Evol. Intel. (2008) 1:113–132
DOI 10.1007/s12065-008-0009-5

RESEARCH PAPER

Improving the reliability of real-time embedded systems using innate immune techniques

Nicholas Lay · Iain Bate

Received: 14 December 2007 / Revised: 12 March 2008 / Accepted: 13 March 2008 / Published online: 24 April 2008

Abstract Previous work has shown that immune-inspired techniques have good potential for solving problems associated with the development of real-time embedded systems (RTES), where for various reasons traditional real-time development techniques are not suitable. This paper examines in more detail the general applicability of the Dendritic Cell Algorithm (DCA) to the problem of task scheduling in RTES. To make this possible, an understanding of the problem characteristics is formalised, such that the results produced by the DCA can be examined in relation to the overall problem difficulty. The paper then contains a detailed understanding of how well the DCA which demonstrates that it generally performs well, however it clearly identifies properties of anomalies that are difficult to detect. These properties are as anticipated based on real-time scheduling theory.

Keywords Artificial immune systems · Dendritic cell algorithm · Real-time systems · Embedded systems · Task scheduling

1 Introduction

Traditional approaches to the development of real-time embedded systems (RTES) often feature a trade-off between using static analysis methods, which are expensive to apply and maintain, or having errors that only manifest themselves infrequently. The static analysis techniques normally involve worst-case analysis of the tasks' scheduling [1], however this does not account for anomalies that may occur in cases other than the worst-case [2, 3]. Even then, it is extremely difficult to obtain the tasks' worst-case execution times (WCET), on which the scheduling and timing analysis relies, on even for the simplest of processors [4]—to date there are only two tools that statically obtain the WCET of tasks, Bound-T [5] and AiT [6], and these only cover a small percentage of the microprocessors available to the embedded systems market.

The problems with static analysis suggest that better techniques are therefore needed to develop RTES in order to avoid in-service problems. In [7], a technique for detecting one class of potential anomalies within RTES – deadline overruns – was presented. The technique, which made use of the Dendritic Cell Algorithm (DCA) first introduced in [8], was based on treating the overrun as a danger signal. However while the results showed the strong potential of the technique, they also showed that considerable further investigation of the concepts was required. From this, the following research challenges are formulated, which form the basis of this paper:

1. Evaluate the technique on more than one example to determine its general applicability;
2. Determine which classes of anomalies are more difficult to detect,
3. Identify the quality attributes of a good solution.

The structure of the paper is as follows. Section 2 gives an overview of the real-time and embedded systems domains, and outlines some of the characteristics possessed by RTES. Section 3 gives a detailed explanation of the specific real-time problems encountered during the development of RTES, and outlines some of the static analysis

N. Lay (✉) · I. Bate
Department of Computer Science,
University of York, York, UK
e-mail: nlay@cs.york.ac.uk

I. Bate
e-mail: iain.bate@cs.york.ac.uk

techniques traditionally used to solve them. Section 4 explores the application of biologically-inspired techniques to problems in real-time and embedded systems and discusses in particular the application of immune-inspired techniques and the DCA. Section 5 gives details of our work employing the DCA to detect deadline overruns. Finally, the results of our experiments, conclusions and ideas for future research directions are presented in Sects. 6, 7.

2 Real-time and embedded systems

The work detailed in this paper is targeted towards solving anomalies in the domain of real-time and embedded systems. This section discusses the characteristics of these systems and the classes of anomalies which occur, and provides motivation for the application of biologically-inspired solutions in solving these anomalies. Further sections will then discuss the application of these techniques in a specific example with a view to solving the problems posed in Sect. 1.

2.1 Real-time systems

In the field of systems engineering, systems are often considered to possess so-called “real-time” requirements. In these systems, the correctness of an operation depends not just on the result of the computation, but also on the time at which that result is produced [1].

Real-time systems are traditionally associated with safety-critical or high-integrity applications, where incorrect behaviour cannot be tolerated as it may result in catastrophic consequences. In addition, a large number of systems exist where real-time properties are desirable, although the failure to meet these will not have the same severe consequences. A system can be considered to be “hard” real-time if it is imperative that all the real-time constraints are met. So-called “soft” real-time systems are ones where ideally real-time requirements should be met, but where the occasional failure can be tolerated. Most systems contain a mixture of “hard” and “soft” tasks.

The properties of real-time systems and the various complexities with their development has been the subject of active research over many years and consequently they are relatively well understood, with a wide range of specialist development and analysis tools available. In the safety-critical domain, software must frequently be analysed and verified to ensure that it is correct and that it meets its timing requirements [9]. However, these techniques are time-consuming and often require specialist knowledge: consequently, the development of reliable real-time software is too expensive for anything except safety or security-critical projects. In addition, many of the techniques used are based on worst-case circumstances and are therefore pessimistic: this leads to an under-utilisation of the system resources in the majority of cases.

2.2 Embedded systems

The adoption of computer systems has increased dramatically throughout the last few decades. In particular, classical computer systems are now significantly outnumbered by embedded computer systems, that is, computers which are encapsulated inside another device. Although used in a number of new devices classes (such as mobile telephones) frequently embedded computer systems are employed as a replacement for discrete control logic or custom control circuits. Particular domains in which embedded systems are frequently found include automobiles and consumer electronics devices [10]. It is widely anticipated that the market for these embedded computer systems will increase exponentially over the next 10 years [11].

There are a number of factors which are specific to the development of embedded systems, particularly those which are to be utilised in mass-market products. Most significant is the need to keep the manufacturing costs down: where a system may be included in hundreds of thousands (or even millions) of units, a small saving on the cost of each unit combines to produce a considerable overall saving [12]. It is important to consider the costs associated with developing the product, as these must be recovered through sales of the final product. In a competitive marketplace, it is crucial that a product enters the market at the correct time: if it is late, it may lose sales to rival products. It is also important that the product is reliable and contains features appropriate for its class.

With the general increase in complexity in computer systems, many embedded systems have started to include components with real-time properties, effectively creating a new class of “real-time embedded systems” (RTES). In some cases, such as in automobiles, these systems are safety-critical, and are generally engineered using real-time systems techniques. Typically, the automotive industry makes use of a standardised range of components from a few specialised suppliers, to reduce development costs. Safety-critical components are generally isolated from non-essential ones, to ensure that their operation is not affected by the failure or malfunctioning of non-essential systems [10].

Systems with definite real-time properties are being increasingly utilised in the consumer electronics domain. These systems are clearly not safety-critical, and consequently it is not economically feasible to engineer them using real-time analysis techniques; however the system must still be engineered with regard to its real-time properties, as inability to meet these may still cause problems. These may range from small issues, such a perception of unresponsiveness or "lag" having a negative effect on usability, through to more severe issues such as complete failure of the system. These conflicting design criteria result in the need for trade-offs during the design and implementation of embedded systems [13].

This is particularly true for digital broadcasting platforms, which transmit digitally-encoded audio and video signals. A digital receiver must be able to decode the incoming signal, separate it into audio and video streams, and these must then be decompressed.

2.3 Characteristics of RTES

The overall operation of a RTES can be thought of as a series of tasks, each of which is responsible for a subset of the system's total functionality. Each of these tasks has a number of properties, which can be combined to form a profile for each task. A typical RTES will contain a set of tasks with a variety of different properties.

The most significant property of any task concerns its temporal behaviour. Some tasks are periodic: they execute repeatedly with a fixed period. Others have no fixed repetition period, and so are classed as aperiodic or sporadic. If the system has multiple modes of operation, it is possible that a particular task might be periodic in one mode, but aperiodic in another.

The majority of other properties relating to a task revolve around that task's execution time. It is rare that a task will execute for the same amount of time every execution cycle: rather, the execution time will vary depending on the data to be processed or the presence/absence of external signals. Consequently, a task's execution time can be considered to fall between two values: a minimum "best-case" execution time (BCET) and a maximum "worst-case" execution time (WCET). These values may differ substantially, and it is also important to consider the average execution time of each task when performing execution time analysis.

Another important property of a real-time task is the task's deadline. This is the point in time where the task must have finished its execution in order for the system to meet its real-time constraints. Typically, any system which employs multi-tasking will incorporate a scheduler, which is responsible for allocating processor time to each task in the system. In a real-time system, the scheduler must allocate sufficient processor time to each task to allow it to complete before the task reaches its deadline.

The utilisation of a task is an important measure from a scheduling perspective. For any given task this is defined as the execution time divided by the task period, giving a proportion of the total time spent executing by that task. In a system where execution times are defined in terms of WCET and BCET, it is normal to also consider utilisation values in terms of best-case and worst-case. The utilisation of an entire system can easily be calculated as the sum of the utilisations of all tasks within that system.

2.3.1 Similarities to job-shop scheduling

At first glance, the problem of scheduling tasks in a real-time system looks similar to job-shop scheduling, to which bio-inspired techniques have been applied with good results [14, 15]. However, there are some significant differences between job-shop scheduling and task scheduling. Job-shop scheduling typically involves a scenario with multiple jobs and multiple machines, where jobs must be processed on a sequence of machines in a specific order. Task scheduling typically only requires that each job be processed on a single processor, although in more complex situations, tasks may require access to specific resources as a part of their execution which may introduce further constraints.

An important differentiation is that job-shop schedules are finite: once a job is completed, it remains completed, and eventually the point will be reached where every possible job is completed. Task scheduling, conversely, has to deal with repeating periodic tasks, and can therefore be considered to run indefinitely. The occurrence in sporadic tasks, and the fact that tasks often experience variable execution times on each execution, result in the overall scheduling scheme being largely unpredictable in advance.

2.3.2 Problems with typical analysis techniques

Traditional analysis techniques used during the development of real-time systems frequently make use of worst-case values when execution times or utilisations are required in calculations. In systems where reliability is frequently the most important consideration, a system which meets all its deadlines in a worst-case scenario should never suffer from a deadline overrun in normal operation.

There are several issues with traditional real-time analysis techniques. To fully analyse a real-time system requires significant amounts of time and specialist knowledge, and consequently is financially expensive. The cost of undertaking this analysis increases the overall engineering costs of the system, which must be earned back through sales revenue.

There are also practical difficulties in obtaining accurate information which can be used in the analysis of systems. For scheduling analysis, it is necessary to obtain accurate values for each task's WCET: this can be achieved through either analysis or measurement. Determination of WCET by analysis requires detailed knowledge of the hardware on which the task will be run, and frequently gives pessimistic results as it is difficult to analyse the effects of caches and out-of-order execution strategies on the WCET of a particular task [16].

