir, M. and Shavit, N. (2005). Obstruction-Free Algorithms Can Be Practically Wait-Free. *Distributed Computing*, Springer Berlin / Heidelberg. **3724:** 78 - 92.

Fidge, C. J. (2002). Real-Time Scheduling Theory. *Technical Report*, Software Verficiation Research Center, University of Queensland.

Ford, H. and Crowther, S. (1922). *My Life and Work*, Garden City, New York, USA: Garden City Publishing Company, Inc.

Fort, B., Capalija, D., Vranesic, Z. G. and Brown, S. D. (2006). A Multithreaded Soft Processor for SoPC Area Reduction. *The 14th Annual IEEE*

Symposium on Field-Programmable Custom Computing Machines (FCCM '06). Napa, CA, USA**:** 131 - 142.

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

Gangoiti, U., Marcos, M. and Estévez, E. (2005). Using Cyclic Executives for Achieving Closed Loop Co-simulation. *The joint 44th IEEE Conference on Decision and Control and the European Control Conference*. Seville, Spain.

Ganssle, J. (2003). *Embedded Systems Dictionary*, CMP Books.

Ganssle, J. (2008). *The Art of Designing Embedded Systems*, Newnes, Elsevier.

Gary, S. and Tyger, K. (2004). Shared Peripheral Architecture. United States. **US2004/0088459 A1**.

Gendy, A. K., Dong, L. and Pont, M. J. (2007a). Improving the performance of time-triggered embedded systems by means of a scheduler agent. *ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE '07)*. Las Vegas, Nevada, USA.

Gendy, A. K. and Pont, M. J. (2007b). *Towards a Generic "Single-Path Programming" Solution With Reduced Power Consumption*. ASME International Design Engineering Technical Conference & Computers and Information in Engineering Conference (IDETC/CIE '07), Las Vegas, Nevada, USA.

Gendy, A. K. and Pont, M. J. (2008a). "Automatically Configuring Time-Triggered Schedulers for Use With Resource-Constrained, Single-Processor Embedded Systems." *IEEE Transactions on Industrial Informatics* **4**(1): 37 - 46.

Gendy, A. K. and Pont, M. J. (2008b). Automating the Processes of Selecting an Appropriate Scheduling Algorithm and Configuring the Scheduler Implementation for Time-Triggered Embedded Systems. *The 27th international conference on Computer Safety, Reliability, and Security*. M. D. Harrison and M.-A. Sujan. Newcastle upon Tyne, UK, Springer-Verlag**:** 440 - 453.

Gendy, A. K. G. (2009). Techniques for scheduling time-triggered resource-constrained embedded systems. *Engineering*. Leicester, University of Leicester. **Doctor of Philosophy:** 199.

Gill, C. D., Levine, D. L. and Schmidt, D. C. (1999). Dynamic Scheduling Strategies for Avionics Mission Computing. *The 17th IEEE/AIAA Digital Avionics Systems Conference (DASC '99)*.

Goossens, J. and Devillers, R. (1997). "The Non-Optimality of the Monotonic Priority Assignments for Hard Real-Time Offset Free Systems." *Journal of Real-Time Systems* **13**: 107–126.

Goossens, J. and Richard, P. (2004). Overview of real-time scheduling problems. *Euro Workshop on Project Management and Scheduling*.

Gordon, J. and Tulip, A. (1997). "Resource Scheduling." *Internattonal Journal of Project Management* **15**(6): 359 - 370.

Griswold, W. G. and Notkin, D. (1993). "Automated Assistance for Program Restructuring." *ACM Transactions on Software Engineering and Methodology* **2**(3): 228 - 269.

Guerin, X. and Petrot, F. (2009). A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-Core SoCs. *The 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '09)*. Boston, MA**:** 153 - 160.

Gupta, N., Mandal, S. K., Malave, J., Mandal, A. and Mahapatra, R. N. (2007). A Hardware Scheduler for Real Time Multiprocessor System on Chip. *The 23rd International Conference on VLSI Design (VLSID '10)*. Bangalore, India**:** 264 - 269.

Hanif, M. A., Pont, M. J. and Ayavoo, D. (2008). Implementing a Flexible Time-Triggered Architecture for Deeply Embedded Applications. *The 4th UK Embedded Forum*. A. Koelmans and K. Maharatna. Southampton, UK, Institute of Engineering and Technology (IET)**:** 106 - 115.

Haynes, B. P. (2008). "An evaluation of the impact of the office environment on productivity." *Facilities* **26**(5/6): 178 - 195.

Heath, S. (2003). *Embedded Systems Design*, Newnes.

Heiser, G. (2007). Virtualization for Embedded Systems.

Helmerich, A., Koch, N., Mandel, L., Braun, P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schätz, B., Wild, T. and Wimmel, G. (2005). Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area. Munich, Germany, FAST GmbH, Technische Universität.

Henzinger, T. A. (2008). "Two challenges in embedded systems design: predictability and robustness." *Philosophical Transactions of The Royal Society A* **366**(1881): 3727 - 3736.

Henzinger, T. A., Horowitz, B. and Kirsch, C. M. (2003). Giotto: A Time-triggered Language for Embedded Programming. *Proceedings of the IEEE*. **91:** 84-99.

Herlihy, M. (1993). "A Methodology for Implementing Highly Concurrent Data Objects." *ACM Transactions on Programming Languages and Systems (TOPLAS)* **15**(5): 745 - 770.

Herlihy, M. and Luchangco, V. (2008). "Distributed Computing and the Multicore Revolution." *ACM SIGACT News* **39**(1): 62 - 72.

Herlihy, M., Luchangco, V. and Moir, M. (2003). Obstruction-Free Synchronization: Double-Ended Queues as an Example. *The 23rd International Conference on Distributed Computing Systems (ICDCS '03)*. Providence, Rhode Island, IEEE Computer Society**:** 522 - 530.

Herlihy, M. and Moss, J. E. B. (1993). Transactional Memory: Architectural Support for Lock-Free Data Structures. *The 20th Annual International Symposium on Computer Architecture*. **21:** 289 - 300.

Hoare, C. A. R. (1981). "The Emperor's Old Clothes." *Communications of the ACM* **24**(2): 75 - 83.

Holden, P. (2005). Develop FFT apps on low-power MCUs. *Embedded Systems Design*. **18**.

Holman, P. and Anderson, J. H. (2006). "Supporting lock-free synchronization in Pfair-scheduled real-time systems." *Journal of Parallel and Distributed Computing* **66**(1): 47 - 67.

Hong, S. H. (1995). "Scheduling Algorithm of Data Sampling Times in the Integrated Communication and Control Systems." *IEEE Transactions on Control Systems Technology* **3**(2): 225 - 230.

Huang, C.-Y., Chang, L.-P. and Kuo, T.-W. (2003). A cyclic-executive-based QoS guarantee over USB. *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium***:** 88 - 95.

Huang, H., Pillai, P. and Shin, K. G. (2002). *Improving wait-free algorithms for interprocess communication in embedded realtime systems*. Usenix Annual Technical Conference.

Hudson, J. M., Christensen, J., Kellogg, W. A. and Erickson, T. (2002). “I’d Be Overwhelmed, But It’s Just One More Thing to Do:” Availability and Interruption in Research Management. *Conference on Human Factors in Computing Systems: Changing our world, changing ourselves*. Minneapolis, Minnesota, USA**:** 97 - 104.

Hughes, Z. (2009). Design and Evaluation of a Predictable Embedded Processor for Use in Time-Triggered Applications. *Embedded Systems Laboratory*. Leicester, UK, University of Leicester. **Doctor of Philosophy:** 199.

Hughes, Z. and Pont, M. J. (2004). Design and Test of a Task Guardian for Use in TTCS Embedded Systems. *UK Embedded Forum*. Birmingham, UK.

Hughes, Z. M. and Pont, M. J. (2008). "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed." *Transactions of the Institute of Measurement and Control* **30**(5): 427-450.

Hughes, Z. M., Pont, M. J. and Ong, H. L. R. (2005). *The PH Processor: A soft embedded core for use in university research and teaching*. The 2nd UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne.

IEEE (1990). "IEEE standard glossary of software engineering terminology." *IEEE Std 610.12-1990*.