Measurement-based techniques rely on repeatedly timing the execution of a specific task and recording the longest observed execution time. Although this is frequently simpler than detailed analysis, there is no guarantee that the WCET has actually been observed during the measurement process, requiring a safety margin to be added on top of the observed worst-case time [17].

Worst-case execution times values derived from both analysis and measurement are specific to the hardware and software analysed, and therefore are rendered irrelevant if either the software or underlying hardware are changed after the analysis has been performed. The result of this is that an independent analysis must be carried out for each different version of a system – knowledge cannot be transferred from one version of a product to the next.

The use of WCET values in the analysis of real-time systems frequently leads to under-utilisation of the available hardware [16]. This is because to ensure reliability a system must be able to cope with all tasks running to their worst case completion times 100% of the time, even though in practice it is likely that those tasks will complete with time to spare in most instances. Clearly in safety-critical systems, where reliability is considered to be more important than cost, over specification of processing can be built into the cost; however this cannot be afforded in situations where the unit cost must be kept as low as possible.

2.4 Anomalies and anomaly detection in RTES

Problems encountered during the development and operation of RTES can cause the system to suffer from a variety of anomalies. There are a number of different anomalies but their effects are similar, normally causing partial or even total non-responsiveness. In [7] three types of anomaly were discussed – deadline overruns, deadlock/livelock and bandwidth bottlenecks. This paper considers deadline overruns in further detail.

Deadline overruns occur when one of a system's constituent tasks fails to complete before its deadline. This may cause problems such as poor response times, or errors in certain operations. It is possible for an overrun in one task to induce further overruns in other parts of the system, potentially leading to a worsening cycle of overruns which may result in total system failure if left unchecked. The presence of deadline overruns in a system can therefore be viewed as evidence that problems exist with the operation of that system. Ideally, deadline overruns should therefore not occur; hence there is a need to reduce the possibility of deadline overruns as a part of the development process. It is also desirable to be able to detect the presence of deadline overruns in a running system, ideally before they actually occur.

There exist techniques to analyse the schedulability of tasks in a system at the design stage, which can be used to determine whether any overruns will occur in a given task set (e.g. [18, 19]). Unfortunately, these techniques are limited in their scope: for instance, it is straightforward to analyse a purely periodic task set, but much more difficult to analyse a system containing sporadic or aperiodic tasks. This is simply because it is not possible to determine the frequency at which a sporadic or aperiodic task is executed. Most techniques represent a sporadic task as a periodic one, with a period equal to the sporadic's minimum inter-arrival time: this is guaranteed to be safe, but is unlikely to be representative of the task's true behaviour and is likely to cause the analysis to be pessimistic. As with most analysis techniques, the results are specific to the particular task set being analysed, and are not valid if even a small change is made to either the system hardware or software. This limitation also causes a priori analysis to be impossible on systems which allow the user to make alterations or additions to the system's software.

Given the characteristics of RTES and the types of anomalies which occur within them, we can begin to identify deficiencies in many of the existing anomaly detection methods. These methods are normally classified as static or dynamic, and both classes have characteristics which can make them unsuitable for application in a RTES scenario.

Static approaches require detailed knowledge of the current system state, which is then used to deduce the presence of problems. This requires a large amount of system analysis, which often makes the techniques inflexible. Particularly, if problems are detected towards the end of the development cycle, it can be difficult to fix them, as any changes to the system require the analysis to be completely redone. Conversely, dynamic approaches rely on easily-computable metrics, such as overall utilisation. These are simpler to apply but can be difficult to generalise for more complex systems.

Our work investigates the use of an adaptive, biologically inspired approach which should allow an accuracy level close to that of static detection methods without the complex analysis procedures normally associated with them.

3 Task scheduling

As outlined in Sect 2.4, task scheduling is a significant problem in the development of RTES, as the ability of tasks to meet their deadlines is important in guaranteeing the system's real-time properties and thus ensuring its reliability.

A variety of scheduling approaches are examined in the real-time literature, each of which has a differing range of properties and characteristics. For the purposes of this work, fixed-priority scheduling is used. The characteristics of this are such that static analysis can be readily applied for small problems, however the run-time ordering is determined dynamically, so giving rise to interesting properties during the course of system operation. In addition, fixed-priority scheduling is one of the scheduling strategies most frequently used in the industrial design of real-time systems [1]. This section explains the fixed-priority scheduling model, along with the static analysis techniques traditionally used for its verification. It also examines the model for deadline overruns and explains the quality attributes for a deadline overrun detection mechanism.

3.1 Fixed priority scheduling

Fixed-priority scheduling was initially examined in [20], in the context of a simple conceptual framework where the executing task at any point in time is always the highest priority task which is runnable, such that the tasks execute in a pre-emptive manner.

A typical implementation of fixed-priority scheduling features three queues: a run queue, which contains tasks which are released but which have not yet commenced execution; a suspend queue, containing those tasks which have commenced execution but have then been suspended, for example due to pre-emption by a higher-priority task; finally the waiting queue contains all tasks which are neither released nor executing.

In [20], tasks are assigned priorities based on their periods, such that the task with the shortest period is that with the highest priority. This policy is known as rate-monotonic priority assignment, and was proven optimal for task sets where each tasks deadline is equal to its period, and where there are zero offsets—i.e. where at some point in the execution of the system, there exists a point where all tasks are released simultaneously, known as a *critical instant*.

3.2 Static analysis of fixed priority scheduling

The purpose of performing static analysis on a scheduling scheme is to determine formally, with the knowledge of the scheme's properties, whether it meets certain real-time requirements. A fundamental aspect of this is schedulability analysis, the purpose of which is to determine whether all tasks in a task set will meet their deadlines when a specific scheduling scheme is employed. For this purpose, there exist a number of schedulability tests which can be conducted. These tests fall into three categories, based on the accuracy of the test results:

- *Sufficient and necessary* The analysis always indicates correctly when a task set is schedulable. In addition, the analysis always indicates correctly when a task set is unschedulable.
- *Sufficient and not necessary* The analysis always indicates correctly when a task set is schedulable. However, there are cases when the task set is schedulable contrary to the results of the analysis.
- *Not sufficient* The analysis indicates a schedulable solution when the task set is in fact not schedulable.

Clearly, the most desirable tests are those which fall into the *sufficient and necessary* category. However, in cases where no *sufficient and necessary* analysis technique is available, or where it is considered computationally infeasible, a *sufficient and not necessary* test may be regarded as acceptable. Tests which are *not sufficient* are undesirable as they give incorrect assurances regarding the schedulability of task sets.

The schedulability of a task set using fixed-priority scheduling can be verified by the application of a simple utilisation-based test [20] given in Eq. (1):

$$U_{\max} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1}$$

where: n is the number of tasks; i is a task in the set of tasks; C_i is the worst-case execution of task i; T_i is the period of task i; $U_{\max}$ is the maximum processor utilisation.

A system is always schedulable provided that the combined utilisation of all tasks in that system is less than the value determined by Eq. (1), according to the number of tasks in the system. Consequently, if n is 1 and $U_{\max}$ is less than or equal to 100%, then the task set is schedulable. As n tends to infinity, then the maximum utilisation guaranteed to be schedulable, $U_{\max}$, tends to 69.31%.

The results provided by this test, however, are pessimistic. An example of this is a task set with two tasks whose period and deadline are both equal to T, and with WCET of $T/2$. The test implies that the combined utilisation of these tasks (100%) is unschedulable since, according to Eq. (1), $U_{\max} = 82.82\%$ where $n = 2$. However, it can be demonstrated that this task set is schedulable: as the tasks are simultaneously released, one task is dispatched immediately and executes for time $T/2$. On its completion the second is dispatched and executes for a further time $T/2$, finishing at time T. Both tasks complete execution within their deadlines, and so the task set is schedulable. Consequently, this test can be considered to be *sufficient and not necessary*.

This pessimism was observed in [21], where it was also noted that rate-monotonic priority assignment is sub-optimal for task sets where task deadlines are not equal to task periods, or where tasks have non-zero offsets (i.e. a critical instant does not exist). The pessimism in this test indicates that a better schedulability analysis method is needed.

An alternative schedulability test was derived in [22], and is valid for task sets where there is a critical instant. The analysis assumes that all tasks have a fixed unique priority, and that the task deadline is not greater than the period.

For each individual task, the response time can be computed using Eq. (2) [22]:

$$R_i = C_i + I_i + B_i \tag{2}$$

where: i is a task in the set of tasks for a given node; R_i is the worst-case response time of task i; C_i is the WCET of task i; B_i is the blocking time of task i; I_i is the interference of task i.

In Eq. (2), the blocking time B_i is the longest time that a runnable task can be prevented from executing by a lower-priority task. This is dependent on the computational model being used. For an ideal pre-emptive system, this blocking time is zero. A non pre-emptive system, or a pre-emptive system where tasks access shared resources, will involve some degree of task blocking which must be accounted for.

When executing in a priority-based system, each task suffers interference from other tasks which are higher in priority than itself. For any task, the maximum interference is suffered at a critical instant, where all tasks higher in priority than itself are released simultaneously. The interference over a period of interest, (in this case the response time of the task being analysed), is derived from the higher priority tasks' periods and WCETs, as shown in Eq. (3).

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3}$$

where: $hp(i)$ is the set of higher priority tasks than task i; I_i is the interference of task I; R_i is the worst-case response time of task i; T_j is the period of task j; C_j is the WCET of task j.

Equations (2) and (3) can be combined to form a recurrence relation as shown in Eq. (4), to give the worst-case response time for each task in the system:

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_i \tag{4}$$

with $R_i^0 = C_i$
which terminates when $R_i^{n+1} = R_i^n$, or $R_i^{n+1} > D_i$
where D_i is the deadline of task i.

If this recurrence successfully indicates that, for each task in the system, $R_i < D_i$, then the task set is schedulable. This test has been found to be *sufficient and necessary* providing the task set meets the assumptions outlined above.

3.3 Deadline overrun model

In general, work on scheduling and timing analysis only considers systems which meet their timing requirements, or providing tests to ensure that requirements are met. Although some papers consider soft real-time requirements, such as [23] this is generally only in the context of providing best effort scheduling. There appears to be little work in the RTS literature which attempts to formalise an understanding of that happens when tasks fail to meet their requirements. This work aims to formalise the effect of deadline overruns.