ITRS (2007). International Technology Roadmap for Semiconductors 2007 Edition.

Jeffay, K., Stanat, D. F. and Martel, C. U. (1991). *On non-preemptive scheduling of periodic and sporadic tasks*. The 12th IEEE Symposium on Real-Time Systems.

Jeffay, K. and Stone, D. L. (1993). Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. *Real-Time Systems Symposium*. Raleigh Durham, NC, USA**:** 212 - 221.

Joseph, M. (1996). *Real-time Systems: Specification, Verification and Analysis*, Prentice Hall.

Joshi, A. (2009). Embedded Systems: Technologies and Markets, BCC Research.

Kalinsky, D. (2001, 26 February 2001). "Context Switch."

Kane, G. (1987). *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Karrenbauer, A. and Rothvo, T. (2009). An Average-Case Analysis for Rate-Monotonic Multiprocessor Real-time Scheduling. *The 17th Annual European Symposium on Algorithms (ESA'09)*. Copenhagen.

Katcher, D. I., Arakawa, H. and Strosnider, J. K. (1993). "Engineering and Analysis of Fixed Priority Schedulers." *IEEE Transactions on Software Engineering* **19**(9): 920 - 934.

Key, S. and Pont, M. J. (2004). *Implementing PID control systems using resource-limited embedded processors*. The 1st UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne.

Key, S., Pont, M. J. and Edwards, S. (2003). Implementing low-cost TTCS systems using assembly language. *The 8th European Conference on Pattern Languages of Programs (EuroPLoP '03)*. Germany**:** 667 - 690.

Kim, B. K. and Shin, K. G. (1997). Task Assignment and Scheduling for Open Real-Time Control Systems. *The American Control Conference*. Albuquerque, New Mexico**:** 3664 - 3668.

Kim, H., Kim, Y., Kim, B. and Yoo, H.-J. (2009). A Wearable Fabric Computer by Planar-Fashionable Circuit Board Technique. *Sixth International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2009)*. Berkeley, CA**:** 282 - 285.

Kim, J., Lee, S. J. and Marschke, G. (2004). Research Scientist Productivity and Firm Size: Evidence from Panel Data on Inventors. *Discussion Paper Series*, Institute of Economic Research, Korea University.

Kirner, R. and Puschner, P. (2003). *Discussion of Misconceptions about WCET Analysis*. The 3rd Euromicro International Workshop on WCET Analysis.

Kohout, P., Ganesh, B. and Jacob, B. (2003). Hardware support for real-time operating systems. *The 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Newport Beach, CA, USA, ACM.

Koopman, P. (1996). Embedded System Design Issues (the Rest of the Story). *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96)*. Austin, TX , USA 310 - 317.

Kopetz, H. (1991a). Event-Triggered Versus Time-Triggered Real-Time Systems. *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, Springer-Verlag.

Kopetz, H. (1991b). *Event-Triggered Versus Time-Triggered Real-Time Systems*. International Workshop on Operating Systems of the 90s and Beyond, Springer-Verlag

Kopetz, H. (2008). The Complexity Challenge in Embedded System Design. *The 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08)*. Orlando, FL**:** 3 - 12.

Kopetz, H., Ademaj, A., Grillinger, P. and Steinhammer, K. (2005). The Time-Triggered Ethernet (TTE) Design. *The 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE Computer Society.

Kopetz, H. and Grünsteidl, G. (1994). "TTP - A Protocol for Fault-Tolerant Real-Time Systems." *Computer* **27**(1): 14 - 23.

Kopetz, H. and Reisinger, J. (1993). The Non-Blocking Write Protocol NBW A Solution to a Real-Time Synchronization Problem. *Real-Time Systems Symposium*. Raleigh Durham, NC, USA**:** 131 - 137.

Kremer, M. and Maskin, E. (1996). Wage Inequality and Segregation by Skill, NBER Working Paper.

Kuacharoen, P., Shalan, M. and Mooney, V. (2003). *A configurable hardware scheduler for real-time systems*. International Conference on Engineering of Reconfigurable Systems and Algorithms.