Figure 1 shows a typical deadline overrun, with four variables. The task's deadline is denoted by D. C indicates the time at which the task completes: in the case of a deadline overrun, the value of C will be greater than D. The value R represents the time at which the task is released, i.e. the point at which it is transferred into the run queue from the waiting queue. Finally, the task dispatch time, shown as Dis, is the time at which the task begins execution following its release. Although in Fig. 1, R and Dis are shown as occurring before and after the deadline D respectively, these points can in fact occur at any point in time providing that $R \leq Dis$, and therefore either can be less than, equal to or greater than D.

One key property of real-world systems is that execution times of tasks is not constant, caused by differences in task behaviour between cycles, and often also by external factors such as user interaction. As a result, the times at which tasks are dispatched and completed relative to their release and deadlines will vary between execution instances. This has a number of implications for this work.

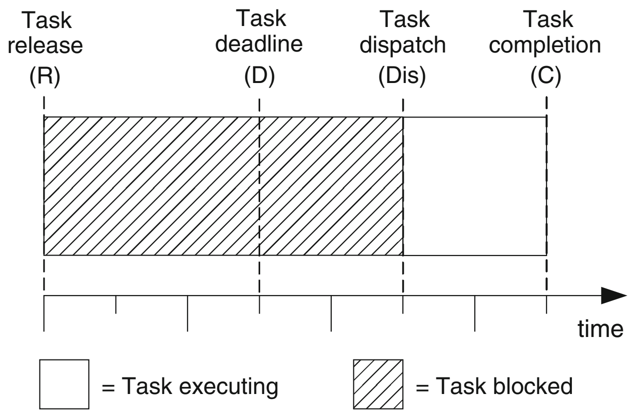


Fig. 1 Typical deadline overrun

The likelihood that a task will overrun varies between the task's release and its completion. It is quite possible that, although the possibility of an overrun is identified at some point during a task's execution cycle, that the task will not actually overrun. In some cases, this is due to errors in the overrun prediction model, or with the information supplied to it by the system. Generally, however, the likelihood of an overrun is reduced as other tasks in the system complete before their worst-case time. Conversely, the release of higher priority tasks into the system during a task's execution cycle may result in an overrun being predicted where previously no risk had been identified. Frequently, these two behaviours will combine, such that the likelihood of a task overrunning is low on its release, increases as other higher priority tasks are released into the system, and decreases again as those tasks complete ahead of their worst-case times.

In addition, in non pre-emptive systems it is possible that overruns can occur when tasks execute for durations less than their WCET. An example of this is shown in Fig. 2. In the first scenario on the left, task A completes after 4 clock cycles, permitting the execution of task C. When task B is released in cycle 5 it is then blocked from executing until task C has completed at cycle 7. If the deadline of task B is 7, the task misses its deadline.

However, in the second scenario, task A executes for 5 cycles. By this point, task B has already been released, and so it is the next to execute after task A completes. The execution of task B is then completed before its deadline at cycle 7.

This complexity makes the analysis and detection of overruns particularly difficult. In a small system, it is possible to exhaustively search through all combinations of task properties to establish all the possible conditions which lead to overruns. However, this quickly becomes infeasible as the complexity of the system increases.

4 RTES problems and biologically-inspired solutions

The use of RTES in consumer electronics devices leads to a significant conflict during the design process. There is a need for the device to meet all its real-time requirements to achieve a high level of reliability, but at the same time, market forces dictate that the development must be both fast and cheap, to ensure the product reaches the market on time and to maximise profitability. As well as being expensive and time-consuming, current real-time development techniques are inflexible, and do not readily support changes during the development process. This makes them unsuitable for use in the majority of CE development.

As well as the traditional real-time development techniques, there are a variety of different methods which can be employed during the design and production of devices, including methods such as model-driven development. Unfortunately, the use of such techniques suffers from problems similar to those encountered with static analysis approaches: problems frequently arise at the implementation stage due to mismatches between the models and the

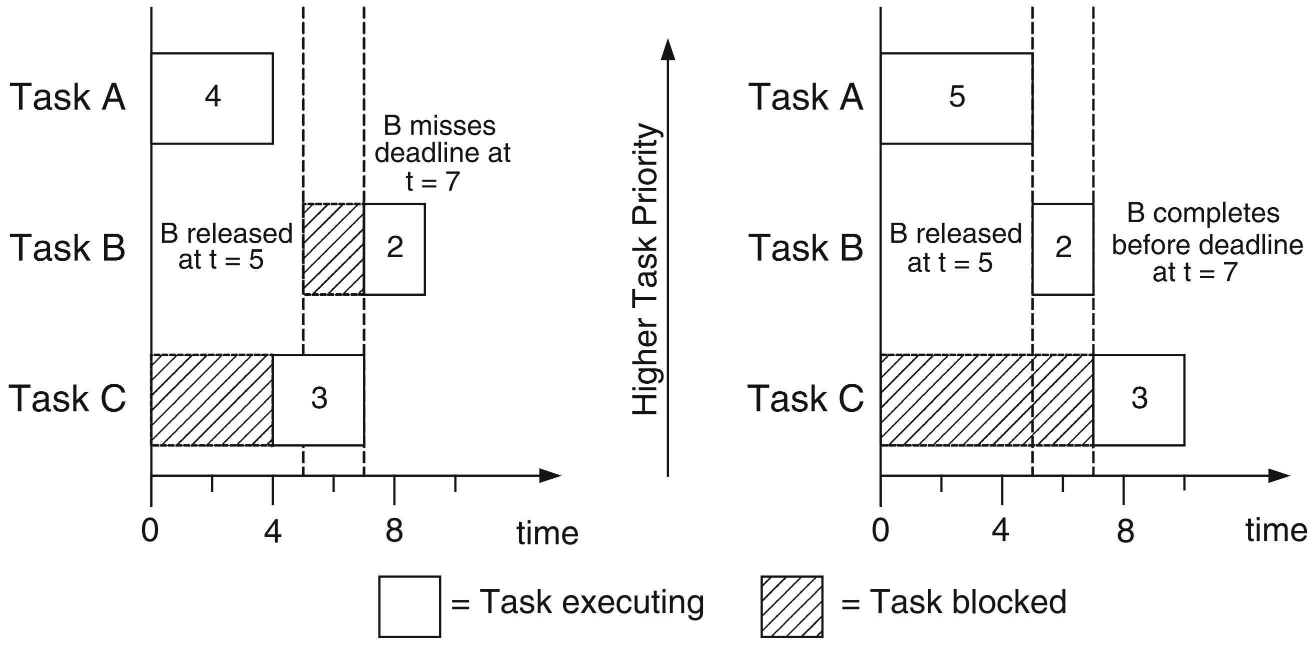


Fig. 2 Overrun caused by task executing for less than worst case time

physical implementations, and solutions derived from iterative development techniques are specific to one particular system configuration, and cannot be transferred to another similar system without restarting the entire development process.

What is required is a mechanism for fault tolerance which can adapt to accommodate differences in the system and its environment. For this, we look to biology for inspiration: in the natural world, biological systems exist which are clearly fault-tolerant and able to adapt successfully to their surroundings. In particular, we draw inspiration from the natural immune system, the purpose of which is to protect its host organism from attack.

The development of biologically inspired computer systems is one of many areas of computer science which is considered to fall outside the realm of "classical" computational methods, despite being a source of inspiration for many of the founders of the computer science discipline [24]. Despite this, there are many models present in modern day computer science which have been successfully derived from biological counterparts. In particular, there is a considerable body of research which has focussed on the development of systems which derive inspiration from immunology, known as "artificial immune systems" (AIS) [25]. This work looks to exploit the anomaly-detection properties of AIS.

4.1 RTES considerations

Our work focuses on the detection of task scheduling anomalies in a device with real-time and embedded characteristics, allowing increases in reliability while at the same time keeping development time and costs to a minimum. AIS has previously been applied to other types of scheduling with positive results [14, 26], and our initial investigations indicate that AIS techniques show good potential in the case of task scheduling [7]. As task scheduling has specific complexities which do not apply to other types of scheduling, and as resource issues are particularly important, we anticipate significant research challenges.

Many of the AIS techniques which have been developed are based on the principles of the adaptive immune system [25]. The adaptive immune system is able to alter its behaviour in response to previously unseen antigens to provide defence against them. Consequently, systems using adaptive techniques are able to "evolve" to provide solutions in previously unknown situations, and to maintain a memory of those situations so that they can be dealt with quickly and effectively should they arise again. This allows a system to change as its environment and circumstances change, resulting in a more effective solution.

Frequently, however, adaptive techniques have significant processing and memory requirements, to support the adaptation and learning properties of the system. In systems where hardware is severely limited, such as embedded systems, it is therefore impractical to make effective use of adaptive AIS techniques.

4.2 Innate immunity and the Danger Model

Current immunological research suggests that the function of the innate immune system is more important to the overall immune behaviour of an organism than previously thought [27]. As a result of this a new class of AIS has been established based around techniques derived from innate immunity [28].

The innate immune system makes significant use of the emission and detection of patterns and chemical signals. These may be specific protein patterns only associated with invading pathogens, or signals generated by the tissues and cells of the body itself as a response to events in the environment or inside individual cells. The presence of these signals induces a response from other immune system components, including those of the adaptive immune system [27].

There have already been a number of applications of innate immune-inspired techniques in the AIS literature, mainly aimed at anomaly detection, such as [29] and [30]. These techniques are unable to adapt to changes in environment or respond to unknown issues, but generally require fewer resources than a fully adaptive system and offer effective solutions to problems that they are designed to solve.

Although innate immune-inspired systems operate in a different manner from "classic" adaptive AIS, they are generally based around the same concept of self/non-self discrimination which has been the basis of immunological theory for nearly 40 years [31]. However, the process of distinguishing "self" from "non-self" has been problematic in artificial systems, particularly when the negative selection model has been applied [32]. In addition, there remain a number of immunological issues which the generally accepted theory is unable to explain adequately. Matzinger's Danger Model [33] suggests a set of fundamentally different principles around which the immune system is based. Rather than the immune system being able to distinguish between "self" and "non-self", instead the Danger Model suggests that the immune system in fact detects signals produced when cells in an organism die unexpectedly.

The Danger Model is based around the idea of signal transmission between cells, the fundamental concept being that cells which die unexpectedly (necrosis) send out signals which are distinct from those sent out by cells which die naturally (apoptosis). These "danger signals" are detected by components of the immune system, which is then mobilised to fight the infection.

Since it was first proposed, the Danger Model has been controversial amongt immunologists and is not widely regarded as being a correct model of how the immune system functions (for example [34]). However regardless of their plausibility or biological correctness, the concepts, structures and methods around which the model is based provide a useful basis from which in silico AIS can be derived.

4.3 The Dendritic Cell Algorithm

The Dendritic Cell Algorithm (DCA) is an innate immune concept derived from the Danger Model. The broad function of the DCA is to provide an indication of danger levels associated with different parts of the system. This is accomplished by the collection and aggregation of signals and antigen derived during the operation of the system. The idea of basing an immune-inspired system on dendritic cells (DCs) was first outlined in [35], and further clarified in [8].

The DCA is based on a series of in vitro observations of the behaviour of individual living DCs. When a DC is created, it starts in an "immature" state. The function of immature DCs is to collect a range of chemical signals, produced when cells belonging to the host organism undergo apoptosis or necrosis, or by other components of the immune system. At the same time, DCs collect samples of potential antigen they encounter.

A DC which experiences high levels of chemical signals associated with "danger", caused for example by cells dying through necrosis, or by the DC encountering high concentrations of pathogenic associated molecular patterns (PAMPs – chemical signals which are directly linked to the presence of pathogenic agents), undergoes transformation into a "mature" state once the number of danger signals encountered within a given time frame exceeds a threshold: it then travels to a lymph node where it presents its antigen. Depending on the response of other DCs, an immune response can then be initiated against that antigen. If a DC detects low levels of danger signals, then it enters a different "semi-mature" state: these semi-mature DCs travel to a lymph node and present antigen in the same way as mature DCs, however their semi-mature status does not cause the initiation of an immune response, instead leading to tolerance of the antigen.

As well as danger and PAMP signals, the DCA allows for other categories of signals, which affect how the DCs mature. In particular, the notion of a "safe" signal is supported: the presence of such a signal often serving to reduce the effect of danger signals within the DC. The notion of inflammation can also be supported, by the generation of specific inflammatory signals when a DC becomes mature; these inflammatory signals are then detected by other DCs in the system.

The method by which DCs link danger to specific antigens relies heavily on guilt by association – that is, an antigen observed in the presence of danger signals is assumed to be the cause of that danger. Although in isolated cases this may lead to individual DCs falsely presenting benign antigens, the combined effect of many DCs presenting the same antigen can be taken to indicate that particular antigen is indeed dangerous and requires the initiation of an immune response against it. The likelihood of an immune response being initiated against an antigen increases as the ratio of mature DCs presenting that antigen increases, compared to the number of times the antigen is presented in total. In [8] this value is referred to as the mean context antigen value or MCAV; the closer this value is to 1, the higher the danger level associated with the particular antigen is.

This method by which DCs function in vivo can be transferred easily to in silico systems, providing them with a mechanism to detect problems. Based on the idea of a DC detecting chemical signals, a virtual DC can be created to detect virtual signals which are derived from specific attributes of the system which they are monitoring. The combination of these input signals causes the virtual DC to mature in the same way as its biological counterpart. By using a population of DCs monitoring different components of the system, the output from a number of DCs can be combined to deduce information about the overall state of the system.

The operation of the DCA is described by pseudocode in Fig. 3, simplified from that in [8]. Each DC can be represented as a data structure storing observed antigen, input signal levels, output signal levels and a migration threshold. This is shown in Fig. 4.

The DCA has been employed effectively to solve anomaly detection problems in a number of problem areas, including intrusion detection [8] and in sensor networks [36]. Its particular advantage is that, due to its origins in innate immunity, it is generally a lightweight solution with little on-line adaptation involved. This requires fewer resources than AIS techniques which are based around adaptive immune principles, and is therefore more likely to be usable in a resource-constrained system. The DCA is therefore a good choice for the detection of anomalies in RTES, where resource availability is a significant concern yet there is a need for reliable anomaly detection strategies.

4.3.1 DC parameters

The operation of the DCA relies on each DC in the population combining its input signals to produce an output. Since the DCA is able to induce an immune response correctly, it is necessary to assign appropriate levels of danger to each of the input signals collected. In vivo, DCs

```
while dc cycle count < max dc cycle count loop
  update antigen and signal levels from environment;

  for all DCs in population loop
    sample associated antigen;
    sample associated signals;
    compute cycle output signal;
    compute cumulative output signal;

    if cumulative output signal > migration threshold then
      DC becomes mature;
      remove DC from population;
      migrate DC to lymph node;
    end if;

  end loop;

  dc cycle count = dc cycle count + 1;

end loop;

all remaining immature DCs become semi-mature;
analyse DC antigen and output signal levels;
```

Fig. 3 Pseudocode for DCA (adapted from [8])

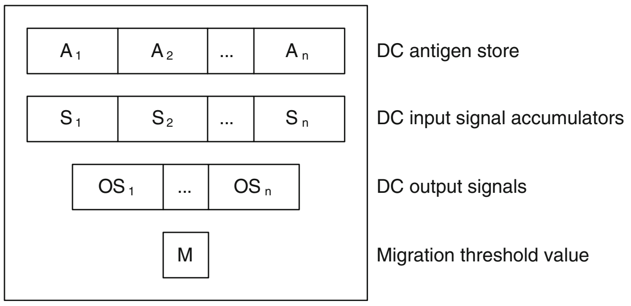


Fig. 4 DC data structure (adapted from [8])

treat some signals as being more dangerous than others, for example, the presence of PAMPs is regarded as more dangerous than the presence of signals generated by necrosis. In silico, this behaviour is emulated by applying weighting factors to the DC input signals, which are then used by the DC when it is computing its output values (Fig. 5). The weightings applied to each input signal during this combination process play a vital part in determining the output value of that DC, as do the thresholds defining the output values at which the DC reaches maturation.

Current implementations of the DCA have made use of fixed weightings specified by the designers. Although this allows the artificial injection of knowledge from in vivo observations (for example, the knowledge that DCs treat PAMP signals as being more dangerous than those caused by necrosis), it does cause the DC to be pre-biased to the concept of what exactly is dangerous and what is not. There is also no guarantee that the designer's weightings

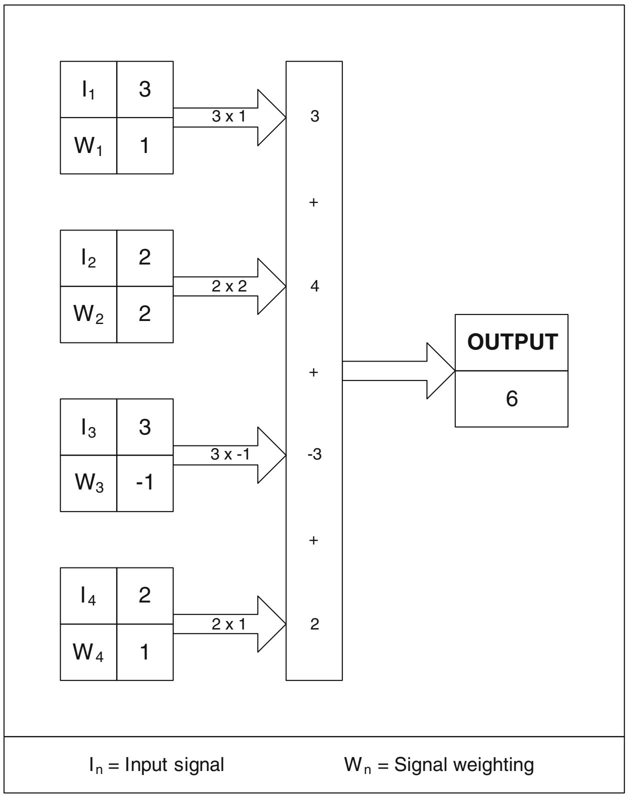


Fig. 5 Calculation of DC output signal using weightings

are optimal for the problem being solved—although they may work acceptably, a better solution may exist. The maturation thresholds are also a fixed range of values; this effectively means that the sensitivity of the DCs is set by the designer and cannot be changed as the system runs.

Our work therefore examines the potential for the DCA parameters to be altered as the system runs. Allowing the weightings for each individual DC to be variable, and changed according to a specific search strategy, such as simulated annealing or an evolutionary algorithm, should allow each DC to be tuned with a specific set of parameters for the problem being solved. Application of this process across a population of DCs should therefore allow for a highly effective (though not necessarily optimal) set of DCs to be found.

This tuning process can be conducted as a part of the system's development process, producing a set of effective DCs which can then be incorporated into the final system. During system development, the evolutionary process can be included as a natural part of a DC's lifecycle; either when they signal danger (i.e. they reach maturity), or when they reach the end of their pre-determined lifecycle (i.e. they reach semi-maturity). The potential search space for DC parameters is large, and as each potential set of parameters requires complex evaluation, the implementation of actual tuning techniques is left as future work.

Any immune system incorporated in the final system is still an innate immune system, however its parameters are derived by adaptive methods to ensure that it is as effective as possible. This can be considered analogous to the continuing evolution of the human innate immune system through mutation and natural variation over the course of many generations.

Table 1 Task properties of scenario system

Task ID	WCET	BCET	Deadline	Period
1	5	4	25	25
2	20	10	50	50
3	30	20	100	100
4	40	25	775	775

5 Task scheduling with AIS

Our work applies immune-inspired techniques, specifically the DCA, to situations which occur in RTES. As an example of a typical RTES problem, we will examine the application of the DCA to task scheduling.

Our implementation incorporates a population of artificial DCs into a system, so allowing that system to detect problems with its scheduling strategy as they arise. Initially, our work is only concerned with identifying scheduling problems, rather than with finding their solutions. There are a variety of methods which can be used to solve scheduling problems, but most of these are reliant on information such as those provided by our solution.

As a part of this work we will enhance the DCA to make it more effective in a limited-resource environment, and establish the minimum population size which is effective at diagnosing anomalies in these situations.

In [7], the DCA was applied to task scheduling with a small set of purely periodic tasks, the properties of which are shown in Table 1. These tasks were engineered such that in most instances they complete normally, but that the relatively infrequent release of a task with a long execution time (task 4) can cause other tasks in the system to overrun. Even when this task is released, the actual occurrence of an overrun is determined by a number of additional factors, such as the execution times and release rates of other tasks; this results in this apparently simple problem being particularly difficult to analyse *a priori*.

In this paper, we aim to evaluate the general applicability of the DCA to a variety of different task sets.