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P. and Farkas, K. I. (2004). Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *The 31st Annual International Symposium on Computer Architecture (ISCA'04)*. **32:** 64 - 75.

Kumar, S., Hughes, C. J. and Nguyen, A. (2007). Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *The 34th annual International Symposium on Computer Architecture (ISCA '07)*. San Diego, California, USA**:** 162 - 173.

Kuo, T.-W., Liu, Y.-H. and Lini, K.-J. (2000). Efficient On-Line Schedulability Tests for Priority Driven Real-Time Systems. *The 6th Real-Time Technology and Applications Symposium (RTAS '00)*. Washington, DC , USA**:** 4 - 13.

Kurian, S. and Pont, M. J. (2007). "The maintenance and evolution of resource-constrained embedded systems created using design patterns." *Journal of Systems and Software* **80**(1): 32 - 41.

Lai, B.-C. C., Schaumont, P. and Verbauwhede, I. (2005). A Light-Weight Cooperative Multi-threading with Hardware Supported Thread-Management on an Embedded Multi-Processor System. *The 39th Asilomar Conference on Signals, Systems and Computers***:** 1647 - 1651.

Lamport, L. (1977). "Concurrent Reading and Writing." *Communications of the ACM* **20**(11): 806 - 811.

Lamport, L. (1986a). "On interprocess communication - Part I: Basic formalism." *Distributed Computing* **1**(2): 77- 85.

Lamport, L. (1986b). "On interprocess communication - Part II: Algorithms." *Distributed Computing* **1**(2): 86 - 101.

Laplante, P. A. (2004). *Real-time Systems Design and Analysis*, Wiley-Interscience.

Laprie, J. C. (1992). *Dependability: Basic Concepts and Terminology*, Berlin, Germany: Springer-Verlag.

Lauzac, S., Melhem, R. and Mossé, D. (1998). Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor. *The 10th Euromicro Workshop on Real-Time Systems*. Berlin , Germany**:** 188 - 195.

Lauzac, S., Melhem, R. and Mossé, D. (2003). "An Improved Rate-Monotonic Admission Control and Its Applications." *IEEE Transactions on Computers* **52**(3): 337 - 350.

Lee, E. A. (2009). "Computing Needs Time." *Communications of the ACM* **52**(5): 70 - 79.

Leen, G. and Heffernan, D. (2002). "Expanding Automotive Electronic Systems." *IEEE Computer* **35**(1): 88 - 93.

Leen, G., Heffernan, D. and Dunne, A. (1999). "Digital networks in the automotive vehicle." *Computing and Control Engineering Journal* **10**(6): 257 - 266.

Leibson, S. (2007). Multicore microprocessors and embedded multicore SOCs have very different needs, Tensilica, Inc.

Lickly, B., Liu, I., Kim, S., Patel, H. D., Edwards, S. A. and Lee, E. A. (2008). Predictable Programming on a Precision Timed Architecture. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. Atlanta, GA, USA**:** 137 - 146.

Lin, K.-J. and Herkert, A. (1996). *Jitter Control in Time-Triggered Systems*. The 29th Annual Hawaii International Conference on System Sciences.

Liu, C. L. and Layland, J. W. (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." *Journal of the Association for Computing Machinery* **20**(1): 46-61.

Liu, J. W. S. and Ha, R. (1995). "Methods for validating real-time constraints." *Journal of Systems and Software* **30**(1-2): 85-98.

Locke, C. D. (1992). "Software Architecture for Hard Real-time Applications: Cyclic Executives vs. Fixed Priority Executives." *Real-Time Systems* **4**(1): 37 - 53.

Maaita, A. and Pont, M. J. (2005). *Using 'Planned Pre-emption' to Reduce Levels of Task Jitter in a Time-triggered Hybrid Scheduler*. The 2nd UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne.

Maier, R., Bauer, G., Stöger, G. and Poledna, S. (2002). "Time-triggered architecture: a consistent computing platform." *IEEE Micro* **22**(4): 36-45.

Marco, L. (1997). "Measuring Software Complexity." *Enterprise Systems Journal*.

Martí, P., Fuertes, J. M., Fohler, G. and Ramamritham, K. (2001). *Jitter Compensation for Real-Time Control Systems*. The 22nd IEEE Real-Time Systems Symposium.

Marti, P., Villa, R., Fuertes, J. M. and Fohler, G. (2001). *On Real-Time Control Tasks Schedulability*. European Control Conference, Porto, Portugal.

Martin, J. (2009). *Parallax Propeller Manual*.

Memik, S. O., Bozorgzadeh, E., Kastner, R. and Sarrafzadeh, M. (2001). A Super-Scheduler for Embedded Reconfigurable Systems. *International Conference on Computer Aided Design*, IEEE Press**:** 391 - 394.

Michael, M. M. and Scott, M. L. (1998). "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors." *Journal of Parallel and Distributed Computing* **51**(1): 1 - 26.

Miller, G. (1956). "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information." *Psychological Review* **63**(2): 81 - 97.

Mok, A. K.-L. (1983). Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. *Department of Electrical Engineering and Computer Science*. Cambridge, MA, USA, Massachusetts Institute of Technology (MIT). **Doctor of Philosophy:** 183.

Monot, A., Navet, N., Simonot, F. and Bavoux, B. (2010). Multicore scheduling in automotive ECUs. *Embedded Real-time Software and Systems*. Toulouse, France.

Mueller, F., Whalley, D. and Harmon, M. (1993). Predicting instruction cache behavior.

Multicore Association (2008). Multicore Communications API Specification V1.063 (MCAPI).

Mwelwa, C., Pont, M. J. and Ward, D. (2005). *Developing reliable embedded systems using a pattern-based code generation tool: A case study*. The 2nd UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne.

Nácul, A. C., Regazzoni, F. and Lajolo, M. (2007). Hardware Scheduling Support in SMP Architectures. *Design, Automation & Test in Europe Conference & Exhibition (DATE '07)*. Nice, France**:** 1 - 6.

Nahas, M. (2008). Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems. *Department of Engineering*. Leicester, University of Leicester. **Doctor of Philosophy**.

Nahas, M., Pont, M. J. and Jain, A. (2004). Reducing task jitter in shared-clock embedded systems using CAN. *The 1st UK Embedded Forum*. A. Koelmans, A. Bystrov and M. J. Pont. Birmingham, UK, Published by University of Newcastle upon Tyne**:** 184-195.

Nebhrajani, V. A. (2007). Asynchronous FIFO Architectures.

Nghiem, T., Pappas, G. J., Girard, A. and Alur, R. (2006). *Time-triggered Implementations of Dynamic Controllers*. The 6th ACM & IEEE International Conference on Embedded Software, Seoul, Korea.

Nissanke, N. (1997). *Realtime Systems*, Prentice-Hall.

Noorden, R. V. (2006). "Moving towards a graphene world." *Nature* **442**: 228 - 229.

O'Conaill, B. and Frohlich, D. (1995). Timespace in the Workplace: Dealing with Interruptions. *Human Factors in Computing Systems*. Denver, CO.

Obermaisser, R., Salloum, C. E., Huber, B. and Kopetz, H. (2009). "From a Federated to an Integrated Automotive Architecture." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **28**(7): 956 - 965.

Oh, S.-H. and Yang, S.-M. (1998). *A Modified Least-Laxity-First scheduling algorithm for real-time tasks*. The 5th International Conference on Real-Time Computing Systems and Applications (RTCSA '98), Hiroshima, IEEE Computer Society.

Oh, Y. and Son, S. H. (1993). Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem. Charlottesville, VA, University of Virginia.

Oh, Y. and Son, S. H. (1995). Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems. Thornton Hall, Charlottesville, VA, USA, Department of Computer Science, University of Virginia.

Olsen, C. (2007). "Getting the most out of EMV with contactless cards." *Card Technology Today* **19**(4): 10 - 11.

Paolieri, M., Quiñones, E., Cazorla, F. J., Bernat, G. and Valero, M. (2009). Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. *The 36th Annual International Symposium on Computer Architecture*. Austin, TX, USA**:** 57 - 68.