5.1 Measures

As outlined in Sect. 2.3, there are a huge variety of different measures from which the status of a particular task can be derived. For the DCA to be effective it is important to consider which of these measures should be used, and how they will be mapped onto the input signals for each individual DC. Some of these measurements relate to the system as a whole, while others are linked only to individual tasks.

System-wide measurements include the total utilisation of all currently-running tasks in the system, giving an indication of the system's workload. Although it is possible for systems to function correctly in overload situations, it is inevitable that a system with a worst-case utilisation level greater than 100% will eventually experience problems. Utilisation is a useful measure of the overall health of a system, but it can be difficult to attribute system failures to any one task.

When considering execution properties of individual tasks, a useful concept is that of slack time. This is the interval between the completion of a task and its deadline. Due to it not always being possible to determine the execution time of a task *a priori* (as outlined in Sect. 3.3), it is also impossible to determine the actual slack time for any given execution of a task until that task is completed. However, by making use of worst-case properties, it is possible to calculate the minimum possible slack time at any point in a task's execution. If this worst-case slack time is less than (or equal to) zero, then the task will complete on time and therefore not miss its deadline. If at any point during the execution cycle the worst-case slack time is negative, there is the possibility that the task may overrun, although as the calculation is by necessity based on worst-case values, it is still possible for the task to complete on time.

An alternative, and arguably more intuitive, measure which is analogous to slack time is a process's overrun time, which is simply the inverse of the slack. A process which has a positive overrun time is one which does not complete before its deadline; a negative overrun time indicates successful completion before the deadline.

The DCA supports multiple categories of input signal which can be considered analogous to the different chemical signals detected by DCs in vivo. Therefore, different measurements are derived from the various task properties outlined above, and these are employed as our input signals (Table 2). Signals are included for the signal categories PAMP, danger and safe: the use of inflammatory signals is not considered in this paper.

5.2 Antigen

The DCA provides a danger level associated with a particular antigen. In this work, the tasks present in the system are taken as the antigen, against which the danger level can

Table 2 Derivation of DC input signals

Event	Signal category	Derivation
Actual overrun	PAMP	Task completion time > task deadline ($C_i > D_i$)
Potential overrun	Danger	At any point from task release to completion, worst-case response time is greater than time to deadline ($R_i > D_i$)
No projected overrun	Safe	At all points from task release to completion, worst case response time is less than time to deadline ($R_i < D_i$)

be measured. A task which experiences a high level of overruns should be flagged as having a high level of danger.

Our implementation of the DCA associates each DC with a subset of the tasks present in the system, allowing the population as a whole to monitor a variety of different combinations of tasks. By combining the output of a number of different DCs, it is possible to build up a picture of the operation of the system as whole and therefore to determine which parts of the system are experiencing problems.

5.3 Learning system behaviour

As a part of investigating general applicability of the DCA to a variety of problems, it is important that the DCA is not provided with any prior knowledge of the system's operation. Therefore, our solution includes support for the DCA to learn the execution properties of each individual task as the system's execution progresses. The parameters learned are used by the DCA in deriving the input signals.

This allows the DCA to be incorporated into a system with no knowledge of how it operates, and for it to be able to derive this knowledge as the system runs. This goes some way towards addressing one of the major problems with static analysis methods, which is the need for large amounts of information about the system properties to be gathered before the analysis can be completed.

Incorporating learning behaviour also allows the DCA solution to go some way towards supporting systems with dynamic run-time properties, for example where additional tasks are added to the system, or where the task properties change as the system runs. Given the increasing complexity of many classes of RTES, it is important for any devised technique to be able to support such systems.

5.4 DC parameters

As discussed in Sect. 4.3.1, a weakness with the DCA is its incorporation of parameters solely derived from in vivo observations of DCs. As the use of these parameters makes potentially unrealistic assumptions about the problem domain and its operation, one of the objectives of this work is to investigate whether there is a set of DC parameters which is universally suitable for solving a large number of problems, and if there is not, the possibility of deriving these parameters such that they are specific to the problem being solved.

Our initial work [7] incorporated rudimentary evolution of some of the DC parameters, driven by random mutation of the DC's weighting values and controlled by a fitness function. This demonstrated that the signals output by the DCA vary radically depending on the values assigned to the weightings within each individual DC. Consequently, it was established that there was good potential for improving the operation of the DCA by allowing the parameters of individual DCs to be altered depending on the correctness of each DC's reports.

Although in [7] mutation is concentrated solely on the DC's internal weighting values, there are a number of parameters in addition to weightings which could potentially be altered by evolution, including DC threshold values, and the bindings between DCs and the parts of the system they monitor.

Initially, to determine the general applicability of the DCA to problems in its current state, we employ the DCA as it has been applied to intrusion detection in [8] and [37], with a fixed set of parameters, to a wide variety of task sets. This allows us to determine the effectiveness of a specific parameter set over a number of similar problems, each with slightly different characteristics. The results obtained from this will allow us to select potentially interesting task sets with which further experimentation and analysis can be carried out using different sets of DCA parameters.

This work makes use of the weighting values given in Table 3. These values were chosen because, initially, there is no need for the DCA to react differently to danger or PAMP signals. It could be argued that, in this situation, the detection of actual overruns is less important than the prediction of potential overruns, as by the time an actual overrun has been detected it is too late for the system to do anything about it.

To maximise the likelihood of the DCA responding to a danger or safe signal, the migration threshold is set to a value of 5. This has the effect that any DC which detects either danger or PAMP, and no further safe signals, will

Table 3 DC weighting values used

Signal	Weighting
PAMP	6
Danger	6
Safe	−6

immediately mature and be analysed. The DC lifecycle is set at 40 DC updates; assuming one update per simulation clock cycle, a DC which experiences no danger or PAMP signals will become semi-mature after 40 clock cycles. Given the recurring nature of the task-scheduling problem, each DC is regenerated into the population once it has reached maturity or semi-maturity. This ensures a constant refreshing of the DCs population, while maintaining a relatively small population size.

The problems evaluated here make use of a population size of 20 DCs. Given random assignment of tasks to DCs, this number gives good coverage of the various permutations of tasks. The assignment of tasks (antigen) to DCs is random at the start of the simulation and does not change throughout the run.

6 Evaluation of AIS-based task scheduling

In this section we evaluate the DCA-based solution for detecting anomalies in task scheduling. This evaluation considers the accuracy of the DCA system for detecting anomalies and the responsiveness of the system. These measures are then considered with respect to the overall utilisation of the system, and also the average slack time of each process.

6.1 Task simulation environment

The results on which the analysis is based are obtained using a task simulation environment. This simulates the execution of a simple task set, using fixed priority non-preemptive scheduling.

The simulator considers task execution time in terms of "cycles"—one cycle being the smallest unit of execution time available. The state of the scheduler is monitored each and every clock cycle, both by static analysis, which possesses knowledge of the task parameters, and by a DCA implementation, which initially has no knowledge of task parameters and must learn these as the simulation is run. Where the static analysis detects the presence of a potential or actual overrun in the system, this is recorded and compared with the output produced by the DCA implementation to give measures of accuracy and responsiveness as defined below.

The simulator supports the execution of different task sets, and is run in a framework which allows the generation of random task sets with rate-monotonic priority assignment. Each task set is simulated 10 times, each run for a fixed number of cycles. The output from the 10 runs is then combined to produce an aggregate output for each task set. The framework repeats the simulation with a number of different task sets.

To produce task sets with a reasonable likelihood of errors, but in which normal execution occurs in the majority of cases, the random task sets are constrained according to the worst-case utilisation of the entire task set.

For this paper, each randomly generated task set comprises four tasks and is engineered to have a total worst-case utilisation between 0.95 and 1.4.

6.2 Measurements: accuracy

The accuracy of the solution is determined by comparison with traditional static analysis techniques for a number of test cases. Both the DCA-based solution and a static analysis solution are run simultaneously, allowing them to monitor a task simulator. The static analysis technique is assumed to be 100% accurate at detecting and predicting overruns. The accuracy of the DCA-based solution is measured against this.

Figure 6 shows the execution of a task, from the start of its execution (at the point labelled Dis) to its completion (at point C). It is assumed that the static analysis solution will detect a potential overrun at the earliest possible opportunity (shown as point SA). Therefore it is assumed that this point is also the earliest that a DCA-based solution can correctly identify the presence of a potential overrun in the system. Any reports of danger produced by DCs in the time window between Dis and SA (labelled 1) are assumed to be incorrect, i.e. the DCA reports a false positive. It is possible, depending on the static analysis technique used, that the point SA may actually occur before Dis.

For the DCA to correctly detect an overrun, it must report the presence of danger in the system at some point in the time window between SA and C (labelled 4). In the case of a task which successfully completes before its deadline, a report of danger in this window indicates successful prediction of a potential overrun. Where the task deadline occurs before the completion of its execution, any report of danger in the time period between SA and D (labelled 2) indicates successful prediction of an actual overrun, while a report between D and C (labelled 3)

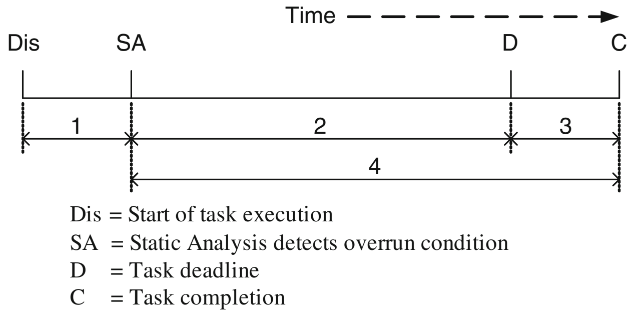

Time
Dis
SA
D
C
1
2
3
4
Dis = Start of task execution
SA = Static Analysis detects overrun condition
D = Task deadline
C = Task completion

Fig. 6 Determining accuracy of DCA-based solution

indicates successful *detection*. For the purposes of this analysis, we do not discriminate between prediction and detection of actual overruns and both are considered equally accurate, however it is noted that in a real-world system prediction is likely to be more desirable than detection.

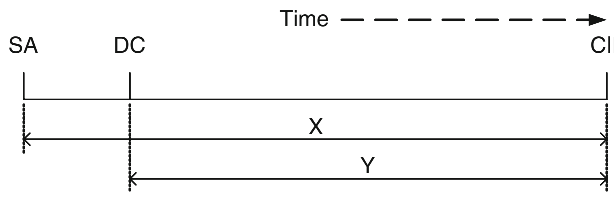
We record the accuracy of the DCA based system for all instances of task execution where static analysis reports the presence of a potential or actual overrun in the system (only one report is made per execution instance). From this, we calculate the accuracy for the DCA-based solution, as the number of correctly detected overruns divided by the total number of overruns present in the system. For the purposes of the analysis in this paper, we consider the accuracy of those execution instances where actual overruns occur separately from those where potential overruns are raised.