Pellizzoni, R. and Lipari, G. (2004). A New Sufficient Feasibility Test for Asynchronous Real-Time Periodic Task Sets. *16th Euromicro Conference on Real-Time Systems (ECRTS '04)*. Scuola Superiore S. Anna, Pisa, Italy, IEEE**:** 204 - 211.

Peng, L., Peir, J.-K., Prakash, T. K., Chen, Y.-K. and Koppelman, D. (2007). Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study. *IEEE International Performance, Computing, and Communications Conference (IPCCC '07)*. New Orleans, LA**:** 55 - 64.

Peterson, G. L. (1983). "Concurrent Reading While Writing." *ACM Transactions on Programming Languages and Systems (TOPLAS)* **5**(1): 46 - 55.

Phatrapornnant, T. and Pont, M. J. (2006). "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling." *IEEE Transactions on Computers* **55**(2): 113 - 124.

Pitter, C. and Schoeberl, M. (2007). Towards a Java Multiprocessor. *The 5th international workshop on Java technologies for real-time and embedded systems*. Vienna, Austria, ACM. **319:** 144 - 151.

Pont, M. J. (2001). *Patterns for time-triggered embedded systems : building reliable applications with the 8051 family of microcontrollers*. Harlow, Addison-Wesley.

Pont, M. J. (2002). *Embedded C*. London, Addison-Wesley.

Pont, M. J., Kurian, S., Wang, H. and Phatrapornnant, T. (2007). *Selecting an appropriate scheduler for use with time-triggered embedded systems*. The 12th European Conference on Pattern Languages of Programs (EuroPLoP '07).

Profeta III, J. A., Andrianos, N. P., Yu, B., Johnson, B. W., DeLong, T. A., Guaspari, D. and Jamsek, D. (1996). "Safety-Critical Systems Built with COTS." *IEEE Computer* **29**(11): 54 - 60.

Puschner, P. and Burns, A. (2002). *Writing Temporally Predictable Code*. The 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.

Puschner, P. and Kirner, R. (2006). From Time-Triggered to Time-Deterministic Real-Time Systems *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, Springer Boston. **225:** 115 - 124.

Raj, H. and Schwan, K. (2007). High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. *The 16th International Symposium on High Performance Distributed Computing*. Monterey, California, USA**:** 179 - 188.

Ramamritham, K. and Stankovic, J. A. (1994). "Scheduling algorithms and operating systems support for real-time systems." *IEEE* **82**(1): 55 - 67.

Redmill, F. (1992). "Computers in Safety-Critical Applications." *Computing & Control Engineering Journal* **3**(4): 178 - 182.

Reeves, G. (1998) Re: What Really Happened on Mars? *Risks-Forum Digest* **19**,

Rodriguez-Andina, J. J., Moure, M. J. and Valdes, M. D. (2007). "Features, Design Tools, and Application Domains of FPGAs." *IEEE Transactions on Industrial Electronics* **54**(4): 1810 - 1823.

Roper, K. O. and Juneja, P. (2007). "Valuation of AW: modeling the impacts of distractions." *Facilities* **25**(13/14): 536 - 553.

Rosén, J., Andrei, A., Eles, P. and Peng, Z. (2007). Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. *IEEE International Real-Time Systems Symposium***:** 49 - 60.

Rouncejield, M., Hughes, J. A., Rodden, T. and Viller, S. (1994). Working with "Constant Interruption": CSCW and the Small Office. *ACM conference on Computer Supported Cooperative Work*. Chapel Hill, North Carolina, United States**:** 275 - 286.

RTCA SC-167 / EUROCAE WG- 12 (1992). DO-178B: Software Considerations in Airborne Systems and Equipment Certification.

Sachitanand, N. N. (2002). Embedded systems - A new high growth area. *The Hindu*. Bangalore, India. .

Sanfridson, M. (2000). Timing problems in distributed real-time computer control systems. Stockholm, Sweden, Mechatronics Lab, Department of Machine Design, Royal Institute of Technology.

Sasso, W. C. (1986). Measuring Office Complexity, Stern School of Business, New York University.

Scheler, F. and Schröder-Preikschat, W. (2006). *Time-triggered vs. event-triggered: A matter of configuration?* GI/ITG Workshop on Non-Functional Properties of Embedded Systems, Nuremberg, Germany, VDE Verlag GmbH.

Schneider, J. (2000). *Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems*. The 21st IEEE Real-Time Systems Symposium (RTSS'00), Orlando, Florida, USA.

Schoeberl, M., Brandner, F. and Vitek, J. (2010). RTTM: Real-Time Transactional Memory. *The 2010 ACM Symposium on Applied Computing*. Sierre, Switzerland**:** 326 - 333.

Schoeberl, M., Puschner, P. and Kirner, R. (2009). A Single-Path Chip-Multiprocessor System. *The 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS '09)***:** 47 - 57.

Schossmaier, K. and Weiss, B. (1999). *An Algorithm for Fault-Tolerant Clock State & Rate Synchronization*. The 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99), Lausanne.

Schultz, K. L., McClain, J. O. and Thomas, L. J. (2003). "Overcoming the dark side of worker flexibility." *Journal of Operations Management* **21**(1): 81 - 92.

Sessions, R. (2009). The IT Complexity Crisis: Danger and Opportunity.

Sha, L., Abdelzaher, T., Årzén, K.-E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J. and Mok, A. K. (2004). "Real Time Scheduling Theory: A Historical Perspective." *Real-Time Systems* **28**(2 - 3): 101 - 155.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990). "Priority Inheritance Protocols: An Approach to Real-time Synchronization." *IEEE Transactions on Computers* **39**(9): 1175 - 1185.

Shavit, N. and Touitou, D. (1997). "Software transactional memory." *Distributed Computing* **10**(2): 99 - 116.

Short, M. and Pont, M. J. (2005). *Hardware in the loop simulation of embedded automotive control system*. The 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005).

Short, M., Pont, M. J. and Fang, J. (2008). Exploring the Impact of Task Preemption on Dependability in Time-Triggered Embedded Systems: A Pilot Study. *Euromicro Conference on Real-Time Systems, 2008 (ECRTS '08)***:** 83-91.

Siemers, C., Falsett, R., Seyer, R. and Ecker, K. (2005). "Reliable event-triggered systems for mechatronic applications." *Journal of Systems and Software: Parallel and distributed real-time systems* **77**(1): 17-26.

Simpson, H. R. (1990). "Four-slot fully asynchronous communication mechanism." *IEE Proceedings on Computers and Digital Techniques* **137**(1): 17 - 30.

Själander, M., Terechko, A. and Duranton, M. (2008). A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. *The 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD '08)*. Parma, Italy**:** 149 - 157.

Sneed, H. M. (2008). Measuring 75 Million Lines of Code. *Software Process and Product Measurement*, Springer Berlin / Heidelberg. **5338:** 271 - 286.

Sorenson, P. G. and Hamacher, V. C. (1975). "A Real-time System Design Methodology." *INFOR* **13**(1): 1 - 18.

Spira, J. B. and Feintuch, J. B. (2005). The Cost of Not Paying Attention: How Interruptions Impact Knowledge Worker Productivity, Basex, Inc.

Spuri, M., Buttazzo, G. and Sensini, F. (1995). *Robust Aperiodic Scheduling under Dynamic Priority Systems*. The 16th IEEE Real-Time Systems Symposium, Pisa, Italy.

Stankovic, J. A. (1988). "Misconceptions About Real-time Computing." *IEEE Computer* **21**(10): 10 - 19.

Stankovic, J. A. and Ramamritham, K. (1990). "What is Predictability for Real-time Systems? ." *Journal of Real-Time Systems* **2**(4): 247-254.

Stankovic, J. A. and Ramamritham, K. (1993). What is Predictability for Real-Time Systems? Amherst, MA, USA University of Massachusetts.

Stankovic, J. A., Spuri, M., Natale, M. D. and Buttazzo, G. C. (1995). "Implications of Classical Scheduling Results for Real-Time Systems." *IEEE Computer* **28**(6): 16 - 25.

Stärner, J., Adomat, J., Furunäs, J. and Lindh, L. (1996). Real-Time Scheduling Co-Processor in Hardware for Single and Multiprocessor Systems. *The 22nd EUROMICRO Conference 'Beyond 2000: Hardware and Software Design Strategies'*. Prague , Czech Republic**:** 509 - 512.