6.3 Measurements: responsiveness

The responsiveness of the solution, as with accuracy, is determined by comparison with static analysis of the system.

We consider the responsiveness of the DCA-based solution for all task execution instances where the static analysis indicates the presence of a potential or actual overrun in the system. The method by which responsiveness is calculated is shown in Fig. 7. For all the tasks where an overrun is detected, we record the cycle time at which the static analysis first highlights the presence of that overrun (point SA in Fig. 7). We record the time at which the task completes, or its deadline if it reaches this before its execution has completed (point C|D). Finally, we record the first report of danger from the DCA-based solution (point DC).

Using these values, the responsiveness of the DCA-based solution can be calculated using Eq. (5). This gives a value for the responsiveness of the DCA-based solution, as a proportion of the responsiveness of the static analysis. In Fig. 7, this is shown as the duration labelled Y divided by the duration labelled X



SA = Static Analysis detects overrun condition
DC = DCA solution detects overrun condition
C|D = Task completion or task deadline (whichever reached first)

Fig. 7 Calculating responsiveness of DCA-based solution

$$\mathrm{Rsp}_{\mathrm{DCA}} = \frac{\mathrm{C|D} - \mathrm{DC}}{\mathrm{C|D} - \mathrm{SA}}. \tag{5}$$

As a consequence of this calculation of responsiveness, any execution instance where the DCA-based solution responds at the same time as the static analysis report has a responsiveness of 1. For any instance where the DCA solution responds on or after the task deadline, or where there is no DCA response at all (i.e. the accuracy for that instance is 0) the responsiveness value can be assumed to be 0. Instances where the responsiveness tends towards 0 are less responsive, while those where the responsiveness tends towards 1 are more so.

6.4 With respect to system utilisation

One of the objectives of this work as outlined in the introduction is to derive some measure of “problem difficulty” for a given task set. Initially, we consider the overall utilisation of a task set as an indication of this. The utilisation of each individual task, which can be combined to give the overall system utilisation, is a classic measure of the complexity of a task set used by many of the traditional static-analysis techniques [1], for example the schedulability test from [20] shown in Eq. (1) (Sect. 3.1). It could potentially be a good base measure of “problem difficulty” in the context of this work.

The task sets used to test the DCA-based solution are generated at random, each task having known BCET and WCET, period and deadline (so allowing static analysis of the task set to take place). From these values, we calculate the best-case and worst-case system utilisation values for each task, and combine these to obtain utilisation values for the entire task set. These values are then used as a measure of difficulty against which we evaluate the accuracy and responsiveness of the DCA-based solution.

6.4.1 Accuracy

The accuracy of the DCA-based solution against total system utilisation is shown in Fig. 8 (worst-case utilisation) and Fig. 9 (best-case utilisation). Each point represents the overall utilisation and average DC accuracy for an individual randomly generated task set, comprising 4 tasks with rate monotonic priority ordering.

It can be seen from both graphs that the detection of actual overruns (shown as triangles), regardless of system utilisation, is close to 100%. This demonstrates that the DCA has good ability to detect overruns in a system when they occur.

However, the ability of the DCA to detect potential overruns (shown as diamonds) is less clear. For some task sets, the detection rate is close to 100%, however for others

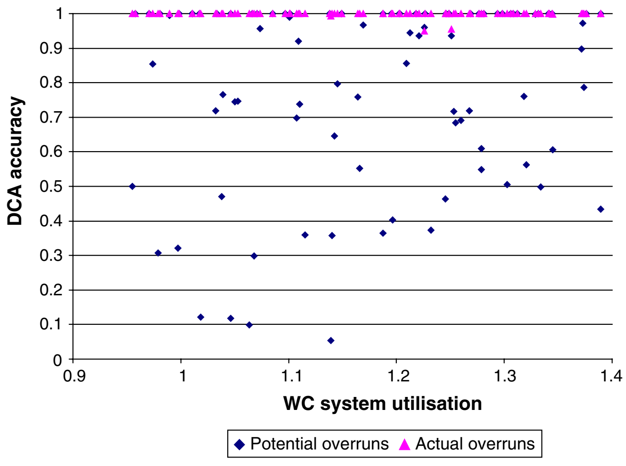


Fig. 8 DC accuracy against worst-case system utilisation

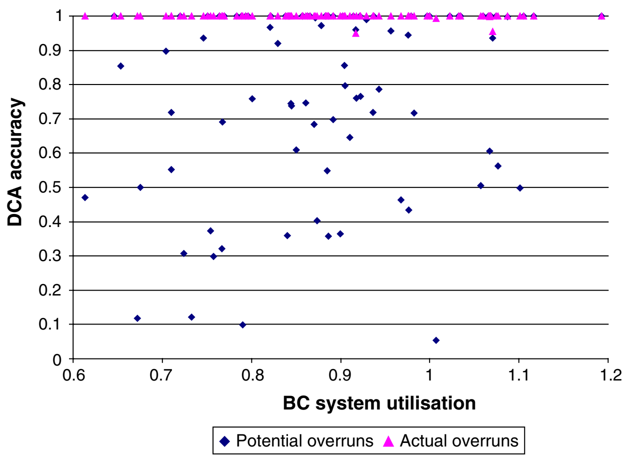


Fig. 9 DC accuracy against best-case system utilisation

the detection rate is considerably lower. There appears to be a slight correlation between utilisation and accuracy in both best- and worst-case graphs—higher utilisations in general showing slightly higher accuracy in the detection of potential overruns than lower utilisations. This is due to the increased probability and magnitude of overruns in higher utilisation task sets, making detection more straightforward. However, the data points are too widely scattered to make any significant correlation between utilisation and accuracy.

6.4.2 Responsiveness

The responsiveness of the DCA-based system with respect to worst-case system utilisation is shown in Fig. 10.

As with the accuracy graph, there is a slight correlation between system utilisation and DC responsiveness; this is for the same reason as with the accuracy graph. Again, however, the data points are too widely scattered to draw

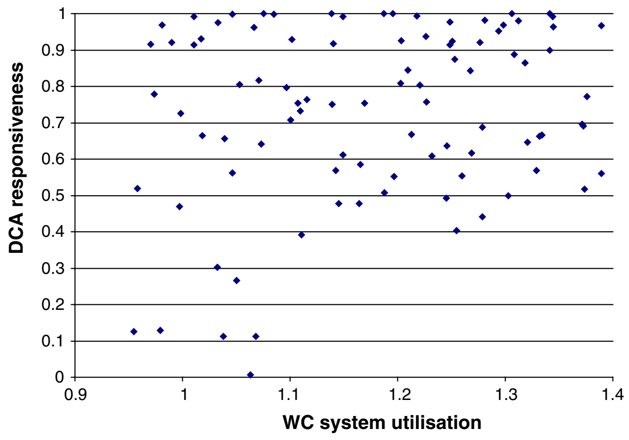


Fig. 10 DC responsiveness against worst-case system utilisation

any definite conclusions regarding the correlation between the total system utilisation and the responsiveness of the DCA. The results were similar when comparing best-case utilisation with responsiveness.

6.5 With respect to overrun time

The results shown above suggest that the overall system utilisation is not an adequately sophisticated measure of system complexity and thus of "problem difficulty".

As outlined in Sect. 3.2, an alternative to using utilisation-based measures is the use of response times. These are measured from the time a task is released, however a task's overrun time can be considered as an analogous measure, taken instead from the task's deadline. For any given task execution instance, its overrun time is an effective measure of whether it has overrun, and if so by how much. It therefore appears that the smaller the overrun time in any given instance, the more difficult it is to detect that overrun.

A measure of overrun time is therefore recorded for each instance of a task's execution where a potential or actual overrun is reported. In the case of actual overruns, the recorded overrun value is positive; for potential overruns, where the task completes before its deadline, the value is negative. Zero overrun of course indicates that the time at which the task completed was exactly the same as its deadline.

The overrun value can then be used for comparison with accuracy or responsiveness, either on a per execution instance basis for a single task set, or aggregated to allow comparisons to be made between multiple task sets.

There are at least three different measures of overrun time which could be used. Firstly, there is the actual overrun time, recorded when the task in question actually completes its execution. Given that tasks can complete on time even where potential overruns are reported, this value can be positive or negative. Secondly, there is the predicted overrun at the time the static analysis first reports the presence of a potential or actual overrun in the system. Because the worst-case response time must be greater than the time remaining before the deadline for an overrun to be reported, this value must always be strictly greater than zero. Finally, there is the worst-case overrun; this is the maximum overrun time reported by the static analysis between its initial report and the time at which the task completes. This is likely to be different from the other measures, as the release and execution of other tasks in the system affects the worst-case response time of a task between the time it is released and the time it completes execution.

For the purposes of this work, two of the above overrun measures are considered; the actual recorded overrun time, and the predicted overrun time. The actual overrun time is chosen because it gives a good indication of the system's actual operation; the predicted overrun time gives a view of the system comparable with that taken in traditional static analysis method.

6.5.1 Accuracy

To establish the accuracy of the DCA solution with respect to overrun time, an average overrun value is computed for each task set used. This value is shown plotted against the DCA accuracy in Fig. 11 (actual overrun time) and Fig. 12 (predicted overrun time). Each point on the graph represents the results of the simulation for a whole task set. As with the earlier accuracy graphs against utilisation, the accuracy of potential overruns and actual overruns is considered separately.