Stewart, D. B. (2001). Real Time. *Embedded Systems Programming*, CMP Media LLC. **14:** 87-88.

Sundell, H. (2004). Efficient and Practical Non-Blocking Data Structures. *Department of Computing Science*. Göteborg, Sweden, Chalmers University of Technology and Göteborg University. **Doctor of Philosophy**.

Sundell, H. and Tsigas, P. (2008). "Lock-free deques and doubly linked lists." *Journal of Parallel and Distributed Computing* **68**(7): 1008 - 1020.

Thiele, L. and Wilhelm, R. (2004). "Design for Timing Predictability." *Real-Time Systems* **28**(2): 157-177.

Tindell, K. (1994). Adding Time-Offsets to Schedulability Analysis. York, England, Real-Time Systems Research Group, Department of Computer Science, University of York**:** 94 - 221.

Törngren, M. (1998). "Fundamentals Of Implementing Real-Time Control Applications In Distributed Computer Systems." *Journal of Real-Time Systems* **14**: 219 - 250.

Treiber, R. K. (1986). Systems Programming: Coping with Parallelism. *Research Report*. San Jose, California, USA, IBM Almaden Research Center.

Tromp, J. (1989). How to Construct an Atomic Variable. *The 3rd International Workshop on Distributed Algorithms***:** 292 - 302.

Tsigas, P. and Zhang, Y. (1999). Non-blocking Data Sharing in Multiprocessor Real-Time Systems. *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*. Hong Kong, China**:** 247 - 254.

TTE Systems (2010). Datasheet and Programming Guide for the TTE32-SM3 Microcontroller (r1).

Turley, J. (1999). Embedded processors by the numbers. *Embedded Systems Programming*. **12**.

Valois, J. D. (1994). *Implementing Lock-Free Queues*. The 7th International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada.

Valois, J. D. (1995). Lock-Free Linked Lists Using Compare-and-Swap. *Annual ACM Symposium on Principles of Distributed Computing*. Ottowa, Ontario, Canada, ACM**:** 214 - 222.

Wang, H. and Pont, M. J. (2008). *Design and Implementation of a Static Pre-emptive Scheduler with Highly-Predictable behaviour*. The 4th UK Embedded Forum, Southampton, UK, Institute of Engineering and Technology (IET).

Wang, H., Pont, M. J. and Kurian, S. (2007). Patterns Which Help to Avoid Conflicts over Shared Resources in Time-triggered Embedded Systems Which Employ a Pre-emptive Scheduler. *The 12th European Conference on Pattern Languages of Programs (EuroPLoP '07)*. Irsee Monastery, Bavaria, Germany.

Ward, N. J. (1991). *The static analysis of a safety-critical avionics control system*. Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, SaRS, Ltd.

Wolf, W. (2002). "What Is Embedded Computing?" *IEEE Computer* **35**(1): 136 - 137.

Wolf, W., Jerraya, A. A. and Martin, G. (2008). "Multiprocessor System-on-Chip (MPSoC) Technology." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **27**(10): 1701 - 1713.

Wolf, W., Ozer, B. and Lv, T. (2002). "Smart Cameras as Embedded Systems." *IEEE Computer* **35**(9): 48 - 53.

Xu, J. (1993). "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations." *IEEE Transactions on Software Engineering* **19**(2): 139 - 154.

Xu, J. and Parnas, D. L. (1990). "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations." *IEEE Transactions on Software Engineering* **16**(3): 360 - 369.

Xu, J. and Parnas, D. L. (1993). "On Satisfying Timing Constraints in Hard-Real-Time Systems." *IEEE Transactions on Software Engineering* **19**(1): 70 - 84.

Xu, J. and Parnas, D. L. (2000). "Priority Scheduling versus Pre-Run-Time Scheduling." *Journal of Real-Time Systems* **18**(1): 7 - 23.

Zhao, W., Ramamritham, K. and Stankovic, J. A. (1987). "Preemptive Scheduling Under Time and Resource Constraints." *IEEE Transactions on Computers* **36**(8): 949 - 960.