As with graphs of system utilisation against accuracy, there is an imperfect correlation between either measure of overrun time and DCA accuracy. However, particularly on the graph showing actual overrun time, it can be seen that

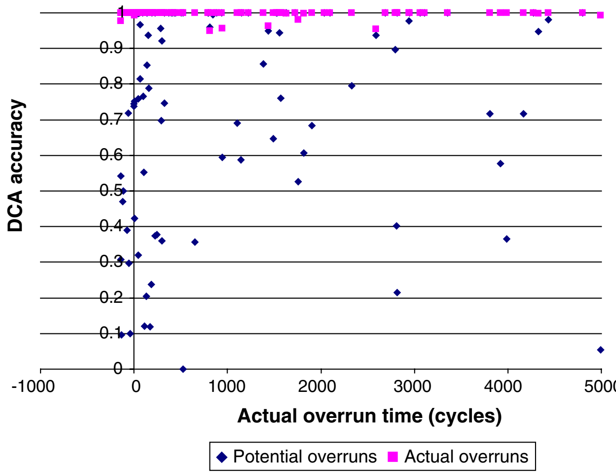


Fig. 11 DCA accuracy against actual overrun time

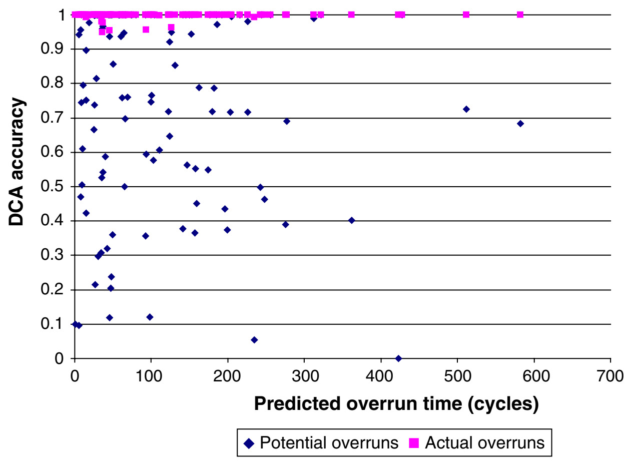


Fig. 12 DCA accuracy against predicted overrun time

in situations where there is low average overrun, the accuracy is spread across the whole range, whereas for higher values of overrun the accuracy tends to be higher. This supports the view that situations with lower overrun values are more difficult to detect. The reasons for the imperfect correlation were considered, and judged to be due to a combination of factors, the main ones being the selection of DC parameters and the fact the learning of the characteristics may cause inconsistent results in the initial stages of the simulation.

Again, the detection of actual overruns does not seem to be affected by the average overrun value of the task sets: this suggests that the DCA in this configuration is well suited to the detection of anomalies. However, it is clearly less suited to the prediction of potential overruns, and there is no apparent correlation between the average overrun value and the accuracy of prediction.

6.5.2 Responsiveness

The responsiveness of the DCA solution, plotted against overrun time, shows more interesting characteristics when plotted on an individual task set basis. Graphs for four different task sets are shown in Figs. 13, 14, 15, 16.

These graphs show distinct and obvious patterns between the overrun time and responsiveness. A reason for the distinctiveness of the results presented here is that the results are at a finer grained level, i.e. they represent the behaviour of one task set rather than the aggregated behaviour across many task sets; each data point on the graph represents one execution instance of a task. The data points on the graph are clustered together in a number of distinct regions, which are caused both by different tasks overrunning, and also by different overrun conditions (for example, the status of the run queue when the overrun is detected). The modal nature of this data can be seen

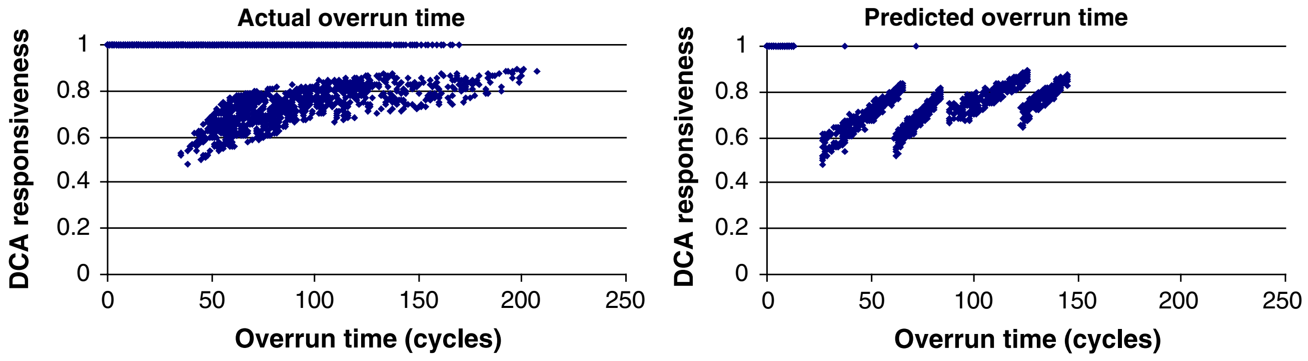


Fig. 13 DCA responsiveness against overrun time–task set A

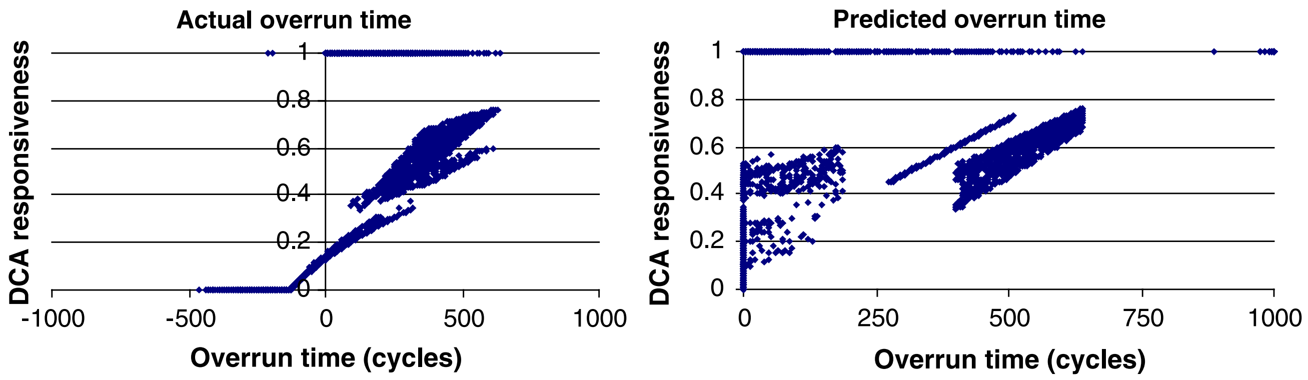


Fig. 14 DCA responsiveness against overrun time–task set B

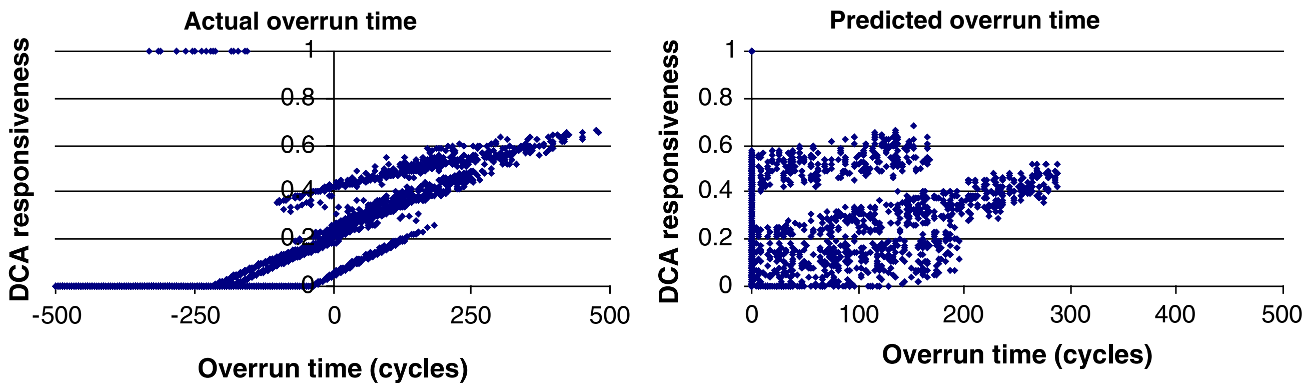
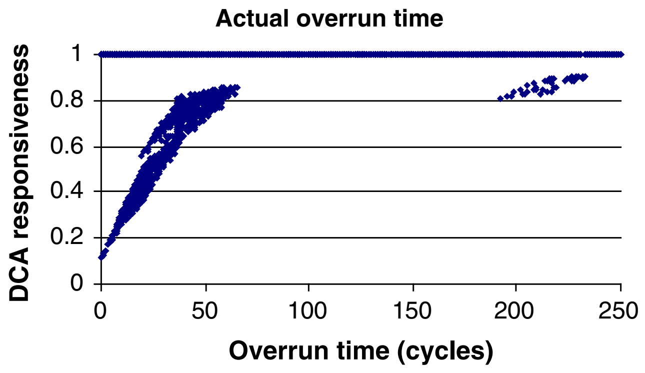


Fig. 15 DCA responsiveness against overrun time–task set C

particularly for the predicted slack graph in Fig. 13; clearly, the pattern produced is the result of overruns being detected in four different system states, each state being responsible for one region on the graph. However, the graph of actual overrun shows that these distinct regions have combined, suggesting either that all the four predicted overrun conditions merge into one, or simply their effects overlap such that the pattern can no longer be easily observed. The reason for this is that predicted overruns tend to be smaller in size, and more transient in their nature than actual overruns. It may be the case that the potential for an overrun may no longer exist due to the system state (e.g. contents of the run queue and the task that is executing) changing before the DCA is able to detect the problem.

It can also be seen in all the graphs that the general trend is for responsiveness to increase as the amount of overrun increases; again, this supports the notion that lower amounts of overrun are more difficult to detect than higher values. In all the task sets, the differences between the



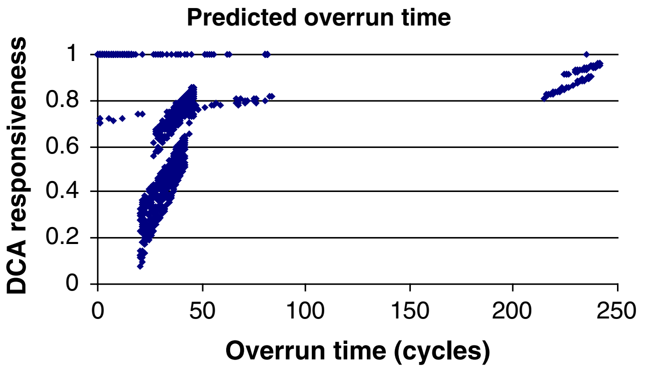


Fig. 16 DCA responsiveness against overrun time–task set D

predicted and actual overrun patterns can be attributed to either the tasks overrunning in a different way to that predicted, or simply the overlapping of the effects altering the patterns observed.

To make comparisons between different task sets, it is necessary to combine the responsiveness and overrun values to produce a single measurements for each task set. These combined measurements are shown in Fig. 17 (actual overrun) and Fig. 18 (predicted overrun).

As with the accuracy graphs, the data points are widely scattered, however particularly with the graph comparing actual overrun and responsiveness there appears to be a pattern to the data. The lowest responsiveness values occur only when the average overrun values are very low or negative, and appears to increase steeply to a peak somewhere shortly before an overrun time of 1,000 cycles. From this peak there appears to be a tailing off as the overrun value increases, although this is not as clear as there are considerably fewer data points in the graph for higher overrun values (the median average slack value is 1,224 cycles).

Overrun values appear to show potential as a measure of problem difficulty, particularly when examining the results obtained for a single task set. However, it is clear that a more sophisticated method of using this information, both for comparison between task sets and for the tuning of DC parameters, is needed. The derivation of a better data aggregation method is left as future work.

6.6 Summary of analysis

From these results, we can see that, even when applied across multiple different task sets, the DCA is successful at detecting the presence of actual overruns in the system. This suggests that our choice of PAMP signal is an effective one, and its high weighting causes DCs to be triggered correctly when overruns do occur. However, the DCA is not as effective at the prediction of potential overruns as the static analysis methods used for comparison—this is to be anticipated as the static analysis method is defined as being a perfect predictor.

From the results, it can be seen that overall system utilisation is not a sufficiently sophisticated measure of problem difficulty, as there is little correlation between the accuracy and responsiveness of the DCA results when compared with utilisation. However, the alternative

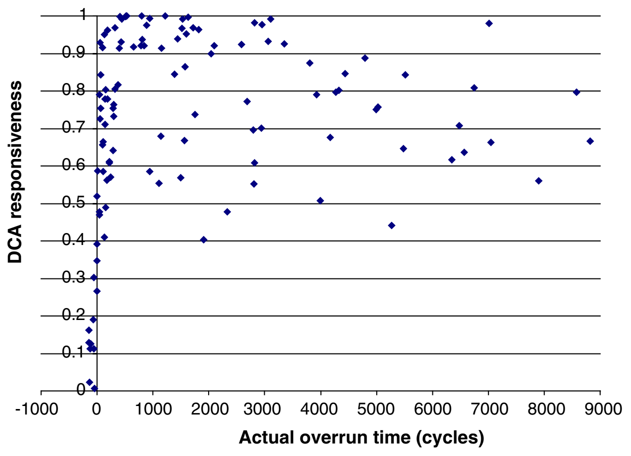


Fig. 17 DCA responsiveness against average actual overrun

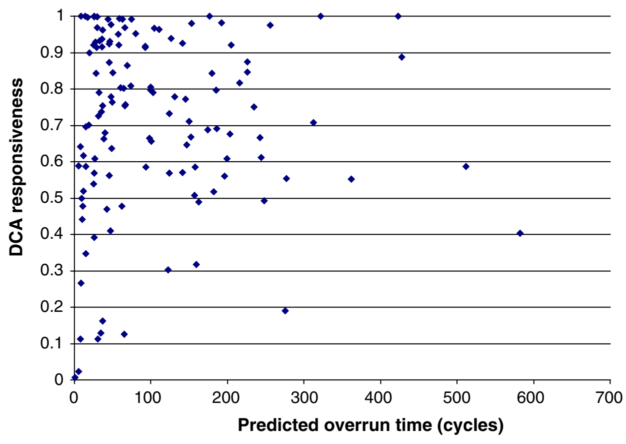


Fig. 18 DCA responsiveness against average predicted overrun

measure based around overrun time appears to give a better correlation, especially when examined in the context of a single task set. This measure should be effective, providing that a better method can be devised which allows comparison of results across multiple task sets.

An important consideration here is that the analysis does not consider the presence of false positives; this is an area which will need to be examined as a part of any further work.

7 Conclusions and future work

This paper has investigated some of the challenges brought about by the application of immune-inspired techniques to a problem associated with the development of RTES. The applicability of the DCA has been examined across a wide variety of task sets, and found to be generally good at detecting deadline overruns but less successful at predicting potential overruns especially when the predicted slack is small. This is as expected as most classification style problems are considered harder when the difference between the object of interest and everything else is small.

To help guide further work, the task scheduling problem has been examined in detail, and a potential measure of problem difficulty has been formulated based on task slack time, against which the performance of the DCA can be evaluated. Future work is to examine how different properties of the DCA can influence the efficiency, effectiveness and robustness.

Acknowledgments This work is supported by EPSRC and Philips Research through a CASE award, reference CASE/CNA/04/78.

References

1. Burns A, Wellings A (2001) Real-time systems and programming languages, 3rd edn, Pearson Education, Upper Saddle River
2. Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM J Appl Math 17:416–429
3. Lundqvist T, Stenstrom P (1999) Timing anomalies in dynamically scheduled microprocessors. In: Proceedings of 20th IEEE real-time systems symposium (RTSS) 1999, pp 12–21
4. Engblom J (2003) Analysis of the execution time unpredictability caused by dynamic branch prediction. In: Proceedings of 9th real-time and embedded technology and applications symposium (RTAS) 2003, pp 152–159
5. Holsti N, Saarinen S (2002) Status of the Bound-T WCET tool, Proc 2nd international workshop on worst-case execution time analysis
6. Theiling H, Ferdinand C, Wilhelm R (2000) Fast and precise WCET prediction by separated cache and path analyses. Real-Time Syst 18(2):157
7. Lay N, Bate I (2007) Applying artificial immune systems to real-time embedded systems. In: Proceedings of the congress on evolutionary computation (CEC) 2007, pp 3743–3750
8. Greensmith J, Aickelin U, Twycross J (2006) Articulation and clarification of the dendritic cell algorithm. In: Proceedings of international conference on artificial immune systems (ICARIS) 2006, pp 404–417
9. Puschner P, Burns A (2000) Guest editorial: a review of worst-case execution-time analysis. Real-Time Syst 18(2):115–128
10. Bouyssounouse B, Sifakis J (2005) Embedded systems design: the ARTIST roadmap for research and development. LNCS, vol 3436. Springer, Berlin
11. Graaf B, Lormans M, Toetenel H (2003) Embedded software engineering: the state of the practice. IEEE Softw 20(6):61–69
12. Vahid F, Givargis TD (2002) Embedded system design: a unified hardware/software introduction. Wiley, New York
13. Eisenring M, Thiele L, Zitzler E (2000) Conflicting criteria in embedded system design. IEEE Des Test Comput 17(2):51
14. Hart E, Ross P, Nelson J (1998) Producing robust schedules via an artificial immune system. In: Proceedings of the world congress on computational intelligence (WCCI) 1998, pp 464–469
15. Hart E, Ross P (1999) An immune system approach to scheduling in changing environments. In: Proceedings of genetic and evolutionary computation conference (GECCO) 1999, pp 1559–1566
16. Colin A, Petters SM (2003) Experimental evaluation of code properties for WCET analysis. In: Proceedings of 24th IEEE real-time systems symposium (RTSS) 2003, pp 190–199
17. Bernat G, Colin A, Petters SM (2002) WCET analysis of probabilistic hard real-time systems. In: Proceedings of 23rd real-time systems symposium (RTSS) 2002, pp 279–288
18. Audsley NC, Burns A, Richardson MF, Wellings AJ (1991) Hard real-time scheduling: the deadline monotonic approach. In: Proceedings 8th IEEE workshop on real-time operating systems and software 1991, pp 133–137
19. Sha L, Rajkumar R, Sathaye SS (1994) Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. Proc IEEE 82(1):68–82
20. Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):40–61
21. Katcher DI, Arakawa H, Strosnider JK (1993) Engineering and analysis of fixed priority schedulers. IEEE Trans Softw Eng 19(9):920–934
22. Harter PK Jr (1987) Response times in level-structured systems. ACM Trans Comput Syst 5(3):232–248
23. Buttazzo GC, Sensini F (1999) Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. Trans Comput 48(10):1035
24. Stepney S, Smith R, Timmis J, Tyrrell A, Neal M, Hone A (2005) Conceptual frameworks for artificial immune systems. Int J Unconv Comput 1(3):315–338
25. de Castro LN, Timmis J (2002) Artificial immune systems: a new computational intelligence approach. Springer, Berlin
26. Hart E, Ross P (1999) The evolution and analysis of potential antibody library for use in job-shop scheduling. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimisation. McGraw-Hill, New York, pp 185–202
27. Medzhitov R, Janeway CA Jr (1998) Innate immune recognition and control of adaptive immune responses. Semin Immunol 10(5):351–353
28. Twycross J, Aickelin U (2005) Towards a conceptual framework for innate immunity. In: Proceedings of international conference on artificial immune systems (ICARIS) 2005, pp 112–125
29. Neal M, Feyereisl J, Rascunà R, Wang X (2006) Don't touch me, I'm fine: robot autonomy using an artificial innate immune system. In: Proceedings of international conference on artificial immune systems (ICARIS) 2006, pp 349–361
30. Zhang X, Dragffy G, Pipe AG, Zhu QM (2005) Artificial innate immune system: An instant defence layer of embryonics. In:

Proceedings of international conference on artificial immune systems (ICARIS) 2005, pp 302–315

31. Burnet FM (1968) Evolution of the immune process in vertebrates. Nature 218:426–430
32. Stibor T, Mohr P, Timmis J, Eckert C (2005) Is negative selection appropriate for anomaly detection? In: Proceedings of genetic and evolutionary computation conference (GECCO) 2005, pp 321–328
33. Matzinger P (1994) Tolerance, danger, and the extended family. Annu Rev Immunol 12(1):991–1045
34. Medzhitov R, Janeway CA Jr. (2002) Decoding the patterns of self and nonself by the innate immune system. Science 296(5566):298–300
35. Greensmith J, Aickelin U, Cayzer S (2005) Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection. In: Proceedings of international conference on artificial immune systems (ICARIS) 2005, pp 153–167
36. Kim J, Bentley P, Wallenta C, Ahmed M, Hailes S (2006) Danger is ubiquitous: detecting malicious activities in sensor networks using the dendritic cell algorithm. In: Proceedings of international conference on artificial immune systems (ICARIS) 2006, pp 390–403
37. Greensmith J, Aickelin U (2007) Dendritic cells for SYN scan detection. In: Proceedings of genetic and evolutionary computation conference (GECCO) 2007, pp 49–56